Understanding the Response to Open-Source Dependency Abandonment in the npm Ecosystem

Courtney Miller, Mahmoud Jahanshahi, Audris Mockus, Bogdan Vasilescu, Christian Kästner

courtneymiller@cmu.edu, mjahansh@vols.utk.edu, audris@utk.edu, vasilescu@cmu.edu

Carnegie Mellon University, Pittsburgh, PA, USA University of Tennessee, Knoxville, TN, USA

***Abstract*—Many developers relying on open-source digital infrastructure expect continuous maintenance, but even the most critical packages can become unmaintained. Despite this, there is little understanding of the prevalence of abandonment of widely-used packages, of subsequent exposure, and of reactions to abandonment in practice, or the factors that influence them. We perform a large-scale quantitative analysis of all widely-used npm packages and find that abandonment is common among them, that abandonment exposes many projects which often do not respond, that responses correlate with other dependency management practices, and that removal is significantly faster when a package's end-of-life status is explicitly stated. We end with recommendations to both researchers and practitioners who are facing dependency abandonment or are sunsetting packages, such as opportunities for low-effort transparency mechanisms to help exposed projects make better, more informed decisions.**

I. Introduction

Many widely-used open source packages serve as digital infrastructure for countless applications downstream [1]. Yet, much of this infrastructure is maintained by a small number of overburdened and underappreciated, often volunteer, developers who may disengage at any point [1]–[3]. Maintainers often disengage for commonly-occurring reasons [4], such as losing interest or switching jobs. More often than not, when that happens, nobody else steps up and the package becomes fully abandoned [5]. This suggests that dependency abandonment will always be a risk that users of open-source infrastructure will be exposed to. And indeed developers worry about abandonment – e.g., because of the increasing incompatibility with other changes and fear of not receiving security patches [6], [7] – to the point that some organizations have explicit policies to restrict the use of end-of-life software components. The tension between this widespread reliance on open source and the lack of certainty surrounding ongoing maintenance efforts is at the heart of the question of *open source sustainability* [1], [7].

Despite the widespread concerns surrounding dependency abandonment, we know very little about its prevalence or how developers react in practice. Research has primarily focused on *preventing* or *predicting* abandonment by reducing disengagement [4], [5], [8] or improving onboarding [9]–[11], rather than studying what happens when abandonment occurs. A key exception is our recent interview study with developers where we studied their *perceptions* of abandonment, but without quantifying the prevalence or reactions in practice [7].

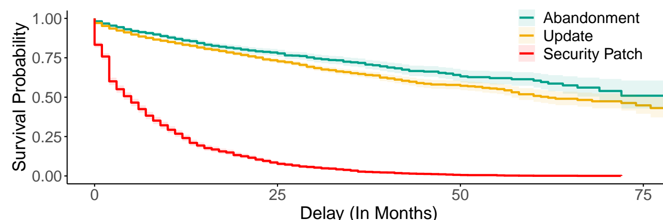


Fig. 1. Survival probability for event "dependency event is not resolved" w.r.t. the date of event occurrence within dependent project's lifetime.

In this paper, we report on a large-scale, quantitative study exploring the prevalence of, impact of, and response to the abandonment of widely-used packages in the JavaScript npm ecosystem. Specifically, we design an approach to detect abandonment at scale, collect a large sample of dependent projects that were exposed to abandonment across all of GitHub, and observe their responses to abandonment. We compare reactions to abandonment with other dependency management practices of updating dependencies with and without known vulnerabilities. Finally, we use statistical modeling to investigate what factors impact likelihood and speed of abandoned dependency removal.

Even with a conservative operationalization, we find that the abandonment of widely-used packages is prevalent, with 15% of widely-used packages becoming abandoned within our six-year observation window. Those abandoned packages expose many dependents, but average direct exposure even for widely-used packages is lower than might be expected, suggesting that collaborative *responsible sunsetting* strategies might be feasible. Developers seem to care about abandonment – 18% of exposed projects remove the abandoned dependency, which is roughly comparable with other dependency management practices such as installing updates (cf. Fig. 1), but reactions to abandonment tend to be delayed – in fact, removal of abandoned dependencies strongly correlates with other good development practices, including regular dependency updates. Finally, making the abandonment status of a package clear can help exposed projects react faster (58% higher chance of reaction on average, at any point in time), suggesting opportunities for low-effort transparency mechanisms to help exposed projects make better, more informed decisions. Overall, our results suggest many opportunities to foster *responsible use*

of open source for developers and *responsible sunsetting* for maintainers.

In summary, we contribute (1) a detailed methodology for detecting abandoned packages at scale; (2) a quantification of the prevalence of widely-used package abandonment in the npm ecosystem; (3) a large-scale analysis of the response to abandonment of exposed projects across GitHub and a comparison to other dependency management practices; (4) a logistic regression model illustrating which dependent project characteristics impact the likelihood of removing abandoned dependencies; and (5) a survival model demonstrating the impact of providing explicit notice of abandonment on dependent project removal rates.

II. Research Questions and Related Work

Reusing open source frameworks, packages, and other abstractions forms *software supply chains* [12], where packages rely on "upstream" dependencies created and maintained by others, that often have their own dependencies, creating the chain. Such reuse speeds up development, but also brings risks "downstream." Dependencies may introduce breaking changes in an update [13], become incompatible with other dependencies [14], [15], contain security vulnerabilities [16]–[19], or even get attacked through supply chain attacks [20]–[24].

A. How Big is the Abandonment Problem?

What We Know. Package abandonment is an understudied risk of open source dependencies. Developers worry about abandonment (e.g., online [25]) particularly from a security perspective, since an abandoned package may no longer receive security patches [6], [7], [21]. Our recent interview study also revealed developers' frustration that they will not receive the new features or support they had hoped for, that the package will become increasingly less useful as requirements and the environment change, and that the package will become incompatible with other evolving infrastructure [7].

What We Don't Know. We have very little data about how prevalent abandonment is among widely-used packages or how many downstream projects are exposed. Sonatype's 2023 State of the Software Supply Chain report finds that 18.6% (24,104) of open-source packages that were maintained the prior year no longer qualify as maintained that year [26], but it is not clear how this data was collected and whether such results generalize to widely-used packages that might be considered critical digital infrastructure. Quantifying the frequency of abandonment and the resulting exposure downstream is needed to understand the scope of the problem, leading us to ask:

- **RQ1a** How prevalent is abandonment among widely-used npm packages?
- **RQ1b** How many open source projects are exposed to abandoned dependencies?

B. What are Downstream Developers Doing About Abandoned Dependencies? ("Responsible Use")

What We Know. Dealing with abandoned dependencies is a facet of *dependency management*. Dependency management practices, especially around versioning and breaking changes [13], [27]–[31] and security patches [16], [17], [32]–[34], have been extensively studied, and are considered highly important [26], [35]. Research suggests that generally keeping dependencies up to date correlates with better security outcomes [36]. Recently, the US White House even mandated the tracking and documenting of dependencies (using software bill of materials, SBOM) for software sold to the government [37].

A central theme in much of the empirical research on dependency management is that the vast majority of open source projects rarely or never update dependencies, even those with known security vulnerabilities [16], [27], [32]–[34], [38]–[44]. For example, Kula *et al.* [34] studied dependency updates across 4,600 GitHub projects and found that the majority tend to not update dependencies even when security vulnerabilities are involved, with 81.5% of projects having outdated dependencies. Similarly, Decan *et al.* [16] estimated that it takes almost 14 months for 50% of projects to install a patch for a vulnerable dependency.

There have been many attempts to improve dependency management practices. Software composition analysis (SCA) tools, such as *dependabot, Sonatype,* and *Snyk*, track dependencies and their updates and alert developers of known vulnerabilities. Studies show that using such SCA tools can improve dependency management outcomes [38], [45], [46]. The adoption of SCA tools is widely seen as a best practice [35], but these tools suffer from many problems, especially high false positive rates and resulting notification fatigue [45]–[48]. Furthermore, semantic versioning with floating dependency versions enables automatic installation of patches, but this practice is controversial since it can also introduce risks of breaking changes and deliberate supply chain attacks [27], [32].

What We Don't Know. Little is known about how developers respond to dependency *abandonment*, and how dealing with abandoned dependencies compares to other dependency management practices. Tools to help with dependency abandonment are rare,[1] to the frustration of practitioners [7]. Generally, developers can choose to continue using abandoned dependencies if they do not (yet) pose actual problems, or they can take various actions that all involve removing the dependency and replacing it with something else. In this paper, we quantitatively study at scale how often and how fast developers respond to abandonment and how (or whether) this differs from other dependency management practices:

- **RQ2a** How often and how fast do dependent projects remove abandoned open source dependencies?
- **RQ2b** How does this compare to how projects update dependencies in general?
- **RQ2c** How does this compare to how projects update dependencies with security vulnerability patches?

We know little also about how individual developers make decisions about removing abandoned dependencies. Our in-

[1] Exceptions are FOSSA's Risk Intelligence service, currently in beta, and a recent research prototype by Mujahid *et al.* [49].

terviews suggest that some developers are very regimented about removing abandoned dependencies (sometimes driven by policies requiring it or a feeling of responsibility) while others prefer to wait for something to break [7]. It is unclear whether the developers that attend to abandoned dependencies are the same ones that follow good dependency management practices and possibly good development practices in general. Thus, we explore whether how developers deal with abandonment associates with other development practices and project characteristics:

RQ3 What dependent project characteristics are associated with removing abandoned dependencies?

C. What Can Maintainers Do? ("Responsible Sunsetting")

What We Know. In open source, developers make many decisions based on publicly available information, without explicit coordination [50]–[56]. This includes complex inferences like choosing which developers to follow [57], [58] or hire [59], [60] and which projects to depend on [13], [61]. A lack of action is often attributed to a lack of awareness – e.g., 69% of surveyed developers reported not having updated dependencies with known security vulnerabilities because they were unaware of the vulnerability [34]. Maintainers can shape how they present their packages to influence the actions of their users and contributors – such mechanisms are often studied in the context of *signaling theory* [50]–[52] and *nudging theory* [45], [46], [56]. For example, developers may include *badges* in their README to signal practices and expectations, such as signaling that a project finds rigorous automated testing and frequent dependency updates important, which may then shape the behavior of contributors [52], [62]. Such nudges can be incorporated in the design of tools, e.g., to accelerate the completion of overdue pull requests [56]. In the CRAN ecosystem, volunteers explicitly coordinate to inform their dependents (within the ecosystem) about breaking changes [13], but this practice is rare otherwise.

What We Don't Know. Little is known about what maintainers can do as their final actions to help the community when they decide to stop maintaining a package. Developers make inferences about the abandonment status of projects with all kinds of information (e.g., the date of the last commit, recent issue discussions, forum discussions), yet they often struggle to determine conclusively whether a project is abandoned [7]. We conjecture that even simple actions like publicly announcing that a project will no longer receive maintenance can shape how affected developers respond to abandonment. As a starting point to explore responsible sunsetting strategies, we ask:

RQ4 How does announcing the abandonment status of a package impact how fast dependent projects remove the abandoned dependency?

III. Detecting Open-Source Package Abandonment

To study abandonment at scale, we first design two conservative heuristics and a manual validation process for the heuristics. Specifically, we look for cases of abandonment with a clear abandonment event, with the evidence coming from either (1) documentation or metadata explicitly indicating a package will not receive further maintenance *(explicit-notice abandonment)*; or (2) shifts in activity patterns from regular maintenance to not receiving any development activity for two years *(activity-based abandonment)*. We intentionally pursue a high-precision detection strategy (detecting real and clear abandonment events), while accepting lower recall (missing some cases of abandonment, e.g., projects that slowly became inactive over an extended period of time). This will result in an undercount in RQ1 (scope of abandonment) but increases confidence in analyses based on our data (RQ2-RQ4).

A. Explicit-Notice Abandonment

We identify packages as abandoned when developers explicitly express their intention to no longer maintain them. First, we manually searched for explicit signals of abandonment (e.g., GitHub "archive" flag) through the repositories of packages that had been likely abandoned based on long periods of observed inactivity. Once we identified some explicit signals, we then searched for additional packages with those signals, verified the precision of the signal, and used the same process to search for additional signals of abandonment. We repeated this process until we found no additional reliable signals. In the end, we used the following three signals:

- *Github archive flag:* In 2017, GitHub introduced the ability to archive a repository [63]. Archived repositories are read-only and display an '*archived*' banner on GitHub. We consider a package as abandoned when its GitHub repository is archived and use the date of archival (shown on GitHub) as the date of abandonment.
- *No-maintenance-intended badge:* Some developers declare their intention not to provide maintenance with a '*no maintenance intended*' badge in their README file [64]. We consider a package as abandoned if its GitHub repository has a README file containing the badge and use the date of the commit introducing the badge as the date of abandonment.
- *Other abandonment description in README:* Finally, developers can textually describe that their package has reached the end of its life in many ways in their README file, such as "*[project] is no longer maintained*" [65]. We consider a package as potentially abandoned if the first 10 lines of the README (excluding URLs and code) contains one of the following text fragments: '*abandoned*', '*deprecated*', '*no longer maintained*', '*no longer supported*', or '*unmaintained*' (fragments identified iteratively as described above). We manually checked all matching packages that did not also have one of the other explicit-notice signals. We consider a package as abandoned if its repository's README states so and use the date of the README change that introduced this description as the date of abandonment.

If a package contained multiple of the above signals, we used the earliest date as the date of abandonment.[2]

[2]Now, npm also has a `deprecation` flag. We did not include it in our analysis because it was introduced near the end of our observation window and does not record the date when the flag was added.

B. Activity-Based Abandonment

Detecting abandonment from a package's activities is non-trivial – some packages may be considered *feature complete* [66] requiring very little maintenance or changes to the code. Therefore, we seek cases where there is a clear shift from regular repository activity levels to a sharp drop indicating abandonment. Conservatively, we look for 2+ years of regular maintenance (operationalized as 10+ total events from contributors per year, including commits, issue comments, and issue close events) representing the *pre-abandonment* phase followed by 2 years with no activity from contributors representing the *abandonment* phase. This provides a relatively clear abandonment point, enabling us to observe reactions downstream. We consider the date of abandonment to be the time of the last commit i.e., the beginning of the *abandonment* phase since that was when the activity dropped off.

We experimented with different heuristics and thresholds (for time and permissible residual activity) and arrived at the design above after reviewing how related work has operationalized abandonment [5], [7], [67]–[70] and exploring activity patterns before and after explicit notices were added to a sample of packages meeting our explicit-notice heuristic. We repeatedly tested the robustness of our heuristic with different thresholds, manually validated the accuracy on a sample of packages, and determined that allowing even a small amount of residual activity (e.g., 3 commits per year) introduced too many false positives of feature-complete but still sporadically-maintained packages; we found similar problems with a smaller observation window. We found that a strict threshold of no residual activity for 2 years had an almost perfect precision while still finding many abandoned packages.

IV. RQ1: Abandonment Prevalence and Exposure

We begin by quantifying the frequency of abandonment among widely-used *packages in npm*. We focus on widely-used packages, rather than including the many more that never gained traction, as a way to focus on digital infrastructure.

We then estimate exposure of abandonment on projects in GitHub that were active and depended directly on an abandoned package at the time of its abandonment (cf. Fig 2). We estimate exposure for *all of GitHub* without restricting the analysis to popular or widely-used projects, because abandonment affects all kinds of users of open source, whether they build popular libraries or applications, or just maintain personal projects. Different users may have different views and perceive different pressures (and may even have policies requiring them to avoid end-of-life dependencies), as we will explore later, but abandonment and sustainability are important for all users of open source who plan to maintain their own project. Users of open source dependencies who write closed-source applications are obviously not captured by our analysis; exposure rates should be considered as a lower-bound estimate.

A. Research Methods

We identify abandoned packages and exposed projects with data from npm, GitHub, and World of Code [71]. We restrict our analysis to abandonment in a six-year observation window from January 2015 to December 2020, for which we can collect all relevant data at scale and which starts after npm has been established and widely used.

Identifying Abandoned Widely-Used npm Packages. To scope our analysis to widely-used packages that can have a substantial impact on the ecosystem if abandoned, we consider only the 36,164 of 1,063,835 npm packages (in 2020) that had at least 10,000 downloads in any month of our observation window (per npm download statistics [72]). We use downloads (rather than reverse dependencies) since they capture both public and private use of packages. Next, we excluded 940 packages because they shared a GitHub repository with other packages, and we cannot clearly attribute the repository-level activity data (e.g., issues) to a single package – this is common when a repository contains the source code for multiple related packages (sometimes called a *project foundry* or *mono-repository*). We also filtered out 7,124 packages that never had more than 10 total activities by contributors (cf. Sec. III-B) in any year of our observation window – they may have been abandoned before our observation window or have low activity levels indistinguishable from abandonment. After filtering, the dataset contains 28,100 widley-used npm packages.

We then identify which of these packages were abandoned and when, as described in Sec. III, using both the explicit-notice and activity-based detection approaches over the entire observation window. Because the activity-based abandonment definition requires two years of activity observation before abandonment and after, it can only occur in the two middle years of our observation window, whereas explicit-notice abandonment can occur in all six years.

Identifying Exposed Dependents. Next, we identify dependent projects across all of GitHub (not just npm packages) that were directly exposed to abandonment. In contrast to prior work on dependency management [16], [27], [38], [42], we explicitly consider all projects rather than just reverse dependencies within npm to capture the impact on open source developers broadly, not just on other package maintainers. Searching across all of GitHub imposes substantial challenges due to its scale and limitations of its search APIs. Instead, we use *World of Code (WoC)* to find all dependents of the detected abandoned packages. WoC is a large scale analysis infrastructure that indexes and curates nearly all public open source code, intended for research studying software supply chains [71], [73]; we use version V, the latest at the time of this analysis.[3] Specifically, in addition to content of the files, commits and trees extracted from all repositories, WoC also parses each version of every file (or blob using git terminology). Projects that rely on npm packages use specific file named packages.json to specify all direct dependencies. As a result, for each commit modifying packages.json file WoC provides a list of upstream dependencies as b2Pkg (blob-to-

[3]GitHub itself has a *Dependency Insights* feature. However, the functionality is closed-source and poorly documented, and our initial experiments showed too many incorrect dependency entries for it to be trustworthy.

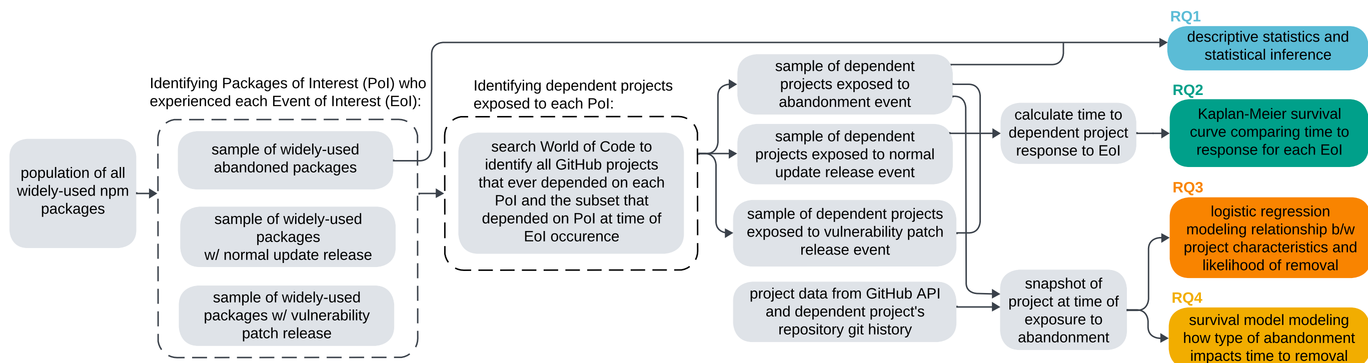


Fig. 2. Overview of our data collection and analysis

package) map. We queried WoC using indexes b2c (blog-to-commit), c2p (commit-to-project), and b2Pkg to retrieve all GitHub projects, excluding forks, that ever depended on any of the abandoned packages in a *package.json* file in their root directory.

Due to the vast number of candidate dependent projects returned by WoC (usually millions) and the nontrivial analysis costs, we perform the analysis on a large sample of 60,000 randomly selected candidate dependents and extrapolate exposure rates to the entire population statistically. We further checked each candidate dependent project in our sample by cloning the project's repository and analyzing the dependencies at the time of abandonment. We use the same step to also detect whether the repository had any commit activity after the time of abandonment. This allowed us to identify the subset of dependent projects that were actively depending on an abandoned project at the time of abandonment who also had at least one commit after abandonment occurred (to ensure they were not entirely inactive themselves).

Limitations. As discussed in Sec. III, construct validity for *abandonment* is difficult to establish. Despite best efforts to design and validate meaningful but conservative heuristics for detection, we may not capture all notions of package abandonment, e.g., adding notices in an external blog or entirely stopping maintenance after years of minimal maintenance activity. For the purpose of this study (actions taken as a result of abandonment) we designed our heuristics to be conservative, hence our abandonment numbers should generally be seen as a lower bound. Additionally, because there are many ways to define abandonment and because we consider multiple definitions of abandonment, there could be packages that meet one definition of abandonment while not meeting another. For example, the maintainers of a package that qualifies as explicit-notice abandoned because they posted a note in the README stating the package will no longer be maintained, could post minor patches down the line potentially making the package not qualify as activity-based abandoned. Even packages with long periods of inactivity and end-of-life notices may be revived years later [5].

We may also miss exposed dependent projects that have since been deleted (and have not been not archived in WoC). In addition, our analysis relies on timestamps of commits, which are usually but not necessarily reliable. In addition to the direct exposure we measure here, there may be instances of indirect exposure where none of the immediate upstream packages are abandoned, but at least one of these packages depends directly or transitively on an abandoned package. We chose not to include these transitive dependencies since developers tend not look beyond direct dependencies (see, e.g., [74]) nor do they have much power to change projects further upstream on such complicated issues as replacing functionality.

Finally, our results are specific to heavily downloaded packages in npm within our observation window. Abandonment may be more common among less downloaded packages and dynamics may be different in other ecosystems. Our study cannot capture the behavior of developers using open source dependencies in closed sourced projects.

B. Results

Of the 28,100 widely-used npm packages in our dataset, we identified 4,108 (15%) as becoming abandoned during our six-year observation window. Abandonment events were distributed fairly uniformly across the observation window, without clear patterns or peaks. In addition, abandoned packages were similar to non-abandoned packages, e.g., in terms of peak downloads and stars (Fig. 3).

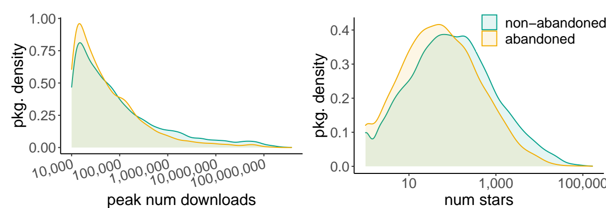


Fig. 3. The distribution of peak download counts during our observation window and current star counts (March 2024) for both non-abandoned and abandoned widely-used npm packages are similar.

Our relatively large sample size for downstream dependent projects affords high generalizability – approximately 0.4% margin of error at 95% confidence level. Assuming the same

abandonment rate of 15%, we estimate[4] that the 4,108 abandoned packages exposed $283,207 \pm 2,096$ GitHub projects (not including forks) who had an abandoned package as a direct dependency at the time of its abandonment (average 69 projects exposed per abandoned package). Of those projects, we estimate that $78,023 \pm 624$ GitHub projects had any commits after exposure, i.e., they were not abandoned themselves at the time and might need to respond.

The directly exposed and subsequently active projects vary widely in their characteristics. They include many small projects, including hobby projects and personal websites, but also popular libraries and end-user products. About half have no stars on GitHub, but 8% have 100 or more stars.

Key Insights: Of the 28,100 widely-used npm packages, 4,108 (15%) were abandoned during our observation window. We estimate that 78,023 dependent projects on GitHub, still active at the time of abandonment, were directly exposed.

V. RQ2: Responding to Abandonment

We detect how often and how fast dependent projects exposed to the abandonment of widely-used packages remove the package after abandonment. Essentially all strategies to respond to abandonment that do not involve preventing it (e.g., contributing financially, taking over maintenance) involve removing the dependency (e.g., replacing it with an alternative, switching to a fork, copying the code, removing the functionality) [7]. Additionally, we compare the response to abandonment to other established dependency management practices, specifically, to developer responses to updates of their dependencies with and without known security vulnerabilities.

A. Research Methods

We detect responses to abandonment (RQ2a) as well as responses to updates (RQ2b) and security patches (RQ2c) with the same three-step research design: (1) We collect a set of *events of interest* among widely used npm packages – package abandonment, package updates, security patches. (2) We identify active projects directly *exposed* to these events. (3) We determine whether and when the exposed projects subsequently *responded* – by removing or updating (cf. Fig 2).

To scale the analysis, we randomly sample both (a) from the set of all possible events to analyze and (b) from the set of all exposed dependents for those dependencies. Note that we use a consistent approach to collect data for different dependency management practices, rather than comparing our abandonment data against previously published results on other dependency management practices – this allows for a direct comparison and avoids potential issues caused by the differences in research design and analyzed populations in past research [16], [32]–[34], [38], [39], [42], [75].

[4]The estimates are based on 2,046,047 candidate exposed projects retrieved with World of Code, from which we randomly sampled and analyzed 60,000. Among those we found 8,305 exposed dependents to the abandoned packages; of those 2,288 had commit activity after exposure.

Removal of Abandoned Dependencies (RQ2a). To analyze the removal of abandoned dependencies, we rely on the data from RQ1, namely the 4,108 abandoned packages we identified and the sample of 2,288 directly dependent projects that were active after the time of abandonment (among the 60,000 dependent projects we analyzed). For each dependent, we analyze their commit history after the abandonment event to measure whether and how long it took them to remove the abandoned dependency from their *package.json* file. While the event must happen within our observation window ending in December 2020, we analyze subsequent reactions until a cutoff date of September 1st, 2023.

Dependency Updates After New Package Releases (RQ2b). To establish a baseline of common dependency management practices, we select a sample of package updates among the 28,100 widely-used npm packages from RQ1 as events of interest. Specifically, to generate a sample similar in size to abandonment, we first identify all releases of widely-used packages released within our six-year observation window, and then randomly sample 6,000 of these releases making sure each is from a unique package. For each update event, we use the same search strategy with WoC described in Sec. IV to detect candidate GitHub projects that directly depended on the package of interest at some point in time. We then analyze a large random sample of those candidate dependents ($> 500,000$), using the same process described in Sec. IV to identify the subset that depended on the package of interest at the time of the update and who had had any commits after the event. If the dependent uses floating version constraints (patterns that match multiple versions, e.g., `^1.4.2` to match release 1.4.2 and any later releases before 2.0.0) that allowed them to automatically update at the time of the event, we discard the dependency from our analysis as it does not require developer intervention to update the dependency. This results in 7,916 observations.

Dependency Updates After Security Patches (RQ2c). As a special version of detecting responses to package releases, we analyze responses to releases that patch known security vulnerabilities, which are usually considered more urgent than other updates. We identified package releases with known security vulnerabilities and corresponding package releases that patch the vulnerability using the OSV database [76]. We select the first release that patches the vulnerability and its release date as the event of interest – that is, we study whether and how fast developers respond to the security patch. We found 442 packages among our set of 28,100 widely-used npm packages that had at least one vulnerable release and corresponding patch release within our six-year observation window. For each of these packages, we randomly selected one patch release, resulting in 442 events of interest. For each security patch release of interest, we use the same search strategy with WoC to detect candidate GitHub projects that depended on the package of interest at some point in time. We then analyze a large random sample of those candidate dependents ($> 500,000$), identifying the subset that depended on a

vulnerable version of the package of interest at the time of the event occurrence and who had had any commits after the event, again discarding issues that were patched automatically due to floating version constraints. This results in 3857 observations.

Analysis. To answer the research questions, we use survival analysis, which specializes in modeling *time-to-event data* and providing estimates of the survival rates for a given population [77]. Survival analysis is designed to account for right-censored data like ours, where the occurrence of the event of interest is only recorded for cases that have experienced the event, and for other cases their data is "censored" because (a) the event was not (yet) observed during the period of observation or (b) they withdrew from the study during the observation period. In our study, we consider a project to be withdrawn and therefore censored if it becomes inactive during our observation period (which we operationalize using the date of its last commit). Additionally, survival analysis can account for the fact that we can observe reactions for different periods of time for different events (for earlier events, developers had longer to react). Specifically, we use the Kaplan-Meier estimator [78], which is a common non-parametric statistic for estimating survival functions [79].

Limitations. To capture the reaction to *average* events, we intentionally do not stratify our analysis by major/minor/patch release or vulnerability severity. Behavior may differ between different subtypes of events, which is not the focus of our study. Similarly, a security patch is not automatically urgent, since the vulnerability may not be exploitable; again, our study only reveals average practices and does not set normative expectations. Finally, our study does not capture the more nuanced behavior of floating dependency declarations when locking dependencies with npm – in such cases, updates matching the versioning pattern may not be fully automated; excluding those cases helps us avoid ambiguity about what actions developers take, but may miss some actions. Limitations from RQ1 also apply.

B. Results

Only 18% of directly dependent projects with any development activity after the abandonment date ever remove the abandoned dependency before our cutoff date (419 of 2,288 in our sample) – the vast majority of dependent projects did not remove the abandoned dependency. Among dependent projects that removed the abandoned dependency, the average time to removal is 13.5 months. Consistent with past research [34], [38], we also observe that many developers do not update dependencies, even those with security vulnerabilities: Only 17% (1,366 of 7,916 in our sample) respond to a random dependency update before our cutoff date with an average time to update of 10.5 months; and 44% (1,720 of 3,857 in our sample) install a patch to a security vulnerability with an average time to update of 8.5 months.

We show survival curves indicating the percentage of dependent projects that react to package abandonment, package updates, and security patches respectively within a given time window in Figure 1, illustrating that security patches are installed at higher rates and faster than other updates and that developers react to abandonment generally at similar rates and with similar latency to random dependency updates. Note that survival rates in the plot are lower than what may be expected from past research, because we censored projects if they became inactive during our observation window – the lack of updates can often be explained by dependent projects becoming inactive whereas dependent projects that remain active for long periods of time after security patch release are much more likely to eventually update.[5]

Key Insights: The response rate for abandonment is similar to updates and lower than the rate for security patches.

VI. RQ3: Characterizing Responsive Dependents

Next, we study population-level differences between the characteristics of projects that remove abandoned dependencies and those that do not.

A. Research Methods

Using our sample of 2,288 dependent projects directly exposed to an abandoned dependency, identified in RQ1, we take a snapshot of each project *at the time of exposure* to abandonment, operationalize numerous factors representing different project characteristics (hypotheses H_1-H_6 described below), and use logistic regression analysis to model the relationship between project characteristics and the likelihood of removing the abandoned dependency.

Hypotheses and Variables. Specifically, the binary response variable is whether the abandoned dependency was removed within two years of abandonment. In addition, we test hypotheses about the association between the following variables and the binomial outcome:

Dependency Count (H_1). We expect that projects with fewer dependencies are less likely to remove abandoned ones. Such projects may pay less attention to dependency management in general and may thus do it less.

Dependency Management Practices (H_2). We expect that projects that manage dependency updates and security patches more actively are also more likely to respond to abandonment. Here we use two variables: First, we model whether there was evidence of software composition analysis (SCA) tool use in the year before exposure (i.e., tool configuration files, README badges, or commits by bots), namely Dependabot, Renovate, Greenkeeper, Snyk, DavidDM, and Gemnasium. Second, we use the standard heuristic by Zerouali *et al.* [42] to calculate the average *technical lag* of all the dependencies the project had at exposure excluding the abandoned one.

Activity (H_3). We expect that more actively developed projects are more likely to respond to abandoned dependencies

[5]As described above, our results do not include dependents that can automatically update dependencies due to floating dependency version declarations. This would account for an additional 33% immediate random dependency updates and an additional 70% immediate security patch updates, also shown in corresponding survival curves in our supplementary material [80].

because developers spend more time on the project overall. We consider two variables: First, we calculate *dependency churn*, i.e., the total number of times the project changed any of its dependencies in the year before exposure. Second, we collect the total *number of commits* in the year before exposure.

Number of Maintainers (H_4). We expect that projects with more maintainers are more likely to respond to dependency abandonment because they have more capacity for maintenance (and dependency management) work. Operationally, we count the number of contributors responsible for the top 80% of commits in the year before exposure.

Corporate Involvement (H_5). We expect that corporate-led projects and projects with more corporate involvement may be more likely to respond to abandoned dependencies because they are more likely to follow explicit end-of-life policies and to explicitly allocate resources to dependency management than volunteer-run projects [81]. Operationally, we check for the presence of commits made in the year before exposure by contributors using a corporate email domain, using the list of corporate domains identified by Spinellis *et al.* [82].

Governance Maturity (H_6). We expect that projects following governance best practices are more likely to respond to abandoned dependencies, conjecturing that responsiveness to abandonment is also seen as a best practice. In particular, we consider six governance practices: having a README, a license, issue templates, pull request templates, contributing guidelines, and a code of conduct, in line with past research associating those with project success [66]. Operationally, for each project, we start collect six binary flags indicating the presence files for each practice at the time of exposure. We then compute a latent trait model [83] to reduce the dimensionality of this dataset. The model assumes that the dependencies between the six observed variables can be interpreted by a few latent variables. The model with a single latent variable fit our data best, confirming that the six indicator variables mostly capture one underlying construct. Finally, we computed the continuous variable *governanceMaturity* representing the factor scores (or "ability" estimates) for the observed response patterns as an operationalization of this latent construct, and used this variable in our subsequent modeling.

For all of the above variables, we collect the relevant data from the GitHub API or from the repository's git history. Where possible, we follow measures developed and validated in past research. We manually validated the construct validity of each factor using a sample of projects to avoid systematic errors by manually verifying that the factor seemed to indeed capture the intended data accurately.

Modeling Considerations. Before estimating the model, we took several steps to ensure model quality and validity. First, after manually examining the distribution of each variable, we removed extreme outliers for variables with highly skewed distributions (top 1% or fewer points), to reduce the risk of high-leverage points biasing our regression estimates. We further removed 7 repositories we failed to clone (likely deleted since) and 14 we failed to compute the average technical lag for, taking our total sample size down from 2,288 to 2,261 after both steps. Next, to reduce heteroscedasticity, we log-transformed the numeric variables as needed [84] (Fig. 4 indicates which variables were log-transformed). The model also includes control variables for repository age (controls for the project's development stage and software evolution) and size (controls for the size of the codebase, measured in bytes).

We further computed the variance inflation factors for each variable to check for presence of multicollinearity [85], checking to ensure each was lower than 5, a common threshold within the statistics community [86]; none of the variables exceeded the threshold. To evaluate the model's overall goodness-of-fit we used McFadden's pseudo-R^2 measure [87]. Finally, we consider model coefficients significant if they are statistically significant at the $\alpha = 0.05$ level. For each variable, we report the exponentiated log-odds regression coefficient so as to report the transformed odds-ratio to aid in interpretation (i.e., the factor by which a 1 unit increase in the predictor increases the odds of the outcome occurring if the odds-ratio is greater than 1, or decreases by if the odds-ration is less than 1), and the significance level (p-value) (cf. Fig. 4).

Limitations. As is usual for this kind of work, despite the careful development and validation described above, the operationalized factors in our model can only capture part of the concepts they are intended to represent and measure. For example, we operationalize governance maturity by detecting six files in the repository: While grounded in prior research and supported by our latent trait (factor) analysis, it likely cannot fully capture the concept of maturity. There may also be additional dependent project characteristics and unobserved confounding factors that we did not include in our model. Our findings should not be considered an exhaustive list, but rather a list highlighting some of the characteristics that *are* constructive when modeling the removal of abandoned dependencies. Hence, as usual, care should be taken when generalizing our results beyond the studied measures.

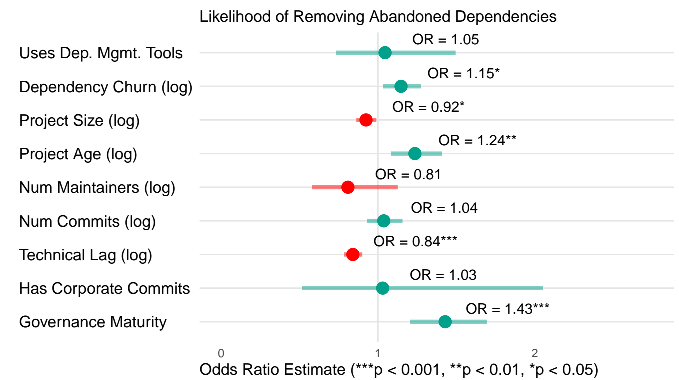


Fig. 4. Summary of the multivariate logistic regression modeling the likelihood of removing an abandoned dependency within two years post abandonment. Confidence intervals (horizontal lines) for the odds ratios (OR) that do not intersect 1 indicate variables with statistically significant effects.

B. Model Results

Regression results in Fig. 4 show five significant effects. One is a strong positive effect of *governance maturity* (supporting H_6): For projects with one standard deviation higher governance maturity score we expect to see about 43% increase in the odds of removing the abandoned dependency. The model also shows that higher *technical lag* is, on average, statistically significantly negatively associated with the likelihood of removal (supporting H_2).

Projects with higher *dependency churn* are generally more likely to remove abandoned dependencies (supporting H_3). To demonstrate the interpretation of the exponentiated regression coefficient, for every factor e ($\simeq 2.72$) increase in the amount of dependency churn (note the log transformation), the odds of removing the abandoned dependency for the average project in our sample multiply by 1.15, holding all else constant. Additionally, as expected we observed a significant effect for both control variables *project age* and *project size*.

The explanatory variables *num dependencies* (H_1), *use of dependency management tools* (H_2), *num commits* (H_3), *num maintainers* (H_4), and *num corporate commits* (H_5) were not significant in the model meaning we have insufficient evidence to reject the null hypothesis that these factors do not impact the likelihood of abandoned dependency removal.

Key Insights: Projects that are more mature, have higher dependency churn, and keep more up to date on dependency updates are more likely to remove abandoned dependencies within two years.

VII. RQ4: Influence of Announcing Abandonment

A. Research Methods

RQ4 extends RQ2 and RQ3 using the same data as RQ2, except we model the distinction in responses to packages that were explicitly declared as abandoned (explicit-notice) as compared to packages that just stopped maintenance (activity-based) as introduced in Sec. III. Similarly to RQ2, we again apply survival analysis to model the time to removal of the abandoned dependencies, except now we use a multivariate Cox proportional-hazards model [88] to jointly control for all factors modeled in RQ3 (see Sec. VI-A for factor definitions). Cox regression is commonly used in medical research for modeling the association between the survival time of patients and one or more predictor variables. In our case, we use Cox regression to estimate the effect of an explicit notice of abandonment on the rate of dependency removal events happening at a particular point in time, i.e., the "hazard rate."

B. Results

We observe after controlling for all the factors we hypothesized are associated with removing abandoned dependencies in RQ2, that there is a statistically significant relationship between the presence of an explicit notice of abandonment for a given dependency and an increased likelihood of the abandoned dependency being removed by downstream projects (cf. Fig. 5). Holding the other covariates constant, having an explicit notice increases the removal hazard by a factor of 1.58, or 58%, with a 95% confidence interval of 1.26 to 1.98. This is in alignment with our expectations, because explicit-notice abandoned packages provide a clear signal to dependents and are more visible sooner.

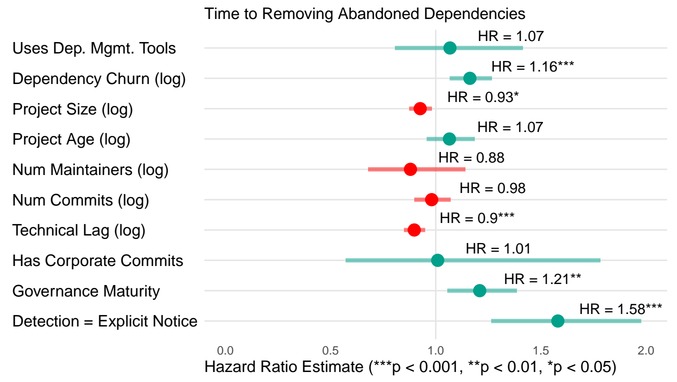


Fig. 5. Summary of the Cox proportional hazards multivariate survival regression modeling the time to removing an abandoned dependency. Confidence intervals (horizontal lines) for the hazard ratios (HR) that do not intersect 1 indicate variables with statistically significant effects.

Key Insights: Packages that provide an explicit-notice of abandonment tend to be removed at significantly faster rates compared to those that do not.

VIII. Discussion and Implications

The Scale of Abandonment. Our study finds that abandonment, even among widely-used npm packages, is fairly common. While many developers carefully analyze signals like the number of stars, responsiveness to issues, or number of contributors when adopting dependencies [61], [89] and past studies have shown several statistical predictors for survival [5], [66], [68], we were surprised by the scale of abandonment among packages that had healthy signals, were among the most popular packages on npm, and were generally similar in their distribution of stars and past activity to those with sustained maintenance. Given that open source maintainers may disengage for all sorts of reasons, such as losing interest, changing jobs, and starting a family [4], users of open source are likely not able to entirely escape abandoned dependencies with careful upfront vetting, but may also need to actively consider strategies to manage abandoned dependencies – an area also called for in our recent interview study [7] for which maintainers have with little existing support.

The Rippling Effects of Abandonment. Although abandonment rates are fairly high, we were surprised at the low rates of direct exposure. While GitHub's *Dependency Insights* page often show thousands to hundreds of thousands of dependent projects for the abandoned packages, the actual *direct* exposure of *active* dependent projects at the time of abandonment was not that high ($\mu = 19$, cf. Sec. IV-B). Many additional dependents of abandoned packages were abandoned even before the package's abandonment, so they

are unlikely to care about it; many others adopted the package even after it was abandoned, possibly knowing and accepting that they will not receive updates.

Package abandonment has vastly more wide-reaching consequences when also considering *indirect* dependencies. On the one hand, this is good news since the few projects that depend directly on an abandoned package can potentially mitigate the consequences of abandonment for the many downstream projects that rely on the abandoned package only transitively. On the other hand, if the maintainers of these intermediate projects do not act, developers have very little means to do anything about indirectly used abandoned packages in their dependency graph. With an increased focus on the entire supply chain through software bill of materials (SBOMs), software composition analysis (SCA) tools and company-wide or agency-wide policies for sunsetting, this can cause a lot of pain for huge numbers of developers, vastly more than those directly exposed. We recommend that **maintainers of popular projects should be particularly attentive to monitoring and reacting to abandoned direct dependencies** due to their outsized lever to benefit the entire ecosystem.

For many abandoned packages in our sample, the small direct exposure would make it feasible to reach out to affected dependent projects (in the context of breaking changes, such proactive actions are not uncommon [13]). However, maintainers currently do not have tools to identify all active direct dependents (e.g., GitHub's Dependency Insights page reports too many false positives, vast numbers of inactive projects, and drowns out direct dependents among many more indirect ones while some dependents may not even be hosted on GitHub). **Researchers or practitioners should explore tools for more targeted outreach to direct active dependents.**

Allocating Resources to Sustain Open Source Communities. Discussions of open source sustainability are often centered on the most widely-used packages that form essential digital infrastructure and usually focus on keeping those projects alive, which may be arguably cheaper in the grand scheme of things than placing the cost for mitigations and replacements on all their dependents. However, the observed low rates of direct exposure may call that balance into question, especially if we can help the exposed projects with a migration guide or through other coordinated action (discussed as "community oriented solutions" in our prior work [7]). There is also a fairness argument regarding the degree to which the often-volunteer maintainers of packages do or should feel responsible to provide ongoing maintenance for their dependents, most of whom never contribute to the package in any way. As we argued previously [7], we believe **it is time to place more emphasis on the *responsible use* of open source rather than attempting indefinite maintenance**.

In our study, we explicitly consider all dependents, not only other packages in npm and not only packages or projects that are popular and form digital infrastructure themselves. That is, many of the exposed dependents are 0-star projects, including personal projects like maintaining a personal website – but all of them were still maintained for some period after exposure to abandonment. Less prominent dependents may have a more relaxed attitude toward abandonment, but they may also be less experienced in dealing with dependencies and likely spend less time on dependency management overall, thus making abandonment possibly even more disruptive. **More research is needed on whether and how to help such developers, rather than only helping and studying the most active developers or the most popular projects.**

Abandoned Dependencies in the Context of Dependency Management. Despite many calls for better dependency management, especially from a security perspective (recently even with the US White House joining in [37]), study after study shows that the majority of developers rarely update dependencies, even those with known vulnerabilities, and even when informed about problems by automated tools [16], [26], [27], [32]–[34], [38]–[44], [75]. If developers do not patch known security vulnerabilities or even add dependencies with known vulnerabilities, should we expect them to care about abandonment? Our results show that different dependency management practices correlate. Developers who generally keep their dependencies up to date are also more likely to react to abandoned dependencies. When (or if) the larger open source community manages to improve dependency management practices in response to perceived higher stakes (e.g., the continuously increasing frequency of supply chain attacks), we expect to also see more people reacting to abandonment – therefore **support to help developers exposed to abandonment will only become more important.**

At the same time, abandonment is different. A dependency does not automatically and immediately become a problem when abandoned – impacts are often delayed and may not even occur in a dependent project's lifespan [7]. A large number of updates and vulnerability fixes can be captured with floating dependency versions (semantic versioning is a common practice in npm [27], [28], automating the "immediate reaction" to 33% of analyzed updates and and 70% of analyzed patch events; although this practice also raises its own security challenges [15], [16], [23]) and various SCA tools can inform and automate update actions. However, there is no equivalent default action or tool automation for abandonment. The decision to remove abandoned dependencies is closer in nature to decisions surrounding technical debt reduction and risk reduction (similar to trying to stay on top of updates to avoid painful large migrations and integration problems later [13]) than the more immediate urgency to patch known vulnerabilities. This is visible in our results (e.g., Fig. 1) where fixing vulnerabilities is more likely and faster than reacting to abandoned dependencies, but reactions to abandoned dependencies are fairly similar to reactions to random dependency updates, even in the absence of any automated tooling.

Responsible Sunsetting and Effectively Signaling Abandonment. Our results show that many developers, though far from all, care about abandoned dependencies but may not be aware of them or may observe a dependency for a lengthy

period before taking action. Dependencies that are clearly marked as abandoned (*explicit notice*, see Sec. III) are removed significantly faster than those that silently stop receiving maintenance (RQ4), suggesting that awareness matters. Based on our research, we can clearly recommend that **maintainers should place an explicit notice about abandonment of their package as their final action to benefit their dependents, costing very little effort to the departing maintainers**. We believe it is time to **establish best practices for responsible sunsetting of packages,** which may include leaving an explicit notice and possibly also reaching out to direct dependents.

In addition, **future research should explore the most effective way to present abandonment notices**, for example, where to place notices to be effective (e.g., placed in README versus using npm's *deprecate* message to create alerts during package installation) and what to include in the message to make it actionable (e.g., alternatives, migration paths). We also expect that there are many opportunities to better communicate the maintenance status of packages beyond already available signals. There are many research opportunities to develop dependabot-style tooling to inform developers about abandoned dependencies and to curate actionable information (e.g., automatically suggest alternatives [49], [90], [91] or even generate patches [92]–[96]). Building on the vast research on signaling theory [50]–[52] and the use of nudging in software engineering [45], [46], [56], **the key challenge for designing such tools will be identifying when and how to inform developers**, as the abandonment of different dependencies may not be equally important to developers [7].

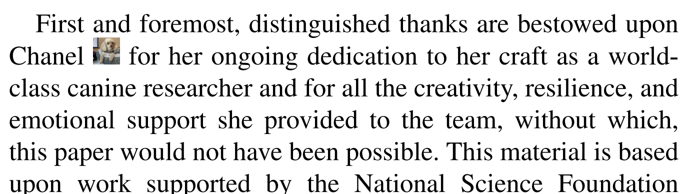
IX. Conclusion

We perform a large-scale quantitative analysis across all widely-used packages in the npm ecosystem, identifying how common abandonment is, measuring exposure and response to abandonment, and performing statistical analysis to understand what factors impact the likelihood of removing abandoned dependencies. We found that abandonment is common and that the majority of exposed dependents do not remove the abandoned dependencies, but also that removal rates are significantly faster for packages that provide explicit notice of abandonment. Based on our finding we recommend strategies for focusing remediation activities, responsible sunsetting, and prioritizing research and tooling.

X. Data Availability

The data and script necessary to reproduce all visualizations and models in the paper are available in the publicly-accessible artifact hosted on Zenodo [80]. DOI 10.5281/zenodo.12687193

XI. Acknowledgments

First and foremost, distinguished thanks are bestowed upon Chanel  for her ongoing dedication to her craft as a world-class canine researcher and for all the creativity, resilience, and emotional support she provided to the team, without which, this paper would not have been possible. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant Number DGE2140739. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Kaestner's work was supported in part by the National Science Foundation (award 2206859). Vasilescu's work was supported in part by the National Science Foundation (awards 2317168 and 1901311). This work was also supported in part by a Google Research Award. Jahanshahi and Mockus' work was supported in part by the National Science Foundation (awards 1901102 and 2120429).

References

[1] N. Eghbal, *Roads and bridges: The unseen labor behind our digital infrastructure*. Ford Foundation, 2016.

[2] N. Forsgren *et al.*, "2020 state of the octoverse: Securing the world's software," *arXiv preprint arXiv:2110.10246*, 2021.

[3] P. Mancini *et al.*, "Sustain: A one day conversation for open source software sustainers – the report," Sustain Conference Organization, Tech. Rep., 2021. [Online]. Available: https://sustainoss.org/assets/pdf/Sustain-In-2021-Event-Report.pdf.

[4] C. Miller, D. G. Widder, C. Kästner, and B. Vasilescu, "Why do people give up FLOSSing? A study of contributor disengagement in open source," in *IFIP International Conference on Open Source Systems*, 2019.

[5] G. Avelino *et al.*, "On the abandonment and survival of open source projects: An empirical investigation," in *Proc. Int'l Symp. Empirical Software Engineering and Measurement (ESEM)*, 2019.

[6] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 995–1010.

[7] C. Miller, C. Kästner, and B. Vasilescu, ""We Feel Like We're Winging It:" A Study on Navigating Open-Source Dependency Abandonment," in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, 2023, pp. 1281–1293.

[8] F. Calefato *et al.*, "Will you come back to contribute? Investigating the inactivity of OSS core developers in GitHub," *Empirical Software Engineering*, 2022.

[9] F. Fagerholm, A. S. Guinea, J. Münch, and J. Borenstein, "The role of mentoring and project characteristics for onboarding in open source software projects," in *Proc. Int'l Symp. Empirical Software Engineering and Measurement (ESEM)*, 2014, pp. 1–10.

[10] I. Steinmacher, T. U. Conte, C. Treude, and M. A. Gerosa, "Overcoming open source project entry barriers with a portal for newcomers," in *Proc. Int'l Conf. Software Engineering (ICSE)*, 2016.

[11] M. Guizani, T. Zimmermann, A. Sarma, and D. Ford, "Attracting and retaining OSS contributors with a maintainer dashboard," in *Int'l Conf. on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, 2022.

[12] C. J. Alberts, A. J. Dorofee, R. Creel, R. J. Ellison, and C. Woody, "A systemic approach for assessing software supply-chain risk," in *Hawaii Int'l Conf. on System Sciences*, IEEE, 2011, pp. 1–8.

[13] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an API: Cost negotiation and community values in three software ecosystems," in *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, 2016, pp. 109–120.

[14] A. Decan, T. Mens, M. Claes, and P. Grosjean, "When GitHub meets CRAN: An analysis of inter-repository package dependency problems," in *Proc. Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[15] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in OSS packaging ecosystems," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*, IEEE, 2017, pp. 2–12.

[16] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proc. Conf. Mining Software Repositories (MSR)*, 2018, pp. 181–191.

[17] J. Hejderup, "In dependencies we trust: How vulnerable are dependencies in software modules? Master's thesis," *Delft University of Technology*, 2015.

[18] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *Proc. Conf. Mining Software Repositories (MSR)*, IEEE, 2017, pp. 102–112.

[19] C. Liu *et al.*, "Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem," in *Proc. Int'l Conf. Software Engineering (ICSE)*, 2022, pp. 672–684.

[20] Sonatype, "2021 state of the software supply chain report," Tech. Rep., 2021. [Online]. Available: https://www.sonatype.com/hubfs/Q3%202021-State%20of%20the%20Software%20Supply%20Chain-Report/SSSC-Report-2021_0913_PM_2.pdf.

[21] N. Zahan *et al.*, "What are weak links in the npm supply chain?" In *Proc. Int'l Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022.

[22] T. Herr, W. Loomis, S. Scott, and J. Lee, *Breaking trust: Shades of crisis across an insecure software supply chain*, 2020. [Online]. Available: https://www.atlanticcouncil.org/in-depth-research-reports/report/breaking-trust-shades-of-crisis-across-an-insecure-software-supply-chain/.

[23] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," in *2023 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2023.

[24] Synopsys, "2024 open source security and risk analysis report," Synopsys, Tech. Rep., 2024. [Online]. Available: https://www.synopsys.com/software-integrity/engage/ossra/ossra-report.

[25] shelly, https://twitter.com/codebytere/status/1567437988908392455, Accessed: 2024-03-17, 2022.

[26] Sonatype, "9th annual state of the software supply chain," Sonatype, Tech. Rep., 2023. [Online]. Available: https://www.sonatype.com/state-of-the-software-supply-chain/about-the-report.

[27] D. Pinckney, F. Cassano, A. Guha, and J. Bell, "A large scale analysis of semantic versioning in npm," *Proc. Conf. Mining Software Repositories (MSR)*, 2023.

[28] A. Decan and T. Mens, "What do package dependencies tell us about semantic versioning?" *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1226–1240, 2019.

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

[30] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016.

[31] J. Dietrich *et al.*, "Dependency versioning in the wild," in *Proc. Conf. Mining Software Repositories (MSR)*, 2019.

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

[33] G. A. A. Prana *et al.*, "Out of sight, out of mind? How vulnerable dependencies affect open-source projects," *Empirical Software Engineering*, vol. 26, 2021.

[34] R. G. Kula *et al.*, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[35] OpenSSF, *FLOSS best practices criteria (all levels)*, Accessed: 2024-03-17. [Online]. Available: https://www.bestpractices.dev/en/criteria.

[36] J. Cox, E. Bouwers, M. Van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *Proc. Int'l Conf. Software Engineering (ICSE)*, IEEE, 2015, pp. 109–118.

[37] *Executive order 14028: Improving the nation's cybersecurity*, https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/, 2021.

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

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

[40] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation? The case of a Smalltalk ecosystem," in *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, 2012.

[41] J. Stringer, A. Tahir, K. Blincoe, and J. Dietrich, "Technical lag of dependencies in major package managers," in *Proc. Asia-Pacific Software Engineering Conf. (APSEC)*, IEEE, 2020, pp. 228–237.

[42] A. Zerouali *et al.*, "An empirical analysis of technical lag in npm package dependencies," in *Proc. Int'l Conf. Software Reuse (ICSR)*, Springer, 2018.

[43] T. Lauinger *et al.*, "Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web," *arXiv preprint arXiv:1811.00918*, 2018.

[44] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the Apache community upgrades dependencies: An evolutionary study," *Empirical Software Engineering*, 2015.

[45] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, IEEE, 2017.

[46] R. He, H. He, Y. Zhang, and M. Zhou, "Automating dependency updates in practice: An exploratory study on GitHub Dependabot," *IEEE Transactions on Software Engineering*, 2023.

[47] B. Rombaut, F. R. Cogo, B. Adams, and A. E. Hassan, "There's no such thing as a free lunch: Lessons learned from exploring the overhead introduced by the Greenkeeper dependency bot in npm," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 32, no. 1, 2023.

[48] I. Pashchenko, D.-L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Proc. Conf. Computer and Communications Security (CCS)*, 2020.

[49] S. Mujahid *et al.*, "Where to go now? Finding alternatives for declining packages in the npm ecosystem," in *Proc. Int'l Conf. Automated Software Engineering (ASE)*, 2023.

[50] J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of social and technical factors for evaluating contribution in GitHub," in *Proc. Int'l Conf. Software Engineering (ICSE)*, 2014.

[51] H. S. Qiu, Y. L. Li, S. Padala, A. Sarma, and B. Vasilescu, "The signals that potential contributors look for when choosing open-source projects," *Proc. of the ACM on Human-Computer Interaction*, 2019.

[52] A. Trockman, S. Zhou, C. Kästner, and B. Vasilescu, "Adding sparkle to social coding: An empirical study of repository badges in the npm ecosystem," in *Proc. Int'l Conf. Software Engineering (ICSE)*, 2018.

[53] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in GitHub: Transparency and collaboration in an open software repository," in *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, 2012.

[54] L. Dabbish *et al.*, "Leveraging transparency," *IEEE Software*, vol. 30, 2012.

[55] J. Marlow, L. Dabbish, and J. Herbsleb, "Impression formation in online peer production: Activity traces and personal profiles in GitHub," in *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, 2013.

[56] C. Maddila *et al.*, "Nudge: Accelerating overdue pull requests toward completion," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 2023.

[57] K. Blincoe *et al.*, "Understanding the popular users: Following, affiliation influence and leadership on GitHub," *Information and Software Technology (IST)*, 2016.

[58] M. J. Lee *et al.*, "GitHub developers use rockstars to overcome overflow of news," in *Extd. Abstracts on Human Factors in Computing Systems*, 2013.

[59] A. Capiluppi, A. Serebrenik, and L. Singer, "Assessing technical candidates on the social web," *IEEE Software*, 2012.

[60] J. Marlow and L. Dabbish, "Activity traces and signals in software developer recruitment and hiring," in *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, 2013.

[61] S. Mujahid, R. Abdalkareem, and E. Shihab, "What are the characteristics of highly-selected packages? A case study on the npm ecosystem," *Journal of Systems and Software*, 2023.

[62] P. Tourani, B. Adams, and A. Serebrenik, "Code of conduct in open source projects," in *Proc. Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, 2017, pp. 24–33.

[63] *Archiving repositories*, Accessed Sep. 2023. [Online]. Available: https://github.blog/2017-11-08-archiving-repositories/.

[64] *Unmaintained tech*, http://unmaintained.tech, Accessed Sep. 2023.

[65] *Automattic/kue*, https://github.com/Automattic/kue, Accessed Sep. 2023.

[66] J. Coelho and M. T. Valente, "Why modern open source projects fail," in *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, 2017.

[67] J. Coelho *et al.*, "Is this GitHub project maintained? Measuring the level of maintenance activity of open-source projects," *Information and Software Technology (IST)*, 2020.

[68] M. Valiev, B. Vasilescu, and J. Herbsleb, “Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem,” in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, 2018.

[69] J. Coelho, M. T. Valente, L. L. Silva, and E. Shihab, “Identifying unmaintained projects in GitHub,” in *Proc. Int’l Symp. Empirical Software Engineering and Measurement (ESEM)*, 2018.

[70] J. Khondhu, A. Capiluppi, and K.-J. Stol, “Is it all lost? A study of inactive open source projects,” in *IFIP Int’l Conf. on Open Source Systems*, Springer, 2013, pp. 61–79.

[71] Y. Ma *et al.*, “World of Code: Enabling a research workflow for mining and analyzing the universe of open source VCS data,” *Empirical Software Engineering*, vol. 26, 2021.

[72] *Npm download statistics*, https://api.npmjs.org/downloads/, Accessed Sep. 2023.

[73] Y. Ma *et al.*, “World of Code: An infrastructure for mining the universe of open source VCS data,” in *Proc. Conf. Mining Software Repositories (MSR)*, IEEE, 2019.

[74] T. Dey, Y. Ma, and A. Mockus, “Patterns of effort contribution and demand and user classification based on participation patterns in npm ecosystem,” in *Proc. Intl. Conf. on Predictive Models and Data Analytics in Software Engineering*, ACM, 2019.

[75] B. Chinthanet *et al.*, “Lags in the release, adoption, and propagation of npm vulnerability fixes,” *Empirical Software Engineering*, 2021.

[76] *Osv database*, https://osv.dev, Accessed Sep. 2023.

[77] S. P. Jenkins, “Survival analysis,” *Unpublished manuscript, Institute for Social and Economic Research, University of Essex, Colchester, UK*, vol. 42, pp. 54–56, 2005.

[78] E. L. Kaplan and P. Meier, “Nonparametric estimation from incomplete observations,” *Journal of the American statistical association*, 1958.

[79] D. R. Cox and D. Oakes, *Analysis of survival data*. CRC press, 1984.

[80] C. Miller, M. Jahanshahi, A. Mockus, B. Vasilescu, and C. Kästner, “Supplementary material for understanding the response to open-source dependency abandonment in the npm ecosystem,” Zenodo, 2024. DOI: 10.5281/zenodo.12686619. [Online]. Available: https://doi.org/10.5281/zenodo.12686619.

[81] L. Ballejos, *Best practices for managing end-of-life software*, https://www.ninjaone.com/blog/end-of-life-software-management/, 2024.

[82] D. Spinellis *et al.*, “A dataset of enterprise-driven open source software,” in *Proc. Conf. Mining Software Repositories (MSR)*, 2020.

[83] D. J. Bartholomew, M. Knott, and I. Moustaki, *Latent variable models and factor analysis: A unified approach*. John Wiley & Sons, 2011.

[84] A. Gelman and J. Hill, *Data analysis using regression and multilevel/hierarchical models*. Cambridge university press, 2006.

[85] Y. Dodge, *The concise encyclopedia of statistics*. Springer Science & Business Media, 2008.

[86] J. Fox, *Applied regression analysis and generalized linear models*. Sage Publications, 2015.

[87] M. R. Veall and K. F. Zimmermann, “Pseudo-r2 measures for some common limited dependent variable models,” *Journal of Economic surveys*, vol. 10, no. 3, pp. 241–259, 1996.

[88] F. E. Harrell Jr and F. E. Harrell, “Cox proportional hazards regression model,” *Regression modeling strategies: With applications to linear models, logistic and ordinal regression, and survival analysis*, pp. 475–519, 2015.

[89] E. Larios Vargas *et al.*, “Selecting third-party libraries: The practitioners’ perspective,” in *Proc. Int’l Symposium Foundations of Software Engineering (FSE)*, ACM, 2020.

[90] H. He *et al.*, “A multi-metric ranking approach for library migration recommendations,” in *Proc. Int’l Conf. Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, 2021, pp. 72–83.

[91] C. Chen, “SimilarAPI: Mining analogical APIs for library migration,” in *Comp. Int’l Conf. Software Engineering (ICSE)*, IEEE, 2020.

[92] L. Wu *et al.*, “Transforming code with compositional mappings for API-library switching,” in *Conf. Computer Software and Applications*, 2015.

[93] K. Chow and D. Notkin, “Semi-automatic update of applications in response to library changes,” in *Proc. Int’l Conf. Software Maintenance (ICSM)*, vol. 96, 1996, p. 359.

[94] R. Ossendrijver, S. Schroevers, and C. Grelck, “Towards automated library migrations with error prone and refaster,” in *Proc. Symp. Applied Computing (SAC)*, 2022, pp. 1598–1606.

[95] H. Alrubaye, M. W. Mkaouer, and A. Ouni, “On the use of information retrieval to automate the detection of third-party Java library migration at the method level,” in *Proc. Int’l Conf. Program Comprehension (ICPC)*, IEEE, 2019.

[96] H. Alrubaye *et al.*, “MigrationMiner: An automated detection tool of third-party Java library migration at the method level,” in *Proc. Int’l Conf. Software Maintenance and Evolution (ICSME)*, 2019.