Empirical Software Engineering (2024) 29:128
https://doi.org/10.1007/s10664-024-10523-y

Dependabot and security pull requests: large empirical study

Hocine Rebatchi[1] · **Tégawendé F. Bissyandé**[2] · **Naouel Moha**[1]

Accepted: 2 July 2024 / Published online: 30 July 2024

Abstract

Modern software development is a complex engineering process where developer code cohabits with an increasingly larger number of external open-source components. Even though these components facilitate sharing and reusing code along with other benefits related to maintenance and code quality, they are often the seeds of vulnerabilities in the software supply chain leading to attacks with severe consequences. Indeed, one common strategy used to conduct attacks is to exploit or inject other security flaws in new versions of dependency packages. It is thus important to keep dependencies updated in a software development project. Unfortunately, several prior studies have highlighted that, to a large extent, developers struggle to keep track of the dependency package updates, and do not quickly incorporate security patches. Therefore, automated dependency-update bots have been proposed to mitigate the impact and the emergence of vulnerabilities in open-source projects. In our study, we focus on Dependabot, a dependency management bot that has gained popularity on GitHub recently. It allows developers to keep a lookout on project dependencies and reduce the effort of monitoring the safety of the software supply chain. We performed a large empirical study on dependency updates and security pull requests to understand: (1) the degree and reasons of Dependabot's popularity; (2) the patterns of developers' practices and techniques to deal with vulnerabilities in dependencies; (3) the management of security pull requests (PRs), the threat lifetime, and the fix delay; and (4) the factors that significantly correlate with the acceptance of security PRs and fast merges. To that end, we collected a dataset of 9,916,318 pull request-related issues made in 1,743,035 projects on GitHub for more than 10 different programming languages. In addition to the comprehensive quantitative analysis, we performed a manual qualitative analysis on a representative sample of the dataset, and we substantiated our findings by sending a survey to developers that use dependency management tools. Our study shows that Dependabot dominates more than 65% of dependency management activity, mainly due to its efficiency, accessibility, adaptivity, and availability of support. We also found that developers handle dependency vulnerabilities differently, but mainly rely on the automation of PRs generation to upgrade vulnerable dependencies. Interestingly, Dependabot's and developers' security PRs are highly accepted, and the automation allows to accelerate their management, so that fixes are applied in less than one day. However, the threat of dependency vulnerabilities remains hidden for 512 days on average, and patches are disclosed after 362 days due to the reliance on the manual effort of security experts. Also, project characteristics, the amount of PR changes, as well as developer and

Communicated by: Igor Steinmacher

Extended author information available on the last page of the article

dependency features seem to be highly correlated with the acceptance and fast merges of security PRs.

Keywords Dependabot · Dependency · Software vulnerability · Software supply chain · GitHub · Pull request

1 Introduction

Our society is becoming more and more dependent on information systems that affect all aspects of human life. The quality of these systems is fundamental to ensure security, reliability, and trust. These systems are developed using a multitude of external packages or third-party libraries. For instance, according to a security researcher from GitHub, 85% to 97% of enterprise software code base comes from open-source components (Kaczorowski 2020). Also, these components have transitive dependencies between them, for example, *npm* has over a million published packages (Mujahid et al. 2023) with an average of 90 direct and indirect dependencies for each package (Garrett et al. 2019). Hence, increasing the influence that propagates from a package to its dependents. Developers tend to trust the authenticity and integrity of third-party packages hosted on commonly used repositories (Hou and Jansen 2023). However, attacks can be conducted by exploiting package updates to compromise dependent systems. These attacks are known as *Supply Chain Attacks*, which is a growing concern in the industry (Andreoli et al. 2023). They often have access to powerful capabilities at the operating system level to create serious vulnerabilities. Additionally, the major challenge lies in the fact that the manual review of such type of vulnerabilities is not as obvious, which amplifies their consequences. For example, the *SolarWinds* attack that was conducted in December 2020 has put the spotlight on these types of attacks, where a group of hackers had breached the company 6 months before planting a malicious code in the software updates of a network monitoring tool (called *Orion*). This incident was then discovered 10 months after the attack was triggered, affecting 18,000 users and at least 9 US federal agencies (Boehm 2023). Also, very recently, a new type of vulnerabilities has emerged under a concept known as *dependency confusion* (Birsan 2021), where over 35 tech companies were in risk of having their systems breached by reverting the use of internal private packages to external packages uploaded on public registries. The research community had taken the lead to raise awareness about these vulnerabilities and propose some security measures to deal with them. Several state-of-the-art works have been proposed to analyze package managers (e.g., *npm, PyPI, RubyGems*, etc.) using heuristics (Duan et al. 2020), unsupervised learning (Garrett et al. 2019; Ohm et al. 2021), supervised learning as well as word embedding techniques (Lin et al. 2017a) to detect and identify vulnerable and potentially malicious package versions. These studies are mainly based on static, dynamic or metadata analysis. Other interesting tools (e.g., Dependabot) have been used for monitoring dependencies and fixing vulnerable versions of packages. However, to the best of our knowledge, no study in the literature investigates the dominance and the added value of using bots and *Software Composition Analysis* tools in handling one the most common strategies to conduct *Software Supply Chain Attacks*, that is by leveraging vulnerabilities in dependencies. Furthermore, few studies examine the countermeasures that are adopted by developers to handle security vulnerabilities in dependencies across different ecosystems. Besides, uncovering the extent to which these approaches are adopted allows to understand what developers seek when dealing

with these vulnerabilities, and it highlights the deficiencies that developers and researchers should address with new approaches.

In our study, we try to shed light on the appropriateness and the limits of the current tools and security measures related to dependency updates and the management of security vulnerabilities in GitHub that lead to threatening the software supply chain. We also attempt to identify the factors and the features that motivate the adoption of such tools. In addition, our study aims to provide a better understanding of the practices used by developers and security experts with regards to mitigating the threat of security vulnerabilities, as well as discovering the dominance and lifetime of these vulnerabilities in dependencies. As such, we sought to unveil the subsequent research problems: the level of popularity of the tool Dependabot, and the reasons behind its wide adoption (***RQ1***); the strategies that are commonly used by developers to handle security vulnerabilities in dependencies (***RQ2***); the intent and time to handle security pull requests, the threat lifetime and fix delay (***RQ3***); and the factors that can potentially act on the decision and time to merge security pull requests (***RQ4***).

1.1 Contributions

In this regard, the major contributions of this paper are summarized as follows:

- To the best of our knowledge, we collect the largest dataset [1] of 9,916,318 pull request-related issues in 1,743,035 GitHub repositories for more than 10 different programming languages.
- We create a pipeline [2] that allows to efficiently collect issues from GitHub based on a search query, and then extract the repositories, pull requests, commits and users related to them.
- A large-scale and lightweight comparative analysis between GitHub bots for dependency update and management with prevalence factors.
- A manual qualitative analysis of pull request commits, patches, and comments to extract the patterns of developers' practices to *identify* and *fix* security vulnerabilities in dependencies.
- A quantitative and feature analysis on the management of security vulnerabilities in dependencies, the vulnerability lifetime as well as the fix delay.

1.2 Summary of Key Takeaways

Following is a summary of the key takeaways of our work:

❶ Bots have enabled an improvement in the monitoring of outdated dependencies, alleviating the difficulty of handling them manually. With Dependabot being the most popular tool that dominates 65% of the dependency management activity, its wide adoption is attributed to its efficiency, accessibility, adaptivity, usability and comprehensibility.

❷ When it comes to handling security vulnerabilities in dependencies, developers rely on different strategies to identify and fix them. But, *Software Composition Analysis* tools along with the automated tasks grant improvements in terms of time, performance, and inspection coverage. However, developers may alternatively rely on some substandard resorts such as dependency downgrades and removal, especially when patches are not yet released.

[1] https://doi.org/10.5281/zenodo.7801356

[2] https://github.com/HocineREBT/GitHub-Miner

❸ The pull-based development model allows significant contributions to identify and fix security vulnerabilities. Indeed, our study shows that developers are highly receptive to manual security PRs, and those created by Dependabot. However, a high percentage of Dependabot security PRs are auto-closed. This is mainly due to the fact that these PRs are left open for too long until new versions supporting the fixes are published. Regarding the responsiveness to security PRs, developers merge their PRs within few hours, and merge Dependabot's in less than 48 hours. But, interestingly, the auto-merge function allows to accelerate the merge into few minutes.

❹ Our work also shows that Dependabot enables quick reaction to vulnerable dependencies after disclosing them in vulnerability databases. However, threat remains unknown in GitHub for 512 days, and patches are disclosed after 362 days from 0-day. Thus, leading to a huge window of exposure, and this is mainly due to the reliance on the manual effort of security experts. But also, some developers tend to use vulnerable dependencies even after their disclosure. We also found that most fixes are performed on patch level, and vulnerabilities with serious severity levels are the most occurring on GitHub. With *Prototype Pollution* being the most exploited vulnerability, *npm* ecosystem (representing *JavaScript* and *TypeScript*) has the most reported vulnerabilities in dependencies, making it a primary target for threat actors.

❺ With regard to merge decision and merge rate, we have found that several factors can potentially act on developers' latency and reaction to security fixes. These factors cover project characteristics, the effort induced by refactoring changes, developer's experience and workload, as well as the severity level of the vulnerability.

The reminder of this work is organized as follows: Section 2 presents required background information. The pipeline used for the dataset collection as well as the description of the gathered datasets are detailed in Section 3. In Section 4, we describe our study design and the results are presented and discussed in Section 5. Section 6 introduces implications that are implied by our findings for the different practitioners and stakeholders in the context of software development. We provide related works in the context of our study in Section 7, then, we follow by the threats that may affect the validity of our study in Section 8. Section 9 concludes the paper and summarizes perspectives for future work.

2 Background

By allowing different third-party members to contribute in the process of the software development and through different stages in the supply chain, it becomes more crucial to introduce new ways that facilitate the tasks that were manually performed (program repair, code review, version update, etc.). Dependabot is one of the most active and popular tools that developers rely on to avoid known security vulnerabilities in their project dependencies.

2.1 Known Vulnerabilities

In the security community, threats that are already identified and reported are consistently documented in a standardized manner. This allows developers and users to remain informed and aware of the security measures and vulnerabilities that affect open-source software. In fact, 85% of vulnerabilities in open source are disclosed with a patch already available (Kaczorowski 2020). The most standard method that is commonly used is through Common Vulnerabilities and Exposures (CVEs), for which many databases (e.g. GitHub Advisory

Database [3], National Vulnerability Database [4], etc.) were created to reference all the information related to computer security flaws. For instance, they provide the type of the vulnerability, its severity level, as well as the patch that allows to correct and remove the threat.

2.2 Dependabot

2.2.1 Description

Dependabot is an automated tool that keeps dependencies secure and up-to-date by managing dependency updates through dependency graph, scanning third party vulnerabilities and sending security alerts. It was released on May 27, 2017, and then, it got acquired by GitHub on May, 2019. During the time of release, Dependabot supported only few programming languages (Ruby, JavaScript), and after multiple upgrades, it currently supports 15 programming languages in total.

2.2.2 Overview of the Working Process

Dependabot's architecture comprises multiple components to handle its core logic as described below. A high-level overview of the working process is illustrated in Fig. 1.

1. **Fetching dependency files**: Given a GitHub or GitLab project as input, this component allows to look for the relevant dependency files that contain a list of all the packages that the project depends on (e.g. `package.json`, `gemfile.lock`, etc.) supporting different programming languages.
2. **Extracting dependencies**: For each dependency file that was previously identified, Dependabot extracts the list of dependencies with their name, version, and the requirements that describe the format of the accepted versions (e.g., `">= 2.8.4"` for the version `2.8.4` or higher).
3. **Checking updates**: Based on the list of dependencies, this component checks whether a given dependency is up-to-date in the common repositories (e.g., npm, PyPI, RubyGems, etc.) as well as the GitHub Advisory Database in case of presence of security vulnerabilities. If the dependency is outdated, this component provides additional information about the newer version.
4. **Updating dependencies**: Dependabot creates PRs to update the outdated dependency to a newer version. Then, the user can review, test, and merge the pull request. It is worth noting that Dependabot could be configured with the *"auto-merge"* function that allows it to automatically merge the changes in the PR. However, this possibility is no longer supported after the migration from Dependabot-preview onto the GitHub-native Dependabot by August 3rd, 2021. This functionality created a debate in the development community, since it can potentially be exploited to propagate malicious packages. Also, some developers believe that they ought to retain control of merging any PRs into their projects and verify the dependencies before merging them.

2.2.3 Usage Example

As part of GitHub repository configuration, Dependabot can be used and setup in an easy and straightforward manner. For each repository that needs to be scanned by Dependabot,

[3] https://github.com/advisories

[4] https://nvd.nist.gov/

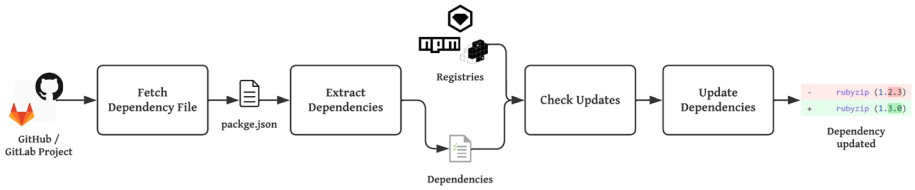

Fig. 1 Overview of Dependabot working process (https://github.com/dependabot/dependabot-core)

developers need to add a configuration file (`dependabot.yml`) where they specify the package manager ecosystem (e.g., npm, maven, etc.), the location of the manifest file that contains the list of the dependencies, and the schedule that defines how often Dependabot should analyze outdated versions (daily, weekly or monthly basis). Note that other specifications can be added to the configuration file for the purpose of managing the dependency update and the users assignment. After that, developers can view the dependency graph that summarizes the direct and transitive dependencies that are referenced in their project. They can also activate Dependabot alerts that send notifications whenever a new version is available or a vulnerability is detected. Figure 2 shows an example of a pull request that was created by Dependabot to update a vulnerable dependency with more information provided by GitHub Advisory Database (GAD).

3 Data Collection

The dataset used in our study consists of a collection of GitHub issues and the associated pull requests, users, repositories and commits. We leveraged the GitHub Search API to create a component that we called GitHub- Miner to download the dataset. Note that we intend to make both the data collection pipeline [5] and the generated dataset [6] publicly available to support reproducibility and future research. Figure 3 illustrates a high-level overview of the data collection pipeline for the GitHub- Miner component.

3.1 Data Collection Steps

The pipeline used to fetch, download and gather the data comprises the following steps:

1. **Search Sampling:** given a search query (combination of search qualifiers and query parameters from GitHub Search API) as input, this step allows us to automatically select an interval for the creation dates of the issues related to the pull requests that are sought in the search query. The main purpose of this component is to download each pull request without losing data due to the API rate limit. It is worth noting that, according to GitHub documentation (GitHub 2021), GitHub search API provides only 1000 search results at a time in a query response. When the search query provides more than 1000 results, we rely on a workaround that consists of using the same search query but during different

[5] https://doi.org/10.5281/zenodo.7801356

[6] https://github.com/HocineREBT/GitHub-Miner

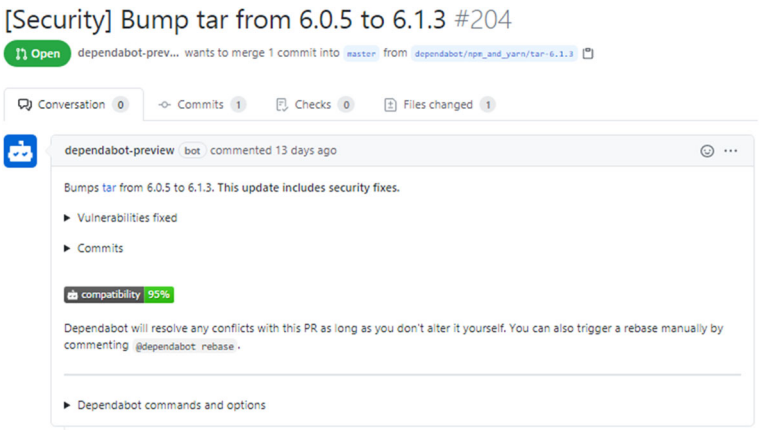

Fig. 2 Example of Dependabot security pull request

time intervals (the time refers to the creation date). In other words, we try to decompose the collection of the entire Dataset into batches/splits, where each batch contain at most 1000 issues that can be retrieved from the search API. We used unauthenticated requests that grant only 10 requests per minute to run this step simultaneously on multiple hosts, and the output of this component is a collection of entries that specify the expected search results *per day(s)* for the given search query.

2. **Outlier Filtering:** this component allows us to perform a filter to separate the entries that contain more than 1000 search results for a time period, so that we can refine and adjust the search query on these entries and narrow the search scope on a smaller interval. The output of this component consists of two collections: The first collection contains all eligible entries with at most 1000 search results, and that is stored and used in step (4). The second collection contains entries with more than 1000 search results (we refer to them as outliers) which require further adjustment in step (3).
3. **Query Adjustment:** this step adjusts the search query to automatically truncate the filtered results from the previous step (having more than 1000 results) into smaller windows of date-time format with a time interval fixed by the user (note that in our study, this value ranges from 2 minutes to 12 hours depending on the results of the previous steps and depending on the dataset). After this step, it is also optional to rerun step (2) to ensure that the entries of the new fetched dates do not contain more than 1000 search results.

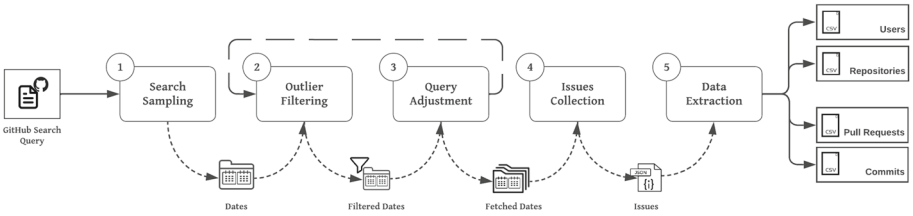

Fig. 3 Data collection pipeline for GitHub- Miner

4. **Issues Collection:** in this step, issues related to pull requests are mined and downloaded by splits from GitHub using the pagination dates that were already filtered and fetched in the previous steps.
5. **Data Extraction:** as a final step, the issues that were previously collected contain all the references that we use to retrieve the data that is related to: the pull requests, the repositories where the PRs were created, the users who create and manipulate these PRs, and the commits that allow the users to apply the changes. Each category is stored in a separate CSV file.

3.2 Dataset Description

In our study, we have considered multiple objectives to provide more insights about dependency updates and security vulnerabilities, and each concern requires a dedicated dataset. Hence, we used the GitHub- Miner to collect four separate datasets to address different aspects of our research questions.

It is worth noting that in our process of selecting the search query for each dataset, we have considered the following:

i. We used a root-based search on the keywords to make sure that we covered the maximum amount of cases without missing any potential PRs. For example, for Dataset (3) we considered multiple combinations of pairs (K_1 and K_2) where K_1 is a keyword from {*Depend, Dependent, Dependence, Dependency, Dependencies*} and K_2 from {*Vulnerable, Vulnerability, Vulnerabilities*}.
ii. We explored synonymous words for the search query as follows: if a keyword K have a set of synonyms S_k, then we compare the number of results for the keyword K with those generated from combining the keyword and each synonym from S_k. If the total number of results does not change, then we consider the keyword K to be more generic and sufficient to cover the dataset search. For example, for Dataset (1) we have considered for the keyword *"Dependency"* a set of synonyms {*"Package", "Library", "Third-party Component"*}. The total number of results for the search query (*"Dependency"* AND *"Update"*) and ((*"Dependency"* OR *"Package"*) AND *"Update"*) is identical. Therefore, we consider the results of searching the keyword *"Package"* a subset included in the results of searching the keyword *"Dependency"* that is more generic.

3.2.1 Dataset (1): Dependency Update

This dataset concerns issues related to pull requests that were created by both users and bots to manage dependency updates in GitHub projects. The search query that we used consists of looking for the keywords *"Dependency, Update"* (*Dependency* AND *Update*) in the title, body or comment of a PR. We have initially performed the collection during the time period between 26/05/2017 and 15/06/2021. Note that the first date corresponds to the launch date of Dependabot that we chose as reference for all our datasets to conserve homogeneity and uniformity of PRs creation dates, and the second date corresponds to the day that we initially executed our collection pipeline process. We obtained for this partition a total of 6,573,489 PR-related issues belonging to a total of 927,007 repositories. Then, to validate our findings on more recent data and ensure our study fits in the current trends, we re-executed the data collection process for the time period between 01/01/2023 and 30/09/2023. The second partition contains a total of 3,342,829 PR-related issues belonging to a total of 816,028

repositories. Due to the large size of the dataset, we executed the GitHub- Miner pipeline up to step (4) to only extract issues.

3.2.2 Dataset (2): Dependabot Security PRs

The second dataset is related to PRs created by Dependabot to handle security vulnerabilities in project dependencies. In the search query, we look for PR-related issues created by *"Dependabot-preview"* or *"Dependabot"* and with the label "*security*", also created during the time period between 26/05/2017 and 30/09/2023. With these parameters, the GitHub API returned 422,388 issues from 47,987 repositories. Table 1 shows some summative statistics for the repositories. The age reveals that most projects have a fair maturity that varies between 36 and 5,728 days with a mean of 1,401 days and a median of 1,241 days. They also have frequent activity (mean of 395 days and a median of 178 days), and the size of these projects differ (from 1 to 3,836,094 Kb, mean of 22,033 Kb and median of 3,032 Kb) with a fair popularity (mean of 212 star and a median of 1 star) in order to expand the scope of our study and cover different project types.

3.2.3 Dataset (3): Manual Security PRs

For this dataset, we were interested in PRs created only by users to handle security vulnerabilities. The used query consists of searching for the keywords *"Dependency, Vulnerable"* (*Dependency* AND *Vulnerable*) in the title, body or comment of a PR created in the time period between 26/05/2017 and 30/09/2023. After the pipeline execution, we excluded the issues that were created by bots by filtering the PR authors with the GitHub user type *"bot"* and only keeping those with the type *"user"*. The final results include a total of 186,186 issues for 60,758 repositories. The descriptive statistics of the repositories age distribution, their latest activity, size and their popularity are indicated in Table 2. This dataset has a similar composition to Dataset (2) with relatively more mature projects (mean of 1,625 days and a median of 1,395 days), that are usually more active (mean of 208 days, a median of 63 days).

3.2.4 Dataset (4): Bots' Security PRs

This dataset is related to PRs created by several bots to handle security vulnerabilities in project dependencies. In the search query, we look for PR-related issues where the keywords *"Dependency"*, and *"Security"*, and *"Vulnerability"* are mentioned in the title, body, or comment of the PR. These PRs are created by one of the following bots: *"Snyk"*, *"Renovate"*, *"Greenkeeper"*, or *"Depfu"*, also created during the time period between 26/05/2017 and 30/09/2023. We describe in the following the obtained results for each bot:

Table 1 Statistics for repositories of dataset (2)

Metric	Min	Median	Mean	Max
Age (days)	36	1,241	1,401	5,728
Latest activity (days)	12	178	395	3,865
Size (Kb)	1	3,032	22,033	3,836,094
Stars	0	1	212	80,749

Table 2 Statistics for repositories of dataset (3)

Metric	Min	Median	Mean	Max
Age (days)	1	1,395	1,625	5,707
Latest Activity (days)	0	63	208	3,759
Size (Kb)	1	3,912	84,161	27,292,660
Stars	0	13	1,423	378,072

- **Snyk:** The GitHub returned a total of 274,363 PR-related issues from 51,032 repositories. Table 3 summarizes the statistics for the set of repositories in this dataset.
- **Renovate:** The results returned by GitHub for this dataset consists of a total of 320,918 PR-related issues from 49,483 repositories. The descriptive statistics for these repositories are indicated in Table 4.
- **Greenkeeper:** In this dataset, we gathered from GitHub a total of 2,894 PR-related issues from 2,367 repositories. We provide in Table 5 an overview of the distribution of some statistics for the repositories in this dataset.
- **Depfu:** For this dataset, The GitHub API returned a total of 30,320 PR-related issues from 1,933 repositories. Table 6 shows some statistics for these repositories.

After the collection of the issues for Dataset (2), (3), and (4), we gathered the related PRs and repositories. Note that some PRs were not downloaded due to the fact that few projects were deleted or transitioned from public to private scope, and therefore we could not retrieve the data related to these cases. For Dataset (3), we performed an additional filter on PRs as we have noticed that a vulnerability management bot called *Snyk* creates PRs on behalf of other users (i.e., the PR author appears as the user instead of *Snyk*) when it does not have permissions to open fix PRs. Table 7 demonstrates the total number of PRs and repositories for each dataset.

With the purpose of interpreting the targeted ecosystems and those with significant number of dependency vulnerabilities, we describe the distribution of the number of repositories and PRs created to handle security vulnerabilities for the 11 most popular programming languages (PLs), these PLs dominate more than 90% of the total instances for all PRs in both datasets. As illustrated in Tables 8 and 9, *JavaScript* and *TypeScript* (*npm* ecosystem) are the most popular with more than 50% dominance. This can be due to the fact that *npm* is the most popular registry and the most used among developers (Szulik 2018; DeBill 2019). It also has increasing number of transitive dependencies as well as the number of reported vulnerable packages (Zerouali et al. 2021). This can also be an indication that threat actors view this ecosystem as primary target to spread their malware. After that comes *Ruby* and *Python* PLs (for *RubyGems* and *PyPI* ecosystem respectively) that have less cases of reported vulnerable dependencies.

Table 3 Statistics for repositories of dataset (4) for the Snyk bot

Metric	Min	Median	Mean	Max
Age (days)	4	1,213	1,399	5,582
Latest activity (days)	1	709	860	3,987
Size (Kb)	1	4,403	73,510	29,119,750
Stars	0	0	64	59,126

Table 4 Statistics for repositories of dataset (4) for the Renovate bot

Metric	Min	Median	Mean	Max
Age (days)	0	1,450	1,549	5,521
Latest Activity (days)	0	497	752	3,964
Size (Kb)	1	1,229	26,131	9,078,404
Stars	0	0	104	377,891

3.3 Dataset Quality and Validity

In the scope of our study, we mainly aim to unveil the aspects related to dependency management and handling security vulnerabilities in dependencies in different projects that cover different software development cultures, multiple backgrounds with several characteristics (age & maturity, size & complexity, popularity, etc.) allowing to achieve better generalizability. However, in order to ensure the quality of our dataset and its validity, we have applied selection criteria to filter the repositories. It is worth noting that a study conducted by Munaiah et al. (2017) demonstrated that some selection criteria that are widely used in the literature can potentially exclude relevant engineered software projects. Therefore, we combine multiple exclusion criteria as suggested by prior studies (Kalliamvakou et al. 2014; Mirhosseini and Parnin 2017; Alfadel et al. 2021). We exclude GitHub repositories that fulfill at least one of the following exclusion criteria:

- *Duplicate & Non-Original Projects:* forked repositories are excluded.
- *Popularity & Activity:* repositories with less than 10 stars and no activity within the last year are also excluded.
- *Small Projects:* repositories with less than 500 `Kb` in size and less than 10 issues are excluded.

With this selection criteria, the excluded repositories from our datasets represent around 15% on average (13% for Dataset (2), 18% for Dataset (3), and 15% for Dataset (4)). Hence, we can safely assume that most of the data (85%) is relevant to our study which supports the generalizability of our results and the validity of their interpretation. Additionally, to maintain the quality of our dataset, we have performed a further inspection on the number of commits and the total number of contributors in a project on a sample extracted using *Stratified Random Sampling*. Our analysis shows that 97% of repositories have more than 10 commits (median: 360, mean: 1503), and 73% have more than 3 contributors (median: 6, mean: 28).

Furthermore, we have conducted an additional analysis to ensure the validity of the keyword search process as well as the appropriateness of the projects, for that, we have extracted a statistically representative sample from Dataset (1) using *Stratified Random Sampling*. The

Table 5 Statistics for repositories of dataset (4) for the Greenkeeper bot

Metric	Min	Median	Mean	Max
Age (days)	1,401	2,548	2,604	5,025
Latest activity (days)	0	336	740	2,863
Size (Kb)	4	1,060	15,899	3,918,869
Stars	0	3	343	37,176

Table 6 Statistics for repositories of dataset (4) for the Depfu bot

Metric	Min	Median	Mean	Max
Age (days)	56	1,512	1,673	5,739
Latest activity (days)	0	353	571	3,069
Size (Kb)	6	2,023	38,570	10,239,100
Stars	0	1	119	73,556

sample comprises 385 PR-related issues from the first partition of Dataset (1) and 385 PR-related issues from the second partition as well. Note that the sample size was determined using the *Andrew Fisher's Formula* with a confidence level of 95% and a margin of error of 5% on the dataset that was previously filtered. Once the sample size is determined, the strata are divided based on the following characteristics: the PR author (*Bot* or *User*), the project creation time, and the programming language. After that, the size of each stratum is determined using (1) as discussed in Section 4.3.2.

The manual assessment consists of checking the following properties that were inspired from prior studies (Kalliamvakou et al. 2014; Munaiah et al. 2017):

- *Engineered software project.* To make sure our dataset is composed of software projects that contain source code, and avoid including repositories that are used for storage purposes or as personal projects, we inspect the description of the project and the `README`, the content of the repository in terms of file type, as well as the number of committers.
- *Relevance & Validity.* For the construction of our dataset, we relied on GitHub API to search for keywords in the title, body, or comment of a PR. Therefore, it is imperative to check whether the results are appropriately accurate and avoid any false positives. We investigate whether the PR was created for the purpose of dependency update by analyzing the PR description (title, body, and comments) and labels as well as the related commits.
- *Active projects.* To ensure that the repositories in our dataset are actively maintained, we investigate the last update date, the number of commits, pull requests, and issues, as well as the number of contributors.

Our assessment has shown that 99.47% of the selected data is relevant and valid for the context of our study, so we can safely rely on the keyword search. As for the other properties, 91.5% of the selected data was found to be fitting our description.

Table 7 Distribution of repositories and PRs per dataset

Dataset	# PRs	# Repositories
Dataset (2)	401,236	47,154
Dataset (3)	115,299	38,126
Dataset (4): Snyk	273,953	50,911
Dataset (4): Renovate	320,634	49,374
Dataset (4): Greenkeeper	2,894	2,367
Dataset (4): Depfu	30,311	1,929

Table 8 Distribution of repositories and PRs per ecosystem for dataset (2) and (3)

Programming language	Dataset (2)		Dataset (3)	
	# Repos	# PRs	# Repos	# PRs
JavaScript	19,557	165,623	9,679	19,452
TypeScript	10,458	84,385	6,512	26,323
Ruby	5,263	52,969	1,255	2953
Python	1,932	15,839	2,806	12,137
HTML	1,660	14,226	818	3,107
PHP	1,522	14,067	824	1,950
Vue	1,512	11,501	330	539
Java	1,094	9,154	6,589	20,810
CSS	950	7,841	343	513
Rust / Go	603	5,667	3,104	8,232
Multiple	789	4,361	1109	3,977
Total	45,340	385,633	33,369	99,993

4 Study Design

In this section, we present the research approach that we relied on to conduct our empirical study, including the research questions (RQs), the features and the factors considered for the dataset assessment, and the methodology that we pursue to address each RQ.

4.1 Research Questions

In this paper, we focus on four distinct aspects related to the adoption of Dependabot in handling dependency updates and security vulnerabilities. Our research is conducted through the following questions:

Table 9 Distribution of repositories per ecosystem for Dataset (4)

Programming language	Dataset (4)			
	Snyk	Renovate	Greenkeeper	Defpu
JavaScript	11,672	17,273	1,677	578
TypeScript	3,685	11,342	505	437
Multiple	13,217	4,741	10	110
Java	4,305	2,368	0	4
Python	7,392	2,221	8	8
HTML	1,889	1,371	54	85
Go	1,335	1,351	0	3
Ruby	1,358	1,191	6	492
PHP	397	1,151	6	31
Vue	319	1,067	13	44
CSS	779	1,037	21	46
Total	46,348	45,113	2300	1838

– **RQ 1** ***To what extent is Dependabot adopted ? Why is Dependabot more adopted than other tools ?***

  Goal : With the first research question, our objective is to measure the degree to which Dependabot is used in GitHub compared to other dependency management tools, as well as the manual activity of developers. This allows us to have more insights about the popularity of this bot and its activity across different projects and ecosystems. Then, we find it very informative to identify the reasons why Dependabot is the most popular and the most active tool based on developers' experiences, so that we can extract important aspects that researchers and tool designers should consider with new tools and approaches.

– **RQ 2** ***What do developers do to handle security vulnerabilities in dependencies ?***

  Goal : The second research question focuses on understanding the practises and the security measures that developers adopt to react to security vulnerabilities in their project dependencies. Our interest lies in determining both the process of identification and the fix that allows to eliminate or mitigate the impact of these vulnerabilities. These trends are extracted manually by analyzing a representative sample of the dataset.

– **RQ 3** ***How fast are security pull requests handled ? How long do vulnerabilities remain unpatched ?***

  Goal : Through the third RQ, we attempt to measure the response time of developers to security PRs and how long it takes to handle them (either accept or refuse). This gives us an indication of how receptive developers are to these PRs and whether they react quickly to fix a security vulnerability in their projects after patch disclosure, so that malicious threat does not persist for an extended period of time. It is also crucial to determine the threat lifetime in GitHub projects prior to the vulnerability's discovery, as well as the fix delay that results from the manual effort to release patches.

– **RQ 4 What are the factors that significantly correlate with the decision and the time to accept security PRs ?**

  Goal : With the final RQ, we try to identify the factors that can potentially act on the decision on whether to merge a security PR created by Dependabot or developers as well as the merge rate. These factors can be very insightful for both contributors and researchers to propose new effective and efficient approaches. The major usefulness lies in discerning the aspects to consider in order to encourage wider and faster adoption of security PRs.

4.2 Descriptive Features

In order to perform our descriptive analysis on the PRs that are created in GitHub projects, a set of features was selected to investigate different aspects related to dependency management and security PRs. These features were inspired from several studies that were conducted in the literature on pull requests merge decision and merge time (Alfadel et al. 2021; Gousios et al. 2014a; Gousios and Zaidman 2014; Soares et al. 2015a), code review in pull requests (Kononenko et al. 2018b; Yu et al. 2015), code contribution in pull requests (Zhu et al. 2016), software bots (Erlenhov et al. 2019a; Wessel and Steinmacher 2020a), and bloated dependencies (Soto-Valero et al. 2021). We have also added some features that can potentially act on the merge decision and merge time of PRs after performing our manual analysis on a representative sample of our dataset (as discussed in Section 4.3.2). The findings induced

by the manual analysis inspired some dependency features that we have noticed developers were considering while handling security vulnerabilities in dependencies. We have later found that some studies also considered these features. Therefore, we sought to substantiate our observations from the manual analysis with a descriptive analysis on these features. Table 10 summarizes the selected features that are separated into the following categories:

4.2.1 Repository Features

These features aim to capture the relationship between the repository as a whole environment and the management of the PRs that are created within the project, as well as the time and effort required to handle these PRs. The first feature represents the `age` of the repository when the PR was created, this provides an indication of the maturity of the project, the level of growth and the experience that change the practices and the development processes that contributors follow (e.g., older projects are usually more stable and developers are more familiar with the code and the different components, so we assume that it would take less time to review and handle a PR). It is calculated as the time difference between the repository creation date and the PR creation date.

Several studies suggest that projects with high and frequent activity tend to handle PRs quickly and within smaller time frames (Alfadel et al. 2021; Gousios et al. 2014a). Therefore, we considered to check how active the repository was at the PR creation time (`latest_activity`) to study the merge rate of PRs. This feature is calculated as the time difference between the repository last update date and the PR creation date. The `size` of the repository can be a pointer to the projects complexity, since larger repositories have more functions (or potentially longer history) which increases the time and effort required to perform manual tasks (e.g., issues management, test, code review, etc.). We also considered the number of users that watch the repository (`# watchers`), so that when new issues and updates are added, notifications are automatically sent to inform them, which increases the chances for new contributions to review and analyze the new PRs by more developers. Similarly, the total number of issues that are still open in a repository (`# open_issues`) indicates to what degree the project is active in regards to supporting new features, handling bug reports, etc. Note that these features are either calculated automatically or provided by the GitHub API.

4.2.2 Pull Request Features

PR features aim to quantify the different contributions and the changes brought by the PRs as well as the exchanges between the owners and the contributors to solve the issues related to the PR. When a pull request is created, developers can be assigned by project maintainers to fulfill several responsibilities (e.g., code review, test run, accepting the PR after the review and the test run, etc.). So, we took into consideration the number of assignees (`# assignees`) and the number of requested reviewers (`# requesed_reviewers`) as a factor that can correlate with the time to merge PRs, in fact, the number of participants has been shown to affect the process time during code reviews (Rigby and Bird 2013).

Also, the changes suggested in the PR affect the merge time, as several studies (Gousios et al. 2014a; Weißgerber et al. 2008) have shown with the patch size and the added and deleted lines of code. In our study, we selected the number of commits (`# commits`), additions (`# additions`), deletions (`# deletions`) and changed files (`# changed_files`) related to the PR which may act on the time required by developers to perform and validate their code review.

Table 10 The descriptive features for the PR study

Category	Feature	Description
Repository	`age [1,6]`	Age of the repository from its creation date until the pull request creation time (in days)
	`latest_activity [7]`*	Time interval between the last update in the repository and the pull request creation time (days)
	`size [1,2,3]`*	Size of the repository (in Kb)
	`# watchers [3]`	Number of GitHub users that register to watch the repository for new updates notifications
	`# open_issues [6]`	Number of the total open issues that are registered and not handled in the repository
Pull request	`# assignees [3]`*	Number of GitHub users that are assigned to the issues related to the PR
	`# requested_reviewers [3]`*	Number of GitHub users that are requested to review the code in the PR
	`# commits [1,2,3,4,5,6]`	Number of commits that perform the changes suggested in the PR
	`# additions [1,2,3,5,6]`	Number of lines of code added in the commits of the PR
	`# deletions [1,2,3,5,6]`	Number of lines of code deleted in the commits of the PR
	`# changed_files [2,3,4,5]`	Number of files changed by the commits of the PR
	`# comments [2,3,5,6]`	Number of comments in the discussion history of the PR
	`discussion_size [8,9]`*	Size of the body of the PR (i.e., words count)
User	`acquaintance [5]`*	Time interval between the creation date of the user account and the PR creation time
	`author_association [4]`	Association of the GitHub user to the project repository (i.e., owner, contributor)
	`commit_history [2,3,5]`*	Number of commits created by the GitHub user before the studied PR is created in the repository
	`# followers [3,6]`	Number of the GitHub user followers
	`# public_repos_gists [5,6]`*	Number of public repositories and gists created by the GitHub user
Dependency	`update_level [1]`	Specification of the version update of the dependency in the PR (i.e., patch, minor, major)
	`severity [1]`	Level of severity for the dependency vulnerability (i.e., low, moderate, high, critical)
	`is_bloated [10]`*	Indicator if the dependency is used in the repository or bloated (not used)

[1] Alfadel et al. (2021); [2] Gousios et al. (2014a); [3] Gousios and Zaidman (2014); [4] Soares et al. (2015a); [5] Kononenko et al. (2018b); [6] Yu et al. (2015); [7] Zhu et al. (2016); [8] Erlenhov et al. (2019a); [9] Wessel and Steinmacher (2020a); [10] Soto-Valero et al. (2021);

(*) Denotes newly suggested features that were not considered in prior studies but were inspired from their findings

Another aspect that we have weighed up is the communication and the exchange between developers while handling the PR. For that, we measured the number of comments in the PR (`# comments`) and the size of the PR body (`discussion_size`) that may introduce more time to understand the issue and cover the different requirements before the merge. It is worth noting that these features are either calculated automatically or downloaded using GitHub API.

4.2.3 User Features

The features related to users are intended to measure the implication of the person who created the pull request on the decision to merge it and the time to process it. In fact, a study conducted by Jeong et al. (2009) has shown that the developer who created a patch has influence on the patch acceptance decision. We also want to inspect if the experience that a developer has can impact the acceptance of his contributions and the time to merge them into a project. Therefore, we consider the degree to which a developer is familiar with GitHub platform and the pull-based development model (`acquaintance`) when the PR was created, it is calculated as the time difference between the user's account creation date and the PR creation date. Another aspect that we want to examine is whether the PRs that are created to handle security vulnerabilities receive special treatment when they are created by core team members or by contributors (`author_association`). Additionally, as previous studies indicated (Pham et al. 2013), developer's track record can be a strong indicator of pull request quality. Hence, we consider the commit history feature (`commit_history`) as a proxy for developers' contribution and track record, as well as their experience and familiarity with the project. Note that this feature can also be an indication of bots' configuration (e.g., usage v.s. non-usage of dependency management bots, frequency of alerts and updates through commits, etc.). It is calculated as the total number of commits that were previously created by the author prior to the creation of the PR in question.

Also, the work conducted by Bird et al. (2007) presented evidence that social reputation has an impact on whether a patch will be merged. So, we consider the number of followers (`# followers`) and the number of public repositories and gists (`# public_repos_gists`) created by a user as a substitute for its reputation on GitHub. Note that these features are either calculated automatically or provided by the GitHub API.

4.2.4 Dependency Features

Dependency features allow us to specify the characteristics of the dependencies that are suggested in the updates of the PRs. Since many developers have concerns related to the breaking changes in their projects, we speculate that the update level (`update_level`) can potentially act on the decision and time to merge a PR. According to the specification of *Semantic Versioning* (Preston-Werner 2021), *patch* updates make backward compatible bug fixes, and *minor* updates are related to adding functionalities in a backward compatible manner, so, we hypothesize that developers can be more receptive to these updates and it would take them less time to review the changes and validate them. However, *major* updates concern incompatible API changes, which may have higher chances to cause breaking changes in the project, which can lead developers to take more time to review the changes and potentially not merge the PRs that cause these breaking changes.

When these PRs are created to fix security flaws, they usually reference the GitHub Advisory Database. In addition to the information that is provided about the security issues

in vulnerabilities databases, they also provide a severity level (`severity`) based on the vulnerability CVSS score (NIST 2021) that describes the impact and the range of malicious behaviors that are exploited by attackers. In our study, we want to examine if PRs are treated differently or handled faster when the dependency updates present high security risks.

Also, an interesting study conducted by Soto-Valero et al. (2021) has shown that bloated (non-used) dependencies tend to increasingly appear through the maintenance of projects, and a considerable number of these dependencies are updated based on Dependenbot PRs as well as those created manually by developers. Hence, one of our concerns is to inspect the correlation between bloat (`is_bloated`) and the manner that developers handle vulnerabilities in bloated dependencies. Note that these features are extracted manually after the analysis performed on the dataset sample as detailed in Section 4.3.2. The analysis of the PR body, comments, and commits showed that some developers give importance to the update level of the dependency, the severity level of the vulnerability, and whether the dependency is used or not. Therefore, we wanted to examine these features closer and provide more evidence to our observations.

4.3 Methodology

We tackle in this subsection the approach that we have conducted to answer the research questions introduced in Section 4.1.

4.3.1 RQ1. Dependabot Popularity

(1) Quantitative & Comparative Analysis. In the first RQ, we start with a global analysis to explore the popularity of Dependabot in the context of dependency management through different ecosystems and repositories that cover multiple time frames, sizes and maturity levels. To that end, we use Dataset (1) (related to *Dependency Update*) to count the total number of PRs that were created to perform dependency updates, grouped by the author who created the PR as specified in the GitHub API. In this process, we separate between bots and the users who manually created these PRs, for that, we check for the user *type* specified in the JSON file returned by the GitHub API. Our goal is to perform a comparison between bots and also analyse the activity improvement compared to the manual effort of users. We also differentiate between open PRs and those that are already handled (merged or closed) to have an overview of their management. In order to extract this information, we check the *state* of the issues related to the these PRs in the JSON file returned by GitHub- Miner. *Open* issues refer to PRs that are still open up to the current date for test and review, and *closed* issues refer to PRs that are either accepted (merged) or rejected (closed). After that, we visualize the creation history of these PRs for each author during the entire period of the dataset lifetime to compare the progress and the growth speed of dependency updates in GitHub projects. For that, we measure the cumulative sum of PRs created on a monthly basis.
(2) Investigation Survey. To answer the second part of this RQ, we have conducted a survey with repository owners from GitHub to understand the reasons behind the adoption of Dependabot and other tools to handle dependency updates. Our survey contains three groups of questions as summarized in Table 11:

i. ***G1:*** Three multiple-choice questions (MCQs) related to the demographic profile about participants and their experience;

Table 11 Survey design for the 1st research question

	Question	Type	Response
G1	*Q1*: The role that best describes the participant's activities	*MCQ*	100%
	Q2: The most used programming languages	*MCQ*	100%
	Q3: The participant's level of experience	*MCQ*	100%
G2	*Q4*: The adopted tool / bot for dependency management	*MCQ*	100%
	Q5: The features that motivate the use of the tool / bot	*MCQ*	100%
	Q6: The popularity reasons	*MCQ*	100%
	Q7: The need for checks / reviews of automatic PRs	*RSQ*	100%
G3	*Q8*: The challenges / problems encountered	*OEQ*	59%
	Q9: The suggested features for potential improvements	*OEQ*	59%

ii. ***G2:*** Three MCQs about the dependency management tool and the features that motivate its use, as well as a rating scale question (RSQ) for the degree to which participants need to review and check the automatically created PRs;
iii. ***G3:*** Two open-ended questions (OEQs) about the challenges and the problems encountered with the adopted tool, and the features that can be added to overcome these deficiencies.

It is worth noting that each of our MCQs provides some options for the participants to select while also leaving the freedom for them to add other options and responses to present open answers in their own words in order to get more valuable and potentially unknown insights. Participants were informed that the survey was anonymous and would take 4-5 minutes to complete.

In this survey, we targeted repository owners that were randomly selected from Dataset (1). First, we considered repositories with several programming languages and we made sure that different dependency management tools were being used within these projects, the initial selection contained 1000 users. After that, we leveraged the user endpoint of GitHub API to retrieve the email of each user, 569 of these users lacked valid email addresses; after sending 431 emails, there were instances of failure, emails that bounced and automatic responses of non-existent emails. Finally, only 164 emails were successfully sent and opened by the repository owners, and we received 22 responses to our survey leading to a response rate of 13% (22/164) which is higher than the suggested minimum response rate of 10% (Groves et al. 2011).

Ethical Considerations To address any ethical concerns related to our research survey we made sure to implement the following: *Confidentiality, Anonymity, & Absence of harm:* we made sure that during the data collection to not retrieve any personal data related to participants, and the survey was conducted anonymously; *Informed consent:* the participants were informed about the purpose of the study, its benefit and impact before taking part in the survey; *Voluntary participation:* participants were free to opt in or out of the study at any point in time. However, when it comes to the fact that e-mails were sent unsolicited from GitHub information, we assumed that, since developers have the option to hide their email addresses from other users and even in the API data, they were interested in participating in our survey especially since it tackles crucial aspects in open-source software development. Additionally, at the end of our survey, we asked the participants if they were interested in keeping contact with us so that we can potentially ask them further questions, and 30% of the participants

agreed and provided their emails. More than two-thirds avoided to be further contacted due to the possibility of them not being comfortable sharing their own email addresses, or the consideration that some participants hesitated since our question did not explicitly mention the time required for the new questionnaire. Finally, we acknowledge that the survey has a small sample size, which can be due to the initial selected subset. However, to ensure using higher quality of data, and acquiring relevant responses that align with the current trends and practices, we focused our survey on projects with key characteristics (activity, maturity, diversity, etc.). Another reason is related to the encountered hindrances (non-valid emails, bounces and failures, etc.) that limit the reception of our survey, and also the low response rate (13%). Yet, it allowed the discovery of valuable insights and it is higher than the minimum response rate (Groves et al. 2011).

4.3.2 RQ2. Security Vulnerabilities in Dependencies

(1) Data Sampling. In this research question, from the global setting of PRs created for the purpose of updating dependencies, we focus on those that allow developers to fix security vulnerabilities. Developers can react differently to vulnerabilities, and they apply several practises to mitigate the threats and conserve the proper functioning of their projects at the same time. Our objective is to understand and examine what do developers do in particular to handle these vulnerabilities. To that end, we qualitatively analyse statistically representative samples of Dataset (2) and (3).

First, we start by selecting a representative set of more than 10% of the total PRs from each dataset, that is a sample of 50,000 PRs (39,677 and 10,323 PRs from Dataset (2) and (3) respectively). The selection process is conducted through *Stratified Random Sampling* with proportionate stratification to conserve homogeneity and the occurrence of each possible sample in the resulting subgroups (*strata*). The subgroups are divided based on specific characteristics that we want to maintain in proportion to the entire dataset. The characteristics that we considered are: the PR author (*Dependabot* or *user*), the PR state (*open*, *merged* or *closed*), the update level in the dependency, the severity level of the vulnerability, and the dominant programming language in the repository. Note that we only considered the 11 most popular programming languages in Dataset (2) and (3). To determine the size of each stratum, we use the following equation:

$$s_i = \frac{N_i}{N} \times n \tag{1}$$

Where: s_i represents the wanted sample size for stratum i; N_i is the population size (from the dataset) for stratum i; N denotes the total population (or dataset) size; and n represents the total sample size. We call the selected sample the *Manual Analysis Dataset* that we intend to use on the investigations of other RQs.

(2) Manual Qualitative Analysis. After the process of selecting all strata, we merge them into one file, then we randomly extract a sub-sample for the manual analysis with a *confidence level* of 95% and a *margin of error* of at most 5% (that is 3060 and 380 sub-samples from Dataset (2) and (3) respectively). After that we randomly split the merged file into 5 distinctive and complementary segments, so that each segment is independently analyzed by a research student. Afterwards, the first author validates the results that were found in each segment. The manual analysis consists of relying on *open coding technique* to examine the commits of each PR to identify the patches and the changes applied on each project file, we also examine the body of the PR and the associated comments in order to inspect the feedback of developers and understand in-depth their practises that solve the security flaws reported through the analyzed

PRs. Finally, using *axial coding*, we summarize the course of action that developers tracked to identify and apply the fixes that eliminate or mitigate the threat in the dependency version. Please note that the coding techniques (open and axial) serve the purpose of our study by enabling data analysis in two facets: first, they reveal undiscovered themes/patterns embedded in the data, and second, they provide an organized plan for the construction of meaning and extraction of knowledge. As for the reasons that motivate the use of such techniques, they are three-fold: (1) Lack of knowledge about the strategies used by developers to identify and fix security vulnerabilities in dependencies. We had no preconceived ideas or theories for this research purpose, and these coding techniques allow us to address this purpose not based on existing theory but based on the knowledge emerging from the data. (2) The inductive process: as the coding process progresses, it enables essential patterns to be identified, codified, and interpreted for the purpose of the research study. It allows us to investigate developers' practices when handling security vulnerabilities to understand: What they do? How they do it? And why they do it? (3) Validation from data: the process allows us to understand closely the data by continuously and iteratively reading (analyzing) and rereading the collected data in order for knowledge to be extracted and systematized. An example of the application of these techniques is provided in Appendix A.

4.3.3 RQ3. Security Pull Requests Management

(1) Quantitative Analysis for PR receptiveness. Security flaws in project dependencies is a critical problem that can be very impactful, and developers ought to fix any security vulnerability as soon as possible. As a matter of fact, the longer a vulnerable dependency remain unpatched, the more damage it causes. And so, after understanding the measures that developers tend to rely on, we start by measuring the responsiveness and receptiveness of developers to PRs related to security fixes. For that, we represent the distribution of the total number of PRs grouped by *state* from Dataset (2) and (3). The state of a PR is provided by the GitHub API as "*open*" if the PR is still on hold for further analysis or under review before accepting or refusing its changes; "*merged*" state represents PRs that are accepted and the changes were indeed applied in the repository; on the other hand, "*closed*" state refers to PRs that are rejected and the proposed changes were not validated. After outlining these statistics, we perform a comparison on how receptive developers are to these PRs based on the author type (*bot* or *user*) and its association (*owner* or *contributor*). For merged PRs, we investigate whether the "*auto-merge*" feature provides any improvements by accelerating the time to merge security PRs. Also, for closed PRs that were initially created for the purpose of improving the project security, we consider that it is crucial to identify the reasons that lead to rejecting these PRs by analyzing the comments, discussions and reviews related to them.
(2) Quantitative Analysis for PR responsiveness. To examine security PRs responsiveness, we measure the merge rate and close rate of PRs created by Dependabot from Dataset (2) and for PRs created manually by users from Dataset (3). The merge rate is calculated as the time difference between the creation date (*created_at*) of the PR and the merge date (*merged_at*) that are both available through GitHub API. Similarly, for the close rate, it is measured as the time difference between the creation date of the PR and its close date (*closed_at*). We represent after that the distribution of the time to handle PRs for both datasets using a boxplot. Finally, to inspect the evolution of security PRs responsiveness over time, we aggregate the merge rate and close rate that were previously calculated on a monthly basis and compute the average and median values.
(3) Quantitative Analysis for Threat Lifetime. Delaying the fixes of security flaws puts the project and its dependents at risk since it spreads the window of exposure. So, for the second

part of this RQ, we investigate the approximate time to discover the vulnerabilities in GitHub projects (*Hidden Threat Lifetime*), and the fix delay spent before fixing the vulnerability and the disclosure of new patches (*0-day to patch disclosure*). For that, We measure the time interval between the PR creation date, the date on which the vulnerable version of the package was first introduced in the project as dependency (*0-day*), and the publication date of the *fix disclosure*, as shown in Fig. 4.

After that, we represent the distribution of these metrics using a violin-plot. A further investigation is conducted on the threat lifetime and the occurrences of security vulnerabilities in dependencies based on their severity and the update level of the suggested upgrades. To that end, we represent the distribution of PRs and aggregate the average threat lifetime based on the different severity and update levels. Finally, we enumerate the list of the most dominant vulnerability types in dependencies with their average threat lifetime. It is worth noting that the PR creation date is provided through GitHub API in our dataset, and the manual analysis of the data sample that was previously selected in RQ2 (i.e., *Manual Analysis Dataset*) allows us to extract when the vulnerable dependency version was first included in the repository by leveraging *"git blame"* to view the change history of the dependency file. The vulnerability disclosure date, the severity and update level, as well as the vulnerability type are all available in the referenced GAD.

4.3.4 RQ4. Merge Decision and Merge Rate

For the last RQ, we attempt to study the factors that can potentially impact the decision to merge or close a PR at a first step, and then study the correlation of these factors with the time to merge these PRs (i.e., *merge rate*). For that, we start by performing a statistical analysis based on the features that were presented in Section 4.2 to evaluate both Dataset (2) and (3). ***(1) Data Pre-processing.*** First, we start with a pre-processing step that comprises a cross-correlation analysis to clean the dataset, and make sure there is no redundancy and all the variables are sufficiently significant and independent. After that, we implement a filtering

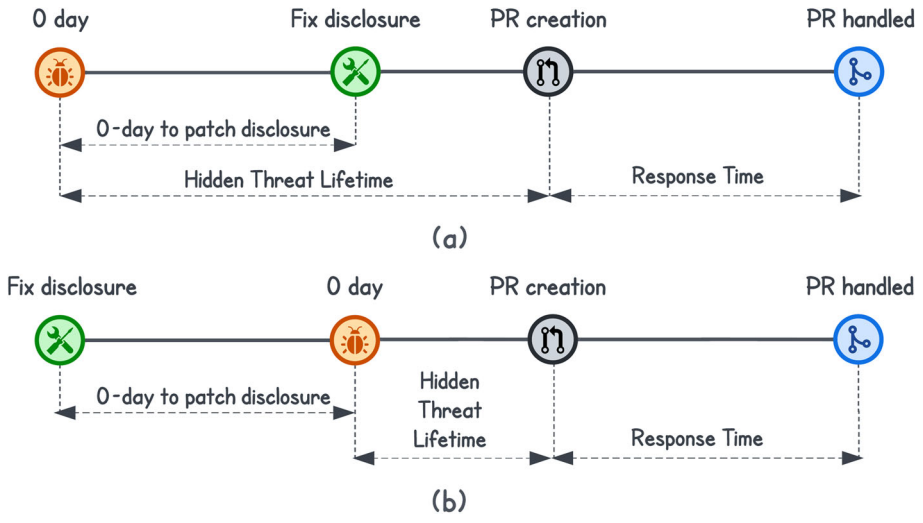

Fig. 4 Timeline for vulnerability discovery time and fix delay; (a) Patch disclosed after adding the vulnerable dependency, (b) Patch disclosed before adding the vulnerable dependency

step that allows us to remove the outliers. The pairwise correlation analysis is conducted by measuring the *Spearman* correlation coefficient $|\rho|$ for each pair of variables in the dataset, and then excluding one of the variables when the correlation is considered high (i.e., $|\rho| \geqslant 0.7$) (Akoglu 2018; Calkins 2005). In our experimentation, we excluded the number of requested reviewers (`# requested_reviewers`) from the total features since it has no variation in the dataset, and therefore it is considered non-significant for our study; we also removed the number of deletions (`# deletions`) since it was highly correlated ($\rho = 0.89$) with the number of additions (`# additions`) in a PR. This high correlation is due to the fact that most of the PRs in both datasets upgrade the vulnerable dependency version, as such, for each deleted version, a new one is added (this also includes related parameters, such as: package hash, download link, etc.). For the outlier filtering step, we have noticed - after analyzing the dataset - that it contains values that significantly differ from other observations, which may skew the analysis and the measurements (e.g. Mean, Standard deviation, etc.) used in our study. Thus we need to exclude these values from the statistical analysis. The outliers are determined by using a box-plot representation of each variable to compute a cutoff value.

Let v be a variable from the set of features considered in our analysis, and $A_v = \{a_1, a_2, a_3, ..., a_n\}$ represents the values of the variable v for the n instances in the dataset. The cutoff c_v for this variable is calculated as following:

$$c_v = \text{Max}\left(Q_3 + \frac{3}{2} \times IQR, \frac{1}{n}\sum_{1}^{n} a_i\right) \tag{2}$$

Where Q_3 and IQR denote *third quartile* and *interquartile* range respectively. Any values from A_v greater than the threshold c_v are filtered out. In our study, the proportion of the excluded outliers is less than 10% of the total instances in our dataset (i.e., 8% for Dataset (2) and 6% for Dataset (3)).

(2) Statistical Tests Analysis. After that, we conduct our statistical analysis. For the merge decision, we start with a *Random Forest* based classifier to perform the feature selection on the filtered variables using the *Gini importance* as feature importance (default property in `sklearn.ensemble.RandomForestClassifier`). Then, we use *Logistic Regression* on quantitative variables, and χ^2 (*Chi-Squared*) test on categorical variables. For the merge rate, we use *Multiple Regression* on quantitative variables; and for categorical variables, we use *Wilcoxon Rank-sum* (i.e. *Mann Whitney U*) test in the case where two groups are being compared, and *Kruskal-Wallis* test for more than two groups. These tests determine whether each variable (i.e., feature) has a statistically significant correlation with the merge decision and merge rate of PRs. We also leverage the R^2 metric to rank the contribution of variables and measure their explanatory power, so that the higher the value, the greater it is acting on the outcome variable.

(3) Validation Survey. Finally we substantiate these quantitative results with a survey that we sent to Dependabot users in GitHub. The survey contains three groups of questions as shown in Table 12:

i. ***G1:*** Three MCQs related to the experience and the demographic profile of participants;
ii. ***G2:*** Two rating scale questions about the receptiveness and responsiveness on security PRs;
iii. ***G3:*** Two MCQs about the reasons that cause the rejection of Dependabot security PRs as well as the reasons for not handling them.

Note that each of our MCQs provides an initial list of options for the participants to select while also giving them the opportunity to add new options and responses to present open answers.

Table 12 Survey design for the 4th research question

	Question	Type	Response
G1	*Q1*: The role that best describes the participant's activities	*MCQ*	100%
	Q2: The most used programming languages	*MCQ*	100%
	Q3: The participant's level of experience	*MCQ*	100%
G2	*Q4*: The frequency of check / review for security PRs	*RSQ*	100%
	Q5: The level of receptiveness to security PRs	*RSQ*	100%
G3	*Q6*: The reasons for rejecting security PRs	*MCQ*	100%
	Q7: The reasons for not handling security PRs	*MCQ*	100%

We initially selected 1000 random users from Dataset (2), but after the transmission fails and the encountered errors, only 128 emails were successfully sent and opened by our recipients, and we have received a total of 18 responses prompting a 14% (18/128) response rate.

Ethical Considerations Similar to our previous survey in RQ1, we made sure to consider any ethical concerns by addressing them is a similar fashion to ensure: *Confidentiality, Anonymity, Absence of harm, Informed consent,* and *Voluntary participation*.

5 Results and Discussion

In this section, we answer each of the four RQs that were presented in Section 4.1.

5.1 RQ1. Dependabot Popularity

Level of popularity First, we start by assessing the activity of different tools and bots that create PRs for dependency management. Figure 5 illustrates the distribution of PRs per author. The graph is only limited to the 9 authors that create more than 99% of the total PRs in the entire dataset, it is worth noting that *Dependabot* was previously known as *Dependabot-preview* that supported the same general functionalities. Therefore, *Dependabot* dominates the dependency management activity with more than 65% of the total PRs. After that, comes *Renovate* that creates around 20% of the total amount of PRs in our dataset, *Renovate* generally shares the same automation functionalities as *Dependabot* and supports more than 9 PLs, which explains its common usage among developers. Also, *Greenkeeper* is a tool that is used for the same purposes but only supports the npm ecosystem which explains its low popularity. Similarly, *depfu* is a paid tool that is usually used by companies and supports only three ecosystems (Ruby, JavaScript and Elixir). The remaining tools (*dotnet-maestro*, *dependencies*, and *github-actions*) are bots that are mainly used on github for general purposes in the context of automating developers workflow (e.g., scheduling tasks on dependency updates, builds and tests on CI/CD pipeline, etc.), so they have a small contribution to update dependencies. Another aspect that is worth highlighting from this diagram is the fact that 79% of the total PRs in the dataset are already handled, so developers react to these PRs either by accepting or rejecting them, which shows their involvement in the activity of dependency management as it is an important concern. Furthermore, based on the total number of created PRs, it is fairly deduced that bots have enabled an improvement in monitoring outdated packages and alleviating the difficulty of handling them manually.

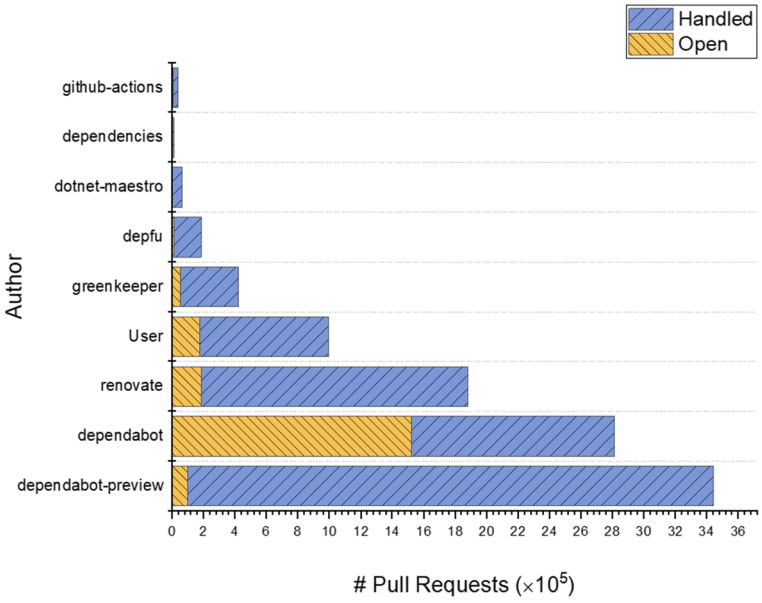

Fig. 5 Pull request distribution per author

In order to see in more depth the activity of dependency management in these repositories and its progress, we visualize the creation history of the PRs for each author over time. Figure 6 depicts the cumulative number of PRs created by the most active authors during the last years from 26/05/2017 to 15/06/2021. We notice through this creation history that Dependabot was increasingly getting more popular, especially from the beginning of 2018 where most of the current programming languages were supported, and the new versions of package managers were added. It is worth mentioning that the other free tools (i.e., *Renovate* and *Greenkeeper*) have a similar progression slope compared to the user activity. *Depfu*, on

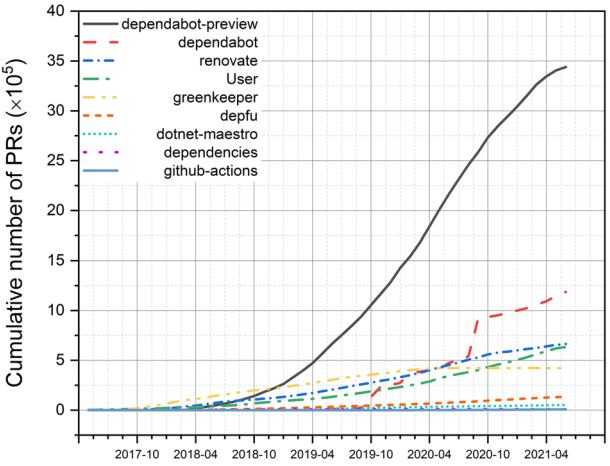

Fig. 6 Pull request creation history per author for the first partition

the other hand, has a slower progression due to the fact that it is a paid tool and supports fewer programming languages. However, Dependabot creates on average around 68,784 new PRs per month - from its launch date - to update project dependencies, which shows the improvement that automated tools provide compared to the manual effort of users.

As a further investigation that we conduct on recent trends for these bots, Fig. 7 depicts the cumulative number of PRs created by the most active authors during the last year from 01/01/2023 to 30/09/2023. The first observation is the absence of *Dependabot-preview* among other bots in the activity of dependency management. This is due to the fact that *Dependabot-preview* is no longer supported since August 2021, and its functionalities were moved to *Dependabot*. The latter is also maintaining its activity while being the most popular among other bots. Interestingly, the activity of *Renovate* and its popularity is the closest to that of *Dependabot*'s, and this popularity is mainly due to the newly supported platforms that were recently added to this tool (e.g., GitHub, GitLab, Bitbucket, etc.). As for the other tools, we observe a similar pattern from the previous time period with a slower progression.

In conclusion, Dependabot is the most popular and the most used tool in GitHub for dependency management, and it still gains more popularity over the course of the last years compared to other tools.

RQ1.1: For dependency management, Dependabot is the most popular tool with more than 65% dominance compared to user activity and other tools. Its activity and popularity is increasing over time especially after the support of different ecosystems.

Popularity reasons In order to identify the reasons behind the popularity of Dependabot, we have conducted a survey with repository owners from GitHub. From the total responses, 86.4% were developers and 13.6% were from the management team. When it comes to the programming languages, 81.8% are mostly familiar with JavaScript, 63.6% use TypeScript, 50% are using HTML/CSS, and 18% to 22% of the participants are familiar with other programming languages (Go, Python, Java). These developers come with different background experiences: 27.3% have less than 5 years experience, 36.4% have 5 to 9 years, 31.8% with 10 to 20 years, and 4.5% have more than 20 years experience. The tools that they mainly

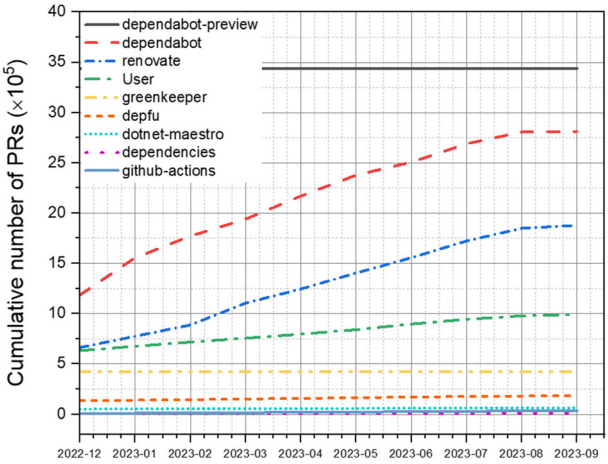

Fig. 7 Pull request creation history per author for the second partition

use are Dependabot and Renovate. We have confirmed that 95.5% of the repository owners use these bots for the purpose of automating dependency updates, and 63.6% are interested in handling security vulnerabilities as an additional features provided by these Software Composition Analysis (SCA) tools.

Concerning the popularity reasons that motivate developers to use Dependabot, Table 13 summarizes the most common features that lead them to adopt this tool. From the total responses, 86.4% use Dependabot because it can be easily integrated and configured into their GitHub projects, which leads to an efficient adoption of the activity of dependency management (Efficiency). Accessibility is also an important aspect, since most of software development conducted on GitHub is open source (Plumb 2022), developers prefer to use free tools to work on both their public and private projects. In fact, it is the second most popular reason that 68.2% of our participants selected. Also, with the new development practices, developers prefer to automate their continuous integration (CI) and continuous development (CD) pipeline where different tasks and tests are performed automatically before building a project. So tools are required to be adapted to these processes (Adaptivity). Hence, the 45.5% of participants that use Dependabot as it can be easily integrated into the CI/CD pipeline. Finally, 31.8% prefer using this tool due to the availability of documentation. If developers struggle to find information about the tool and its usage, more time is required for learning, and they may encounter several issues that may increase the effort and the cost for the development (Usability & Comprehensibility). Therefore, developers are usually more comfortable with tools and practices that are easily adopted and integrated into their environment, which is also validated by the findings of the work conducted by Lawall and Muller (2018) on patch generation in the Linux kernel.

> **RQ1.2:** The adoption of Dependabot is motivated by its efficiency in terms of integration and configuration, the accessibility of the tool, the adaptivity to modern software development and CI/CD processes, as well as its usability and comprehensibility when it comes to documentation support.

Encountered challenges & Potential improvements With the intent to provide more insights about the challenges that developers encounter when using dependency management tools, as well as the potential improvements that allow to overcome these deficiencies, the previous survey was enriched with more inquiries to cover these aspects.

Problems & Challenges. Our previous results demonstrated that automation has allowed an improvement in terms of activity and management of dependency updates. However, automated tools can present some practical issues for users. Uncovering these problems can be highly beneficial for tool designers and researchers. We summarize in the following the major challenges experienced by developers:

– Overwhelming number of PRs and commit changes that *"pollute"* notifications. This problem intensifies with more dependencies, and developers have also mentioned the

Table 13 Dependabot popularity reasons

Reason	Selection rate
Ease of integration into GitHub projects	86.4%
Accessibility (free tool)	68.2%
Ease of configuration into CI/CD pipeline	45.5%
Availability of documentation	31.8%

problem of having many PRs for the same dependencies across different repositories. A participant stated *"Right now we get 10+ PRs for a single repository and every single of them triggers a full run of tests and other automated checks"*.
- Extensive communication and notifications (email, integrated communication tools, etc.).
- Breaking changes and the refactoring effort with major updates.
- Lack of track for the subsequent code changes from the dependency updates. When packages are updated, developers struggle to have a clear idea of what needs to be changed at the source code level. Some tools also rely on change-log updates that are not maintained in all projects.
- Absence of auto-merge feature in some bots and the new version of Dependabot. Developers have to rely on other alternatives (e.g., `github-actions` bot).
- Merge requirements and configuration of auto-merge. Developers are highly concerned about breaking changes, and the auto-merge feature can be either enabled or disabled for all dependency updates. Whereas merge requirements are preferred to be extended to include more options (e.g., update level).
- Level of test coverage to enable automatic merge. Since the adoption of CI/CD process is becoming more popular, as our participants mentioned, it can be difficult to get enough test coverage to allow for auto-merges.
- Peer requirement problem, where the new version update is incompatible with other dependent packages.
- Trust issues. Developers expressed that they often need to update dependencies by hand in local environment, as one of the participants indicated *"Even with CI integration, it's hard to feel completely confident that the upgrade PRs aren't breaking my app in some subtle way my automated tests aren't catching."*.

Features & Improvements. Our survey has also provided some valuable insights regarding the potential improvements that allow to enhance dependency management and SCA tools while overcoming the previous challenges. Developers suggested the following features:

- Group and combine multiple dependency updates into a single PR that contains an overview of the dependencies that need updates, and from which developers can choose and edit the suggested changes.
- Assistance reports and the auto generation of refactoring changes that need to be applied on source code after the new dependency update.
- Support of auto-merge with easier setup for developers. And provide more options when it comes to enabling and disabling the auto-merge feature. For example, the ability to enable security-labeled updates only; the ability to disable auto-merge for *"specific"* dependencies to be blacklisted, or allow auto-merge for patch and minor updates but never for major updates.
- Support more automation for the configuration of tasks that can be executed before, after, or during updates. For instance, participants suggested a potential feature *"if X gets updated run this command, or update X & Y together"*.
- Improve the communication qualitatively and quantitatively by providing *"better"* and *"more"* information in the generated markdown of the PR.
- Dashboard for the monitoring and tracking of the activity of dependency updates as well as the fixes for security vulnerabilities.

5.2 RQ2. Security Vulnerabilities in Dependencies

Vulnerability Identification & Fix In this research question, we investigate the strategies that developers adopt to eliminate or mitigate the threat of security vulnerabilities in their dependencies. We distinguish between *Automatic* (***A***) and *Manual* (***M***) strategies to perform the *Identification* (***I***) and *Fix* (***F***) as depicted in Fig. 8. The most adopted courses of actions that we have manually extracted are summarized as follows:

1) **Exploiting SCA tools (AI)**: This is the most utilized strategy among developers and projects on GitHub. With the availability of different tools and bots in GitHub Marketplace, developers can adopt free or paid tools that provide the most appropriate functionalities and that meet their project requirements (programming languages, supported versions of the package managers, scheduled tasks, etc.). Also, the level of automation in these tools present more facility concerning the suggested fixes that need to be applied, and more efficiency concerning the continuous checks that these bots perform on a regular basis, which requires a lot of effort and time from a human perspective. Some of the bots that we noticed developers use the most are: *Dependabot*, *Renovate*, *Snyk*, and *Greenkeeper*. These tools mainly rely on creating PRs when a vulnerable version is detected in the project, and depending on the availability of disclosed patches, they provide changes in the PR that update the package to a new safer version. However, even with the automation that these tools provide, developers react differently when it comes to applying the fixes in the generated PRs, and we summarize the different approaches that developers adopt with the PRs management through the following practises:

   (a) ***Auto-merge (AIAF):*** A new functionality provided on GitHub and that developers commonly use to increase the development velocity and reduce the manual effort on checks and reviews, is the automatic merge of PRs. With this option, developers can configure a set of requirements that need to be met before merging the PR. These requirements are usually related to code review, code coverage, successful test runs, etc. So, when a PR is created by a bot, tests are run after that along with other required configurations. In case of success, the PR commits are merged into the project automatically, otherwise, the PR remains open for further review. This type of practise is useful when adopting a CI/CD process. Some bots like *Dependabot* initially supported the *"auto-merge"* function, but then it was removed since it was argued that this function can be exploited by attackers to spread their malicious code even faster. Yet, developers can still leverage the features provided by other tools (e.g., *github-actions*) to configure the automatic merge of security PRs.

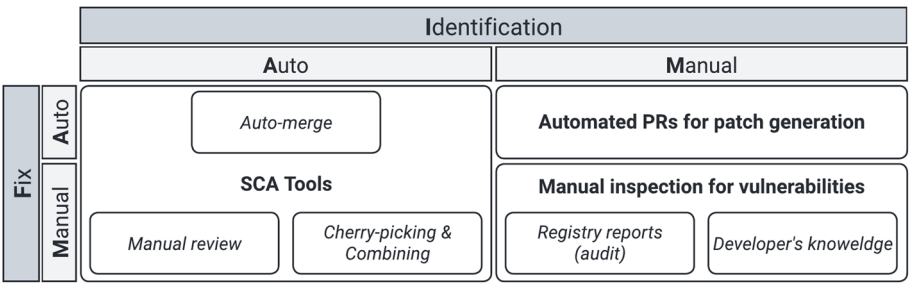

Fig. 8 Patterns of developers' practices to identify and fix vulnerabilities

(b) ***Manual review (AIMF):*** After the automatic generation of PRs by a bot, some developers prefer to manually review the suggested changes and only accept the PR when it meets the project requirements without affecting the development process in a way that causes more tasks or potentially produces malfunctions. This is usually the case for developers that are concerned by *breaking changes* and also those who are more critical about using core dependencies.

(c) ***Cherry-picking & combining (AIMF):*** Similar to the previous practise where developers manually review the PR. However, when it comes to applying the changes, they hand select a subset of commits from the PR to merge in the project. Also, when the bot creates many PRs, some developers combine multiple dependency updates into a single PR to keep the project history clean and easier to maintain.

2) **Manual inspection for vulnerabilities (MIMF)**: In several cases, developers don't rely on automated tools due to trust, awareness issues or even the fact that these tools do not meet the requirement specifications of the projects on GitHub. Therefore, they perform their own inspection manually via:

(a) ***Registry reports (audit):*** Developers can trigger the inspection process for a potential security vulnerability by relying on the public registry reports, for instance the '*npm audit*' command that scans and flags the potential outdated versions that were disclosed as vulnerable in npm registry. Note that this inspection can also be launched automatically during the installation of packages.

(b) ***Developer's knowledge:*** This refers to all the external sources that developers explore manually in order to maintain their awareness about security issues, including: vulnerability databases (NVD, GAD, etc.), published security reports and articles, test reports, security policies and advisories on GitHub Security and external analysis tools (e.g., *OWASP Dependency Check*).

However, these types of practises have many limitations in terms of time, performance and inspection coverage. For instance, developers may fail to find which version introduces the vulnerability and what version incorporates the fixes. As a result, in some cases, they completely change the dependency to a different package, or update all the dependencies in the project through a single PR hoping it would solve the security issues.

3) **Automated PRs for patch generation (MIAF)**: When developers and security researchers identify a security flaw in a package, applying manual changes to different repositories can be a tedious and time consuming task. Thus, knowing the appropriate course of action to remove the vulnerability or replace the package, they can leverage auto-generated PRs to perform hand-crafted changes. The vulnerability is identified by searching the threat signatures (e.g., package name, package version, configurations settings, etc.) in different GitHub projects through different search approaches such as *semantic code search* provided by continuous security analysis tools (e.g., LGTM [7]).

Change Practices For all the aforementioned strategies, we have identified the common set of actions applied in the patches and commit changes. These fixes can be conducted either automatically or manually to remove the vulnerabilities from the project's code. Sorted from the most to the least adopted practices in GitHub projects, security vulnerabilities in dependencies are handled by:

[7] https://lgtm.com/

- Upgrading the dependency version to the new safer one as disclosed in the vulnerability databases; changing the related attributes (the download link and the hash that ensures the package integrity); and updating the transitive dependencies.
- Leveraging *Selective Dependency Resolutions* to block the update of direct and transitive dependencies in the dependency chain to only use the ones that are specified (i.e., preventing the usage of the '^' and '∼' symbols).
- Changing the dependency to a completely different package. In cases where no safe versions of the vulnerable package are yet released, developers prefer to switch to other secure packages with similar functionalities.
- Downgrading the dependency to a lower version that is vulnerability-free. In some rare cases when patches for new versions are not yet available, a previous stable version is used instead, especially in the absence of other packages with similar functionalities to which developers can transition.
- Removing the dependency from the project when the package is confirmed to be bloated (non-used). When packages are no longer used or referenced in the project due to updates, functionality removal, or transitive dependency change, it is preferred to delete non-used packages.

RQ2: Known security vulnerabilities in dependencies are commonly disclosed in vulnerability databases and public registries. These vulnerabilities are mainly handled by upgrading the vulnerable version, restricting the version update process, or completely modifying the exposed dependency. The automation employed by developers and SCA tools enables an amelioration to the process of identification and fix.

5.3 RQ3. Security Pull Requests Management

Time to handle security PRs To explore how "*receptive*" developers are to security PRs, we start by assessing the distribution of PRs by state as illustrated in Tables 14 and 15. For PRs that are created manually by users, 74% are merged and accepted by developers which shows that they are highly receptive to the changes that are suggested by both contributors and owners. This is due to the fact that when a PR is created manually, the developer is aware of the threat that is lingering in the dependency, and the changes are usually made conforming to the project acceptance level and its requirements so that breaking changes are avoided in most scenarios. Also, 15% of the PRs are closed when the test runs fail or when the suggested changes do not meet the specified requirements, and only a minority (11%) are still open. Interestingly, we noticed that most PRs are created by contributors (90,418/115,299) and not the project owners. A *Chi-Squared* test for independence shows a statistically significant difference ($p\text{-}value < 0.001$) between the two groups in terms of the total PRs that are created. This shows that even for security management, the pull-based development model has allowed new significant contributions where developers outside the core team members collaborate to identify the security flaws and propose new fixes.

When it comes to Dependabot security PRs, only 25% are merged. From the total merged PRs, 70% (71,131/101,616) are manually handled by developers which shows their receptiveness to Dependabot security PRs, and 30% (30,485/101,616) are merged by Dependabot using the auto-merge function. The auto-merged PRs are rapidly processed where most of them (78%) are merged within few minutes, and only few of them (22%) require more than 1

Table 14 Distribution of security PRs per state and author for Dataset (2) & (3)

State	Manual			Dependabot
	Contributor	*Owner*	*Total*	
Merged	68,486	17,060	85,546 (74%)	101,616 (25%)
Closed	14,765	3,025	17,790 (15%)	176,895 (44%)
Open	7,167	4,796	11,963 (11%)	122,725 (31%)
Total	90,418	24,881	115,299	401,236

day due to project requirement (e.g., code review, code coverage, successful test runs). On the other hand, for PRs that are manually merged by developers, 53% are merged within 2 days and 47% take more than two weeks (median: 17 days, mean: 43 days). This data suggests that the auto-merge feature allows to accelerate the time to merge security PRs. However, the majority of Dependabot PRs (44%) are closed, and a considerable amount of them (31%) are still open. Even tough this data may suggest that developers are not highly receptive to Dependabot security PRs, we performed a further investigation on who closes these PRs and why they are closed. Surprisingly, we found that only 8% of the total closed PRs are rejected by developers, mainly because they lead to breaking changes, fail test runs, or affect the dependencies required by core components; and most of them (92%) are actually closed by Dependabot itself, which supports our previous assertion that developers are receptive to these PRs. From the total closed PRs, 86% were closed because they were *"superseded"*, which means a newer PR was proposed to update the vulnerable dependency to a higher version, and Dependabot automatically shuts the previous PR to introduce a more up-to-date PR. Also 9% were closed due to dependencies being updated manually, and for the few remaining (5%), it was due to peer requirement (i.e., version-specific reliance) that makes some dependencies not updatable, dependency removal from the project, and update errors.

Concerning the other tools, *Renovate* has the highest acceptance ratio (56%), and our first RQ has shown that this bot is gaining popularity along with Dependabot. On the other hand, *Depfu* has the highest rejection ratio (64%), and *Snyk* is the bot that has most of its PRs (85%) open and not handled.

For Dependabot security PRs that are still open, 122,725 PRs is a significant amount of security fixes that are pending. Therefore, we conducted a survey with GitHub repository owners from the datasets used in this study to understand the reasons that can lead developers to leave these PRs open (Section 5.4). It was carried out in a similar fashion to the one we performed in the first research question. Most developers consider the updates proposed in these PRs to be low priority for the project. Then, some developers bind this behavior to the lack of time to check and review these PRs, and finally, others consider keeping the versions of packages with low severity due to their small impact. And so, Dependabot has a

Table 15 Distribution of security PRs per state and author for Dataset (4)

State	Snyk	Renovate	Greenkeeper	Depfu
Merged	17187 (6%)	180322 (56%)	820 (28%)	6634 (22%)
Closed	23170 (8%)	83631 (26%)	1033 (36%)	19507 (64%)
Open	233596 (85%)	56681 (18%)	1041 (36%)	4170 (14%)
Total	273953	320634	2894	30311

significant impact on managing and fixing known security vulnerabilities in dependencies, and developers are majorly receptive to its PRs.

After analyzing how receptive developers are to security PRs, it is crucial to tackle their responsiveness and measure how long it takes to handle them. In the pull-based development model, developers need enough time to review and check the changes before they make the decision to accept (merge) the PR or refuse (close) it, and such decision is more impactful when the PR is security-related. By visualizing a violin plot in Fig. 9 to compare the time required to handle (merge or close) a PR created by Bots against those created manually by users, we observe that most Dependabot PRs are merged in less than 24 hours (median: 1 day, mean: 16 days) and those created by developers are merged even faster (median: 1 day, mean: 7 days). As for the other tools, Renovate performs the best when it comes to merging PRs (median: 0 day, mean: 9 days), then Depfu (median: 1 day, mean: 9 days) and Snyk (median: 1 day, mean: 23 days), and finally Greekeeper's PRs takes the longest to be merged (median: 2 day, mean: 62 days).

Regarding closed PRs, changes that are suggested by Bots tend to take longer to be closed overall (median: from 11 to 160 days, mean: from 58 to 295 days) compared to the manual

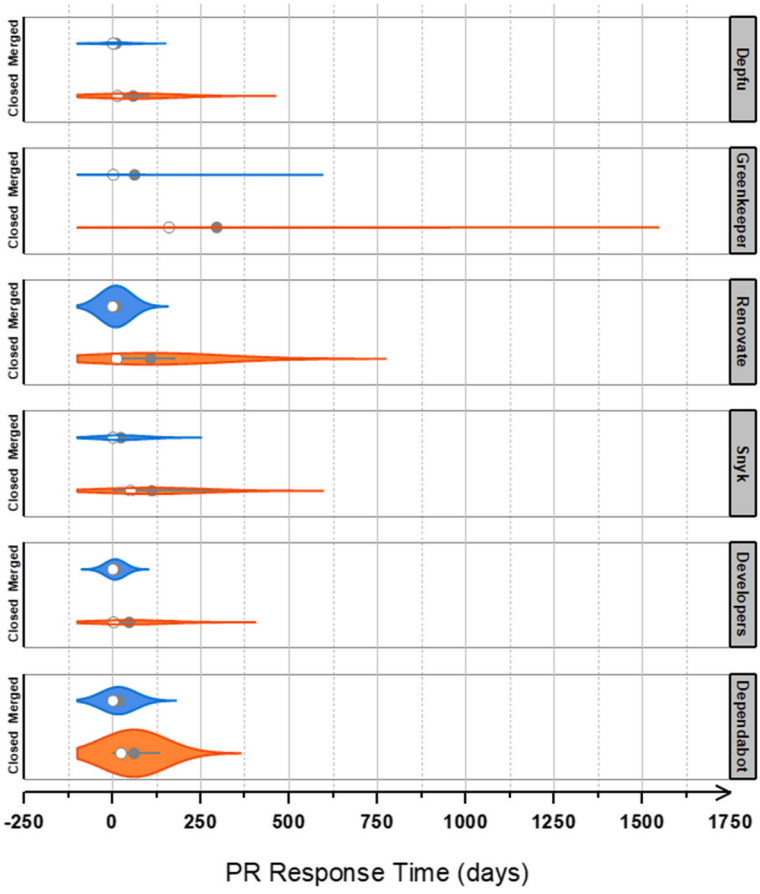

Fig. 9 Violin-plot for the time to handle security PRs

changes made by developers (median: 3 days, mean: 47 days). However, some Dependabot security PRs are handled automatically by Dependabot itself and others are handled manually by developers. As a result, we perform a further investigation on whether the auto-merge function allows to accelerate the time to merge security PRs.

Based on Fig. 10 that shows box-plots with a normal distribution for the merge rate and close rate of security PRs created by Dependabot and developers, but also separating those that are handled automatically and manually, we can make the following observations: *(1)* PRs are merged much faster than they are closed whether they are created by Dependabot or by users, which means PRs containing security fixes are either assessed and integrated quickly, or left hanging open until being closed as found by Alfadel et al. (2021); Also, *(2)* when it comes to merging these PRs, Dependabot performs the best with the auto-merge function (median: 0, mean: 5.67), and interestingly, developers merge their PRs faster (median: 1, mean: 7.52) than those created by Dependabot (median: 2, mean: 20.48). However, *(3)* pertaining to the closed PRs, the ones created by developers are closed faster (median: 3, mean: 47.00), and Dependabot PRs take longer to be closed whether automatically (median: 24, mean: 59.42) or manually (median: 36, mean: 81.70), this is due to the fact that users are familiar with their project requirements and suggest changes that comply to its acceptance level, which makes reviews and tests run faster and without conflicts, unlike Dependabot PRs that suggest general changes on the dependency version that can potentially cause some breaking changes or some refactoring on some components to support the newer version. In addition, the auto-closed PRs are mainly dismissed after being left open for a long time until newer changes are available.

Finally, we want to study the evolution of merge rate and close rate of these security PRs in order to assess if the developer's reaction to these PRs had changed over time. As shown in Fig. 11 that illustrates box-plots for the merge rate and close rate of PRs created by

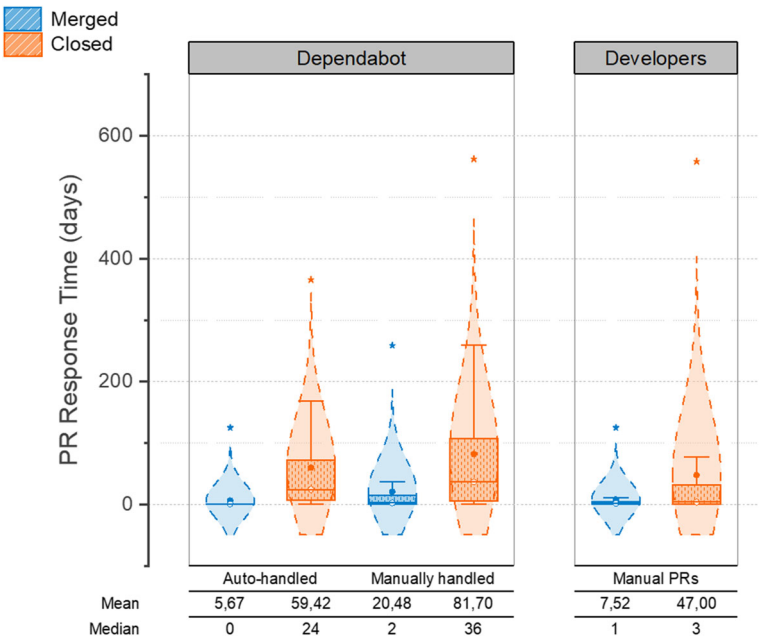

Fig. 10 Box-plot for merge and close rate of security PRs

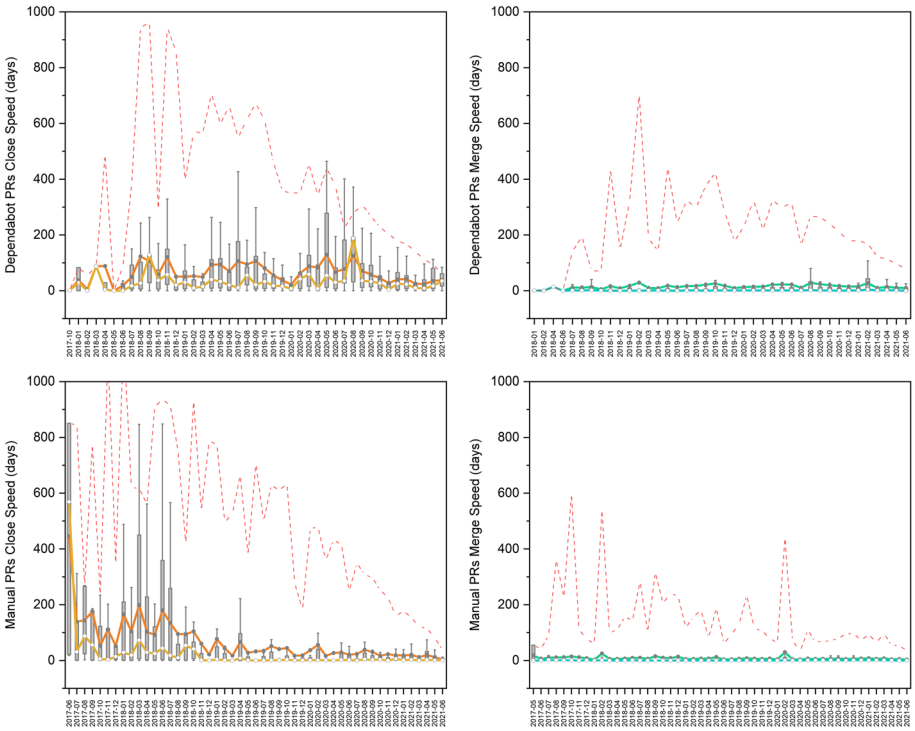

Fig. 11 Box-plot for the evolution of merge and close rate

Dependabot and developers grouped by month of creation with connected lines for the mean and median values, we notice that the close rate was initially high especially with the PRs created by developers, but then it decreases through the course of the last years (especially in terms of max values) which suggests that developers became more familiar with these PRs. Concerning the merge rate, the values keep a week variation, in terms of mean and median, which means that the time to merge PRs does not change for both developers and Dependabot.

> **RQ3.1:** Developers are highly receptive and responsive to security PRs where the acceptance time is no more than 1 day, and the automation provided by Bots allows to maintain and improve the merge rate. The auto-merge function accelerates the merge rate into few minutes. However, developers tend to handle their PRs much faster than those created by other tools. And most of Dependabot PRs are automatically closed due to the release of newer versions of vulnerable dependencies after PRs are left open for long period of time.

Threat lifetime In this subsection, our focus is to determine how long the threat in dependency vulnerabilities remain silent and undetected in projects, and also analyse this threat in terms of persistence, popularity and exposure range based on the severity level of vulnerabilities and the update level of their patches. Based on Fig. 12 that demonstrates violin-plots for the

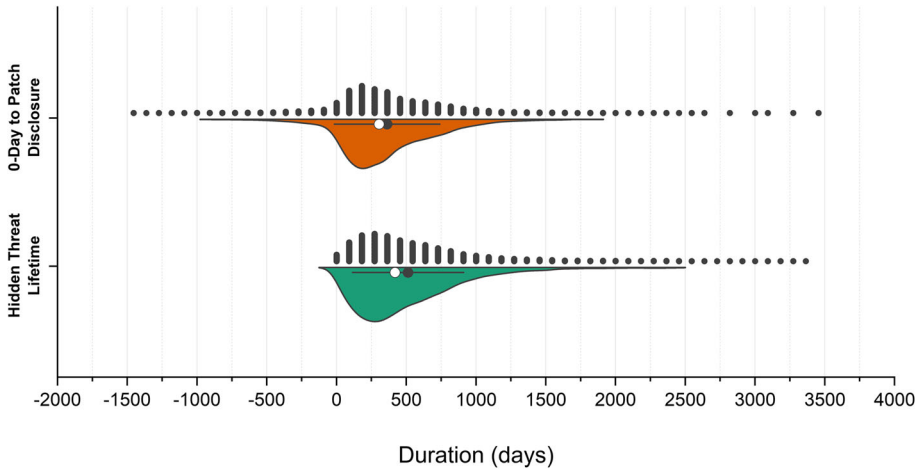

Fig. 12 Violin-plot for threat lifetime and fix delay

hidden threat lifetime and the fix delay of vulnerabilities, we notice that the threat persists unknown in GitHub projects on average for **512** days (median: 419 days) which is a huge window of exposure that can affect many users. When it comes to releasing patches, they can be already disclosed before adding the vulnerable version into the project, hence the negative values for 0-day to patch disclosure in the Fig. 12. In such cases, Dependabot immediately creates PRs to fix them; However, the time required to publish these fixes, after adding the vulnerable version into the project, is on average **362** days (median: 305 days), this is due to the fact that these patches are made after inspecting the code manually by security experts. Also, remarkably, the close median and mean values between the hidden threat lifetime and the 0-day to patch disclosure time make small time difference between these two events, which suggests that changes are made quickly in GitHub projects soon after adding the patches in the CVE databases (GAD, NVD, etc.), which shows one of the advantages of SCA tools in general, and Dependabot in particular. However, the detection is only for *known* security vulnerabilities; Besides, the analysis and the patches still rely on the manual effort of security experts, which calls for the need of an automated real-time analysis tool that can identify new potential vulnerabilities in these dependencies as soon as they are added to the project.

As an indication of how serious the impact of such vulnerabilities can be, we try to examine the average threat lifetime and the number of PRs created to fix these vulnerabilities based on the severity level. Table 16 shows that most vulnerabilities affecting GitHub projects are

Table 16 Threat lifetime based on the severity level

Severity level	# PRS	Average threat lifetime (days)
High	6587	540
Moderate	2607	481
Critical	1653	427
Medium	691	653
Low	502	595

those with *high* severity and they last for 540 days on average without being discovered, then comes the *moderate* and *critical* level; On the other hand, vulnerabilities with *low* severity are the least occurring, which highlights the crucial and significant impact of serious threats that occur very often. Interestingly, vulnerabilities with the highest severity level (*critical*) have the smallest threat lifetime (427 days), so they seem to receive fixes more quickly compared to other levels considering their severe outcomes.

Tables 17 and 18 demonstrate the threat lifetime and its prevalence based on the ecosystem and the severity level. We observe that across different ecosystems, the vulnerabilities that linger the longest without being discovered and/or fixed are those with *low* and *medium* severity. However, the vulnerabilities that seem to occur the most are those with high severity, and *npm* (*Javascript & Typescript*) is the most affected with them compared to other ecosystems. Interestingly, these findings are also supported by the results of the study conducted by Decan et al. (2018) in the npm ecosystem.

Regarding the proposed patches, the update changes in the newer dependency version vary from patch, minor, to major level. Table 19 shows an interesting pattern, the number of PRs created to deal with the dependency vulnerabilities decreases as the update level gets higher, such that most fixes are made on patch level, and major update fixes are the least occurring in GitHub projects. Similarly, the average threat lifetime of these vulnerabilities increases as the update level gets higher, such that patch level updates are published sooner after the introduction of the vulnerability into the project, and major updates take the longest time to be released (957 days on average), this is due to the fact that major updates usually induce huge changes (incompatible API changes according to *semantic versioning*), and therefore they require more time to be achieved.

To perform a further inspection across the different ecosystems, we represent in Tables 20 and 21 the threat lifetime and its prevalence based on the ecosystem and the update level. The data suggests that regardless of the ecosystem, the vulnerabilities that remain undiscovered and/or unfixed for a longer period of time are those requiring *major* updates. This is due to the common concern of breaking changes and the refactoring effort that comes with it. On the other hand, most of the vulnerabilities in dependencies are fixed with patch releases. This

Table 17 Average threat lifetime based on severity level and programming language

Programming language	Low	Medium	Moderate	High	Critical	Total
JavaScript	581	623	477	524	433	507
TypeScript	587	589	469	510	397	493
Ruby	658	713	516	581	477	576
Python	431	623	477	486	372	474
HTML	611	723	**625**	496	384	529
PHP	**924**	653	383	481	423	476
Vue	535	704	511	452	356	465
Java	523	660	370	470	**590**	472
CSS	698	561	531	**619**	483	**584**
Rust	354	234	429	530	338	464
Multiple	512	**2019**	347	477	532	483

Table 18 Threat dominance based on severity level and programming language

Programming language	Low	Medium	Moderate	High	Critical	Total
JavaScript	**233**	**246**	**1122**	**2907**	**834**	**5342**
TypeScript	105	114	577	1513	337	2646
Ruby	63	217	332	823	187	1622
Python	18	20	140	266	58	502
HTML	23	29	112	269	66	499
PHP	14	20	96	208	34	372
Vue	11	10	86	191	44	342
Java	15	10	51	120	23	219
CSS	10	19	40	139	35	243
Rust	2	4	26	77	24	133
Multiple	8	2	25	74	11	120

can be an indication for an encouragement to make patch releases to fix vulnerabilities and not including them in major releases when possible since it can lead to a better and wider adoption of less vulnerable dependencies.

Another aspect related to threat longevity in dependencies that we want to tackle is: the popularity/dominance of vulnerability types. Table 22 enumerates the most exploited vulnerabilities in the dependencies of GitHub projects with their average threat lifetime (ATL). These vulnerabilities represent the most popular malicious behaviors that attackers exploit in dependencies, they dominate 80% of the total PRs created by Dependabot. We notice that most of the occurrences are those with *Prototype Pollution* vulnerability that remains undetected for 478 days on average. The popularity of this injection threat is due to not only its simplicity to tamper the logic of the application but also the fact that it can allow hackers to further conduct other subsequent attacks (e.g., Denial of Service, Remote Code Execution). Besides, it targets the JS runtime (*npm* ecosystem) which is highly adopted among developers. It is also worth noting that *Cross Site Scripting (XSS)* vulnerability is the one with the highest lifetime (ATL) of 629 days, which may suggest that this type of threat is harder to identify manually and/or requires a lot of time when implementing the fixes. On the other hand, the vulnerability that seems to be fixed the fastest is the *Usage of broken/risky cryptographic algorithms*. In these cases, we hypothesize that the security flaw is clearly apparent at the source code level and the fixes are usually predefined by the appropriate standard algorithms that are commonly used.

Table 19 Threat lifetime based on the update level

Update level	# PRs	Average threat lifetime (days)
Patch	8316	435
Minor	4808	580
Major	713	957

Table 20 Threat lifetime based on the update level and programming language

Programming language	Patch	Minor	Major	Total
JavaScript	444	564	938	506
TypeScript	436	560	770	491
Ruby	424	674	1046	**577**
Python	407	569	943	495
HTML	426	602	1141	530
PHP	425	438	**1348**	468
Vue	390	495	1115	458
Java	411	574	1138	522
CSS	**472**	**716**	724	574
Rust	449	432	870	452
Multiple	450	702	1282	535

RQ3.2: Dependabot enables quick reaction to vulnerable dependencies after disclosing them in vulnerability databases. However, developers tend to use some vulnerable dependencies even after their disclosure. The vulnerabilities remain hidden for 512 days on average, and most of them are labeled with high severity levels. As for the fix delay, repairs are disclosed after 362 days on average, and mostly perform patch level updates.

5.4 RQ4. Merge Decision and Merge Rate

Merge decision In order to identify the factors that potentially have correlation with the decision to merge security pull requests, we perform a statistical analysis as described in Section 4. As a reminder, a PR can be either merged or closed (we excluded the open PRs since it would be hard to draw conclusions as the PRs are not yet handled). We perform our analysis on Dependabot PRs and those created manually by users. Note that PRs that

Table 21 Threat dominance based on the update level and programming language

Programming language	Patch	Minor	Major	Total
JavaScript	**3671**	**2081**	**251**	**6003**
TypeScript	1902	943	149	2994
Ruby	1016	802	167	1985
Python	339	202	33	574
HTML	336	185	35	556
PHP	297	164	20	481
Vue	235	133	17	385
Java	155	92	20	267
CSS	171	108	13	292
Rust	94	65	4	163
Multiple	100	33	4	137

Table 22 Most exploited vulnerabilities in dependencies

Vulnerability (Malicious behavior)	# PRs	ATL (days)
Prototype pollution	4525	478
Regular expression denial of service	1627	561
Denial of service	539	543
Signature malleabillity	435	499
ReDoS and prototype pollution	430	353
XSS vulnerability	299	629
Command injection	280	284
Path traversal	243	536
Arbitrary code execution	236	375
Resource allocation without throttling	225	276
Using risky cryptographic algorithm	181	197
Potential memory exposure	174	619
OS command injection	172	440
Remote memory exposure	150	623
Arbitrary file overwrite	143	442
Possible information leak / Session Hijack	126	464
Remote code execution	117	368

are auto-merged or superseded by other PRs are included in our analysis. The management of these PRs can potentially be influenced by some factors. For example, auto-merged PRs usually have to pass test runs and code coverage requirements which can potentially be influenced by the amount of changes introduced in a PR. Thus, it is important to include them in our investigation to uncover these influences. Table 23 shows the results of *Logistic Regression* and *Chi-squared* tests for Dependabot security PRs sorted by features importance

Table 23 Tests results on merge decision for dataset (2)

Feature	Coef.	z	p-value
`# comments`	−2.1931	−202.477	< 0.001
`discussion_size`	−0.7786	−97.675	< 0.001
`# assignees`	0.2704	42.805	< 0.001
`# additions`	−0.2018	−36.532	< 0.001
`# changed_files`	0.1328	19.490	< 0.001
`commit_history`	0.6187	14.600	< 0.001
`latest_activity`	−0.0782	−14.361	< 0.001
`# open_issues`	0.0577	7.637	< 0.001
`age`	0.0193	3.492	< 0.001
`# commits`	−0.0125	−1.922	0.055
`# watchers`	−0.0104	−1.632	0.103
`size`	−0.0039	−0.682	0.495
`update_level`	–	2105.170	< 0.001
`severity`	–	224.866	< 0.001

(i.e., `z-value = coef./std err`), and similarly Table 24 demonstrates the results of *Logistic Regression* and *Chi-squared* test analysis on Dataset (3).

Pull Request Features. The number of comments in a PR (`# comments`) along with the discussion size (`discussion_size`) have the most significant correlation with the state of Dependabot PRs and those created manually by users. The coefficient sign suggests that the more comments are made in a PR and the bigger the discussion, the more likely a PR is closed. This is due to the fact that most of Dependabot PRs are automatically closed due to newer versions and changes that are published before merging the PR. In most of these cases, Dependabot informs the developers at each step about the upcoming changes and the actions that were performed through comments (e.g., superseded PRs, removed dependencies). So, the data suggests that the merge decision is supported by descriptions of small size, and we hypothesize that developers perceive Dependabot communication as *'too much'*, or they may hesitate to accept PR changes with large descriptions as they can indicate more changes in the project. Interestingly, the number of assignees of a PR (`# assignees`) has a significant correlation on the merge decision so that the more developers are assigned to it, the more likely it would be merged. Also, as we speculated, a high number of additions (`# additions`) and file changes (`# changed_files`) in a PR can cause the rejection of the suggested patches, this is due to the fact that developers are concerned by breaking changes in their projects and the more additions/deletions are made in a PR, the more effort is required to maintain the project's functioning and more test and reviews are performed.

Repository Features. As demonstrated by several previous studies (Gousios et al. 2014b; Kononenko et al. 2018a; Soares et al. 2015b), the repository characteristics also have an influence on the PR merge decision. Our data suggests that the more a project is active (`latest_activity` and `# open_issues`), the more likely a Dependabot PR is merged, this hypothesis is also supported by our previous finding where Dependabot automatically closes PRs after being left open for so long until newer changes are suggested, therefore, when developers are actively reviewing and considering the changes in these PRs, they

Table 24 Tests results on merge decision for dataset (3)

Feature	Coef.	z	p-value
`latest_activity`	−14.0161	−65.587	< 0.001
`# assignees`	0.4259	22.609	< 0.001
`discussion_size`	−0.2729	−17.569	< 0.001
`acquaintance`	0.2142	12.903	< 0.001
`# watchers`	−0.1185	−6.617	< 0.001
`# additions`	0.0741	4.417	< 0.001
`# public_repos_gists`	−0.0606	−3.708	< 0.001
`age`	0.0573	3.570	< 0.001
`# comments`	−0.0576	−3.568	< 0.001
`size`	−0.0413	−2.824	0.005
`commit_history`	0.1735	1.478	0.140
`# changed_files`	0.0202	1.165	0.244
`# followers`	−0.0137	−0.712	0.476
`# open_issues`	−0.0114	−0.676	0.499
`# commits`	0.0070	0.396	0.692
`author_association`	–	14.792	< 0.001

merge the changes quickly before newer versions are available. This observation also applies to manual PRs, since the PRs are created and handled manually by developers and neither bots nor tools are involved in the process, we hypothesize that developers are more active and involved in the process of fixing the vulnerabilities in their dependencies which has a relationship with their decision on handling PRs. We can also support this observation with the number of developers who watch and observe the repository (`# watchers`) having a significant correlation with the merge decision. The repository age (`age`) also seems to have a statistically significant correlation with the merge state, so that mature (i.e., older) projects favor merging Dependabot and manually created PRs. This can be due to the fact that developers are more familiar with the tool, or the maturity level makes the project more robust and adaptive to the changes that are suggested in security PRs. In addition, the tests also suggest that small size repositories (`size`) are more likely to accept security PRs created manually by developers.

Dependency Features. For dependency features, the *Chi-squared* test yielded very low p-values, so we can safely reject the null hypothesis that the update level and the severity have no relationship with the merge decision. In other words, Dependabot PRs that handle dependency vulnerabilities with different update levels and severity levels are statistically different in terms of merge state (merged vs. closed). Besides, as we have previously demonstrated with the Average Threat Lifetime of vulnerabilities, developers seem to prioritize PRs with critical levels due to their serious impact, as well as patch level updates since they don't introduce any breaking changes.

User Features. Interestingly, the commit history (`commit_history`) is highly correlated with the merge decision of PRs created by Dependabot, this can be an indication to the influence of the bot configuration in the project and its past contribution, as well as the adoption of the auto-merge function that enables frequent updates. Indeed, the coefficient sign suggests that the larger the commit history gets, the more likely the PRs are merged. However, we do not observe this correlation when it comes to PRs created by developers. Nonetheless, the acquaintance of the developer who created the PR (`acquaintance`) and its contribution (`# public_repos_gists`) are significantly correlated with the merge decision, these observations were also noticed in a similar study (Gousios et al. 2014b) on the acceptance of PRs in the pull-based development model, and since our dataset contains security related PRs, it is expected that developers put more emphasis and priority on the merge activity. Unlike previous work (Gousios et al. 2014b), our data suggests that the author association can influence the merge decision of manually created PRs. By comparing the acceptance rates, owners have 82% of their PRs merged and contributors have 79% despite the fact that the latter have more activity and create more PRs. Also a *Chi-Squared* test for independence shows a statistically significant difference ($p\text{-}value < 0.0002$) between the two groups. Since the created PRs are security related and act on crucial aspects of the project, it would be expected that developers trust core team members more than other contributors.

> **RQ4.1:** The management of security pull requests can be influenced by the project characteristics and its activity, developers experience and their collaboration, the amount of changes induced by the PR, as well as the severity level of the vulnerabilities and the update level of their fixes.

Merge rate Concerning the time required to merge a security PR, Table 25 shows the results of *Multiple Regression* and *Kruskal-Wallis* test analysis on the merge rate for auto-generated PRs by Dependabot, and Table 26 illustrates the results of *Multiple Regression* and *Wilcoxon*

Table 25 Tests results on merge rate for Dataset (2)

Feature	Coef.	z	p-value
# commits	1.6082	9.898	< 0.001
# changed_files	1.2759	7.728	< 0.001
commit_history	−5.4416	−5.472	< 0.001
# comments	0.7519	4.653	< 0.001
discussion_size	0.6749	4.160	< 0.001
age	−0.6404	−3.910	< 0.001
size	−0.6161	−3.276	0.001
# additions	0.2599	1.597	0.110
latest_activity	−0.2039	−1.263	0.207
# open_issues	−0.2479	−1.085	0.278
# assignees	−0.1043	−0.647	0.517
# watchers	−0.0785	−0.383	0.701
severity	–	16.791	0.002
update_level	–	8.498	0.014

rank-sum (*Mann Whitney U*) test analysis on the merge rate for manually created PRs in Dataset (3). Please note that, similar to our previous analysis on merge decision, PRs that are auto-merged or superseded by other PRs are also included in this analysis. We aim to uncover the potential influences of different factors on the reaction to these PRs and the response time as well.

Pull Request Features. The features with the most significant importance are related to the changes introduced in security PRs, that is, the number of commits (`# commits`) for both datasets, and the number of changed files (`# changed_files`) for Dependabot PRs.

Table 26 Tests results on merge rate for dataset (3)

Feature	Coef.	z	p-value
# comments	4.1299	18.532	< 0.001
# public_repos_gists	3.4833	15.432	< 0.001
age	2.3858	10.257	< 0.001
# commits	0.8149	3.610	< 0.001
discussion_size	0.7945	3.251	0.001
size	−0.4939	−2.217	0.027
# watchers	−0.5248	−2.104	0.035
commit_history	12.4632	1.886	0.061
acquaintance	−0.2772	−1.158	0.247
# changed_files	0.2295	0.983	0.326
# followers	−0.1798	−0.814	0.416
latest_activity	0.1535	0.714	0.475
# additions	0.1190	0.527	0.598
# open_issues	−0.0707	−0.282	0.778
# assignees	0.0215	0.093	0.926
author_association	–	7.903	< 0.001

This is due to the fact that more changes can potentially induce more time for code reviews, and having more changes (especially breaking changes) increases the chance for test runs to fail, which also increases the effort and time to merge the changes and update the dependent components in the project. In addition, when these PRs are created manually by users, the suggested changes are not limited to updating the dependency version only. As shown in ***RQ2*** (Section 5.2), developers adopt different courses of action that impact the project differently (upgrade, downgrade, change dependency, etc.). Therefore, more changes lead to more commits which entails more time to merge. This observation is also supported by the coefficients sign. As an additional investigation, we have separated the PRs in dataset (2) into: PRs that are auto-merged v.s. PRs that are manually-merged. Interestingly, we have found that whether PRs are automatically merged by Dependabot or manually merged by developers, these PRs usually suggest fixes in one commit (median: 1, mean: 1) by mostly changing one file (median: 1, mean: 1.1) which refers to Dependabot changing the dependency files (e.g., `package.json`). However, the PRs that are merged manually by developers tend to introduce more changes on average (Mean [`# additions`]: 45) compared to those auto-merged by Dependabot (Mean [`# additions`]: 20) which demonstrates that Dependabot automatically handles PRs affecting dependency files while developers are involved in the process of merging PRs when more changes are required. Furthermore, our statistical tests with Logistic Regression show a high correlation between the changes in a PR (`# commits`, `# additions`, `# changed files`) and whether the PR is merged automatically or manually. Indeed, the auto-merges co-occur with less changes in a PR (based on the test coefficient sign) which validates our previous observation. Another interesting aspect is the communication within the repository. Indeed, the number of comments (`# comments`) and the PR body size (`discussion_size`) are highly correlated with the merge rate of Dependabot PRs and those created manually by users. According to the coefficients sign of these features, the more comments are posted and the more discussions are extensive, the longer it will take to merge these PRs. Large discussions and the numerous comments can indicate: the active feedback of developers as communication is required to discuss the strategy to fix the dependency vulnerabilities; the extensive comments generated by Dependabot; or the configuration commands posted from Dependabot's actions. Nonetheless, the data suggests that the longer the discussion goes, the more time is required to merge.

Repository Features. Concerning the project characteristics, fast merge seems to co-occur with larger size (`size`) repositories for both datasets. We hypothesize that larger projects have more contributors that work on reviewing and managing PRs. The age (`age`) of the repository is also highly correlated with the merge rate: for Dependabot PRs that are merged faster in mature repositories, it can be an indication of the familiarity of the projects with Dependabot PRs and their adaptivity to integrate new changes, and this adaptivity is acquired through long time experience; as for manually created PRs, recent projects merge their security PRs faster than older ones unlike the management of Dependabot PRs that are favoured by the latter. Interestingly, the number of watchers (`# watchers`) is also an important feature to consider in merge rate for manual PRs, as more developers watch and monitor the project issues, security PRs are handled much faster due to their activity and implication in the review process.

Dependency Features. Another interesting finding is the strong correlation between the merge rate and both the severity and update level of the dependencies. As we have discussed previously, the hidden threat lifetime of vulnerabilities shows that PRs with higher levels of severity are handled faster than others (especially those with critical level), and the same applies to PRs with low update levels.

User Features. Interestingly, the commit history (`commit_history`) is one the features that are mostly correlated with the merge rate of PRs created by Dependabot such that fast merges co-occur with larger commit history. This finding also supports our observation with the merge decision which demonstrates the influence of bots' configuration and the adoption of the auto-merge function as well as the extent to which developers are familiar with such updates. Regarding the developer's characteristics, the number of created repositories and gists (`# public_repos_gists`) is one way to measure his contribution. This feature is highly correlated with the merge rate, so that the more repositories and gists a developer has, the longer it takes to merge security PRs. This observation is also supported by the study that was conducted by Zahan et al. (2022), the authors stated that one of the weak link signals in the software supply chain is the *overloaded maintainers*. This refers to maintainers who own many projects and packages, and therefore may not have enough time to maintain the security of all their projects. Our survey results also validate this finding, as developers indicated that some of the reasons that lead them to leave Dependabot security PRs open are the *"manual effort"* and *"insufficient time to check and review the PR"*. The author association (`author_association`) has a statistically significant correlation with the merge rate as well based on the *Mann Whitney U* test. The data suggests that PRs that are created by project owners are merge within 5.4 days on average, whereas, those created by contributors are merge within 7.7 days on average.

RQ4.2: The fast merge co-occurs with small changes in the PRs in terms of commits and changed files that lead to more refactoring effort, the efficient communication between developers, high severity vulnerabilities, and low update levels. The merge rate also fluctuates depending on project's size and maturity level, as well as the developer's workload and association.

Survey discussion In order to substantiate the results of our statistical test analysis, we discuss in the following the responses of our survey with GitHub developers regarding the reasons that lead them to close security PRs created by Dependabot, and also the reasons behind leaving some PRs open and neither merge nor close them. From the total responses, 94.4% were from developer team and 5.6% were from the research team. When it comes to the programming languages, 72.2% are mostly familiar with TypeScript, 66.7% use JavaScript, 44.4% are using Python, 33.3% are using HTML/CSS, 27.8% are familiar with Ruby, and 5.6% to 16.7% of the participants are familiar with other programming languages (Go, Java, C, etc.). These developers come with different background experiences: 16.7% have less than 5 years experience, 50% have 5 to 9 years, 27.8% with 10 to 20 years, and 5.6% have more than 20 years experience. We have confirmed through this survey that most participants (77.8%) are highly receptive to security PRs created by Dependabot, and they frequently (66.6%) check and review these PRs.

Concerning the reasons that lead to the rejection of Dependabot security PRs, Table 27 shows the percentage of the selected responses for *Why do developers close / reject security PRs ?* The data suggests that the most common reason is the concern of breaking changes, and this finding supports our statistical test analysis where features related to changes in the PR (`# commits, # additions, # changed_files`) were found to be highly correlated. Another reason that leads developers to close security PRs is the test runs that fail. Some developers also close security PRs when the vulnerable dependency version is required by another package, or when the dependency is already updated manually. In some cases, the PRs are considered to be lacking information about the vulnerability which leads to hesitating

Table 27 Reasons for the rejection of security PRs

Reason	Responses
Breaking changes	56%
Test runs fail	50%
Required dependency version	33%
Previously updated dependency version	22%
Lack of information in the pull request	17%
Availability of newer dependency version	17%
Previously removed dependency version	11%
Bloated dependency	6%
Combining multiple PRs for different dependencies	6%

to merge the suggested updates. Occasionally, developers close Dependabot security PRs when newer dependency versions are available, or when the dependency is already removed. However, our survey also showed that these PRs can be created for dependencies that are not used (*bloated*), which presents on one hand a potential improvement for Dependabot and SCA tools; and on the other hand, it hints to a potential threat that developers need to take into consideration. Some studies (Ohm et al. 2020b) showed that malicious code can be triggered during installation (through install scripts) and not only at runtime. Therefore, a bloated dependency can present a threat solely by remaining in the dependency file. Finally, in some rare cases, developers combine multiple PRs into one PR that applies all changes while closing the others since PRs are raised per single dependency.

When it comes to security PRs that are left open, Table 28 demonstrates the percentage of the selected reasons that lead to leaving these PRs open. Most participants consider the updates in these PRs to be low priority. Other developers claim that the PRs are not yet handled due to the lack of time to review and check the PR. Interestingly, half participants deem the impact of keeping the current version to have a low severity. Finally, some developers provided other reasons that include: lack of information in the PR, high frequency of updates, the manual effort especially when several repositories have these PRs, and the lack of meaningful / interpretable information in some measures such as the CVSS score.

6 Implications

Our findings provide viable, empirically justified implications for improving dependency management in general and handling the security vulnerabilities in project dependencies.

Table 28 Reasons for not handling security PRs

Reason	Responses
Low priority update	78%
Not enough time to review and check the pull request	67%
Low severity for the impact of keeping the current version	50%
Other reasons	22%

➤ ***Tool designers.***

Bot-Human Interaction. Several dependency management bots and Software Composition Analysis tools allow users to avoid known security vulnerabilities that were already discovered in the wild. However, it is crucial that tool creators consider improvements for the bot-human interaction. Our survey revealed that many developers consider Dependabot alerts to be overwhelming and may pollute their project notifications. Especially when several dependencies are found to be vulnerable in a small period of time. A potential improvement is to combine the different changes into a single PR (that developers can then edit and select the appropriate changes) instead of making a separate PR for each dependency issue. In addition, the information provided in the PRs can have a huge impact on handling the security vulnerabilities, and some of the developers that were surveyed emphasized on providing more and better views on the importance of the suggested fixes; One of our participants mentioned that *"Nothing meaningful in the CVSS score to tell me whether it's worth it"*.

Bots' Deficiencies & Improvements. Another major concern is the breaking changes, the high frequency of updates and the huge manual effort, especially when multiple repositories receive PRs for the same dependency, and each PR triggers a full run of tests and other automated checks. Tools should assist developers in case of major updates are needed by locating the code fragments that need to be modified for the newer dependency. This information can also be induced from other repositories that already applied the changes. It is also recommended to support the auto-merge feature and offer restriction options on specific dependency versions (e.g., allow patch and minor updates, but not major updates), so that PRs that are less likely to cause breaking changes can be handled automatically, and developers can solely focus on those that require refactoring effort.

Threat Lifetime & Fix Delay. This study also shows that even by using one of the most popular tools (i.e., Dependabot), vulnerabilities still linger in GitHub projects for a huge amount of time, which increases the window of exposure and the impact of these attacks. Therefore, more effective and efficient tools are required to identify the vulnerabilities sooner using available package data (e.g., source code, artifacts, metadata, etc.) and without relying on information that may take long period of time to be found and disclosed due to the manual effort of security experts involved in the process (e.g., vulnerability databases).

Features to Improve Automation. Concerning the tools that are currently being used, users do not react quickly to their alerts, and the fixes require long time to be merged and are sometimes rejected. Through our study, we have identified several features and factors that tool designers should consider. This includes: *efficiency* with the configuration and the integration into development environments, the *accessibility* of the tool to support wider adoption, *adaptivity* to the new development processes, *usability* and *comprehensibility* with the documentation support, and the *generalizability* to different ecosystems and technologies.

➤ ***Repository maintainers / owners.***

Security Awareness. Security is a major concern for many users. And since third-party packages are dominantly used in software development, maintainers and repository owners share the responsibility to ensure the safety of their products. Our study revealed that some maintainers do not use any tools or bots to manage dependency vulnerabilities due to trust issues or project requirements. However, they should maintain a regular level of

awareness. They can perform manual inspections and audits that are supported by package managers. They also should refer to external sources such as vulnerability databases, security reports, and security advisories.

Limiting Threat Propagation. Another concern that can be addressed by maintainers is the fix delay. After a vulnerability is discovered in a package, fixes take time to be published through a new version. During that window, users are still exposed to the threat and the package functionalities are often required for the proper functioning of the dependent software. Therefore, in the absence of a safe version that corrects the vulnerabilities in a given dependency, advisories can suggest a substitute package that developers can use until new fix patches are released. Our data related to the hidden threat lifetime also suggests that developers use vulnerable packages even after patch and fix disclosure. Package maintainers should strive to limit the propagation of disclosed vulnerable versions of the packages, and inform the users during installation about the potential threats.

Vulnerability Listing. We have provided through our analysis, a list of the most exploited vulnerabilities in dependencies so that maintainers are informed about the most common strategies to inject malicious code. These insights allow to avoid the flaws that lead to such attacks and to be prepared with the countermeasures in the early stages. Additionally, this information allows security teams to evaluate the safety of the source code and their security measures against the most common attacks in a similar fashion to *OWASP Top Ten* [8].

➤ ***Developers.***

Transparency and Collaboration. Similar to repository owners, developers ought to maintain an awareness about the safety of the components that they integrate into their projects. Besides, they should not assume that suppliers will always handle security concerns for them. Also, when handling the security vulnerabilities manually, our ***RQ4*** showed that the discussion size and the number of comments may act on the merge decision, thus developers should be concise and *make a long story short* when making contributions to fix vulnerabilities.

Responsiveness to Fixes. Regarding the adoption of Dependabot, the fact that most of the auto-generated security PRs are closed by Dependabot itself mainly because they become superseded implies that developers leave these PRs open for too long up until newer versions are released.

Limited Dependencies & Separation. Based on our manual analysis, we also found that developers adopt different strategies to maintain the security of their projects. But, one the most important incidents that were perceived is the removal of the unused (bloated) dependencies after they were signaled to be vulnerable. It is a golden rule to keep the dependency graph clean from redundant and bloated dependencies, and only use packages that are needed. Otherwise, it would create opportunities for attackers to inject malicious code, and opens more flaws that expose the project and its dependents.

7 Related Work

This section presents the works and studies most related to ours in the context of vulnerabilities that lead to Software Supply Chain Attack (SSCA), security and contribution in the pull-

[8] https://owasp.org/www-project-top-ten/

based development model, and software bots that support the development and management of open-source software.

Software Supply Chain Attack In an interesting work, Ohm et al. Ohm et al. (2020b) conducted the first manual dataset analysis - *to the best of our knowledge* - for open-source malicious packages. In addition to the constructed dataset, and following an iterative manual approach, the authors defined attack vectors for the injection of malicious code. They also defined the circumstances in which these attacks can be triggered. The authors found that the majority of malicious packages trigger their routines at installation, and almost half of the packages use code obfuscation to avoid detection. A recent study conducted by Coufalíková et al. (2021) looks into SSCA events and proposes strategies that allow to protect against such compromises and narrow the window of exposure. One of the safeguards that the authors mentioned is the importance of performing audits periodically on program components to avoid the threat in old and unpatched software. Similarly, Ladisa et al. (2022) have developed an attack taxonomy for SSCAs, and by analyzing real-world exploits as well as literature studies, the authors proposed numerous attack vectors for the injection of malicious code into open-source artifacts. They also mapped safeguards to the listed attack vectors as mitigating means to prevent, detect and correct the malicious behaviors. Zahan et al. (2022) performed an empirical study on package metadata to extract indicators that expose the package to potential risk of SSCAs. The study aims to inform developers and security practitioners about potential threat signals. One of the interesting findings that the authors uncovered is the threat lingering in unmaintained packages since updates, bug fixing and security issues require from maintainers to retain regular activity. However, these studies lack automation and mainly rely on manual decision making which requires a lot of time and effort.

Some of the main studies rely on using unsupervised learning such as clustering and anomaly detection to build a model that is able to extract and detect the syntactic or semantic similarities in packages. Garrett et al. (2019) developed an anomaly detection-based approach to identify suspicious packages. Despite the ability of this approach to reduce the effort of manual review, it requires the availability of as many cases in the dataset, and the package must have at least two versions. Furthermore, it may fail to find some cases with the sophisticated attacks that are currently exploited (e.g., code obfuscation). For campaign attacks that are conducted via packages that share the same or similar malicious code snippets, Ohm et al. (2020a) proposed ACME, an approach that extracts AST-based representations of the code, and uses Markov Cluster Algorithm (MCL) to compare the obtained representations and perform their clustering. Then, for each cluster, signatures are extracted to identify the fingerprints of each feature that allows to detect instances with similar behavior. This approach helps to detect known attacks and reduce the effort of manual review. However, the extracted signatures require manual refinement to reduce false positives, and this type of strategy is not suitable for detecting new vulnerabilities.

Previous research has also been conducted on identifying and detecting vulnerabilities at function-level using Representation Learning (Lin et al. 2017b), Deep Learning with feature extraction (Russell et al. 2018) and NLP techniques (Lin et al. 2019), GNNs and Word Embedding Models (Zhou et al. 2019; Wang et al. 2020a) to extract different code representations that capture semantic, syntactic and structural data. These recent studies have yielded promising results for vulnerability detection in source code. Our work complements the studies presented in the context of SSCAs as it provides a broader context and a direct motivation for the proposed approaches. More specifically, our results provide qualitative and quantitative evidence on the importance of identifying and fixing security vulnerabilities in dependencies, as well as the features and factors that researchers should consider while

developing their tools. Our study also unveils the currently adopted strategies than can be automated and further enhanced to handle dependency threats while highlighting the most exploited vulnerabilities. And therefore, our findings mainly aim to provide more knowledge for detection-oriented approaches.

Security Vulnerabilities and Pull Requests ben Othmane et al. (2015) performed a qualitative case study to identify the factors that impact the time to find, fix, and address security vulnerabilities by conducting a survey with security experts. The authors identified several potential factors, including those related to the software structure, the communication and collaboration within developers' team, and their experience and knowledge. Our study provides quantitatively measured evidence in open-source software that supports these findings. In another study (Ben Othmane et al. 2017), the authors built a predictive model for the fix time of security issues, but the models yielded mixed and conflicting results due to the fact that models are dependent on the collection period of the data. They also conducted a statistical analysis on the importance of factors that impact the vulnerability fix time in an industrial case study. The authors found that features related to project characteristics, software structure and development groups have significant impact. An interesting study performed by Decan et al. (2018) aims to discover how and when vulnerabilities are discovered and fixed in npm. The authors conducted an empirical study of 399 security reports for Javascript packages. They found that vulnerabilities in dependencies are growing with serious vulnerability levels. Our results confirm their findings regarding the time to fix and discover vulnerabilities (e.g., the authors found that 50% of vulnerabilities are discovered after 24 months, and our study shows that vulnerabilities remain undiscovered for more than 17 months on average), as well as the prevalence of vulnerabilities with high severity. However, our study also complements their work by considering more ecosystems (more than 10 PLs), and by investigating the support provided by bots for vulnerability discovery. Another study conducted by Wang et al. (2020b) investigates the usage, the update, and the risk of Java libraries. The authors conducted their empirical analysis on 806 Java projects, and they found that outdated libraries are commonly used. Our results confirm their findings concerning the considerable amount of projects that leave their dependencies non-updated. However, concerning the update delay, our results contest their findings. The authors found that developers have slow reaction to new dependency releases, whereas, our study showed that developers are highly receptive and responsive to security updates. However, our study focuses on software bots that allow to accelerate developers' reaction, and we focus on updates for security vulnerabilities that are expected to require more attention from developers. We also complement this study with larger dataset for more ecosystems to ensure better generalizability, and provide qualitative evidence for these findings. In a recent research conducted by Canfora et al. (2020), the authors performed an empirical and statistical investigation to identify the main differences between the fixing process of security-related and canonical bugs, the study also aims to extract the common patterns in these processes and pinpoint the factors influencing their dynamics. Using metrics that reflect different dimensions of the development process, the authors claim that security vulnerabilities require a greater number of contributors, longer discussions, and more effort to define the fixes. Our study differs from the literature work in this context by studying factors impacting fixing vulnerabilities in third-party packages (i.e., dependencies) that can potentially lead to Software Supply Chain Attacks in open-source software development.

For dependency management, Pashchenko et al. (2020) conducted a qualitative study to investigate functional and security strategies used by developers. Through semi-structured interviews, the authors uncovered some of the methods and tools used by developers to han-

dle vulnerable dependencies, as well as the workarounds applied when no fixes are available. Our study complements this work with empirical and quantitative data. Prana et al. (2021) performed an empirical study on dependency vulnerabilities to uncover the dominant vulnerability types, and examine their relationship with the characteristics of GitHub development model entities. The authors relied in their study on Veracode, and they considered features related to GitHub projects and commits for 3 programming languages. Whereas our work looks into 5 other Bots including Dependabot that we have demonstrated to be more popular, using a dataset of more than 10 different programming languages with a broader set of features. Similarly, Zerouali et al. (2022) conducted an empirical study on security vulnerabilities affecting npm and RubyGems packages. The authors were interested in the time and process required to find and fix these vulnerabilities, they found that the latter are increasingly emerging with a considerable severity level, and the discovery time is greatly larger than the fix time. The authors mentioned that automated tools are required to improve the discovery and fix time. In our study, we demonstrate that even with dependency management and SCA tools, fixes take a considerable amount of time to be applied; and we add a comparison of the automated process of tools with developer's manual effort. Another empirical study (Imtiaz et al. 2022) was conducted on security fixes and releases in OS packages. The authors performed their analysis in different registries, and focused on the aspects related to documentation and breaking changes. They identified interesting factors that can cause several windows of delay in publishing security fixes. But even after publishing them, they still take time to be applied in dependent projects for unknown reasons that we aim to uncover through our analysis.

Pull-based Development Model Peterson (2013) performed an analysis on GitHub repositories to demonstrate the influence of the pull-based development process on the basic aspects of open-source software development. The author provided evidence that there is no significant correlation between the number of repository forks and the time to fix issues, and the development process in GitHub is mostly similar to traditional OSS development. However, the study was conducted on repositories owned by GitHub organizations only (generalizability concern), and it lacks the security aspects besides the fact that the author considered repository related features solely. Gousios and Zaidman (2014) constructed a dataset of GitHub repositories and listed a set of features related to pull requests, developers, and project characteristics. The authors claim that the dataset can be exploited for research purposes in the context of pull-based distributed development model in projects and their underlying activities. But further effort is required to inspect security related activities. In a following study (Gousios et al. 2014b), Gousios et al. conducted a qualitative and quantitative analysis about pull request usage and the factors that affect the decision and time required to merge a pull request, as well as the reasons that lead to rejecting them. The authors found that merge time is affected by code reviews, while merge decision is influenced by the latest activity in the repository, and the developer's track record. Concerning the acceptance of pull requests in GitHub projects, Soares et al. (2015b) conducted a mining approach to investigate the factors that can lead to better and faster merge. Also, Kononenko et al. (2018a) conducted an empirical study on a commercial project to investigate the merge nature and dynamics of pull requests. The authors applied data mining and conducted survey with developers to investigate the factors that can improve the merge time and decision. In contrast to previous studies, our work focuses on automated and manual pull requests that handle security issues and vulnerabilities in project dependencies.

Bots in Software Development For bot adoption in open-source software (OSS) development in GitHub, Wessel et al. (2018) conducted a study that investigates a sample of GitHub projects to classify the usage and impact of bots. The authors relied on metrics that are measured before and after the adoption of bots, and conducted a survey with developers to gather the problems and the deficiencies in the adopted tools. Our study focuses on dependency management and security vulnerabilities and provides more insights by considering a wider range of features to compare the automated process with the manual tasks performed by developers. For bot classification, Erlenhov et al. (2019b) proposed a taxonomy of Bots that support software development (DevBots) through a general faceted analysis. The authors looked into several DevBots that are widely used in OSS development including Dependabot, but the taxonomy is based on theoretical setting and lacks quantitative evidence. A similar study (Wessel and Steinmacher 2020b) looked into the use and effects of software bots that assist maintainers and contributors with pull requests. The authors categorized GitHub bots based on their activities, and highlighted their negative aspects. The authors claim that the interaction between human and bots can be disruptive and perceived as unwanted, especially when they introduce communication noise. Dey et al. (2020) proposed an approach to automate bot identification based on the author name in GitHub and commit activity (messages, files and projects features). However, relying on these heuristics can lead to false positives and false negatives. In our study, we focus on dependency management and security aspects, and in addition to commits characteristics, we also consider those related to repositories, pull requests, and users. Concerning security aspects, Angermeir et al. (2021) focused on automated security activities related to open-source software products. The authors mined repositories and conducted surveys with project maintainers to investigate the adopted security tools and extract the characteristics related to security adoption. They found that Dependabot is the most used tool for vulnerability scanning. However, security tool detection was performed heuristically using manually created search patterns, and the analysis was carried out in CI setup with dataset projects that use Travis and GitHub CI service only, which may narrow the viewing of some security activities. The study also showed that the latter are rarely performed in CI which calls for the need to inspect manual security activities.

The recent study performed by Alfadel et al. (2021) on Dependabot security pull requests is the closest to our work. The authors performed an empirical study to investigate the use of Dependabot to handle security pull requests, the reasons behind their rejection, and the factors correlated with the fast merging. The authors provided evidence that open-source JS projects are highly receptive to Dependabot security PRs, and the fast merge time is related to the project activity level, the past experience with Dependabot, and the adoption of automerge feature. In our work, we perform the study on a larger dataset of projects from different ecosystems (more than 10 PLs), with a variety of maturity levels and project sizes to ensure a better generalizability. We also consider a dataset for security PRs created manually by developers and 4 other Bots to compare the activity and the contribution of Dependabot as opposed to the manual effort. Furthermore, through a manual analysis, our study uncovered interesting patterns when it comes to the strategies used to identify and fix security vulnerabilities automatically and manually. Finally, we consider larger number of factors that can potentially act on the merge rate and merge decision of security PRs while substantiating our findings with a survey conducted with GitHub developers who also provided valuable insights on the encountered challenges and some potential improvements. Other recent studies also focused on the adoption of Bots in software projects. Wessel et al. (2021), Wessel et al. (2022) were interested in the benefits and challenges presented by software bots. Our results confirm their findings concerning the added value of bots with PRs merge activity in the context of dependency management and handling security vulnerabilities. Our study

also provides quantitative and qualitative evidence regarding the challenges encountered with bots in terms of adoption and interaction while complementing with potential improvements for these challenges. Similarly, Santhanam et al. (2022), Moguel-Sánchez et al. (2022) conducted mapping studies for Bots in software engineering and software development. The authors provided a systematic classification for Bots and their application. Our study focuses on PR management Bots, and our findings confirm their wide adoption in project management for the task of handling security vulnerabilities. We also complement these studies by providing quantitative evidence for the adoption of software bots and their activity, as well as the reasons that lead to their wide adoption.

✦ ***Comparative Summary***
In order to demonstrate the significance of our work and its added value with regards to prior work in the literature, Table 29 summarizes a comparison with the related work by highlighting the methodological variations and showcasing the similarities and differences in terms of: the study focus, scope, context, the adopted approach, and its setup, as well as the results and findings.

8 Threats to Validity

Construct Validity Threats to construct validity can affect the degree to which an assessment that we have performed is well suited to evaluate the concept it was meant to represent. In our study, this can refer to the extraction of the data from PRs that are meant to be security-related. For Dependabot PRs, we exploited the label that is created automatically by the bot and always set as *"label: security"* if it handles dependency vulnerabilities. For manual PRs, we perform a query-based search using multiple root options. The threat to this validity can also be related to the concept of popularity in dependency management tools. We consider the number of PRs that are created by each tool/bot across multiple repositories as an indication to its popularity since it reflects its activity and dominance level. However, we argue that it would be more appropriate to consider the number of repositories. To mitigate this threat, we inspected the popularity of each tool based on the number of PRs and also the number of repositories, and we found that the relationship between the repositories and the created PRs is *linear*, and dependency management tools create 7 PRs per repository (1:7) on average. We intentionally measured the popularity in terms of PRs to inspect the state of the activity (handled vs. non-handled). Threats could also be associated to the concept of discovery time of security vulnerabilities. In our study, we consider the hidden threat lifetime to be the duration between 0-day and the date of creating the PR suggesting the fixes. However, the vulnerability can be discovered by developers before creating the PR. To mitigate this threat, we inspected with our manual analysis in RQ2 the source that developers rely on to discover vulnerabilities, and their comments have shown their quick reaction to create the PRs to fix the discovered vulnerabilities. For Dependabot and other bots, they rely on the data provided in vulnerability databases, and therefore, the PR are created quickly after the vulnerability disclosure (as shown in RQ3). Finally, to measure the developers experience, we used the age of their GitHub accounts as proxy. This metric may inaccurately portray the experience of a developer with an older account who is not actively involved in the project as an experienced developer. To mitigate this threat, we consider the account age to indicate the degree to which the developer is familiar with the platform, the pull-based development model, and software

Table 29 Comparative summary for the related works

Study	Method / Approach	Similarities	Differences
Ohm et al. (2020b)	Knowledge systematization - Manual analysis	**(1)** Raising awareness about SSCAs **(2)** Inspection of packages / dependencies	**(1)** Our study provides empirically extracted evidence on the threat of SSCAs and vulnerability emergence **(2)** Our quantitative and qualitative analyses on vulnerabilities provide complementary proof for literature findings (e.g., impact of developers' activity and workload on vulnerability identification and fix) **(3)** We consider both manual & automated approaches to identify vulnerabilities in packages / dependencies
Coufalíková et al. (2021)			
Ladisa et al. (2022)			
Zahan et al. (2022)			
Garrett et al. (2019)	ML-based approach	**(1)** Identification of vulnerable packages	
Ohm et al. (2020a)			
Lin et al. (2017b)	Deep representation learning-based approach	**(1)** Focus on software vulnerabilities	
Russell et al. (2018)			
Lin et al. (2019)			
Zhou et al. (2019)			
Wang et al. (2020a)			

Table 29 continued

Study	Method / Approach	Similarities	Differences
ben Othmane et al. (2015)	Empirical analysis	**(1)** Investigating factors that impact vulnerability fixes **(2)** Investigating the time to discover and fix security vulnerabilities **(3)** Investigating the usage and impact of outdated and bloated dependencies	**(1)** Our study covers broader set of features & factors **(2)** Our study targets more ecosystems (> 10 PLs) **(3)** Our study complements qualitative analyses with quantitative evidence **(4)** Our study investigates the contribution of bots in vulnerability discovery
Ben Othmane et al. (2017)			
Decan et al. (2018)			
Wang et al. (2020b)			
Canfora et al. (2020)			
Pashchenko et al. (2020)			
Prana et al. (2021)			
Zerouali et al. (2022)			
Imtiaz et al. (2022)			
Peterson (2013)	Empirical analysis	**(1)** Investigating the aspects of the pull-based development process **(2)** Investigating the factors that affect the merge time and merge decision of PRs	**(1)** Our study focuses on security concerns in software **(2)** Our study inspects more characteristics in the pull-based development model **(3)** We investigate both auto-generated & manual PRs
Gousios et al. (2014b)			
Kononenko et al. (2018a)			
Gousios and Zaidman (2014)	Knowledge systematization		
Soares et al. (2015b)	Data mining-based approach		

Table 29 continued

Study	Method / Approach	Similarities	Differences
Erlenhov et al. (2019b)	Knowledge Systematization	**(1)** Exploring software bots and their adoption **(2)** Investigating security activities in bots **(3)** Discovering bots limitations	**(1)** Our study focuses on dependency management bots and SCA tools **(2)** Our study compares bots' *v.s.* developers' activities **(3)** Our study covers a broader set of features **(4)** Our study enables better generalizability with several bots and different ecosystems (> 10 PLs)
Dey et al. (2020)	Heuristic-based Approach		
Wessel et al. (2018)	Empirical Analysis		
Wessel and Steinmacher (2020b)			
Angermeir et al. (2021)			
Alfadel et al. (2021)			
Wessel et al. (2021)			
Wessel et al. (2022)			
Santhanam et al. (2022)	Mapping Study		
Moguel-Sánchez et al. (2022)			

development. We have also included the number of public repositories and gists that can indicate the contribution of the developer and its activity.

Internal Validity Threats to internal validity refer to the level of confidence to which a cause-effect relationship established in our study is not due to other influences. In our experimentation this can be related to the reasons Dependabot PRs are closed, the difference between Dependabot and manual PRs in terms of time to handle security PRs, and the hypotheses about the factors that can influence the merge decision and merge time. To alleviate this threat, we attempted to explore several studies in the literature that support our analyses and findings and which we have stated. We also performed our manual analysis on a statistically representative dataset sample to confirm our observations. The analysis was conducted by 4 research students and further validated by the first author. Finally, we conducted a survey with developers to substantiate our findings and support our data with their own experience.

External Validity Threats to external validity concern the extent to which our findings can be generalized to other situations and broader contexts. To address this threat, we made sure our dataset comprises projects of more than 10 different programming languages, so our study covers multiple development ecosystems including the most popular. We also considered looking into projects with different characteristics (size, maturity level, etc.) to ensure better generalizability. In addition, we inspected repositories that use automation with different Bots and those that perform their activities manually on managing security vulnerabilities. However, the scope of our work is projected onto open-source software due to its wide adoption and the enlarged attack surface, therefore further studies are required for closed-source software.

9 Conclusion

Security vulnerabilities that are injected into project dependencies is one of the most popular strategies exploited by hackers to conduct Software Supply Chain Attacks. The recent incidents have shown the magnitude of such attacks and their huge impact and attack surface due to their reliance on the chain of open-source dependencies. In our study, we investigate the appropriateness and the limits of the current tools and security measures used to deal with this major concern. Then, we attempt to identify the factors and features that encourage the usage of these security measures and tools so that researchers and security experts know what to avoid and what to empower in new approaches. After that, we try to understand the practices retained by developers and security experts so that new tools can potentially consider automating these strategies. Finally, our work aims to discover the dominance and the lifetime of the threat of security vulnerabilities in dependencies. For that, we created a pipeline that allows us to collect the data related to issues, pull-requests, repositories, commits, and users on GitHub platform. The constructed dataset comprises 9,916,318 PR-related issues made in 1,743,035 GitHub projects for more than 10 different programming languages.

Through our analysis, we found that Dependabot is the most popular tool in dependency management due to its efficiency, accessibility, adaptivity, usability, and the documentation support that developers consider as primary features in software bots. We manually extracted the security measures that developers adopt to manually handle security vulnerabilities in dependencies, and we found that developers mainly upgrade vulnerable dependency versions,

but also rely on some inadequate resorts in the lack of new fixes. After that, we measured the receptiveness and responsiveness of developers towards the auto-generated security PRs, and we discovered that developers are highly receptive to security PRs with a merge rate of less than 1 day, and the automation level provided by Software Composition Analysis (SCA) tools allows to improve the response time. Concerning the threat lifetime, our study revealed that vulnerabilities remain unpatched for a huge amount of time (512 days on average) which increases the window of exposure even with the adoption of SCA tools, but most importantly, some vulnerable dependencies are still being used even after their disclosure in vulnerability databases. In addition, we listed the most exploited vulnerabilities that attackers use to conduct these attacks to shed light on the major focus when it comes to risk assessment and prevention measures. Finally, we perform a statistical analysis to identify the factors that have high correlation with the merge decision and merge rate of security PRs. The analysis showed that the project characteristics (maturity and size) and its activity, the developers experience, and the refactoring effort are strongly correlated with high chances of adopting security patches.

In future work, we aim to conduct a further investigation on the vulnerabilities that lead to software supply chain attacks in order to discover their peculiarities compared to general vulnerabilities, as well as locating the vulnerable changes in package versions. In an effort to improve the discovery time and the fix delay of these vulnerabilities, we also intend to explore more advanced approaches that rely on readily available package data (source code, artifacts, etc.) to detect security vulnerabilities in dependencies.

A Open & Axial Coding Techniques

In this section, we provide an example to illustrate the adoption of the open coding and axial coding to perform our manual analysis in order to answer RQ2. Figure 13 showcases an example of the results of open and axial coding analyses with the derived quotes and codes.

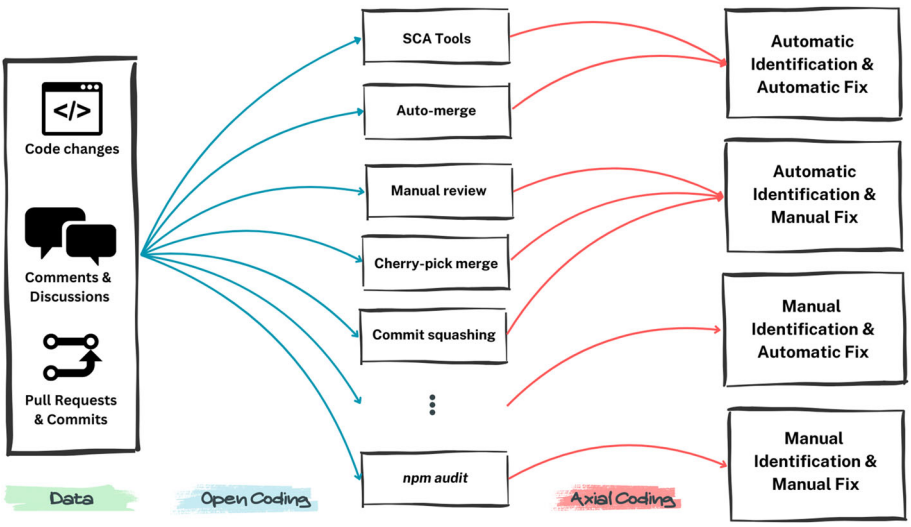

Fig. 13 Example of open coding and axial coding analyses

Acknowledgements This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 949014).

Author Contributions
- Conceptualization: Hocine Rebatchi, Tégawendé F. Bissyandé, Naouel Moha
- Data curation: Hocine Rebatchi
- Formal analysis: Hocine Rebatchi
- Investigation: Hocine Rebatchi
- Methodology: Hocine Rebatchi, Tégawendé F. Bissyandé, Naouel Moha
- Resources: Hocine Rebatchi, Tégawendé F. Bissyandé, Naouel Moha
- Software: Hocine Rebatchi
- Supervision: Tégawendé F. Bissyandé, Naouel Moha
- Visualization: Hocine Rebatchi
- Writing – original draft: Hocine Rebatchi
- Approval of the final version of the manuscript: Hocine Rebatchi, Tégawendé F. Bissyandé, Naouel Moha

Funding This study was supported by (1) the Natural Sciences and Engineering Research Council of Canada (NSERC), and by (2) the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 949014).

Materials and/or Code Availability The datasets that were generated and analyzed in the current study are stored within a Zenodo repository: https://doi.org/10.5281/zenodo.7801356 under Creative Commons Attribution 4.0 International License (CC-BY 4.0). The artifacts and the source code are also hosted on a GitHub repository: https://github.com/HocineREBT/GitHub-Miner.

Declarations

Consent All the authors give their consent to submit this work.

Data Our datasets support our results and comply with the field standards.

Ethics Approval Our manuscript is not submitted to another journal for simultaneous consideration, and our work is original. The authors also declare that this manuscript follows the best scientific standards, in particular, w-r-t to acknowledgment of prior works, honesty of the presentation of results, and focus on the demonstrability of the statements. This manuscript and the work that led to it do not carry any specific ethic issue. As such, it was considered unnecessary to seek formal approval from our institution Ethics Committee specifically for this work.

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

References

Akoglu H (2018) User's guide to correlation coefficients. Turk J Emerg Med 18(3):91–93. https://doi.org/10.1016/j.tjem.2018.08.001, https://www.sciencedirect.com/science/article/pii/S2452247318302164

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. https://doi.org/10.1109/MSR52588.2021.00037

Andreoli A, Lounis A, Debbabi M, Hanna A (2023) On the prevalence of software supply chain attacks: empirical study and investigative framework. Forensic Sci Int: Digital Investigation 44:301508

Angermeir F, Voggenreiter M, Moyón F, Mendez D (2021) Enterprise-driven open source software: a case study on security automation. In: 2021 IEEE/ACM 43rd International conference on software engineering: software engineering in practice (ICSE-SEIP). IEEE, pp 278–287

ben Othmane L, Chehrazi G, Bodden E, Tsalovski P, Brucker AD, Miseldine P (2015) Factors impacting the effort required to fix security vulnerabilities. In: Lopez J, Mitchell CJ (eds) Information security. Springer International Publishing, Cham, pp 102–119

Ben Othmane L, Chehrazi G, Bodden E, Tsalovski P, Brucker AD (2017) Time for addressing software security issues: prediction models and impacting factors. Data Sci Eng 2(2):107–124

Bird C, Gourley A, Devanbu P, Swaminathan A, Hsu G (2007) Open borders? immigration in open source projects. In: Proceedings of the fourth international workshop on mining software repositories. IEEE Computer Society, USA, MSR '07, p 6. https://doi.org/10.1109/MSR.2007.23

Birsan A (2021) Dependency confusion: how I hacked into apple, microsoft and dozens of other companies. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

Boehm C (2023) Supply chain attacks: how to protect against attack and sabotage. https://assets.sentinelone.com/supply-chain-attacks/how-to-protect-against-attack-and-sabotage-en

Calkins KG (2005) Correlation coefficients. https://www.andrews.edu/~calkins/math/edrm611/edrm05.htm, publisher: Andrews University

Canfora G, Di Sorbo A, Forootani S, Pirozzi A, Visaggio CA (2020) Investigating the vulnerability fixing process in oss projects: peculiarities and challenges. Comput Secur 99:102067

Coufalíková A, Klaban I, Šlajs T (2021) Complex strategy against supply chain attacks. In: 2021 International conference on military technologies (ICMT). IEEE, pp 1–5

DeBill E (2019) Module counts. http://www.modulecounts.com/

Decan A, Mens T, Constantinou E (2018) 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

Dey T, Mousavi S, Ponce E, Fry T, Vasilescu B, Filippova A, Mockus A (2020) Detecting and characterizing bots that commit code. In: Proceedings of the 17th international conference on mining software repositories, pp 209–219

Duan R, Alrawi O, Kasturi RP, Elder R, Saltaformaggio B, Lee W (2020) Towards measuring supply chain attacks on package managers for interpreted languages. arXiv:2002.01139

Erlenhov L, de Oliveira Neto FG, Scandariato R, Leitner P (2019b) Current and future bots in software development. In: 2019 IEEE/ACM 1st International workshop on bots in software engineering (BotSE). IEEE, pp 7–11

Erlenhov L, Gomes de Oliveira Neto F, Scandariato R, Leitner P (2019a) Current and future bots in software development. In: 2019 IEEE/ACM 1st International workshop on bots in software engineering (BotSE), pp 7–11. https://doi.org/10.1109/BotSE.2019.00009

Garrett K, Ferreira G, Jia L, Sunshine J, Kästner C (2019) Detecting suspicious package updates. In: Proceedings of the 41st International conference on software engineering: new ideas and emerging results. IEEE Press, ICSE-NIER '19, p 13–16. https://doi.org/10.1109/ICSE-NIER.2019.00012

GitHub (2021) Github rest api. https://docs.github.com/en/rest/reference/search

Gousios G, Pinzger M, Deursen Av (2014a) An exploratory study of the pull-based software development model. In: Proceedings of the 36th international conference on software engineering. Association for computing machinery, New York, NY, USA, ICSE 2014, pp 345–355. https://doi.org/10.1145/2568225.2568260

Gousios G, Pinzger M, Deursen Av (2014b) An exploratory study of the pull-based software development model. In: Proceedings of the 36th international conference on software engineering, pp 345–355

Gousios G, Zaidman A (2014) A dataset for pull-based development research. In: Proceedings of the 11th working conference on mining software repositories. Association for computing machinery, New York, NY, USA, MSR 2014, pp 368–371. https://doi.org/10.1145/2597073.2597122

Groves RM, Fowler FJ Jr, Couper MP, Lepkowski JM, Singer E, Tourangeau R (2011) Survey methodology, vol 561. John Wiley & Sons

Hou F, Jansen S (2023) A systematic literature review on trust in the software ecosystem. Empir Softw Eng 28(1):8

Imtiaz N, Khanom A, Williams L (2022) Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages. IEEE Trans Softw Eng

Jeong G, Kim S, Zimmermann T, Yi K (2009) Improving code review by predicting reviewers and acceptance of patches. Research on software analysis for error-free computing center Tech-Memo (ROSAEC MEMO 2009-006), pp 1–18

Kaczorowski M (2020) Secure at every step: what is software supply chain security and why does it matter? https://github.blog/2020-09-02-secure-your-software-supply-chain-and-protect-against-supply-chain-threats-github-blog/

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

Kononenko O, Rose T, Baysal O, Godfrey M, Theisen D, De Water B (2018a) Studying pull request merges: a case study of shopify's active merchant. In: Proceedings of the 40th international conference on software engineering: software engineering in practice, pp 124–133

Kononenko O, Rose T, Baysal O, Godfrey M, Theisen D, de Water B (2018b) Studying pull request merges: a case study of shopify's active merchant. In: 2018 IEEE/ACM 40th International conference on software engineering: software engineering in practice track (ICSE-SEIP), pp 124–133

Ladisa P, Plate H, Martinez M, Barais O (2022) Taxonomy of attacks on open-source software supply chains. arXiv preprint arXiv:2204.04008

Lawall J, Muller G (2018) Coccinelle: 10 years of automated evolution in the Linux kernel. In: Proceedings of the 2018 USENIX conference on usenix annual technical conference. USENIX Association, USA, USENIX ATC '18, pp 601–613

Lin G, Xiao W, Zhang J, Xiang Y (2019) Deep learning-based vulnerable function detection: a benchmark. In: International conference on information and communications security. Springer, pp 219–232

Lin G, Zhang J, Luo W, Pan L, Xiang Y (2017a) Poster: vulnerability discovery with function representation learning from unlabeled projects. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security. Association for computing machinery, New York, NY, USA, CCS '17, pp 2539–2541. https://doi.org/10.1145/3133956.3138840

Lin G, Zhang J, Luo W, Pan L, Xiang Y (2017b) Poster: vulnerability discovery with function representation learning from unlabeled projects. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp 2539–2541

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

Moguel-Sánchez R, Martínez-Palacios CS, Ocharán-Hernández JO, Limón X, Sánchez-García ÁJ (2022) Bots and their uses in software development: a systematic mapping study. In: 2022 10th International conference in software engineering research and innovation (CONISOFT). IEEE, pp 140–149

Mujahid S, Abdalkareem R, Shihab E (2023) What are the characteristics of highly-selected packages? a case study on the npm ecosystem. J Syst Softw 198:111588

Munaiah N, Kroh S, Cabrey C, Nagappan M (2017) Curating github for engineered software projects. Empir Softw Eng 22:3219–3253

NIST (2021) Vulnerability metrics. https://nvd.nist.gov/vuln-metrics/cvss

Ohm M, Kempf L, Boes F, Meier M (2020a) Supporting the detection of software supply chain attacks through unsupervised signature generation. arXiv preprint arXiv:2011.02235

Ohm M, Kempf L, Boes F, Meier M (2021) Supporting the detection of software supply chain attacks through unsupervised signature generation. arXiv:2011.02235

Ohm M, Plate H, Sykosch A, Meier M (2020b) Backstabber's knife collection: a review of open source software supply chain attacks. In: International conference on detection of intrusions and malware, and vulnerability assessment. Springer, pp 23–43

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

Peterson K (2013) The github open source development process. http://kevinp.me/github-process-research/github-processresearch.pdf (visited on 05/11/2017)

Pham R, Singer L, Liskin O, Figueira Filho F, Schneider K (2013) Creating a shared understanding of testing culture on a social coding site. In: 2013 35th International conference on software engineering (ICSE). IEEE, pp 112–121

Plumb T (2022) GitHub's Octoverse report finds 97% of apps use open source software. https://venturebeat.com/programming-development/github-releases-open-source-report-octoverse-2022-says-97-of-apps-use-oss/

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. Empir Softw Eng 26(4):1–34

Preston-Werner T (2021) Semantic versioning 2.0.0. https://semver.org/

Rigby PC, Bird C (2013) Convergent contemporary software peer review practices. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering. Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2013, pp 202–212. https://doi.org/10.1145/2491411.2491444

Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, Ellingwood P, McConley M (2018) Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE international conference on machine learning and applications (ICMLA). IEEE, pp 757–762

Santhanam S, Hecking T, Schreiber A, Wagner S (2022) Bots in software engineering: a systematic mapping study. PeerJ Computer Science 8:e866

Soares DM, de Lima Júnior ML, Murta L, Plastino A (2015a) Acceptance factors of pull requests in open-source projects. In: Proceedings of the 30th annual ACM symposium on applied computing. Association for Computing Machinery, New York, USA, SAC '15, pp 1541–1546. https://doi.org/10.1145/2695664.2695856

Soares DM, de Lima Júnior ML, Murta L, Plastino A (2015b) Acceptance factors of pull requests in open-source projects. In: Proceedings of the 30th annual ACM symposium on applied computing, pp 1541–1546

Soto-Valero C, Durieux T, Baudry B (2021) A longitudinal analysis of bloated java dependencies. In: Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. Association for Computing Machinery, New York, USA, ESEC/FSE 2021, pp 1021–1031. https://doi.org/10.1145/3468264.3468589

Szulik K (2018) Dependency management and your software health. https://blog.tidelift.com/dependency-management-and-your-software-health

Wang H, Ye G, Tang Z, Tan SH, Huang S, Fang D, Feng Y, Bian L, Wang Z (2020a) Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Trans Inf Forensics Secur 16:1943–1958

Wang Y, Chen B, Huang K, Shi B, Xu C, Peng X, Wu Y, Liu Y (2020b) 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

Weißgerber P, Neu D, Diehl S (2008) Small patches get in! In: Proceedings of the 2008 international working conference on mining software repositories. Association for Computing Machinery, New York, USA, MSR '08, pp 67–76. https://doi.org/10.1145/1370750.1370767

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. Proc ACM Hum-Comput Interaction 2(CSCW):1–19

Wessel M, Wiese I, Steinmacher I, Gerosa MA (2021) Don't disturb me: challenges of interacting with software bots on open source software projects. Proc ACM Hum-Comput Interaction 5(CSCW2):1–21

Wessel M, Gerosa MA, Shihab E (2022) Software bots in software engineering: benefits and challenges. In: Proceedings of the 19th International conference on mining software repositories, pp 724–725

Wessel M, Steinmacher I (2020a) The inconvenient side of software bots on pull requests. In: Proceedings of the IEEE/ACM 42nd international conference on software engineering workshops. Association for Computing Machinery, New York, USA, CSEW'20, pp 51–55. https://doi.org/10.1145/3387940.3391504

Wessel M, Steinmacher I (2020b) 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

Yu Y, Wang H, Filkov V, Devanbu P, Vasilescu B (2015) Wait for it: determinants of pull request evaluation latency on github. In: 2015 IEEE/ACM 12th Working conference on mining software repositories, pp 367–371. https://doi.org/10.1109/MSR.2015.42

Zahan N, Zimmermann T, Godefroid P, Murphy B, Maddila C, Williams L (2022) What are weak links in the npm supply chain? In: 2022 IEEE/ACM 44th International conference on software engineering: software engineering in practice (ICSE-SEIP). IEEE, pp 331–340

Zerouali A, Mens T, Decan A, De Roover C (2022) On the impact of security vulnerabilities in the npm and rubygems dependency networks. Empir Softw Eng 27(5):1–45

Zerouali A, Mens T, Decan A, Roover CD (2021) On the impact of security vulnerabilities in the npm and rubygems dependency networks. arXiv:2106.06747

Zhou Y, Liu S, Siow J, Du X, Liu Y (2019) Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv Neural Inf Process Syst 32

Zhu J, Zhou M, Mockus A (2016) Effectiveness of code contribution: from patch-based to pull-request-based tools. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering. Association for Computing Machinery, New York, USA, FSE 2016, pp 871–882. https://doi.org/10.1145/2950290.2950364

Authors and Affiliations

Hocine Rebatchi[1] · Tégawendé F. Bissyandé[2] · Naouel Moha[1]

✉ Hocine Rebatchi
hocine.rebatchi.1@ens.etsmtl.ca

Tégawendé F. Bissyandé
tegawende.bissyande@uni.lu

Naouel Moha
naouel.moha@etsmtl.ca

1 École de Technologie Supérieure, Montreal, Canada

2 University of Luxembourg, Luxembourg City, Luxembourg