Noname manuscript No.
(will be inserted by the editor)

Securing Dependencies: A Comprehensive Study of Dependabot's Impact on Vulnerability Mitigation

Hamid Mohayeji · Andrei Agaronian · Eleni Constantinou · Nicola Zannone · Alexander Serebrenik

Abstract The growing use of third-party libraries in software development poses a hidden security risk, as vulnerabilities in these libraries can easily spread to dependent applications. Project maintainers must remain vigilant regarding updates and patches for these external libraries, a responsibility that is facilitated by automated tools, also known as *bots*. This study centers on Dependabot, a widely adopted bot that offers security and version updates. We aim to scrutinize the impact of Dependabot on mitigating vulnerabilities arising from dependencies, preventing potential prolonged security issues in open-source software. We investigate how developers react to security updates provided by Dependabot within engineered and actively maintained JavaScript projects. We also delve into how project attributes, including the integration of tests and continuous integration (CI) tools, influence the acceptance rate of security updates. Additionally, we perform a detailed analysis of the lifespan of each vulnerability to demonstrate how they are dealt with when Dependabot is in use. Our findings reveal a significant reliance on Dependabot by developers for managing security vulnerabilities in dependencies, with most updates being

Hamid Mohayeji
Eindhoven University of Technology, The Netherlands
E-mail: h.mohayeji.nasrabadi@tue.nl

Andrei Agaronian
Eindhoven University of Technology, The Netherlands
E-mail: andrei.agaronian@gmail.com

Eleni Constantinou
University of Cyprus, Cyprus
E-mail: constantinou.a.eleni@ucy.ac.cy

Nicola Zannone
Eindhoven University of Technology, The Netherlands
E-mail: n.zannone@tue.nl

Alexander Serebrenik
Eindhoven University of Technology, The Netherlands
E-mail: a.serebrenik@tue.nl

merged swiftly within days. We find that projects equipped with tests and CI tools are more likely to merge security updates. Conversely, when developers opt not to merge a security update, they often manually address the identified vulnerability. This manual approach, however, could span over several months, potentially exposing projects to security risks. Crucially, in many instances, the manual fixes are potentially inspired by earlier security updates, underscoring Dependabot's pivotal role in safeguarding dependencies.

Keywords Security, Vulnerability, Bot, Dependency Management, Dependabot

1 Introduction

Contemporary open-source software increasingly builds upon reusable code components distributed through language-specific online registries (e.g., `npm` and `Maven`). Software reuse is an established practice that facilitates the development of complex systems (Frakes and Kang, 2005). It also contributes to rapid software evolution by reducing costs (Irshad et al., 2016; Krüger and Berger, 2020), efforts, and delivery time (Basili et al., 1996; Lim, 1994; Mohagheghi et al., 2004). Despite the advantages of reuse, there is also the risk of inheriting security vulnerabilities present in the imported libraries (Thompson, 2003). Although vulnerabilities are mitigated in package newer releases, developers are generally reluctant to update outdated and vulnerable dependencies due to the perceived extra workload and responsibility associated with dependency management (Kula et al., 2018). In fact, after interviewing developers of open-source projects with known vulnerable dependencies, Kula et al. report that 69% of the interviewees were simply unaware of them. Furthermore, a study by Decan et al. (2018b), which investigated the propagation of security vulnerabilities and their fixes in the `npm` package registry, revealed significant time lags in mitigating vulnerable dependencies. As reported, it requires nearly 14 months for 50% of the dependent software projects to address a vulnerable dependency. Additionally, a recent study by Alfadel et al. (2023) on Node.js applications highlighted that the primary reason for the applications being affected by public vulnerabilities is the absence of dependency updates despite the availability of fixes. Consequently, applications continue to be affected by public vulnerabilities for extended periods.

Given the time-intensive nature of dependency management, automation has become increasingly prevalent (Decan et al., 2018b). Various software bots have been developed to monitor releases and security reports, detecting outdated or vulnerable dependencies, and subsequently generating pull requests to update them (Dependabot, s.d.; Neighbourhoodie Software, 2020; Snyk Limited, s.d.; Mend, s.d.; Gebauer, 2015). Nonetheless, the use of bots is not without its drawbacks. As revealed by Wessel et al. (2021) through interviews, developers identified challenges arising from bots in pull requests, with some perceiving certain bot actions as disruptive noise when dealing with a heavy workload.

On May 2019 (Hutchings, 2019), GitHub acquired one of the most popular dependency management bots, *Dependabot-preview* (Dependabot, s.d.), resulting in a new natively integrated service, *Dependabot security updates* (GitHub, s.d.). Upon receiving an alert, Dependabot automatically opens a pull request, *i.e.*, security update, upgrading the dependency to the minimum non-vulnerable version. Among the many dependency management tools, such as Greenkeeper (Neighbourhoodie Software, 2020), Renovate (Mend, s.d.), and Depfu (Depfu, 2020), Dependabot distinguishes itself due to its seamless integration with GitHub and its free availability, making it one of the most accessible and widely used dependency management tools (Erlenhov et al., 2022). Dependabot provides automated pull requests to resolve vulnerable dependencies (Wyrich et al., 2021) and the security issues they may cause. Given its widespread adoption across numerous GitHub repositories, evaluating its effectiveness in maintaining the security of dependencies is crucial.

In this study, we aim to comprehend how Dependabot security updates help mitigate dependency vulnerabilities. Specifically, we examine Dependabot from two perspectives: its role in automatically fixing vulnerabilities within dependencies through security updates and its function in flagging security issues to assist developers in addressing them. To achieve this, we initially examine the merge ratio of Dependabot security updates across various disjoint groups of projects, considering the number of security updates they receive. Additionally, we explore potential correlations between the merge ratio of security updates and certain project attributes, such as popularity and the use of tests and/or CI tools. This helps us in identifying project characteristics that influence the acceptance of security updates. To assess the effectiveness of Dependabot in alerting developers to security vulnerabilities, in contrast to the qualitative method utilized by Alfadel et al. (2021), we quantitatively measure the extent to which developers fix vulnerabilities manually in the presence of Dependabot. Our study is centered on vulnerabilities rather than security updates, allowing for a more detailed analysis compared to prior research, given that security updates may encompass fixes for multiple vulnerabilities. To this end, we derive each instance of vulnerability from the parent commit of every security update, a process facilitated by leveraging GitHub Advisory Records. We then present a method to recursively mine the commit history of each repository to identify the potential fixing commit of each vulnerability, which in turn allows us to monitor and analyze the lifecycle of each instance of vulnerability individually. Following this, we conduct a survival analysis of this data to investigate the degree to which developers react to the vulnerabilities promptly and study persistent cases. We also uncover the correlation between the severity of vulnerabilities and the time required for developers to respond to them, a finding that contradicts the conclusions presented by Alfadel et al. (2021). Finally, for a deeper understanding of Dependabot's impact on aiding developers in addressing security vulnerabilities, we examine cases of manual fixes potentially inspired by Dependabot. In these instances, even though developers choose to address vulnerabilities manually, their fixes are possibly inspired by earlier security updates provided by the bot.

In this work, we scrutinize 4,195 security updates associated with 978 mature and actively maintained JavaScript projects that are based on `npm` or `yarn` package managers. We discovered that 57% of the bot-initiated security updates are merged. We also found that certain project attributes, such as the incorporation of tests and CI services, influence the developers' responsiveness to security updates, likely due to diminished concerns regarding potential breaking changes. Moreover, our findings indicate that bot fixes are almost two times more frequent than manual fixes. While the majority of vulnerabilities associated with ignored security updates were fixed manually, our results reveal that manual fixes take considerably more time. This denotes that rejecting security updates exposes packages to vulnerabilities for extended periods, potentially resulting in security issues (Cox et al., 2015). Furthermore, we observe that some manual fixes are likely to be inspired by the bot, since the authors of these fixes follow precisely the version suggested in prior security updates, even when newer versions are available at the time they manually address vulnerable dependencies. Overall, our study highlights that Dependabot is highly effective in mitigating security vulnerabilities. Still, factors such as apprehensions regarding compatibility issues, restricted configurations, and the bot's automatic deployment without prior notification discourage maintainers from adopting the security updates.

This article builds upon and extends our previous work (Mohayeji et al., 2023) by incorporating two additional research questions. In the original paper, we assess the merge ratio of various project groups based on the frequency of security updates they receive ($\boldsymbol{RQ_1}$), quantify the incidence of manual resolution of dependencies when Dependabot is in use ($\boldsymbol{RQ_3}$), and determine the duration developers require to rectify vulnerabilities spotted by Dependabot ($\boldsymbol{RQ_4}$). In the first additional research question, we examine the correlation between various project attributes, such as popularity metrics, utilization of testing frameworks, and the impact of CI tools on developers' receptiveness to Dependabot security updates ($\boldsymbol{RQ_2}$). We also seek to uncover potential instances of inspired fixes as the second additional research question. In these cases, developers opt to implement changes mirroring those found in prior security updates without formally merging them ($\boldsymbol{RQ_5}$). Conducting these supplementary examinations yields additional valuable insights into the characteristics of projects adopting security updates, alongside a more comprehensive understanding of Dependabot's efficacy in securing dependencies. To summarize, this paper makes the following contributions:

- Through an examination of the correlation between the merge ratio of Dependabot security updates across various project groups and attributes such as popularity, as well as the adoption of testing frameworks and continuous integration tools, this study offers insights into the association between projects' characteristics and receptivity of security updates.
- We introduce a mechanism for extracting all individual vulnerability instances from the parent commit of each security update, followed by a

recursive algorithm for fix detection. This approach aids us in studying the lifecycle of each individual vulnerability instance.
- The presented work leverages a survival analysis approach to monitor the persistence of vulnerabilities in cases where developers opt not to apply security updates.
- To enhance our assessment of Dependabot's efficacy in aiding developers with the management of vulnerable dependencies, we conduct an analysis on potential inspired manual fixes, *i.e.*, the cases where developers precisely follow the patch version suggested by a prior security update to address vulnerable packages, without merging the security update itself.

The subsequent sections of the paper are structured as follows. The following section presents background on dependencies in JavaScript projects and Dependabot. Section 3 provides our research questions. Section 4 presents the methodology used for data collection and analysis, and Section 5 reports the results. Section 6 discusses our findings and presents the threats to validity. Finally, Section 7 discusses related work, and Section 8 concludes the paper.

2 Background

A *dependency* (also known as *package* or *library*) is a piece of code directly utilizable within a program. To streamline the installation, upgrading, removal, and distribution of software packages, developers rely on package managers (Burrows and Fernandez Montecelo, 2016). A plethora of studies (Kula et al., 2018; Decan et al., 2018b; Backes et al., 2016; Zerouali et al., 2018; Decan et al., 2018a; Derr et al., 2017; Wang et al., 2020; Chinthanet et al., 2021; Decan et al., 2017; Zerouali et al., 2019; Alfadel et al., 2020, 2023) demonstrate that dependency updates suffer from considerable time lags, sometimes even measured in the orders of years (Lauinger et al., 2018). This phenomenon can be observed across various ecosystems, both decentralized, such as Android (Backes et al., 2016; Derr et al., 2017), and centralized, such as Java (Kula et al., 2018; Wang et al., 2020; Xu et al., 2022) and JavaScript (Decan et al., 2018b; Zerouali et al., 2018; Decan et al., 2018a; Chinthanet et al., 2021; Wang et al., 2023). The reluctance to update dependencies can ultimately lead to security issues (Cox et al., 2015).

Decan et al. (2018b) analyzed the spread of security vulnerabilities in the `npm` dependency network, reporting that more than 20% of the projects have direct dependencies on vulnerable packages, with many of these projects containing at least one release that relies on an affected version of such packages. This underscores the prevalence of vulnerabilities within `npm` packages and the imperative to address them. Furthermore, they found that it usually requires close to 14 months for half of dependent packages to address a vulnerable dependency. Notably, these dependent packages not only require a substantial amount of time to mitigate vulnerabilities but also significantly more time compared to upstream packages. Likewise, Prana et al. (2021) demonstrate that the prolonged existence of a vulnerable dependency is mainly attributable to

delayed updates within the dependent application, rather than the persistence of vulnerabilities across releases of the upstream package. They further observed that the most significant correlation factor for the number of vulnerable dependencies is the total dependency count, which also signifies the complexity of the dependency network. This suggests the necessity for automation to assist developers in managing dependencies. While it is strongly encouraged to practice careful dependency management by ensuring that dependencies are kept up to date, it is not consistently practiced. Through a survey of developers involved in software projects with identified vulnerable dependencies, Kula et al. (2018) found that developers prioritize the effort required to address a vulnerable dependency over the persistence of the security issue. Furthermore, the study revealed that 69% of developers were unaware of the presence of vulnerable dependencies in their projects. This underscores the potential benefits for the community from automated notifications of security issues in projects' dependencies, motivating our investigation into the extent to which developers respond to such notifications, and their broader impact on ensuring the security of dependencies.

2.1 Dependency Management in JavaScript

To ease the installation, upgrading, removal, and distribution of software packages, developers rely on package managers (Burrows and Fernandez Montecelo, 2016). This work considers projects that utilize either of the two primary package managers available for JavaScript software: `npm` and `yarn`. In these projects, dependencies are typically declared in a *dependency file* associated with the repository. Specifically, all dependent JavaScript projects contain a *manifest file*, called `package.json`, which specifies the set of *direct* dependencies, *i.e.*, upstream packages referenced within the source code. Each dependency is identified by the name of the required upstream package, associated with a *dependency constraint* that explicitly defines the range of acceptable releases, effectively excluding versions deemed undesirable or incompatible. In addition to a manifest file, developers are encouraged to include a *lock file* in the source repository. This file is generated when the installation command is executed on a manifest file and records the precise *dependency tree*. This tree delineates both direct and transitive (*indirect*) dependencies, along with pertinent metadata for each node, such as the integrity hash and the resource path in the registry.

2.2 Dependabot

Dependabot is a dependency management tool provided by GitHub that is designed to aid developers in the tasks of updating dependencies and fixing identified vulnerabilities. This tool encompasses four interrelated services: *database of security advisories*, *security alerts*, *security updates*, and *version updates*.

GitHub Advisory Database GitHub curates a list of security vulnerabilities found in software packages across six distinct ecosystems, including `npm` and `yarn`. Each advisory entry contains detailed information about the identified vulnerability, comprising its description, the affected package's name and associated ecosystem, as well as the versions impacted and patched releases. Additionally, every security advisory is categorized with a severity level (*low*, *moderate*, *high*, or *critical*), aiding maintainers in assessing and prioritizing the risks posed to affected systems.

Security Alerts Dependabot security alerts is a native GitHub service designed for the efficient management of vulnerable dependencies. It continuously scans the project's dependency graph, comparing it to the GitHub security advisory database. Upon detecting a vulnerable dependency version, it prompts developers with a security alert. Dependabot leverages the *dependency graph* to execute vulnerability scans. To generate this graph, it parses both manifest and lock files residing in the repository's *default branch* and constructs a comprehensive representation of the complete dependency tree.

Security Updates GitHub announced the acquisition of Dependabot-preview in 2019, which resulted in the introduction of a newly integrated service known as Dependabot security updates. Upon receiving a security alert, Dependabot constructs a pull request, *i.e.*, security update, containing the necessary modifications to the dependency file(s) to upgrade each upstream package to the minimum required non-vulnerable release, whenever feasible. An illustration of a security update is provided in Figure 1. The proposed modification is visible in both the title and body sections of the pull request (A1 & A2). Furthermore, Dependabot includes the changelog for each version between the current and suggested releases (B), along with a badge indicating the compatibility score (C). This score is dynamically computed based on the percentage of successful CI runs in other public repositories where an identical security update has been applied. However, details regarding the concerned vulnerability are not directly provided within the pull request. Instead, users are alerted by a yellow panel, signaling the presence of a security vulnerability (D). On the far-left side, it offers a hyperlink to the relevant security alert, while on the far-right side, it presents the severity level of the vulnerability. It is worth noting that users lacking the requisite access privileges, *i.e.*, external observers, are unable to view this information.

Version updates Version updates refer to automatically generated pull requests designed to keep dependencies up-to-date. Both security and version updates can be concurrently enabled within a single repository. However, external observers cannot distinguish between these two types of pull requests.

A recent investigation by Alfadel et al. (2021) on the adoption rate of security pull requests authored by Dependabot-preview, the predecessor of GitHub's service, manifested that most of such automated suggestions are merged within

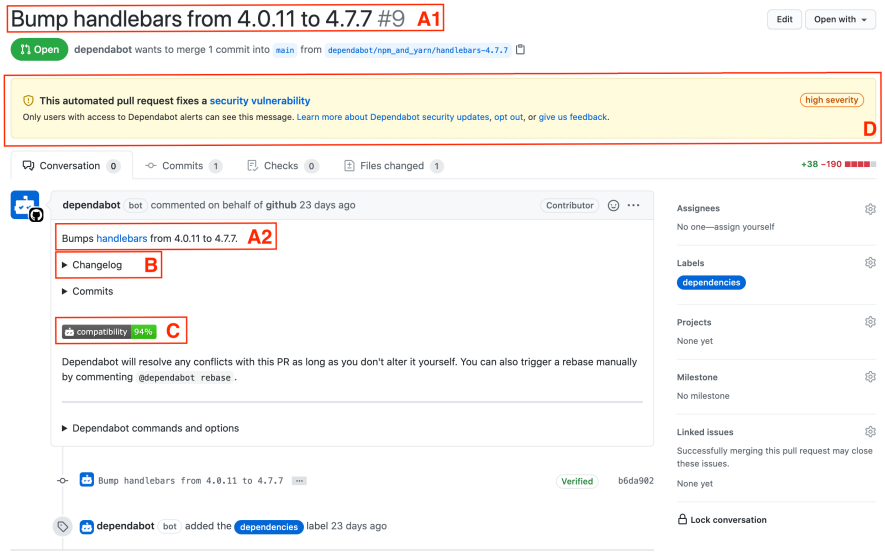


Fig. 1: Screenshot of a security update.

a day. They further examined the factors that affect the rapid merge of security pull requests generated by Dependabot-preview, observing that notably, the severity level of identified vulnerability has no substantial impact. Nevertheless, the findings of Alfadel *et al.* might not necessarily reflect the developer reception and usage of Dependabot security updates provided by GitHub (McDonald, 2021). In fact, there are significant differences between the two services. In particular, the core function of Dependabot-preview is to handle version updates, aiming to ensure that dependencies are consistently up-to-date. This suggests that Dependabot-preview generates pull requests with greater frequency, potentially leading to increased noise. Moreover, when encountering a vulnerable dependency, Dependabot-preview consistently recommends upgrading to the latest secure version, in contrast to Dependabot security updates, which propose the minimum required version. As a consequence, it often occurs that if a more recent version is available, Dependabot-preview supersedes the previous security update with a fresh one. Indeed, Alfadel *et al.* noted that the bot itself closes the majority of the rejected pull requests. Besides, Dependabot-preview provides an *auto-merge* feature, which allows the bot to merge its pull requests without developer involvement. In this study, we investigate Dependabot security updates, analyzing their effectiveness in identifying and resolving security vulnerabilities and helping developers secure software dependencies within their projects.

3 Research Questions

Dependabot is recognized as one of the most accessible tools offering automated security updates to address vulnerabilities that could potentially lead to security issues (Wyrich et al., 2021). The purpose of Dependabot can be seen as twofold. On one hand, it offers automatic security updates that simplify the process of merging and fixing security vulnerabilities. On the other hand, it raises awareness about project security and assists developers in addressing vulnerabilities at their own pace and in their preferred manner. In this regard, the degree of acceptance regarding Dependabot security updates, the project's attributes influencing its adoption, the lifespan of vulnerabilities and their management in its presence, as well as its precise impact on maintaining the security of dependencies, need to be investigated. Thus, we center our attention on the five research questions outlined below.

The first research question in this study focuses on evaluating how receptive developers are to Dependabot security updates. Our objective is to measure the extent to which developers merge these updates, focusing on the frequency with which they receive them and identifying any potential trends in merge rates. Therefore, we pose the following question:

RQ$_1$: *To what degree do developers merge Dependabot security updates?*

Moving forward, we aim to understand which types of repositories are more likely to adopt security updates at higher rates. To achieve this, we examine how various repository attributes, such as popularity, project size, age, number of contributors, commit count, or the adoption of test frameworks and CI/CD tools, may influence the willingness of maintainers to merge Dependabot security updates. This analysis could help us identify the factors that drive greater adoption of security updates:

RQ$_2$: *How do repositories' attributes impact security updates merge ratio?*

As indicated by prior studies (Alfadel et al., 2021; Pashchenko et al., 2020), developers might choose to close a security update and manually implement the bot's recommendations. The characteristics of the fix, such as the type of version bump (e.g., *major*, *minor*, or *patch*), could raise concerns about breaking changes and affect developers' decisions to apply the fixes manually. Alternatively, they might opt for removing the dependency on a vulnerable package entirely. Envisioning various scenarios where a security update is declined, yet the identified vulnerability is still addressed within the project, we inquire:

RQ$_3$: *How frequently do developers opt for manual resolution of a vulnerable dependency when Dependabot security updates are in place?*

Furthermore, we aim to evaluate Dependabot's effectiveness in notifying developers of vulnerabilities. Specifically, we measure how quickly developers respond to these identified vulnerabilities when Dependabot is involved. This analysis also allows us to compare the time required to manually address vulnerabilities versus the time it takes to resolve them by merging security updates. The following research question captures these concerns:

RQ$_4$: *What is the duration required to address a vulnerable dependency identified by Dependabot?*

At last, to uncover fixes indirectly influenced by the Dependabot and enhance our understanding of the role security updates play in maintaining the security of dependencies, we examine the manual cases that are likely inspired by the bot. This refers to instances where despite the availability of newer versions of the concerned library, developers apply the precise changes suggested by the bot without actually merging the security updates:
RQ$_5$: *What proportion of manual fixes are potentially inspired by Dependabot security updates?*

4 Methodology

This section outlines the methodology utilized for data collection and the analytical techniques employed to address the research questions presented in this study, as summarized in Figure 2.

4.1 Data Collection

To investigate the frequency with which developers merge security updates of Dependabot (***RQ$_1$***), we collect security updates from GitHub. Moreover, we retrieve the number of stars and forks for each project, serving as their popularity metrics, and seek evidence of testing frameworks and CI tool adoption by examining their commit history. This exploration aims to investigate a potential relationship between these features and the merge ratio of security updates (***RQ$_2$***). Subsequently, we extract instances of vulnerabilities from each security update and then analyze the commit history of the projects to examine the resolution of vulnerable dependencies (***RQ$_3$***, ***RQ$_4$***). Lastly, we investigate occurrences of potential inspired fixes. We consider a fix *inspired* by the bot if developers opt to manually implement the precise changes suggested by an earlier security update, despite the existence of more recent versions of the concerned library. In this regard, for every manual fix within each project, we scrutinize all preceding security updates to detect indications of inspired fixes (***RQ$_5$***).

For our empirical study, we concentrate on JavaScript projects for two primary reasons. Firstly, data from the GitHub annual survey[1] indicates that JavaScript is ranked as the most popular programming language. Moreover, JavaScript projects demonstrate the most extensive distribution of package dependencies compared to other programming languages (GitHub, 2020), thereby rendering them more susceptible to inheriting vulnerabilities through dependencies (Decan et al., 2017, 2019). The combination of these factors accentuates the significance of our findings for a wide community of developers.

[1] `https://octoverse.github.com/2022/top-programming-languages`. Accessed on January 18, 2023.

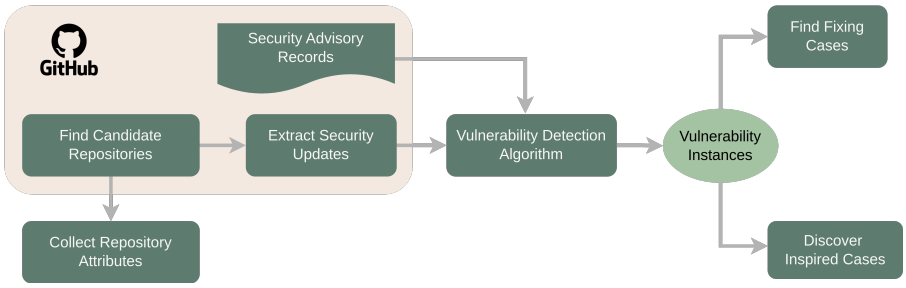


Fig. 2: Illustration of the process for extracting repositories and their attributes, identifying security updates and vulnerabilities, followed by analysis to discover fixes and possible inspired cases.

Given our lack of access to projects' settings, determining whether a project utilizes both version update and security update services is not feasible. The coexistence of both services within a single project may introduce confounding factors in the resolution of security updates, as their pull requests are indistinguishable. Therefore, we restrict the collection of security updates to the period between the introduction of security updates (June 1, 2019) and the introduction of version updates (May 31, 2020), ensuring that only security updates are considered.

Utilizing the *GitHub Search API*[2], we discovered 155,065 repositories that were starred and non-forked, created before the commencement of the data collection period, and received at least one update thereafter. We selected projects that demonstrated continuous maintenance throughout the entire data collection period and had a minimum of 100 commits at the beginning of this period (Kula et al., 2018), with at least one commit made each month during the collection period. Given our interest in projects utilizing the `npm` and `yarn` package management tools, we exclusively considered projects containing a `package.json` manifest file at the root of the repository. This criterion resulted in a subset of 3,587 projects for further analysis.

As demonstrated by Kalliamvakou et al. (2014), a significant proportion of GitHub repositories serve experimental, storage, or academic purposes. Given that the presence of such repositories may introduce noise into the analysis, we employed `Reaper` (Munaiah et al., 2017) to exclude them. Reaper utilizes several quantifiable software engineering practices (*e.g.*, *architecture*, *community* and *continuous integration*), referred to as *dimensions*, to characterize a repository. It calculates these dimensions based on the repository's source code and the project's historical data. This process yielded 3,151 engineered projects for further examination. From this dataset, we identified 1,492 repositories that had received at least one security update from Dependabot. Subsequently, we excluded 390 projects utilizing multiple dependency management bots (*e.g.*, Dependabot-preview (Dependabot, s.d.), Greenkeeper (Neighbourhoodie Soft-

[2] `https://docs.github.com/en/rest/search`. Accessed on January 18, 2024.

Table 1: Characteristics of the selected projects.

Metric	Min.	Max.	Median	Mean
Forks	0	33,022	33	391.88
Stars	2	180,228	63	2,121.08
Core contributors*	1	477	4	8.58
Security updates	1	67	3	4.50
Commits before col. period	101	48,807	890	2,019.81
Commits during col. period	25	15,306	346	670.80

ware, 2020), Snyk-bot (Snyk Limited, s.d.)), to mitigate potential confounding factors arising from the use of various tools. Furthermore, we filtered out 124 projects that had received Dependabot security updates targeting ecosystems other than `npm` and `yarn` (*e.g.*, `Maven`, `RubyGems`). This curation process resulted in a refined dataset of 978 projects. Table 1 provides an overview of these projects. For every project included in our selection, we retrieve all security updates generated by Dependabot during the collection period. At this point, our dataset comprises 4,416 security updates, with 2,391 classified as *merged*, 1,804 as *closed*, and 221 as *open*. In light of developers' indecision regarding whether to merge an open security update, we utilize the 4,195 non-open updates for our analysis. A non-open security update refers to a security update with a state designated as "closed" or "merged". In $\boldsymbol{RQ_5}$, however, our focus shifts to every security update that precedes a vulnerability fix and is not classified as "merged". To investigate the possible connection between the security updates merge ratio and certain projects' attributes, such as popularity or the adoption of testing and CI tools, we need to extract metadata for each repository. We measure the popularity of the repositories by the number of their GitHub stars and forks. To determine if a project includes tests, one approach could be to scan the history of each repository to find actual test cases. This strategy, however, would introduce additional complexity to our study. In contrast to certain other programming languages such as Java, where developers tend to follow established conventions when writing tests, there is no universally adopted method for writing test code in JavaScript projects. To determine whether a repository employs testing, we leverage a list of widely recognized testing frameworks studied by Delčev and Drašković (2018), supplemented by the most popular JavaScript testing frameworks based on the number of stars on GitHub. The resulting catalog comprises Mocha, Jest, Jasmine, Karma, Puppeteer, Nightwatch, Cypress, Playwright, and Selenium. Afterward, for each repository, we traverse their commit history to monitor the presence of each testing tool in their dependency file. If a testing tool is available under direct runtime or development dependency in the package.json file corresponding to the beginning and the end of our observation period, the repository is considered to be utilizing that testing tool.

Regarding CI pipelines, their usage in a repository is reflected in the presence of specific configuration files. For instance, a *.travis.y(a)ml* file indicates that

Travis CI has been configured for this repository while a *.y(a)ml* file in the *.circleci/* directory or a *circle.y(a)ml* in the root directory triggers CircleCI. In this study, we scan repository commits to find configuration files of the 7 most prominent CI tools on GitHub identified by Golzadeh et al. (2022): TravisCI, GitHub Actions, CircleCI, Jenkins, GitLab, AppVeyor, and Azure. It is worth noting that although certain repositories may contain CI configuration files, they may not necessarily be actively employing the CI tool in their development processes. For instance, repositories might be experimenting with the integration of a certain CI service (Golzadeh et al., 2022). The frequency of commits to the configuration files can serve as an indicator of active CI tool usage in a repository. Therefore, consistent with our criteria for project extraction, we consider a repository as actively utilizing a CI tool only if its configuration file is updated, on average, every month throughout our analysis timeframe.

It is important to highlight that, in order to investigate the potential relationship between the merge ratio and project attributes (***RQ₂***), as well as to identify manual fixes possibly inspired by Dependabot (***RQ₅***), an examination of the history of each repository is necessary. However, as these two analyses were conducted at a later time compared to the rest, out of the original dataset of 978 projects, were able to access the commit history of 954 projects. We noticed that the remaining were either deleted or rendered private by their respective owners. It is noteworthy that to preserve as many repositories as possible, we leveraged both *GitHub* and *Software Heritage* (Di Cosmo and Zacchiroli, 2017)[3]. Given that the 954 accessible projects represent a substantial majority, exceeding 97% of our initial repository count, we anticipate that the outcomes of ***RQ₂*** and ***RQ₅*** can be interpreted within the same context as the other research questions.

4.2 Detecting Vulnerabilities in Dependencies

To uncover vulnerabilities, we accessed the security advisories via the *GitHub GraphQL API* on March 27, 2021. Subsequent removal of vulnerabilities without known fixes resulted in 1,063 security advisory records. Considering the continuous evolution of the GitHub advisory database, we examined each retrieved security update to determine if there is at least one associated security advisory from the collected set. As information regarding the vulnerabilities targeted by a security update is inaccessible without specific project permissions, we identify the association of an advisory with a security update by extracting information from the title of security updates. These titles typically contain details about the vulnerable upstream package, its current installed version, and the suggested upgrade release by Dependabot (cf. Figure 1). We found 28 records (0.6%) in our dataset that lacked corresponding security advisory records. Hence, these records were excluded from further analysis.

[3] `https://www.softwareheritage.org/`. Last accessed 11 January 2024.

The proposed vulnerability detection algorithm leverages three input parameters: (1) the name of the concerned upstream package with known security vulnerabilities, (2) the database of security advisories, and (3) the dependency files of the selected repository. Utilizing this input, the algorithm determines the set of vulnerabilities inherited through the specified upstream package dependency. A hosted repository is considered vulnerable if any of its dependency files include a version declaration for an upstream package known to contain security vulnerabilities. However, the aforementioned definition applies differently to manifest and lock files.

A manifest file is considered to declare a dependency on a vulnerable release when the specified upstream package is directly present as either a runtime or development dependency, while its most recent version that satisfies the defined dependency constraint is affected by the identified vulnerability. It is noteworthy that this definition expands upon the one adopted by Decan et al. (2018b), incorporating both runtime and development dependencies. As a final consideration, there is an additional rule that applies if the repository follows the *monorepo* paradigm (Brito et al., 2018), indicating it encompasses multiple projects. In this scenario, the direct dependencies of the hosted sub-modules, whose relative paths are defined through the "workspaces" field in the manifest file, are considered direct dependencies of the entire top-level module.

Regarding lock files, two scenarios are considered for determining if they declare a dependency on a vulnerable release. In the first scenario, the specified upstream package with a known security vulnerability is declared as a direct runtime or development dependency in the corresponding manifest file. To determine whether the examined lock file declares a dependency on a vulnerable release of the upstream package in question, only the release associated with the node in the dependency graph representing this direct dependency is checked for vulnerability, and the nodes associated with the transitive dependencies are disregarded. In the second scenario, the specified upstream package with a known security vulnerability is not declared as a direct runtime or development dependency in the manifest file. Then, contrary to the previous case, we recursively traverse every dependency node object in the graph, examining those defined by the name of the concerned upstream package and collecting the releases locked to them. If the locked release of at least a single node in the dependency graph belongs to the range of the affected versions, then the lock file is deemed to declare a dependency on a vulnerable release of an upstream package.

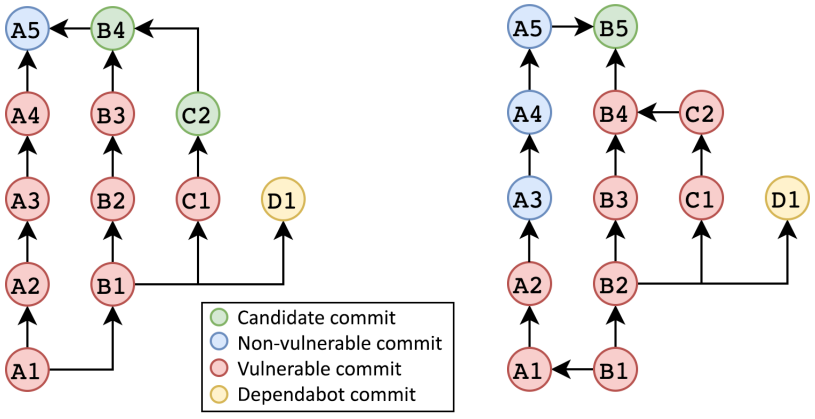
The vulnerability detection algorithm is assessed through a binary classification approach leveraging Dependabot security updates. The *parent commit* of a security update, denoting the commit upon which the modification is built, reflects the project's state with a vulnerable dependency. This is because every security update addresses at least one vulnerability within the dependencies. Conversely, the *merge commit*, representing the result of merging the security update, marks the state wherein the vulnerable dependency is addressed. For each repository, the algorithm is fed the dependency file associated with each of these two commits. The algorithm's evaluation proceeds as follows: For the

parent commit of a security update, the algorithm should return at least one security advisory associated with the selected upstream package, published before the corresponding security update was generated. In contrast, for the merge commit, no related vulnerabilities should be returned.

Upon executing the algorithm on all security updates, it shows no false negatives, accurately identifying vulnerable dependencies in every parent commit associated with security updates in our dataset. Nonetheless, the analysis uncovered 133 cases of false positives for merged security updates. In these instances, the algorithm detected the existence of vulnerabilities despite Dependabot's intended resolution. Closer examination attributed these false positives to a bug within Dependabot itself, rather than an issue with the algorithm. In such a situation, Dependabot inadvertently ignores all but one range of vulnerable releases and adjusts the yarn.lock file accordingly. Consequently, at least one dependency resolution block continues to reference a vulnerable release even after the bot's modification (Dependabot, 2020). For good measure, we replicated the described conditions within a GitHub repository, activating both the Dependabot security alerts and security updates. We observed that even after merging the automated security update generated by Dependabot, the corresponding security alert persists. The presence of this bug is also confirmed in a discussion over a Dependabot security update (Dependabot, 2020).

4.3 Fixing Cases Discovery

To identify cases where a vulnerable dependency was fixed, we employed the vulnerability detection algorithm described in the previous section on the parent commits of each security update. Subsequently, we generated distinct entries for every reported security advisory. Accordingly, each entry is distinguished by the following four attributes: (1) the slug, denoting the name of the repository owner, (2) the corresponding security advisory, which includes the name of the affected upstream package, (3) the relevant (sub-)modules containing the dependency files declaring a vulnerable dependency, and (4) the parent commit of the associated security update. The latter attribute enables pinpointing the initial moment within a repository's history where Dependabot identified the specific vulnerability. We identified 5,089 vulnerabilities in our initial analysis. However, Dependabot can generate new security updates for existing vulnerabilities if a newer version of the upstream package is required. This means some of the collected events captured the state when a security update was recreated, not the initial identification of the vulnerability. To address this, we linked superseded and superseding updates and removed redundant entries, resulting in a collection of 4,978 vulnerabilities. Lastly, to collect the fixing cases, we trace the *fixing commit* for each identified security vulnerability. We define a fixing commit as the earliest modification to the dependency files that resolves the specified vulnerability in the dependencies, eventually merging into the default branch of the repository. To identify the fixing commit, we designed an algorithm that recursively explores the descendants of each security



(a) Simple scenario (b) Complex scenario

Fig. 3: Candidate and fixing commits scenarios.

update's parent commit and determines whether the relevant vulnerability has been resolved. Since the repository history often comprises more than one branch, the algorithm may yield multiple *candidate fixing commits*. The rationale behind this is that when the original modification carrying the fix, located within a particular development branch, is incorporated into another branch via a merge operation, the earliest node with the resolved vulnerable dependency on that subsequent branch becomes the merge commit. Therefore, as the algorithm traverses each development branch autonomously, it may return multiple commits. Nevertheless, there is also a possibility that none of the candidates is the fixing commit. We regard such a scenario as *complex*, as opposed to *simple*, where one of the candidates is the fixing commit. Figures 3a and 3b showcase excerpts of the repository history. Each node denotes a commit, while the directed edges depict the parent-child relationships between them. In Figure 3a, the parent commit of the security update generated by Dependabot is `B1`, with the candidate commits `B4` and `C2`. In this simple scenario, the fixing commit `C2` is a descendant of `B1`. Conversely, in the complex scenario depicted in Figure 3b, the fixing commit `A3` is not among the candidates, as the commit `B2`, which is the parent of the commit initiated by Dependabot, is not its ancestor. This discrepancy arises from the fact that branch `A`, from which the fix originated, was forked before `B2` was created.

The presence of the complex scenario elucidates the challenge of achieving complete automation in discovering the fixing commit. Attending to such scenarios requires traversing the network graph backward, which significantly increases the number of nodes to be visited. Hence, we adopted a semi-automated approach where a human rater manually determined the fixing commit based on the list of candidate commits. In the complex scenario, however, the earliest

candidate commit arises from a merge, necessitating manual investigation of its ancestors to identify the fixing commit. To assess the extent of accidental errors or the potential bias resulting from manual assignment, we recruited another independent rater. They were presented with 50 events identifying a security vulnerability, each accompanied by a list of candidate fixing commits. Half of the events pertained to a complex scenario, without disclosure to the rater. Our comparison revealed no discrepancies between the fixing commits reported by the original and the second rater, enhancing our confidence in the accuracy of the collected fixing cases.

4.4 Discovering Inspired Fixes

At times, it is observed that the security update proposed by Dependabot remains unmerged or is rejected, while an identical update (as the one suggested by Dependabot) is executed manually by developers. Moreover, during the fixing process, a more recent version of the concerned package is available. Despite this, developers choose to utilize the precise version proposed by the security update in their manual fix. In this scenario, there exists a possibility that the fix is inspired by the bot. While it may not be feasible to identify such cases with absolute certainty, we adhere to the following heuristics. Upon generating a security pull request, Dependabot recommends elevating the version of the vulnerable package to the minimum non-vulnerable version, rather than the latest available version. This implies that when a developer chooses to upgrade a package to the precise version suggested by the bot, it is likely inspired by a previous security update, particularly when there is a more recent version available than the one selected by the developer. Hence, in a nutshell, *we recognize a manual fix as being potentially inspired by the bot only if the fixing version precisely matches the one suggested by the bot and differs from the latest available version of the package.*

In our dataset of manual fixes, each entry contains a *fixing version* within its dependency file and/or lock file. Regarding the dependency file, we consider the minimum version satisfying the version criteria as the fixing version, while for the lock file, the fixing version corresponds to the specific version assigned to the concerned package. For each manual fix, we search for a prior security update recommending the same fixing version. If this version differs from both *the latest version* and *the latest stable version* of the package at the time of the fix, we mark this manual fix as inspired. To extract the latest version and the latest stable version of each vulnerable package, we leverage the *view* command within the npm package manager, which grants us access to version history for every package.

4.5 Data Analysis

This section describes the techniques employed to answer our research questions.

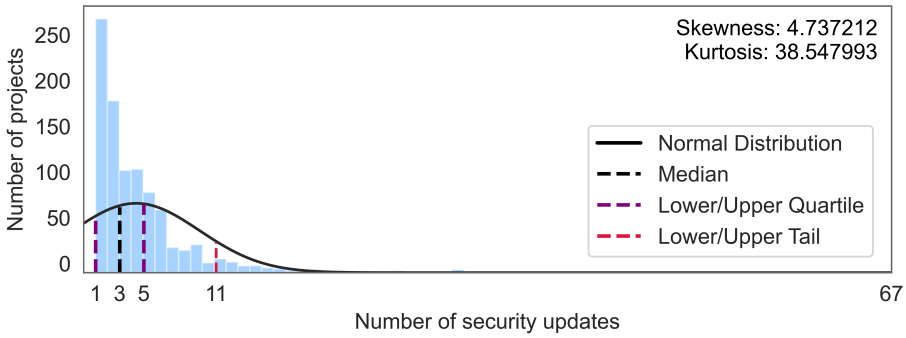


Fig. 4: Distribution of the number of security updates.

Table 2: Project classification based on the number of security updates.

Group	Number of security updates	
	Constraint	Interpretation
Very low	[lower quartile, median)	[1, 3)
Low	[median, upper quartile)	[3, 5)
High	[upper quartile, upper tail)	[5, 11)
Very high	[upper tail, maximum]	[11, 67]

4.5.1 ***Addressing RQ$_1$***

To gauge the frequency with which developers merge the security updates created by Dependabot, we compute the *merge ratio*, defined as the proportion of non-open security updates that were merged. Nonetheless, project practices can vary, leading to different responses to security updates. Therefore, we conduct a more detailed analysis by assessing this metric on a per-project basis. It is important to note that if a project receives only a small number of security updates (*e.g.,* 1 or 2), the merge ratio is extremely likely to be either 0% or 100%. Thus, if this is the prevailing case, the distribution of merge ratios will mainly be shaped by these projects, leading to a significant skew toward both extremes.

Figure 4 illustrates the distribution of the number of security updates for the projects in our collection, along with the coefficients of *skewness* and *kurtosis* (Kokoska and Zwillinger, 2000), whose acceptable values for normality lie between -1 and 1 and between -2 and 2, respectively. It is evident that the distribution exhibits a very high positive skew. Hence, we adopted a quantile classification approach (Lanza and Marinescu, 2007) to categorize projects into four distinct groups based on the number of security updates they receive, namely *Very low*, *Low*, *High*, and *Very high*. Table 2 presents the cut-off points used for this classification.

4.5.2 ***Addressing RQ$_2$***

To address this research question, we first identify relevant project attributes. We choose popularity metrics, such as stars and forks, because previous studies indicate that more popular projects are more likely to adopt test automation practices (Lin et al., 2021) and may have fewer reservations about potential disruptions from merging pull requests, potentially leading to a higher merge ratio. Additionally, we consider project size (size of the total code of the project (Joy et al., 2018)) as a factor that reflects the complexity of the projects, as well as the age of the projects, which might indicate their maturity and maintenance practices. Finally, we gather data on the number of contributors and commit counts, as a greater number of contributors could increase the likelihood of prompt merges, while higher commit counts may suggest greater activity within the repository (Joy et al., 2018)).

Given the non-normal distribution of our data, we employ Spearman's rank correlation test (Zar, 2005) to examine whether there is a correlation between the merge ratio of pull requests and selected features. To discover a potential connection between the merge ratio of security updates and the presence of testing and CI tools, we compare the mean merge ratio of projects utilizing these tools with those that do not. We designate a project as employing testing frameworks or CI tools if we find evidence of at least one such tool in their repository's dependency history. Consistent with our approach in addressing **RQ$_1$**, for a more thorough evaluation, we also report a mean merge ratio for each project group, considering the availability of testing and CI tools.

4.5.3 ***Addressing RQ$_3$***

To evaluate how frequently developers manually resolve vulnerabilities in dependencies despite the presence of a Dependabot security update, we measure the percentage of vulnerabilities in the collected dataset that are (1) fixed by Dependabot, (2) fixed by a developer, or (3) remain unaddressed. We determine the third category by the absence of a fixing commit while discerning between the first and second categories is more intricate. We define a vulnerable dependency as fixed by Dependabot only if the fixing commit is authored by the bot; otherwise, we attribute the fix to the developers.

To ensure consistency with **RQ$_1$**, we also calculate the percentages for the four groups of projects separately: (1) the percentage of vulnerabilities that are addressed versus not addressed, and (2) out of all fixes, the proportion contributed by a bot versus implemented by a human. In case of an observable discrepancy in the results across the various groups, we measure the statistical significance of this variation by constructing the contingency table with the absolute values and applying Pearson's χ^2 test (Pearson, 1900) of independence. This test allows us to assess whether there is a relationship between two categorical variables (Turhan, 2020). When two variables are associated, the probability of one variable taking on a certain value depends on the value of the other variable. The calculations for the chi-square test of independence

are based on observed frequencies, representing the number of observations in each combined group (McHugh, 2013). We reject the null hypothesis H_0, which posits no relationship between the number of security updates received by a project and the expected response to a vulnerability if $p < 0.05$ (the 5% significance level). We also present the effect size (APA, 1994) (the magnitude of this relationship) when Pearson's χ^2 test indicates rejection of the null hypothesis. Despite the plethora of methods available for computing the effect size (APA, 1994), for a contingency table whose size exceeds 2×2, it is recommended (Healey, 2009; Howell, 2007) to use Cramér's V (Cramér, 1946), denoted by ϕ_V. This metric ranges between 0 and 1, with the former corresponding to a lack of association. We adhere to the interpretation of ϕ_V proposed by Cohen (1988), which suggests that for a 4×2 contingency table, the association between two variables is deemed *trivial* if $\phi_V < 0.10$, *small* if $0.10 \leq \phi_V < 0.30$, *medium* if $0.30 \leq \phi_V < 0.50$, and *large* if $\phi_V \geq 0.50$. However, the rejection of the null hypothesis across an entire contingency table and observing a non-negligible effect size do not inherently suggest statistical significance in differences between each pair of groups, nor do they specify which groups exhibit significant distinctions. Therefore, we finalize the analysis by conducting $\binom{4}{2} = 6$ pairwise comparisons, *i.e.*, Pearson's χ^2 tests, to further scrutinize the differences between groups. To mitigate the elevated risk of a type I error resulting from multiple statistical tests, we apply the Benjamini-Hochberg correction procedure (Benjamini and Hochberg, 1995) to adjust the p values and control the false discovery rate.

We also examine the characteristics of both manual fixes and those made by the bot, focusing on whether the fix involves a major version bump (e.g., from 1.0.0 to 2.0.1), a minor version bump (e.g., from 1.1.0 to 1.2.0), or a patch update (e.g., from 3.2.1 to 3.2.2). This analysis helps us determine whether the type of version bump influences developers' decisions to address vulnerabilities manually or delegate them to Dependabot. While minor and patch updates typically maintain backward compatibility, major version changes may introduce breaking changes, potentially discouraging developers from merging them automatically and leading them to handle these updates manually instead.

4.5.4 ***Addressing RQ$_4$***

In examining the fourth research question, our objective is to gauge the timeliness of developers' reactions to vulnerabilities identified by Dependabot. We consider the time required to resolve a vulnerable dependency as the duration between the moment the vulnerability was flagged by Dependabot through a security update and the time the fixing commit was made.

In line with prior research (Decan et al., 2018b, 2017; Lin et al., 2017; Samoladas et al., 2010; Constantinou and Mens, 2017), we rely on survival analysis (Aalen et al., 2008) to assess the distribution of time-to-event. This approach offers the advantage of accounting for vulnerabilities that remain unaddressed by the end of the observable period, referred to as *censored* observations. We employ the Kaplan-Meier estimator (Kaplan and Meier, 1958),

a non-parametric statistic, to construct a survival analysis model for estimating the survival rate of vulnerabilities in dependencies. This survival rate signifies the expected duration until an actionable reaction to a vulnerability occurs over time. Consequently, the survival function demonstrates the probability of a vulnerability persisting beyond a particular time point.

We also conduct the survival analysis with respect to the severity levels of vulnerabilities. This enables us to examine whether there is a correlation between the risks posed by a vulnerability and the time taken by developers to respond to it. In other words, we investigate whether developers prioritize vulnerabilities based on their severity. To inspect the significance of any observable difference between each pair of severity levels, we conduct $\binom{4}{2} = 6$ pairwise comparisons using the log-rank test (Fleming and Harrington, 2011), which is the standard testing procedure for comparing time-to-event distributions. The null hypothesis H_0 posits that there is no difference in the survival distributions between the two groups. In the absence of a meta-test that evaluates all four survival curves concurrently, we control the family-wise error rate, which represents the probability of committing at least one type I error. Following the Bonferroni approach (Mittelhammer et al., 2000), we test each individual hypothesis at a significance level of 0.83% ($= \alpha/T$ where $\alpha = 5\%$ denotes the desired overall significance level and $T = 6$ represents the number of comparisons). Lastly, we measure and compare the durations required to resolve vulnerabilities between fixes executed by the bot and those performed manually. Since vulnerabilities that remain unresolved by the end of the observation period cannot be attributed to either bot or human intervention, there are no instances of censored observations. Thus, conducting survival analysis becomes unnecessary , and instead, we assess the distributions of bot and human fixes using violin and box plots. To statistically confirm any disparities, we employ the one-sided non-parametric Mann-Whitney U test (Nachar et al., 2008) at a conventional 5% significance level. The decision to use the one-sided alternative is driven by the expectation that vulnerability fixes attributed to Dependabot require less time than those implemented manually.

Finally, we also explore whether maintainers fix vulnerabilities in batches, potentially within a short timeframe, such as during their development sprints. In such cases, the resolution of security vulnerabilities may be postponed or take longer to complete. To investigate this, we conducted a temporal analysis for each repository, examining fixes made through separate commits to identify groups of fixes implemented within a one-month period (equivalent to the maximum length of an agile sprint (Abrahamsson et al., 2017)).

4.5.5 ***Addressing RQ_5***

To discern manual fixes potentially inspired by Dependabot, we conduct a stringent comparison of the fixing version extracted from the fixing commit with the version suggested by each preceding security update, the latest version, and the latest stable version of the relevant vulnerable package. If the fixing version matches one of the earlier security updates and is different from both

the latest version and the most recent stable version of the relevant package at the time of the fix, we categorize this manual fix as inspired. We categorize and report inspired cases based on whether the fixing version originates from the dependency file and/or the lock file. We also investigate whether higher severity vulnerabilities are more likely to inspire manual fixes. Furthermore, we conduct a manual analysis of the commit messages associated with inspired fixes to find possible links to security updates. Specifically, three authors independently categorize each message into one or more of the following groups: 1) explicitly referencing a Dependabot security update, 2) addressing security vulnerabilities, and 3) updating dependencies. If a commit message referenced a pull request, we investigated whether it pertained to a previous Dependabot security update. In cases where the commit message was unclear or ambiguous, we closely examined the actual code changes to clarify the nature of the modifications.

To assess the time required by developers in implementing inspired fixes compared to those that are not inspired, we need to compare the distribution of the time spent by developers on each type of fix. In particular, this comparison will provide insights into whether inspired fixes are typically implemented more swiftly, especially given that the developer uses the exact fixing version directly suggested by a received security update. In this regard, to showcase the two distributions visually, we initially employ Boxen plots (or letter-value plots) (Heike Hofmann and Kafadar, 2017), an advanced visualization of boxplots designed to provide a more accurate representation of distributions. Additionally, similarly to $\mathbf{RQ_4}$, we utilize the Mann-Whitney U test to examine whether the overall time distributions of the two types of fix are statistically different. To achieve a more detailed analysis, we also conduct a per-repository Mann-Whitney U test. Since a valid Mann-Whitney U test requires a sample size of at least 4 when $\alpha = 0.05$ (Fay and Proschan, 2010), we conduct the test for each repository containing at least 4 manual fixes. We report any distinctions observed in the time distributions of inspired and non-inspired fixes within each repository.

5 Results

5.1 RQ1: To what degree do developers merge Dependabot security updates?

To address **RQ1**, we compute the merge ratio for non-open Dependabot security updates (cf. Section 4.5.1). The findings reveal that out of the 4,195 non-open Dependabot security updates in our dataset, 57% are merged. To examine whether manual interventions are necessary after they are merged, we investigate if project maintainers ever roll back these updates. Specifically, we analyze the commit history of each repository during our observation period to identify any occurrences of merged security updates being reverted. Our findings show no such rollback incidents, suggesting that these automated updates integrate seamlessly into projects without requiring further manual adjustments.

Table 3: Distribution of merge ratios for the four project groups.

Group	Min	25%	Median	75%	Max	Avg	Std
Very low	0%	0%	50%	100%	100%	49%	47%
Low	0%	0%	75%	100%	100%	59%	43%
High	0%	0%	80%	100%	100%	57%	42%
Very high	0%	26%	78%	94%	100%	61%	38%
Total	0%	0%	67%	100%	100%	54%	44%

Table 3 displays the distribution of merge ratios for the projects examined in our study. It is evident that, on average, projects merged 54% of their non-open security updates (7th column in Table 3), and the median project merged 67% of its non-open security updates (4th column in Table 3). Moreover, the mean merge ratio for the projects in all categories except *Very Low*, is 75% or higher. Nonetheless, we refrain from drawing firm conclusions about the trend in specific project groups due to the relatively narrow range of average values (ranging between 49% and 61%) and the considerable standard deviation (ranging from 38% to 47%).

To assess potential differences in the distributions of merge ratios between project groups, we conduct multiple Mann-Whitney U tests to validate the null hypothesis stating that "the samples in category X come from the same population as the ones in category Y", where X and Y represent the four project groups (*Very low*, *Low*, *High*, *Very high*). Note that due to multiple comparisons over the same data, we apply Bonferroni's correction (Armstrong, 2014) and report the corrected p-values. The results indicate that the null hypotheses (one for each distinct pair of project groups) cannot be rejected (adjusted $p \geq 0.15$), suggesting no significant differences between repositories of different project groups concerning their willingness to merge Dependabot security updates.

To investigate potential trends throughout our observation period, such as variations in the merge ratio of security updates or the resolution time of vulnerabilities across different intervals, we partition our dataset into three segments (P1, P2, P3), each representing a period of 4 months. Our examination across the three temporal segments indicates that projects within all four categories demonstrate a greater merge ratio during P1. However, since the majority of projects receive only a small number of security updates, we are unable to draw conclusions about the adoption within projects throughout the observation period. When we conduct the analysis exclusively on projects that receive security updates throughout the entire observation period, we observe that the average merge ratio per category remains relatively consistent across all periods.

Table 4: Mean merge ratio categorized by the presence of tests or CI tools within each project group.

Group	Tests		CI		Tests + CI	
	✓	×	✓	×	✓	×
Very low	53%	50%	40%	53%	38%	51%
Low	52%	59%	64%	55%	**94%**	61%
High	57%	54%	66%	54%	**61%**	52%
Very high	60%	63%	69%	60%	**63%**	61%
Total	54%	54%	**55%**	54%	**58%**	55%

5.2 RQ2: How do repositories' attributes impact security updates merge ratio?

To answer this research question, we performed Spearman's rank correlation analysis between the merge ratio and popularity metrics, as well as project size, age, number of contributors, and commit count. The results indicate a weak correlation with stars, forks, and the number of contributors, with $\rho = 0.16$ for stars and forks, and $\rho = 0.12$ for contributor count. We observe similar results when performing the correlation analysis across different project groups. The only exception is the group with a *high* number of security updates, where the correlation is stronger: both stars and forks show a moderate correlation at $\rho = 0.31$, and contributor count at $\rho = 0.27$, with all p-values being less than 0.001. Overall, these findings suggest that, although popular projects may exhibit a greater inclination towards testing practices (Lin et al., 2021), we cannot conclude that these projects are inherently more receptive to Dependabot security updates. The same observation applies to projects with a larger number of contributors. This means that while these repositories may boast more contributors or a larger number of test cases, this may not necessarily translate to a reduced concern about potential breaking changes and, consequently, a higher merge ratio. We also did not find a significant correlation between the merge ratio and other repository attributes, such as project size, age, and commit count.

Table 4 presents the mean merge ratio of various project groups based on their adoption of tests, CI tools, or both. The results show that the merge ratio remains consistent at 54% for both projects that adopt testing and those that do not. Moreover, projects with CI exhibit a merge ratio of 55%, compared to 54% for projects where no evidence of CI utilization was found. When comparing repositories that have both testing practices and enabled CI services to those with neither, the disparity in merge ratios widens, with figures of 58% and 55%, respectively. In particular, for the project groups with *low* and *high* number of security updates, the mean merge ratio is considerably higher when both testing and CI are implemented. Although these results do not provide strong evidence that having testing and CI directly contributes to a higher merge ratio of security updates, we do observe that the presence of these tools in projects slightly enhances the likelihood of merging a security

Table 5: Percentages of vulnerabilities addressed by human/bot and non-resolved per project group.

Group	Response to vul.		Fixed by	
	Fixed	Not fixed	Bot	Human
Very low	76.35%	**23.65%**	52.83%	**47.17%**
Low	**90.77%**	9.23%	61.36%	38.64%
High	86.04%	13.96%	64.11%	35.89%
Very high	84.07%	15.93%	**71.11%**	28.89%
Total	84.62%	15.38%	63.85%	36.15%

update. As the potential for breaking changes impacts the decision to merge pull requests (Decan et al., 2018b; Haenni et al., 2013; Bogart et al., 2016; Chinthanet et al., 2021; Huang et al., 2019), this could suggest that such tools could ease concerns related to merging security updates. Consequently, project maintainers merge security updates with greater confidence, knowing that any potential bugs would promptly be detected by running the test cases through continuous integration pipelines. This argument is bolstered by the findings of our qualitative analysis of project artifacts in Section 6, where compatibility challenges emerge as one of the primary reasons for maintainers to decline security updates. It is noteworthy that the absence of concrete evidence linking testing and CI tools to a higher merge ratio may be due to these tools not being effectively utilized in practice.

5.3 RQ3: How frequently do developers fix a vulnerable dependency manually in the presence of a Dependabot security update?

To answer this question, we calculate the percentage of vulnerabilities that were (1) fixed by Dependabot, (2) fixed by a developer, or (3) remain unattended. There are 4,978 security vulnerabilities in our collection, derived from the 4,195 security updates discussed in $\mathbf{RQ_1}$. The findings reveal that 53.48% (2,662 out of 4,978) of the vulnerabilities are mitigated by merging Dependabot security updates, 30.27% (1,507 out of 4,978) are manually resolved, and the remaining 16.25% (809 out of 4,978) remain unfixed. It is noteworthy that nearly one-third of the vulnerabilities in our dataset were resolved manually, despite the presence of Dependabot security updates. Considering the proportion of unresolved vulnerabilities, bot fixes occur 1.8 times more frequently than manual fixes. This suggests that, on the whole, Dependabot is largely entrusted with the task of vulnerability resolution.

Table 5 outlines the distribution of vulnerabilities across project groups based on their resolution status (meta-column *Response to vulnerabilities*). A distinct contrast is evident in these percentages. Pearson's χ^2 test confirms the significance of this difference ($p = 1.19e\text{-}15$), with a non-negligible (small) effect size of $\phi_V = 0.12$. Also, the group characterized by a *very low* number of

Table 6: Computed p-values for the pairwise comparisons between the project groups. Significance: '***' < 0.001, '**' < 0.01, '*' < 0.05. Blue cells highlight the cases when $p < 0.05$.

fixed vs not fixed \ *fixed by bot vs. fixed by human*	*Very low*	*Low*	*High*	*Very high*
Very low		6.41e-15***	3.99e-09***	1.57e-05***
Low	1.70e-03**		7.66e-04***	7.43e-06***
High	3.95e-06***	2.08e-01		1.04e-01
Very high	4.32e-14***	1.24e-05***	1.85e-04***	

received security updates exhibits the lowest proportion of resolved vulnerabilities. Conversely, the group with a *low* number of security updates demonstrates the highest ratio of resolved vulnerabilities, marking a 14% contrast with the first group. Moreover, with an increase in the number of security updates, there is a corresponding rise in the proportion of unresolved security vulnerabilities. This trend may stem from developers becoming overwhelmed with notifications about vulnerable dependencies. To substantiate this conjecture, we compare the proportions of security fixes executed by the bot and humans (as indicated in the meta-column *Fixed by* in Table 5). We observe that the reliance on Dependabot for security fixes could escalate with the rise in the number of security updates ($p = 9.89e-14$ and $\phi_V = 0.12$, *i.e.*, the difference is significant with a small effect size). This discovery could potentially be attributed to a deeper experience with the bot, as denoted by Alfadel et al. (2021). Furthermore, the increased reliance on the bot for vulnerability resolution may spring from several factors: (1) the intricate nature of the project and the relationship between its dependencies, which impede manual vulnerability mitigation without recalculating the dependency tree, (2) the effort and time required to address numerous vulnerabilities or (3) a combination of both these points.

However, as evidenced by the outcomes of the post hoc pairwise comparisons reported in Table 6, probably, the reality lies somewhere in the middle. Put differently, both experience with the bot and the intricacies of the project likely contribute to the observed phenomenon. In all cases except for two, we observe a significant difference (at an overall significance level of $\alpha = 5\%$). The first exception pertains to the trivial difference in vulnerability response between the *high* and *very high* groups, despite a substantial discrepancy in the delegation of security fixes. This indicates that a majority of projects within the second group offset the elevated intricacy or increased number of vulnerabilities by assigning the additional security workload to the bot. Hence, there is a substantial rise in the ratio of security fixes executed by the bot without a corresponding surge in the degree of vulnerability resolution. The second exception to the trivial difference concerns the assignment of security fixes between the *low* and *high* groups, regardless of a significant discrepancy in vulnerability response. This implies that projects affiliated with the latter group might struggle to offset the increased complexity by entrusting the resolution

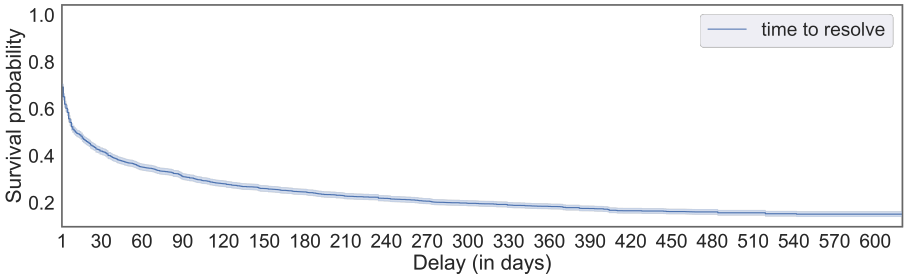


Fig. 5: Survival curve for the event "vulnerability identified by Dependabot is addressed".

of vulnerabilities to Dependabot, possibly in light of insufficient experience or trust in the bot's capabilities.

Our analysis of both manual and bot-generated fixes reveals that the proportion of minor and patch version bumps is similar across both types of fixes Specifically, 51.72% of bot fixes and 53.74% of manual fixes are patch updates, while 42.78% of bot fixes and 38.35% of manual fixes involve minor version bumps. Nonetheless, manual fixes that include a major version bump occur at a rate nearly 1.5 times higher than those done by the bot (6.78% compared to 4.65%). This could suggest that maintainers have a higher tendency to handle major version updates manually, likely due to the potential for breaking changes that require more careful review and testing.

5.4 RQ4: What is the duration required to address a vulnerable dependency identified by Dependabot?

To answer this question, we employ survival analysis to undertake a time-to-event evaluation. Figure 5 depicts the survival curve of the vulnerable dependencies identified by Dependabot. We can observe that the probability of a vulnerability remaining unaddressed within the first day is below 70%. In practice, this implies that nearly a third of vulnerable dependencies are typically resolved within 24 hours of receiving the security update. Moreover, the findings indicate that the majority of vulnerable dependencies are addressed within the initial two-week period. Therefore, we infer that developers often act promptly on Dependabot's recommendations. Nonetheless, there remains an 18% probability that a vulnerable dependency flagged by the bot stays unaddressed for over a year, implying that approximately one in five vulnerabilities could affect users of dependent projects for at least an entire year since the advisory is published. This underscores the need to reinforce developers' incentive to promptly resolve vulnerable dependencies.

The assessment of the association between vulnerability survival and severity indicates a negative correlation between severity level and survival probability when comparing critical and high-severity vulnerabilities to moderate and

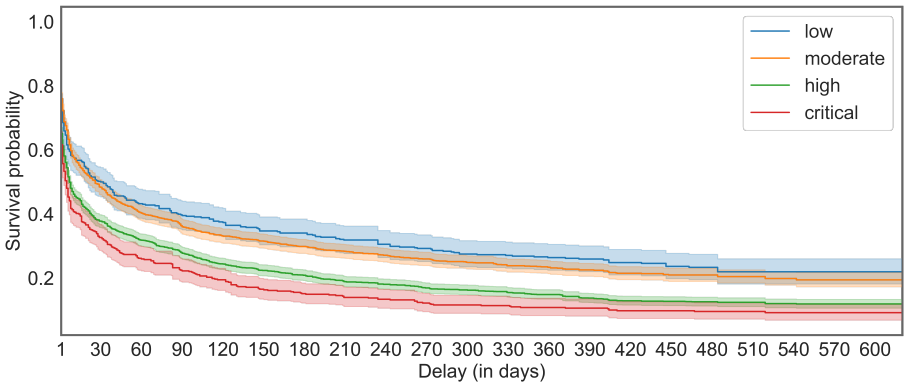


Fig. 6: Survival curves for the event "vulnerability identified by Dependabot is addressed" based on severity level.

low ones, contradicting the findings of Alfadel et al. (2021). Evidently, a distinct contrast exists between the first two classes of vulnerabilities. When considering an equivalent period, a critical severity vulnerability consistently exhibits a lower probability of remaining unattended. This predominance is further accentuated when compared to vulnerabilities of low and moderate severity. Specifically, a critical severity vulnerability identified by Dependabot typically exhibits 15-20% higher likelihood of eliciting an actionable response within an identical period. Throughout an entire year following the notification, a low severity vulnerability is 2.3 times more inclined to persist than a critical one and 1.7 times more than a high one. The analysis of pairwise log-rank tests confirms a significant difference between each severity level, except for the comparison between low and moderate severity vulnerabilities. This suggests that developers consider the severity of vulnerabilities when determining the priority of a dependency update.

Predictably, when comparing bot fixes with those implemented manually, we observe that the former requires significantly less time than the latter (at a significance level of 0.01%). The distributions of fixing times depicted in Figure 7 illustrate a notable contrast between the two classes of vulnerability resolutions. Approximately 50% of the security updates proposed by Dependabot are merged within a day, with another 25% merged within eleven days. Conversely, less than a quarter (precisely 18%) of manual fixes are implemented within a day, while half of them require at least 1.5 months. This disparity suggests that security fixes are either swiftly resolved by the bot or demand a significantly longer duration when addressed manually by developers. When partitioning our dataset into three segments (P1, P2, P3), each representing 4 months, we find vulnerabilities identified in the initial period (P1) exhibit a tendency to be resolved more swiftly compared to those in the subsequent two periods (P2, P3). For instance, within 60 days, approximately 75% of vulnerabilities were resolved in P1, whereas the figures for P2 and P3 were around 50% and 60%

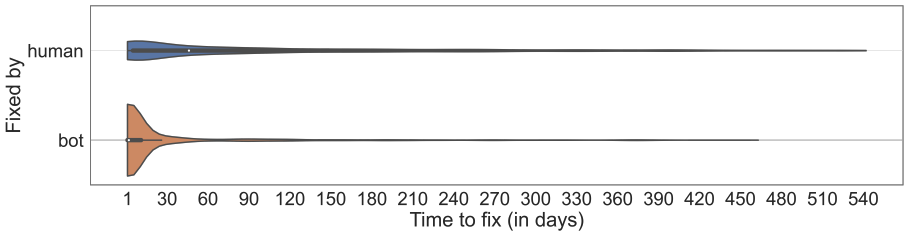


Fig. 7: Violin plots for the distributions of the bot and manual fixing times.

respectively. Hence, vulnerabilities were addressed more rapidly at the outset of our observation period.

Our temporal analysis, focused on identifying batch fixes that might delay the resolution of security vulnerabilities, reveals that in bot-generated fixes, 48 repositories (approximately 5%) contained at least one instance of batch fixes, where at least 5 vulnerabilities were resolved together during a timeframe. In contrast, we found only 2 repositories where manual fixes were batched in this way. This indicates that batch fixes are uncommon in the repositories we analyzed and primarily occur in bot-driven fixes. Furthermore, their rarity means they are unlikely to contribute to delays in resolving vulnerabilities.

5.5 RQ5: What proportion of manual fixes are potentially inspired by Dependabot security updates?

Our objective in answering this research question is to understand to what extent manual fixes executed by the developers are potentially inspired by a security update they have received beforehand. In other words, when developers encounter a security update, they may choose to implement the exact changes suggested by the bot manually, often due to concerns about potential compatibility issues or breaking changes. Referring to the findings from $\mathbf{RQ_3}$, we know that 30.27% of the fixes are handled manually, constituting 1,507 cases out of the total 4,978 fixes identified. Some of these fixes involve the complete removal of the vulnerable dependency, as developers possibly recognize that the concerned package is not actively utilized in the project. After filtering out such instances, we are left with 983 fixing records that meet two criteria: 1) executed by a human, not the bot, and 2) contain a fixing version in one of their dependency files.

Our findings suggest that 14.60% (220 out of 1507) of records that are manually fixed by developers are likely inspired by Dependabot. Table 7 provides a breakdown of these cases, indicating whether the fix originates from the dependency file, the lock file, or both. The predominant number of inspired entries is addressed via the lock file, indicating a probable transitive fix stemming from adjusting the version of an upstream package. In other words, the developer manually bumps the version of an upstream package, causing the vulnerable transitive dependency to be updated to a non-vulnerable version.

Fix Origin	Inspired Cases	Percentage
Dependency file	45	2.99%
Lock file	203	13.47%
Total	220	14.60%

Table 7: Proportions of manual fixes derived from the dependency file and/or lock file.

Among all the inspired fixes, 6.82% (15 cases) addressed vulnerabilities of critical severity, 45% (99 cases) involved high-severity vulnerabilities, 43.18% (95 cases) were related to moderate vulnerabilities, and 5% addressed low-severity ones. Given that the Mann-Whitney U test shows no significant difference between the distribution of inspired fixes and the others ($p \geq 0.6$), we cannot conclude that vulnerability severity has a significant influence on the likelihood of fixes being inspired. This suggests that developers use security updates as a source of inspiration for manually addressing vulnerabilities, regardless of their severity.

Our manual analysis of the commit messages associated with these fixes, after resolving disagreements, led to a full consensus on the final labels. Out of the 220 records, 5 commits explicitly mentioned Dependabot or referenced a *security update*, 20 commits stated that they were aimed at resolving *security issues*, and 108 commits were identified as specifically focusing on *updating dependencies*. Although these results do not offer strong evidence of a connection to security updates, most commit messages suggest that the primary focus was on updating dependencies. We hypothesize that this motivation to update dependencies likely arises from the security updates they received earlier.

Figure 8 presents the distributions of fixing time for both inspired fixes and non-inspired ones. While there is a noticeable difference in the median time taken by developers to manually address non-inspired cases compared to inspired ones (44 days versus 68 days), we observe no statistical difference between the two distributions according to the Mann-Whitney U test ($p \geq 0.8$). Particularly, when considering fixes executed in less than 1 year, they are more likely to be statistically different from each other ($p \geq 0.21$). Moreover, in both distributions, the 75th percentile is nearly 140 days, indicating that 75% of all fixes are completed within less than 140 days after receiving a security update. Additionally, considering that a sample size of at least 4 is necessary for a valid Mann-Whitney U test (Fay and Proschan, 2010), we conducted a per-repository Mann-Whitney test for repositories containing a minimum of 4 manually fixed vulnerabilities. The findings indicate that in 94.09% of cases (430 out of 457), the fixing time for both inspired and non-inspired cases is derived from the same distribution. In other words, we observe that the time developers invest in manually addressing security issues is not correlated with whether the changes are inspired by the bot or not. Also, this could suggest that other influential factors impact fixing time, such as widespread use of the vulnerable package within the repository or how developers handle potential breaking changes associated with non-vulnerable versions.

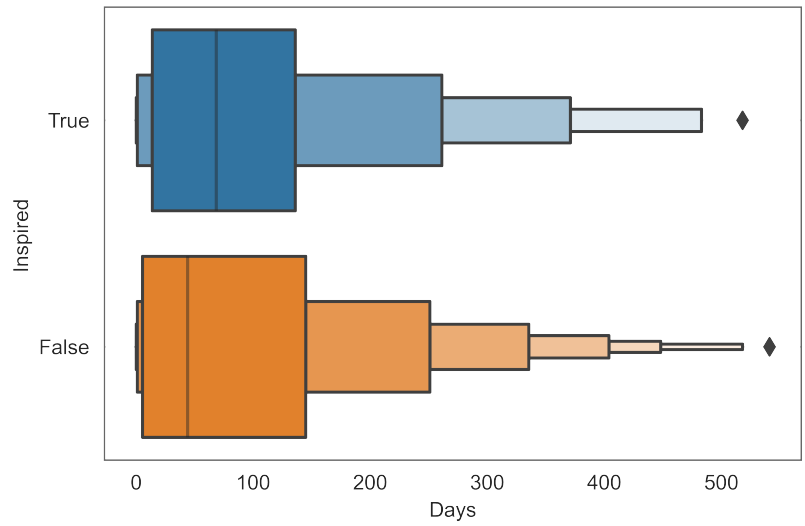


Fig. 8: Boxen plot depicting the distributions of time required to complete fixes inspired by Dependabot and those not inspired.

Overall, our results highlight that, although the 57% merge ratio identified in the results of $\mathbf{RQ_1}$ is notable, this percentage could be even higher if we also account for inspired cases. This means that Dependabot plays a more prominent role in drawing attention to security vulnerabilities and aiding project maintainers in addressing them. While the maintainers sometimes may choose not to merge the security updates, at times, they are still inclined to acknowledge them and resolve the vulnerable packages precisely as recommended by the bot.

6 Discussion

When comparing the merge ratio of security pull requests created by Dependabot to those of Dependabot-preview, as reported by Alfadel et al. (2021), we observe a reduction of 9% in receptivity. We conjecture that this contrast primarily stems from the auto-merge functionality of Dependabot-preview. Moreover, the automatic activation of Dependabot by GitHub, provided that the repository meets certain requirements, could elucidate the reluctance to merge pull requests, as developers did not enable the bot themselves.

To uncover the reasons influencing developers' judgments regarding vulnerability resolution, including decisions to not address vulnerabilities, manual implementation, or rejection of Dependabot proposals, we utilize the expertise of two separate raters. They assess various textual artifacts, such as git commit

messages submitted with the fixing commit, comments accompanying security updates initiated by Dependabot, and other communication texts. Accordingly, each rater assigns concise labels to express the underlying reasons for each case. Each label is accompanied by one to two describing sentences. When assigning a label, we rely solely on the rationale provided by the developers themselves, without speculating on possible events or intents not explicitly specified. Throughout the analysis, we notice that the majority of non-merged security updates are rejected without explanation, whereas vulnerability fixes often come with only a superficial message. During the manual review process, the first author identified 213 explicitly motivated and actionable responses to vulnerability notifications. We organized these into 22 unique reasons, *i.e.*, labels, and further clustered these fine-grained labels into six generalized groups. Additionally, to mitigate subjectivity, the second rater was engaged to independently conduct a separate round of labeling. To assess the agreement between the two raters, we computed Cohen's κ, recognized as the standard measure. Upon comparing the classifications of the two raters, we observed $\kappa = 0.963$, which following the interpretation proposed by Viera and Garrett (2005), corresponds to *perfect agreement*.

We observe that in 31.92% (68) of the cases, the choice not to merge a Dependabot security update and instead address the vulnerability manually is attributed to ***project management peculiarities***. These peculiarities encompass scenarios such as *external management* (50), where the repository serves as a mirror for the project, while development and management occur through a separate third-party platform (*e.g.*, Gerrit Code Review). Another contributing factor within this category of challenges is the *automatic closure* of a security update (11) by another bot. In accordance with previous studies (Haenni et al., 2013; Bogart et al., 2016, 2015), we find that ***compatibility challenges*** constitute a significant concern for developers in 27.70% (59) of cases. Most commonly, developers assert that an update introduces *breaking changes* (40), requiring further modifications to the source code and adjustments to other project dependencies. We also find instances where developers opt to dismiss a security update due to the *absence of tests* (1), indicating uncertainty about breaking changes. At times, developers explicitly request Dependabot to *ignore dependency* entirely (14), while in certain scenarios, they express a desire to upgrade to a more recent release, seeking a *higher version* (3) than suggested by the bot, to leverage increased performance or access new features. ***Dependency usage***, specifically unused dependencies, accounts for 18.31% (39) of the cited reasons. We also observe that developers occasionally exhibit interest in the *manifest updates only* (13), indicating they are solely concerned with updates to direct dependencies and are not focused on transitive dependencies. Additionally, it is noted that in 10.33% (22) of the cases, certain ***bot limitations*** can influence the decision to decline its contributions. For instance, there are restricted configuration settings offered by Dependabot. This issue recurs as a significant hurdle in bot adoption, as indicated by Wessel et al. (2021). At times, developers express that the vulnerable dependency has been *already resolved* (9) on the development branch by the time Dependabot

generates the corresponding security update. In other cases, they indicate that the proposed change was made on the *wrong branch* (4), implying developers would have merged the suggestion if it had been based on their preferred branch. Moreover, we discover the developers expressing ***bot dissatisfaction*** in 9.39% (20) of the cases. Notably, developers have deliberately removed all lock files in some cases (11) to *prevent future security alerts*, and in other scenarios, they have marked security updates as *spam* (5). This suggests that Dependabot generates a certain level of noise, which aligns with the most prevalent and central complication in interacting with software bots as documented in prior studies (Wessel et al., 2018; Wessel and Steinmacher, 2020; Wessel et al., 2021). We also observe several cases where developers explicitly express *reluctance* (4) to utilize Dependabot when it is deployed automatically. Lastly, we identified three reasons that occur too infrequently to constitute a separate category yet are distinct from the others. These reasons are categorized under ***miscellaneous*** (**2.35%**, 5 cases). Specifically, they include instances of developer *confusion* (3), which arises from discrepancies between the title or contents of the security update, resulting in Dependabot's proposition being rejected.

6.1 Implications to Practitioners

The observed distribution of merge ratios indicates that once project maintainers merge at least one security update, they tend to accept all or the vast majority of future Dependabot suggestions. This trend, coupled with our qualitative analysis revealing around 9% dissatisfaction in explicitly motivated rejections, suggests that the benefits of Dependabot outweigh any drawbacks.

Despite the availability of more tests in popular projects (Lin et al., 2021), our findings reveal that there is no significant correlation between a project's popularity and the adoption rate of security updates. This underscores that developers need to remain vigilant on security, even if their project is dependent on popular repositories with a possibly larger community. On the contrary, we conclude that the utilization of tests and CI tools boosts developers' confidence in security updates, potentially mitigating concerns over introducing breaking changes, which in turn results in a higher merge ratio.

The survival analysis conducted in this work provides evidence that manual resolution of vulnerabilities can be a time-consuming process, potentially leaving projects susceptible to security issues for extended periods. We also observe that even if developers choose not to directly merge Dependabot suggestions, their manual fixes may draw inspiration from the security updates, thereby reinforcing Dependabot's role in identifying vulnerabilities. Thus, taking everything into account, we strongly encourage developers to consider experimenting with Dependabot on a trial basis, as the evidence demonstrates its significant benefits in effectively managing security vulnerabilities.

6.2 Implications to Dependabot Maintainers

Considering the largest group of projects, those with two or fewer security updates (approximately 45%), we observe a clear tendency among maintainers to either accept all security updates or reject them entirely. We believe this behavior may stem from confusion or reluctance to merge pull requests from an unfamiliar bot without prior notification, leading maintainers to opt for manual resolution of vulnerabilities instead. This is supported by the fact that the highest percentage of developer fixes occurs within this group of projects. To address this issue, we suggest that GitHub consistently deploys Dependabot with an introductory or welcoming message explaining its purpose.

The challenge of restricted configuration options is a widespread issue in bot adoption, as highlighted in prior research (Wessel et al., 2021), and Dependabot is no exception. One potential setting that could enhance customization to suit user preferences is the ability to limit the number of open security updates, as we observe instances where developers become overwhelmed by them. Alternatively, Dependabot could offer project maintainers the option to prioritize the delivery of security updates based on the severity level of vulnerabilities. This could be particularly beneficial for maintainers, as we observe a strong correlation between the vulnerability survivability and the severity level.

One possible explanation for maintainers' lack of interest in merging security updates is that, as previous studies have indicated (Hejderup et al., 2022; Nielsen et al., 2021), some detected vulnerabilities are not reachable. In other words, the vulnerable functions are not invoked in the source code, rendering the security update ineffective. Thus, we recommend enhancing the analysis performed by Dependabot by scanning the source files to determine whether the identified vulnerable package is actually imported and used in the code. This approach could also help reduce the number of false alarms generated by Dependabot.

6.3 Implications to Researchers

We spot that developers merge a security update generated by Dependabot in 57% of cases. This statistic differs from the findings reported in previous works, such as that of Wyrich *et al.* (Wyrich et al., 2021), who reported that only 37.38% of bot pull requests in their collection were merged. The contrast in results could be attributed to the extensive project filtering conducted in our study. However, we are more inclined to attribute the observed discrepancy primarily to the selection of tools, as in the approach of Wyrich *et al.*, bots issuing pull requests are not distinguished and are examined collectively. This conjecture is further supported by comparing our findings to those reported by Mirhosseini and Parnin (2017) in their study on the usage of Greenkeeper, which is a bot that offers automated pull requests for updating stale dependencies. The authors concentrated on starred and non-forked JavaScript projects with at least 20 commits and discovered that only 32% of pull requests generated

by Greenkeeper were merged, which is 1.8 times lower than the percentage recorded in our study. Despite Greenkeeper and Dependabot being fashioned to update dependencies and employing nearly identical developer interaction mechanisms, *i.e.*, pull requests, their distinct objectives likely influence their receptivity. Hence, when analyzing bot usage among developers, we recommend categorizing them based on the specific goals and tasks each bot is designed to fulfill.

There are alternative dependency management bots, such as Snyk (Snyk Limited, s.d.) and Renovate (Mend, s.d.), assisting developers in managing vulnerable dependencies. Although these bots also possess constraints, certain characteristics and features they offer are supplementary to those of Dependabot. Dependabot's security updates, for example, include a compatibility score, which informs developers about the likelihood of encountering breaking changes when updating a dependency. Consequently, this feature could potentially prompt a greater merge ratio for its security updates. Snyk, on the other hand, furnishes a priority score feature, which aids developers in filtering and prioritizing discovered issues based on their significance, risk level, occurrence frequency, and ease of resolution. Employing multiple bots within a repository for similar purposes, however, could potentially instigate increased noise. As outlined in our paper, noise represents one of the contributing factors to vulnerabilities remaining unaddressed. The utilization of complementary tools might aggravate this noise, consequently prolonging the resolution of vulnerabilities. Nonetheless, additional research is required to investigate the repercussions of employing multiple bots in software repositories.

6.4 Threats to Validity

Below we review the issues that might have threatened the validity of our conclusions following the traditional typology (Shadish et al., 2002). We are aware of the limitations of this conceptualization both within the context of empirical software engineering (Verdecchia et al., 2023) and outside of it (Reichardt, 2011), as well as the recently proposed alternatives such as trade-offs (Robillard et al., 2024).

Construct validity. A threat to construct validity concerns our definition of a security fix made by Dependabot. We employ a high-precision strategy and only consider a fix to be contributed by the bot if Dependabot is the author of the fixing commit. To identify potential fixes inspired by Dependabot, wherein developers replicate the exact update generated by the bot despite the availability of newer versions for the concerned library, we adopt a strict approach to enhance the accuracy of our method. Nonetheless, uncovering such cases with absolute certainty may require additional measures, such as conducting interviews with developers.

Internal validity. The main internal threat pertains to the quality of the obtained collection of projects. The presence of abandoned projects or immutable forks could have downgraded the overall merge ratio or significantly affected

survival curves for vulnerabilities. To address this threat, we applied an extensive filtering procedure and removed projects having less than one commit each month of the collection period. Also, we only considered engineered projects selected using the `Reaper` tool. Despite the fact that this tool may not be perfectly accurate, given the other criteria we impose on the projects, we are confident the presence of personal repositories in the considered sample is minimal.

External validity. This work only focuses on JavaScript projects and the `npm` and `yarn` ecosystems. Therefore, our findings might not apply to other types of projects and ecosystems due to the different policies, practices, and culture in each ecosystem (Decan et al., 2017; Bogart et al., 2016). This is supported by the findings of Zerouali et al. (2022), who found that vulnerabilities in `npm` are fixed sooner compared to `RubyGems`, which is a much small ecosystem compared to `npm`. Moreover, our analysis is strictly limited to Dependabot, whose interaction traits and core logic can be different from the ones of other bots. Further studies are needed to verify our findings for other ecosystems and bots.

7 Related work

Prior research on bots, as one of the most dominant assistant tools in software development, has focused on identifying the challenges in interaction (Wessel et al., 2021, 2018; Wessel and Steinmacher, 2020; Liu et al., 2020), assessing their impact on development artifacts and software quality (Mirhosseini and Parnin, 2017; Wessel et al., 2020; Kavaler et al., 2019), and quantitatively measuring their adoption and developer receptivity towards their assistance (Alfadel et al., 2021; Wyrich et al., 2021; Brown and Parnin, 2019; Mohayeji et al., 2022).

Wyrich et al. (2021) discovered that pull requests from humans are accepted and merged almost twice as often as bot pull requests (72.53% vs. 37.38%), suggesting that such tools may not be fully utilized. Moreover, despite bot pull requests containing fewer changes on average, they take significantly longer to be interacted with and merged. Nonetheless, this analysis is not scoped to a specific bot or the characteristics of the projects employing them. In contrast, our study focuses on GitHub's Dependabot and well-established, actively maintained JavaScript projects, aiming to assess the impact of security updates in safeguarding dependencies. Our findings reveal that the majority of Dependabot suggestions are typically merged, and a notable portion of those handled manually are also influenced by security updates. This highlights that, overall, Dependabot is largely relied upon to maintain the security of dependencies. We also find that bot fixes often require less time than those carried out manually by developers.

Despite the consensus within the research community regarding the complexity and burdensome nature of dependency management faced by developers, only a few studies analyze tools designed to support this task. Mirhosseini and Parnin (2017) explored the functionality of Greenkeeper, a bot that generates pull requests for updating obsolete dependencies in JavaScript projects. Their findings indicate that, on average, projects utilizing this bot upgrade their dependencies 1.6 times more often than those not utilizing it, implying a notable

practicality of such a tool. While the authors note report 32% of automated pull requests in their dataset were merged, it remains unclear whether developers disregarded the updates or executed them manually. Also, the extent to which developers delegate dependency updates to the bot is not studied in this research. Our results, however, indicate that Dependabot pull requests achieve a notably higher merge rate of 54%. To investigate the additional workload caused by dependency management bots, Rombaut *et al.* (Rombaut et al., 2022) analyzed 93k issue reports initiated by Greenkeeper in the npm ecosystem. They found that Greenkeeper imposes a noteworthy overhead on client projects, contributing to half of all reported issues within them. Notably, many of these issues arose due to Greenkeeper taking incorrect action on a dependency update.

The study conducted by Pashchenko et al. (2020) suggests that bots can be merely utilized to detect issues within dependencies, while the update itself is executed by developers. Also, He et al. (2022) highlights Dependabot's effectiveness in reducing technical lag, with developers demonstrating strong acceptance of its pull requests. While this is consistent with our findings, in our research, we advance this understanding by not only examining developers' receptivity to bot suggestions but also quantifying the frequency with which they manually address issues despite automated pull requests being in use. We also investigate the persistence of vulnerabilities, as well as manual fixes that are potentially influenced by the security updates.

The research conducted by Alfadel et al. (2021) is the most pertinent to our study. In their work, the authors investigated developers' receptivity to security pull requests generated by Dependabot-preview, the forerunner to GitHub Dependabot. The findings indicate that the majority of such pull requests are often merged within a day. They also observe that the severity level of the vulnerabilities does not significantly affect their resolution time. This contrasts with our findings, which reveal a negative correlation between vulnerability severity and the time required to address them. Moreover, their results suggest that the risk of breaking changes does not exert a significant influence on the merging of pull requests, in contrast to findings from other studies (Decan et al., 2018b; Haenni et al., 2013; Bogart et al., 2016; Chinthanet et al., 2021; Huang et al., 2019). It is noteworthy to highlight that the primary function of Dependabot-preview, studied by (Alfadel et al., 2021), is providing version updates. It automatically supersedes a security update with a newer one when a more recent release of the vulnerable package becomes available. Crucially, Dependabot-preview offers an *auto-merge* feature, implying that the findings mentioned may not accurately reflect developers' reception of its security pull requests, as numerous merges and rejections are executed automatically. On the contrary, scrutinizing Dependabot enables us to account for these confounding factors since Dependabot consistently suggests the minimum required non-vulnerable version. Besides, dissimilar to related studies, we concentrate our analysis on the vulnerabilities rather than solely on the pull requests. This approach yields a more precise estimation of the time developers require to resolve vulnerabilities. We also examine the correlation between project attributes and the merge ratio of security updates, a type of analysis not explored in previous studies.

Several other studies investigate dependency updates and vulnerabilities arising from dependencies without considering the potential mitigating effects of bots. Wang et al. (2023) recently conducted an empirical study to explore the scope of packages hindering the spread of vulnerability fixes within the npm ecosystem. They found that there are 45k blocking packages and 358k blocking chains in the recent snapshot of the npm ecosystem. They also introduced a technique for devising practical remediation strategies. Likewise, Liu et al. (2022) presented a dependency tree-based vulnerability remediation method for npm packages. They introduced a graph-based dependency resolution technique that resolves the inner dependency relations of dependencies as trees and analyzes transitive vulnerability propagation paths. To evaluate the discoverability of vulnerabilities inherited through dependencies, Alfadel et al. (2023) investigated Node.js applications. Their findings unveiled that a considerable portion of the affected applications depended on undisclosed vulnerable packages, with only a small percentage being exposed to dependencies with publicly known vulnerabilities. They concluded that the primary reason for the prolonged susceptibility of applications to security issues is the absence of dependency updates despite the availability of fixes. In a comparable investigation focusing on the PyPI and Maven ecosystems, Xu et al. (2022) showcased that nearly 83k projects within these ecosystems directly or indirectly rely on vulnerable libraries compiled from C projects. Additionally, they propose an automated tool to identify these vulnerabilities and analyze the vulnerable APIs involved. Pashchenko et al. (2022) presented a methodology for counting *actually vulnerable dependencies* in Java libraries. Their approach tackles complexities such as the inflation of unexploitable vulnerabilities, and vulnerabilities introduced through transitive dependencies. Their proposed method is transferable to other ecosystems, including JavaScript. Jafari et al. (2023) examined the association between npm package characteristics and the chosen dependency update strategy by its dependents to comprehend developers' decision-making and strategy adjustments. They develop a predictive model to discern the common dependency update strategy for each package. Additionally, they observe that factors such as breaking changes can impact the evolution of chosen update strategies as the project evolves.

8 Conclusion

In this study, we conducted comprehensive research to assess the usage of Dependabot and its efficacy in safeguarding the security of dependencies within JavaScript projects hosted on GitHub. In particular, we analyzed 4,195 security updates associated with 978 engineered and actively maintained JavaScript projects. We studied developers' receptivity towards Dependabot security updates and the correlation between the attributes of projects and the merge ratio of security updates. Moreover, we scrutinized the practice of fixing the vulnerability manually in the presence of an automated pull request, assessed how proactive developers are in solving the vulnerable dependencies alerted by Dependabot, and traced manual fixes potentially inspired by the bot.

The findings manifest that more than half of the bot suggestions are merged. We observed that, on average, the merge ratio is higher for repositories that adopt both testing frameworks and CI tools. Moreover, we concluded that when developers do not accept a security update, they often address the associated vulnerability manually. At times, these manual fixes are possibly inspired by a preceding security update, underscoring the effectiveness of Dependabot in aiding project maintainers with managing vulnerable dependencies. By levering a survival analysis, we found that developers are often proactive in addressing vulnerable dependencies identified by Dependabot and prioritize fixes based on the severity level of vulnerabilities. Nevertheless, if developers opt for manual implementation of fixes, the process might extend significantly.

We call for the continuation of our work by further analyzing the impact of Dependabot on keeping the project dependencies secure. One possible research avenue involves inspecting the actual usage of vulnerable APIs within the code repository. This is crucial because receiving a security update or having a vulnerable dependency does not automatically imply that the repository is vulnerable. Examining these cases can also assist Dependabot in creating fewer security updates and reducing the noise level.

9 Data Availability

Following Open Science policies, all scripts and data utilized to produce the results of this research are publicly available [4].

References

Aalen O, Borgan O, Gjessing H (2008) Survival and event history analysis: a process point of view. Springer Science & Business Media

Abrahamsson P, Salo O, Ronkainen J, Warsta J (2017) Agile software development methods: Review and analysis. arXiv preprint arXiv:170908439

Alfadel M, Costa DE, Mokhallalati M, Shihab E, Adams B (2020) On the Threat of npm Vulnerable Dependencies in Node.js Applications. arXiv preprint arXiv:200909019

Alfadel M, Costa DE, Shihab E, Mkhallalati M (2021) On the use of dependabot security pull requests. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pp 254–265, DOI 10.1109/MSR52588.2021.00037

Alfadel M, Costa DE, Shihab E, Adams B (2023) On the discoverability of npm vulnerabilities in node.js projects. ACM Trans Softw Eng Methodol 32(4), DOI 10.1145/3571848, URL https://doi.org/10.1145/3571848

APA (1994) Publication manual of the American Psychological Association, 4th edn, American Psychological Association, pp 16–18

[4] https://github.com/piwvh/dependabot-emse

Armstrong RA (2014) When to use the Bonferroni correction. Ophthalmic and Physiological Optics 34(5):502–508

Backes M, Bugiel S, Derr E (2016) Reliable third-party library detection in android and its security applications. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp 356–367

Basili VR, Briand LC, Melo WL (1996) How reuse influences productivity in object-oriented systems. Commun ACM 39(10):104–116, DOI 10.1145/236156.236184, URL https://doi.org/10.1145/236156.236184

Benjamini Y, Hochberg Y (1995) Controlling the false discovery rate: a practical and powerful approach to multiple testing. Journal of the Royal statistical society: series B (Methodological) 57(1):289–300

Bogart C, Kästner C, Herbsleb J (2015) When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), IEEE, pp 86–89

Bogart C, Kästner C, Herbsleb J, Thung F (2016) How to break an API: cost negotiation and community values in three software ecosystems. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp 109–120

Brito G, Terra R, Valente MT (2018) Monorepos: a multivocal literature review. arXiv preprint arXiv:181009477

Brown C, Parnin C (2019) Sorry to bother you: designing bots for effective recommendations. In: 2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE), IEEE, pp 54–58

Burrows D, Fernandez Montecelo MA (2016) What is package manager? In: aptitude user's manual, Free Software Foundation, https://www.debian.org/doc/manuals/aptitude/pr01s02.en.html

Chinthanet B, Kula RG, McIntosh S, Ishio T, Ihara A, Matsumoto K (2021) Lags in the release, adoption, and propagation of npm vulnerability fixes. Empirical Software Engineering 26(3):1–28

Cohen J (1988) Statistical power analysis for the behavioral sciences, 2nd edn, L. Erlbaum Associates, pp 224–226

Constantinou E, Mens T (2017) An empirical comparison of developer retention in the rubygems and npm software ecosystems. Innovations in Systems and Software Engineering 13(2):101–115

Cox J, Bouwers E, Van Eekelen M, Visser J (2015) Measuring dependency freshness in software systems. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, IEEE, vol 2, pp 109–118

Cramér H (1946) Mathematical methods of statistics, Princeton University Press, chap 21, p 282

Decan A, Mens T, Claes M (2017) An empirical comparison of dependency issues in oss packaging ecosystems. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 2–12

Decan A, Mens T, Constantinou E (2018a) On the evolution of technical lag in the npm package dependency network. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 404–414

Decan A, Mens T, Constantinou E (2018b) On the impact of security vulnerabilities in the npm package dependency network. In: Proceedings of the 15th International Conference on Mining Software Repositories, pp 181–191

Decan A, Mens T, Grosjean P (2019) An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Software Engineering 24(1):381–416

Delčev S, Drašković D (2018) Modern javascript frameworks: A survey study. In: 2018 Zooming Innovation in Consumer Technologies Conference (ZINC), pp 106–109, DOI 10.1109/ZINC.2018.8448444

Dependabot (2020) chore(deps): bump acorn from 5.6.2 to 5.7.4. pull request #1008 carbon-design-system/carbon-addons-iot-react. `https://github.com/carbon-design-system/carbon-addons-iot-react/pull/1008`, [Online] Last accessed January 11 2024

Dependabot (s.d.) Dependabot-preview. `https://github.com/marketplace/dependabot-preview/`, [Online] Last accessed 21 March 2021

Depfu (2020) Depfu. `https://depfu.com/`, [Online] Last accessed 18 October 2024

Derr E, Bugiel S, Fahl S, Acar Y, Backes M (2017) Keep me updated: An empirical study of third-party library updatability on android. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp 2187–2200

Di Cosmo R, Zacchiroli S (2017) Software heritage: Why and how to preserve software source code. In: iPRES 2017-14th International Conference on Digital Preservation, pp 1–10

Erlenhov L, de Oliveira Neto FG, Leitner P (2022) Dependency management bots in open-source systems—prevalence and adoption. PeerJ Computer Science 8:e849

Fay MP, Proschan MA (2010) Wilcoxon-mann-whitney or t-test? on assumptions for hypothesis tests and multiple interpretations of decision rules. Statistics surveys 4:1

Fleming TR, Harrington DP (2011) Counting processes and survival analysis, vol 169. John Wiley & Sons

Frakes WB, Kang K (2005) Software reuse research: Status and future. IEEE Transactions on Software Engineering 31(7):529–536

Gebauer J (2015) Pyup. `https://pyup.io/`, [Online] Last accessed 18 January 2023

GitHub (2020) Securing the world's software: The 2020 State of the Octoverse. `https://octoverse.github.com/static/github-octoverse-2020-security-report.pdf`, [Online] Last accessed 18 January 2023

GitHub (s.d.) About Dependabot security updates. `https://docs.github.com/en/code-security/dependabot/dependabot-security-updates/about-dependabot-security-updates`, [Online] Last accessed 11 January 2024

Golzadeh M, Decan A, Mens T (2022) On the rise and fall of ci services in github. In: 2022 IEEE International Conference on Software Analysis,

Evolution and Reengineering (SANER), IEEE, pp 662–672
Haenni N, Lungu M, Schwarz N, Nierstrasz O (2013) Categorizing developer information needs in software ecosystems. In: Proceedings of the 2013 international workshop on ecosystem architectures, pp 1–5
He R, He H, Zhang Y, Zhou M (2022) Automating dependency updates in practice: An exploratory study on github dependabot. DOI 10.48550/ARXIV.2206.07230, URL https://arxiv.org/abs/2206.07230
Healey JF (2009) Statistics: A Tool for Social Research, 8th edn, Wadsworth Cengage Learning, pp 316–317
Heike Hofmann HW, Kafadar K (2017) Letter-value plots: Boxplots for large data. Journal of Computational and Graphical Statistics 26(3):469–477, DOI 10.1080/10618600.2017.1305277, URL https://doi.org/10.1080/10618600.2017.1305277, https://doi.org/10.1080/10618600.2017.1305277
Hejderup J, Beller M, Triantafyllou K, Gousios G (2022) Präzi: from package-based to call-based dependency networks. Empirical Software Engineering 27(5):102
Howell DC (2007) Statistical methods for psychology, 5th edn, Wadsworth, p 165
Huang J, Borges N, Bugiel S, Backes M (2019) Up-to-crash: Evaluating third-party library updatability on android. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P), IEEE, pp 15–30
Hutchings J (2019) Introducing new ways to keep your code secure. https://github.blog/2019-05-23-introducing-new-ways-to-keep-your-code-secure/, [Online] Last accessed 11 January 2024
Irshad M, Torkar R, Petersen K, Afzal W (2016) Capturing cost avoidance through reuse: systematic literature review and industrial evaluation. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, Association for Computing Machinery, New York, NY, USA, EASE '16, DOI 10.1145/2915970.2915989, URL https://doi.org/10.1145/2915970.2915989
Jafari AJ, Costa DE, Shihab E, Abdalkareem R (2023) Dependency update strategies and package characteristics. ACM Transactions on Software Engineering and Methodology 32:1 – 29, URL https://api.semanticscholar.org/CorpusID:258887857
Joy A, Thangavelu S, Jyotishi A (2018) Performance of github open-source software project: an empirical analysis. In: 2018 Second International Conference on Advances in Electronics, Computers and Communications (ICAECC), IEEE, pp 1–6
Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining github. In: Proceedings of the 11th working conference on mining software repositories, pp 92–101
Kaplan EL, Meier P (1958) Nonparametric estimation from incomplete observations. Journal of the American Statistical Association 53(282):457–481
Kavaler D, Trockman A, Vasilescu B, Filkov V (2019) Tool choice matters: Javascript quality assurance tools and usage outcomes in github projects.

In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pp 476–487

Kokoska S, Zwillinger D (2000) CRC standard probability and statistics tables and formulae. Crc Press, Section 2.2.24

Krüger J, Berger T (2020) An empirical analysis of the costs of clone- and platform-oriented software reuse. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2020, p 432–444, DOI 10.1145/3368089.3409684, URL https://doi.org/10.1145/3368089.3409684

Kula RG, German DM, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? Empirical Software Engineering 23(1):384–417

Lanza M, Marinescu R (2007) Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer Science & Business Media

Lauinger T, Chaabane A, Arshad S, Robertson W, Wilson C, Kirda E (2018) Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. arXiv preprint arXiv:181100918

Lim W (1994) Effects of reuse on quality, productivity, and economics. IEEE Software 11(5):23–30, DOI 10.1109/52.311048

Lin B, Robles G, Serebrenik A (2017) Developer turnover in global, industrial open source projects: Insights from applying survival analysis. In: 2017 IEEE 12th International Conference on Global Software Engineering (ICGSE), IEEE, pp 66–75

Lin JW, Salehnamadi N, Malek S (2021) Test automation in open-source android apps: A large-scale empirical study. Association for Computing Machinery, New York, NY, USA, ASE '20, p 1078–1089, DOI 10.1145/3324884.3416623, URL https://doi.org/10.1145/3324884.3416623

Liu C, Chen S, Fan L, Chen B, Liu Y, Peng X (2022) Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE) pp 672–684, URL https://api.semanticscholar.org/CorpusID:245853604

Liu D, Smith MJ, Veeramachaneni K (2020) Understanding user-bot interactions for small-scale automation in open-source development. In: Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems, pp 1–8

McDonald M (2021) Goodbye Dependabot preview, hello Dependabot! GitHub blog, https://github.blog/2021-04-29-goodbye-dependabot-preview-hello-dependabot/, [Online] Last accessed 11 January 2024

McHugh ML (2013) The chi-square test of independence. Biochemia medica 23(2):143–149

Mend (s.d.) Renovate. https://github.com/marketplace/renovate/, [Online] Last accessed 11 January 2024

Mirhosseini S, Parnin C (2017) Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 84–94

Mittelhammer RC, Judge GG, Miller DJ (2000) Econometric foundations. Cambridge University Press

Mohagheghi P, Conradi R, Killi O, Schwarz H (2004) An empirical study of software reuse vs. defect-density and stability. In: Proceedings. 26th International Conference on Software Engineering, pp 282–291, DOI 10.1109/ICSE.2004.1317450

Mohayeji H, Ebert F, Arts E, Constantinou E, Serebrenik A (2022) On the adoption of a todo bot on github: A preliminary study. In: 2022 IEEE/ACM 4th International Workshop on Bots in Software Engineering (BotSE), IEEE Computer Society, Los Alamitos, CA, USA, pp 23–27, DOI 10.1145/3528228.3528408, URL https://doi.ieeecomputersociety.org/10.1145/3528228.3528408

Mohayeji H, Agaronian A, Constantinou E, Zannone N, Serebrenik A (2023) Investigating the resolution of vulnerable dependencies with dependabot security updates. In: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), pp 234–246, DOI 10.1109/MSR59073.2023.00042

Munaiah N, Kroh S, Cabrey C, Nagappan M (2017) Curating github for engineered software projects. Empirical Software Engineering 22(6):3219–3253

Nachar N, et al. (2008) The mann-whitney u: A test for assessing whether two independent samples come from the same distribution. Tutorials in quantitative Methods for Psychology 4(1):13–20

Neighbourhoodie Software (2020) Greenkeeper automated dependency management. https://greenkeeper.io/, [Online] Last accessed 11 January 2024

Nielsen BB, Torp MT, Møller A (2021) Modular call graph construction for security scanning of node. js applications. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp 29–41

Pashchenko I, Vu DL, Massacci F (2020) A qualitative study of dependency management and its security implications. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp 1513–1531

Pashchenko I, Plate H, Ponta SE, Sabetta A, Massacci F (2022) Vuln4real: A methodology for counting actually vulnerable dependencies. IEEE Transactions on Software Engineering 48(5):1592–1609, DOI 10.1109/TSE.2020.3025443

Pearson K (1900) On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 50(302):157–175

Prana GAA, Sharma A, Shar LK, Foo D, Santosa AE, Sharma A, Lo D (2021) Out of sight, out of mind? how vulnerable dependencies affect open-source

projects. Empirical Software Engineering 26(4):1–34
Reichardt CS (2011) Criticisms of and an alternative to the shadish, cook, and campbell validity typology. New Directions for Evaluation 130:43–53, DOI https://doi.org/10.1002/ev.364, URL https://onlinelibrary.wiley.com/doi/abs/10.1002/ev.364
Robillard MP, Arya DM, Ernst NA, Guo JL, Lamothe M, Nassif M, Novielli N, Serebrenik A, Steinmacher I, Stol KJ (2024) Communicating study design trade-offs in software engineering. ACM Transactions on Software Engineering and Methodology
Rombaut B, Côgo FR, Adams B, Hassan A (2022) There's no such thing as a free lunch: Lessons learned from exploring the overhead introduced by the greenkeeper dependency bot in npm. ACM Transactions on Software Engineering and Methodology 32:1 – 40, URL https://api.semanticscholar.org/CorpusID:248435571
Samoladas I, Angelis L, Stamelos I (2010) Survival analysis on the duration of open source projects. Information and Software Technology 52(9):902–922
Shadish WR, Cook TD, Campbell DT (2002) Experimental and quasi-experimental designs for generalized causal inference. Houghton, Mifflin and Company
Snyk Limited (s.d.) Snyk. https://github.com/marketplace/snyk/, [Online] Last accessed 11 January 2024
Thompson HH (2003) Why security testing is hard. IEEE Security & Privacy 1(4):83–86
Turhan NS (2020) Karl pearson's chi-square tests. Educational Research and Reviews 16(9):575–580
Verdecchia R, Engström E, Lago P, Runeson P, Song Q (2023) Threats to validity in software engineering research: A critical reflection. Inf Softw Technol 164:107329, DOI 10.1016/J.INFSOF.2023.107329, URL https://doi.org/10.1016/j.infsof.2023.107329
Viera AJ, Garrett JM (2005) Understanding interobserver agreement: the kappa statistic. Family medicine 37(5):360–363
Wang Y, Chen B, Huang K, Shi B, Xu C, Peng X, Wu Y, Liu Y (2020) An empirical study of usages, updates and risks of third-party libraries in java projects. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 35–45
Wang Y, Sun P, Pei L, Yu Y, Xu C, Cheung SC, Yu H, Zhu Z (2023) Plumber: Boosting the propagation of vulnerability fixes in the npm ecosystem. IEEE Transactions on Software Engineering 49:3155–3181, URL https://api.semanticscholar.org/CorpusID:256710267
Wessel M, Steinmacher I (2020) The inconvenient side of software bots on pull requests. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, pp 51–55
Wessel M, De Souza BM, Steinmacher I, Wiese IS, Polato I, Chaves AP, Gerosa MA (2018) The power of bots: Characterizing and understanding bots in oss projects. Proceedings of the ACM on Human-Computer Interaction 2(CSCW):1–19

Wessel M, Serebrenik A, Wiese I, Steinmacher I, Gerosa MA (2020) Effects of Adopting Code Review Bots on Pull Requests to OSS Projects. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 1–11

Wessel M, Wiese I, Steinmacher I, Gerosa MA (2021) Don't disturb me: Challenges of interacting with softwarebots on open source software projects. Proc ACM Hum-Comput Interact 5(CSCW2), DOI 10.1145/3476042, URL https://doi.org/10.1145/3476042

Wyrich M, Ghit R, Haller T, Müller C (2021) Bots don't mind waiting, do they? comparing the interaction with automatically and manually created pull requests. In: 2021 IEEE/ACM Third International Workshop on Bots in Software Engineering (BotSE), pp 6–10, DOI 10.1109/BotSE52550.2021.00009

Xu M, Wang Y, Cheung SC, Yu H, Zhu Z (2022) Insight: Exploring cross-ecosystem vulnerability impacts. Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering URL https://api.semanticscholar.org/CorpusID:255441443

Zar JH (2005) Spearman rank correlation. Encyclopedia of Biostatistics 7

Zerouali A, Constantinou E, Mens T, Robles G, González-Barahona J (2018) An empirical analysis of technical lag in npm package dependencies. In: International Conference on Software Reuse, Springer, pp 95–110

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

Zerouali A, Mens T, Decan A, Roover CD (2022) On the impact of security vulnerabilities in the npm and RubyGems dependency networks. Empir Softw Eng 27(5):107