A Deep Dive into Deprecation Declarations in the Rust Package Ecosystem

Minyu Shu, Meng Fan, Yuxia Zhang, Tao Wang, and Hui Liu

Abstract—Utilizing third-party open source libraries is fundamental to modern software development because it can enhance productivity and software quality. However, libraries may cease maintenance and become deprecated, negatively impacting the projects that rely on them. Promptly identifying and addressing deprecated libraries can help developers mitigate potential risks within their projects. As a programming language known for its emphasis on safety, Rust's package manager currently does not provide a direct mechanism for deprecation. Nevertheless, Rust developers can still declare deprecation using certain methods offered by GitHub and the official Rust package registry, crates.io. However, the current usage of these deprecation mechanisms in the Rust ecosystem, as well as their effectiveness, remains underexplored. This paper addresses this gap by empirically studying the prevalence of deprecation declarations in Rust libraries, the effectiveness of different ways of declarations, and the reasons for using deprecated libraries to understand how deprecation information is disseminated and perceived in the current Rust ecosystem. We found that: 1) Among the 13,289 inactive libraries in the Rust ecosystem, only 11% of them indicate their deprecated status; 2) Among the packages that released a new version after their dependent library declared deprecation, 38.9% still chose to use the deprecated library in their new releases; 3) Despite developers being able to actively or passively discover deprecated libraries within their projects through various means, unawareness of library deprecation is a significant reason for developers using deprecated libraries. Based on these findings, we discuss practice insights to help improve the deprecation mechanism and mitigate software dependency risks.

Index Terms—Software supply chain, Rust, Deprecation declaration, Software maintenance

1 Introduction

Don't *reinvent the wheel.* Modern software development always utilizes existing libraries, frameworks, or other modules to create new products, forming complex *software supply chains* [1]. Software reuse can effectively accelerate the pace and enhance the quality of software development while reducing associated costs [2]. Nowadays, open source software (OSS) has played a fundamental role in information technology, and the majority of reused code is open source. As reported recently, up to 97% of commercial software contained open source code [3].

However, incorporating OSS libraries into projects can also introduce risks [4, 5]. Numerous OSS packages may become deprecated due to factors such as heightened competition, a lack of development time and interest from core developers [6], or dependence on outdated technologies [7, 8]. Once a library becomes unmaintained, any new vulnerabilities in that library will remain unaddressed, and the fixes for these vulnerabilities in upstream libraries will become inaccessible, thereby affecting all software relying on that library [9].

To alert users of a library about its maintenance status, some package managers, such as NPM, allow developers to mark a library as deprecated, then provide a deprecation warning to all who attempt to install it [10]. Known for its safety and high performance, Rust has been voted as the most loved programming language by developers for seven consecutive years [11]. However, Rust's package manager, Cargo[1], has not yet provided an official way to deprecate a library. Nevertheless, Rust library developers can proactively declare deprecation through methods provided by the Rust package retrieval website, i.e., crates.io, and GitHub. Among these deprecation methods, if a library becomes unusable after being deprecated, we refer to it as mandatory deprecation. If the deprecation serves only as a warning or recommendation, we refer to it as advisory deprecation. There is currently a lack of research on how these two types of deprecation are used and how they impact downstream packages.

Regarding deprecated libraries, Zhong et al. [12] and Miller et al. [13] have conducted studies on deprecated libraries in the Python and NPM ecosystems, respectively, exploring their impact on downstream dependencies. However, their definitions of deprecated libraries do not account for the unique deprecation mechanisms of each ecosystem, making their findings not directly applicable to the Rust ecosystem. Additionally, their research did not investigate the broader impact of deprecated libraries on the software supply chain. To have a thorough understanding of the usage and impact of deprecation declarations in the rising Rust ecosystem, we conducted this empirical study by mining and analyzing packages from Rust's official package registry (crates.io). Specifically, we formulated and answered the following research questions:

RQ1: How many libraries that have not been maintained for a long time explicitly declare deprecation?

We started our research by investigating inactive Rust libraries hosted on GitHub that have not been maintained

- *Minyu Shu, Meng Fan, Yuxia Zhang, and Hui Liu are with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China. E-mail: yuxiazh@bit.edu.cn. (Corresponding author: Yuxia Zhang.)*
- *Tao Wang is with the School of Computer Science, National University of Defense Technology, Hunan, China.*

1. https://doc.rust-lang.org/cargo/guide/why-cargo-exists.html

for over 1 year, as these inactive libraries are potential candidates for explicit declaration of deprecation. We found that among 13,289 inactive libraries, only 11% of them declared deprecation, leaving the maintenance status of the majority of libraries unknown.

RQ2: What impact does the declaration of library deprecation have on downstream dependencies?

We categorize the ways to deprecate libraries into two types: mandatory deprecation and advisory deprecation. We found that the former effectively forces all alive downstream packages to abandon their dependency on these libraries. However, this can cause issues for packages that directly or indirectly depend on them. The latter, serving as a warning, exhibits a limited impact on deterring new packages from adopting deprecated libraries. Among the packages that released a new version after their dependent library declared deprecation, 38.9% still chose to use the deprecated library in their new releases.

RQ3: Why do some developers still use libraries that have been declared deprecated?

Given the discovery in RQ2 that many active packages continue to use deprecated libraries, we surveyed developers of these packages that are still maintained to understand the reasons. We categorized the reasons into four main categories, with unawareness of the existence of deprecated libraries in the project being the most commonly cited reason.

To the best of our knowledge, this is the first study to investigate how deprecation declarations are published and perceived, and their impacts in the Rust ecosystem, one of the fastest-growing programming language ecosystems. Our findings can shed light on the shortcomings of the current Rust library management mechanism and offer insights for improvements to the deprecation mechanism.

In the remaining sections of the paper, Section 2 provides an introduction to the background of our study. Section 3 introduces the source of our experimental data. Sections 4 to 6 each address a specific research question. Section 7 discusses the implications of our findings. Section 8 discusses the threats to the validity of our study. Section 9 concludes this paper.

2 Background

To better elucidate the content of our experiments and make them more comprehensible, in this section, we introduce the related work, discuss the dependency management system of Rust's package manager Cargo, and define the terminology used in this paper, along with the corresponding computational methods.

2.1 Related Work

In this section, we present related work about the software supply chain and software deprecation mechanisms.

2.1.1 Software Supply Chain

With the emergence of a series of influential software dependency issues, such as left-pad[2] and log4j[3] incidents, security concerns related to software dependencies have garnered increasing attention in both academia and industry. Similar to the supply chain in industrial production, the concept of the software supply chain is used to analyze security issues related to software artifacts that have dependencies [1]. Numerous studies have already analyzed the dependency structures and evolution within various software ecosystems [14–16]. However, due to the complexity of software supply chain networks, dependency conflicts may arise, potentially leading to problems during project building or execution [17–19]. In a dependency network, a library carrying a vulnerability affects not only the packages directly dependent on it but also those packages that indirectly depend on these libraries [20–23]. Despite upstream libraries releasing new versions that fix vulnerabilities, downstream packages typically take a considerable amount of time to incorporate the patched versions [24–26]. When an upstream library is no longer maintained and existing vulnerabilities cannot be addressed, downstream packages may choose to remove their dependency on the deprecated library and opt for another more appropriate alternative [27–29].

2.1.2 Deprecation Mechanism

In software systems, there are different levels of deprecation granularity. Deprecation can occur at the API, release, and package levels. When developers want to discourage users from using some APIs, releases, or packages, deprecation declarations can serve as a communication mechanism with users, reminding them of the associated risks.

For API-level deprecation, the Rust language provides the "`#[deprecated]`" attribute to mark APIs that are not recommended for use. When other developers depend on this library and use the deprecated API, the compiler will issue a warning [30]. For release-level deprecation, Rust's package manager, Cargo, provides the `yank` method to deprecate a specific version, rendering that version unusable [31]. However, Rust currently does not offer a direct method to deprecate an entire software package. Nevertheless, *Yanking all releases* can serve as an indirect way to deprecate an entire package, as it makes all versions unavailable for use. This is explained in detail in Section 4.1. In contrast, using "`#[deprecated]`" is not suitable for indirectly deprecating an entire package. This is because it requires publishing a new version of the package that includes the deprecated API annotation. The code of older versions remains unchanged, so users depending on those versions will not receive any warnings. Furthermore, if an API annotated with "`#[deprecated]`" is not used in a project, no warning will be triggered.

Regarding research on API-level deprecation, Robbes et al. [32] analyzed deprecated APIs in the Squeak and Pharo software ecosystems, addressing research questions about the frequency, magnitude, duration, adaptation, and consistency of the ripple effects caused by API changes. Sawant et al. [33] conducted semi-structured interviews with 17 third-party Java API producers and surveyed 170 Java developers. They found that the current deprecation mechanism in Java and the proposed enhancements do not meet all the developers' needs.

In the context of deprecation at the release level, Cogo et al. [34] studied the usage of deprecated releases in the

2. https://www.sciencealert.com/how-a-programmer-almost-broke-the-internet-by-deleting-11-lines-of-code

3. https://nvd.nist.gov/vuln/detail/CVE-2021-44228

JavaScript language software ecosystem, finding that 27 percent of all client packages directly adopt at least one deprecated release, and 54 percent of all client packages transitively adopt at least one deprecated release. In their subsequent study on deprecation at the release level in the Rust ecosystem [35], they found that the yanked releases propagated through the dependency network resulted in 1.4% of releases in the ecosystem having unresolved dependencies. Kula et al. [36] investigated developers' responsiveness to important awareness mechanisms, such as new release announcements and security advisories, in the context of library updates. They found that as many as 81.5% of systems were still using outdated versions of libraries.

For deprecated packages, previous studies [7] have shown that developers may stop developing OSS projects due to job changes, economic issues, lack of interest, etc. Xia et al. [37] studied 361 popular GitHub projects that have been archived, supplementing the reasons for the deprecation of OSS projects and describing the process of deprecation. Additionally, there are studies aimed at identifying projects that are no longer maintained, to assist developers in assessing the maintenance status of OSS projects [38–40].

Previous research on deprecated packages typically focused on the projects themselves, while lacking exploration into how declarations of deprecation affect other packages. Recently, Zhong et al. [12] and Miller et al. [13] have conducted studies on deprecated libraries in the Python and NPM ecosystems, respectively, exploring their impact on downstream dependencies. However, the deprecation methods examined in these two studies are not ecosystem-specific and primarily focus on deprecation practices found on GitHub. Since different ecosystems adopt different deprecation mechanisms, the findings of these studies cannot be directly applied to research within the Rust ecosystem. Moreover, neither study explores how developers identify deprecated libraries and the impact of deprecated libraries within the software supply chain. Therefore, our study on deprecated libraries in the Rust software supply chain helps fill a gap in the existing research on library deprecation. In Section 7.3, we discuss in detail how the results of our study differ from prior work and highlight the unique contributions.

2.2 Dependency Management in Cargo

The official package manager for the Rust programming language is Cargo[4]. In a Cargo project, to introduce a dependency on a third-party library, developers need to specify the library's name and its corresponding semantic versioning specification[5] in the project's configuration file, i.e., $Cargo.toml$[6]. Versioning specifications use symbols like "$*$", "$\wedge$", and "$\sim$" to define allowed version ranges. Version numbers follow the MAJOR.MINOR.PATCH format, representing breaking changes, backward-compatible features, and bug fixes, respectively [41]. This structure helps developers understand the level of change and potential impact when updating dependencies. For example, "^1.2.3" represents a range of "$\geq$ 1.2.3 and $<$ 2.0.0". If a library has multiple version numbers that meet the versioning specification, Cargo will select the highest version that meets the criteria as a dependency for the project.

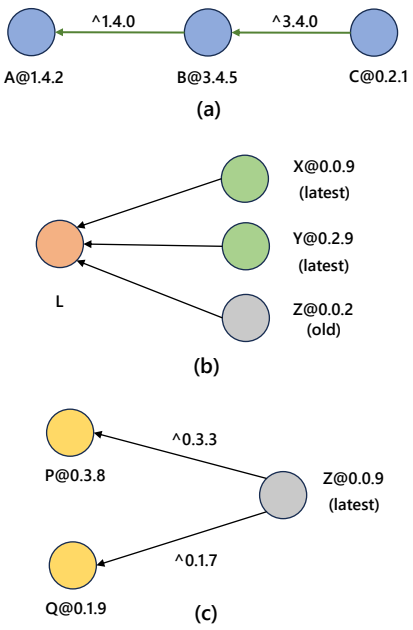

Fig. 1: Dependency Relationships Between Packages

2.3 Terminology

Dependency Types: If we consider a specific version (x.y.z) of a package (P) as a node (denoted as $P@x.y.z$), then the various versions of packages will form a dependency network structure. Figure 1(a) illustrates the dependency relationships between three packages, A, B, and C. $B@3.4.5$ depends on $A@1.4.2$, and $C@0.2.1$ depends on $B@3.4.5$. We refer to these adjacent dependencies as direct dependencies. For example, $B@3.4.5$ is a direct downstream package of $A@1.4.2$, and $A@1.4.2$ is a direct upstream Package of $B@3.4.5$. Dependencies that are not adjacent are referred to as transitive dependencies. For example, $C@0.2.1$ is a transitive downstream package of $A@1.4.2$.

Direct Downstream Packages: In the Rust package management system, if Package B wants to stop depending on Library A, Package B needs to create a new version in which Library A is not used. However, in the previous versions, Package B still depended on Library A. Therefore, to determine the true direct downstream packages of Library A at a specific moment, when the highest version of a direct downstream Package B is currently dependent on Library A, we consider Package B as the direct downstream package of Library A at that moment. We consider the highest version of Package B, which depends on Library A, as the current latest intent of Package B. In Figure 1(b), $X@0.0.9$, $Y@0.2.9$, and $Z@0.0.2$ all directly depend on Library L. Versions of Package X and Y are the highest versions currently, while the version of Package Z is not the current highest version. Figure 1(c) shows the dependency situation of the current highest version of Package Z, where Package Z only directly

4. https://doc.rust-lang.org/cargo/guide/why-cargo-exists.html#enter-cargo
5. https://github.com/steveklabnik/semver
6. https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html

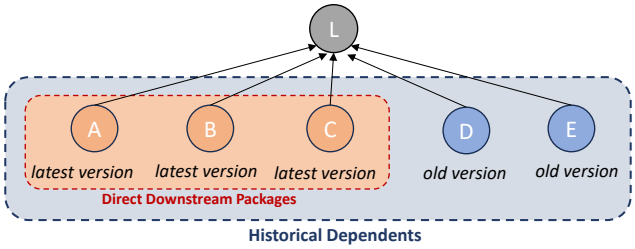

Fig. 2: Direct Downstream Packages of Library L

depends on Libraries P and Q, without a direct dependency on Library L. Therefore, we consider Package X and Y as the direct downstream packages of Library L at the moment, while Package Z is not a direct downstream package of Library L. In other words, the current number of direct downstream packages of Library L is 2.

Historical Dependent Count: The number of direct downstream packages for a library can vary at different points in time, and it may not necessarily increase, as some packages may abandon their dependencies on it. Therefore, having a small number of current direct downstream packages for a library does not necessarily imply that it was seldom used in the past. To reflect the historical usage of a library, we use "History Dependent Count" to represent the number of times a library has been directly depended upon by other packages since its creation. In Figure 1(b), Packages X, Y, and Z each have a version that directly relies on Library L, so the History Dependent Count for Library L is 3.

Figure 2 intuitively illustrates the above concept. For library L, packages A, B, C, D, and E directly depend on library L. Packages A, B, and C are all in their latest versions, whereas the latest versions of packages D and E no longer depend on library L. Therefore, the set of Direct Downstream Packages for library L consists of packages A, B, and C. The Historical Dependents of library L include A, B, C, D, and E, and the Historical Dependent Count of library L is 5.

3 Dataset

As the official package index website for Cargo, crates.io provides the official database dumps[7] of all packages' metadata. With this service, we can easily download the dataset that details the basic information of Rust packages and their dependencies. Table 1 describes the key fields in the dataset that are primarily used in this study, and their respective descriptions. Our research uses data exported on January 17, 2024, which records basic information of 134,224 packages and 1,001,235 releases.

To facilitate subsequent data processing, we performed an inner join on the three tables `crates`, `versions`, and `dependencies` based on the `crate_id` and `version_id`, creating a new table named `crates_dep`. This table records each version of each package's version dependency range for every direct upstream dependency. The main fields and examples are shown in Table 2, which contains a total of 9,068,558 dependency records.

7. https://crates.io/data-access

TABLE 1: Dataset Description

Table	Field	Description
crates	id	The identifier of a package.
	readme	The content of a package's README in crates.io.
	repository	The URL of a package's source code repository.
	description	The description of a package.
	downloads	The total number of downloads of the package as of the time of data collection.
badges	attributes	The values corresponding to each type of badge.
	crate_id	The id of the package that possesses the badge.
versions	id	The identifier of a release.
	crate_id	The id of the package to which a release belongs.
	created_at	The time when a release was published.
	yanked	The flag indicating whether a release is deprecated.
	num	The version name of a release.
dependencies	version_id	The id of the release to which this dependency belongs.
	req	The dependency scope of a release on a direct upstream package.
	crate_id	The direct upstream package id that this dependency points to.

TABLE 2: Field Description of crates_dep Table

Field Name	Example	Description
name	A-Mazed	Package name (unique in crates.io)
num	0.1.0	Version number
dep_name	rand	Name of the direct upstream dependency
req	^0.8.5	Version requirement specification for the dependency
created_at	2018-06-07	Creation timestamp of the version

4 RQ1: Prevalence of Deprecated Libraries

4.1 Method

1. Selection of the libraries studied.[8] As of the date of our data collection, there were a total of 134,224 packages on crates.io, with 101,610 (75.6%) of packages having their source code available in GitHub repositories. Since GitHub's rich open API and data export capabilities can facilitate the study of library maintenance status and deprecation declarations, our experiment chose to investigate libraries having their source code available in GitHub repositories. Therefore, we excluded packages that do not provide a GitHub repository URL. Besides, this study aims to explore the impact of library deprecation on downstream projects that depended on them, therefore, we excluded packages that had never been used by other packages from our study. Never being used by other packages means that the Historical Dependent Count of this library is 0. If the name of a certain package appears in the *dep_name* field of the *crates_dep* table (Table 2), it indicates that the Historical Dependent Count is greater than

8. The method for selecting libraries in this section is specific to RQ1. In other sections of the paper, the scope of Rust packages is based on all packages available on crates.io.

0, because this package has been depended upon by other packages. Therefore, it is only necessary to exclude those packages whose name does not appear in the dep_name field of the table *crates_dep*.

As a result, we collected 40,636 packages with no less than one Historical Dependent Count and open source code on GitHub as the libraries for our research.

2. Identification of inactive libraries. Before studying deprecated libraries, we first identify libraries that have not been maintained for a long time and consider them inactive libraries, which are potential deprecation candidates. Existing studies considered OSS projects with no commits for over a year as inactive [7, 42, 43]. Therefore, following prior studies, we used the absence of commits for more than one year as one of the criteria to identify inactive repositories. If a longer threshold—such as two or three years—were applied, the stricter filtering would exclude a considerable number of inactive packages. Specifically, applying two-year and three-year thresholds would retain only 66% and 42% of the inactive libraries identified using the one-year threshold, respectively, which would in turn affect the total number of deprecated libraries included in our analysis[9].

Additionally, different libraries may have different development rhythms [44, 45]. If a library takes longer than expected to release a new version, its development status may have changed. Inspired by these criteria, we consider a library to be inactive when it meets both of the following conditions:

1) We identified repositories with no commits for over a year by retrieving the last commit timestamps via the GitHub REST API[10].
2) The time since the last release of a new version exceeds any previous adjacent version release time interval. (This condition is considered only when this library has at least two releases.)

To validate the effectiveness of our methods for identifying inactive libraries, in Section 8, we conducted a spot check on inactive libraries and confirmed that the vast majority do not become active again.

3. Identification of deprecation-declared libraries. We examined the following information to confirm whether these inactive libraries declare deprecation. If a library uses at least one of the methods mentioned below, we consider it to be a deprecated library.

First, GitHub and crates.io provide mechanisms as follows to directly indicate the maintenance status of the library.

1) *Archiving in GitHub*. GitHub launched the Archive Feature that allows users to explicitly declare the project as deprecated and set the repository to a read-only state [37]. We used the GitHub REST API to determine if a GitHub repository is in an archived state.
2) *Maintenance Badge*. Developers of libraries can add badge-related configurations in the $Cargo.toml$ file to indicate the maintenance status of the library, such as $maintenance = \{status = \text{“}actively - developed\text{”}\}$. This allows for specifying status badges that can be displayed on crates.io when the package is published[11]. Records related to badges can be obtained from the data collected in Section 3. We considered that when a library's maintenance badge is *“deprecated”*, it signified the library had been declared deprecated.
3) *Yanking all releases*. When a package yanks all of its releases, none of the versions of the library can be used, and the package developers can use this method to voluntarily deprecate the package [35]. We identified libraries that yanked all releases using the data introduced in Section 3 and considered them as libraries deprecated using this method. It is worth noting that this method represents an informal deprecation strategy intentionally adopted by developers, rather than an official technical mechanism provided by Cargo.

Second, the homepage of GitHub and crates.io provide **description** information and the content of **README** for the project, allowing developers to publish information related to the maintenance status of the project at these locations to inform users of the current status of these projects. We used the GitHub REST API to obtain the description on the GitHub homepage and the content of the README file. The description content on the crates.io homepage and the content of the README file can be obtained from the data collected in Section 3.

After obtaining the text content at these four locations for all inactive libraries, we filtered the text at these four locations using keywords such as 'deprecate', 'discontinue', 'archive', 'abandon', and similar terms, including their root forms and variations, to select text containing these keywords. We provided the full list of keywords in our online appendix[12]. Subsequently, the first two authors of the paper judged whether these filtered texts could indicate the maintenance status of the library. Developers can explicitly declare deprecation by including relevant text descriptions in the *description* or *README*, such as: "This crate is deprecated in favor of *[another Rust crate name]*." We could determine whether it is deprecated by analyzing the semantic meaning of the text. Additionally, in the README, some developers used deprecation badges to indicate abandonment, such as a badge labeled "No Maintenance Intended."[13] When we identified such badges in the README, we also considered the content as confirmation that the corresponding library was deprecated.

Sometimes, the deprecation-related descriptions in the *description* or *README* are not clearly stated. For example, some libraries claim to be "soft-deprecated." In these special cases, the judgments of the two authors might differ. When discrepancies arose, the two raters reached a consensus by analyzing relevant information such as the library's commits and issues. The Cohen's kappa coefficient of agreement between them was 0.89, indicating a high level of consistency. Finally, we separately tallied the number of libraries declared deprecated at these four locations.

4. Calculation of Direct Downstream Packages Count. To determine the number of direct downstream packages of a

9. After a thorough examination, we found that using two- and three-year thresholds would reduce the number of identified deprecated libraries to 60% and 54% of the original, respectively.

10. https://docs.github.com/en/rest

11. https://doc.rust-lang.org/cargo/reference/manifest.html#the-badges-section

12. https://doi.org/10.5281/zenodo.15266092

13. https://unmaintained.tech/

library L at a given time T, we extracted all packages whose latest versions prior to T declared a direct dependency on L, based on records in Table 2. We then removed duplicates by package name to ensure each package was counted only once. The final count of unique package names represented the number of direct downstream packages of L at time T.

5. Calculation of Historical Dependent Count. To compute the Historical Dependent Count of a library L at time T, we used the `crates_dep` table (Table 2) to obtain all packages that had declared a direct dependency on L before T. We then extracted the unique package names, and the number of these distinct packages represented the Historical Dependent Count of L at time T.

4.2 Results

Among the 40,636 open source Rust libraries that host their repositories on GitHub, we identified a total of 13,289 inactive libraries. It means that 32.7% of open source Rust libraries have ceased receiving code changes for over a year. The blue line in Figure 3 shows the change in the proportion of inactive libraries, measured as the number of inactive libraries relative to the total number of packages whose repositories were hosted on GitHub each month from January 2017 to January 2024. The figure illustrates that as the number of packages stored on crates.io increases, the number of libraries that have ceased receiving commits also rises. Moreover, the proportion of these inactive libraries is growing over time.

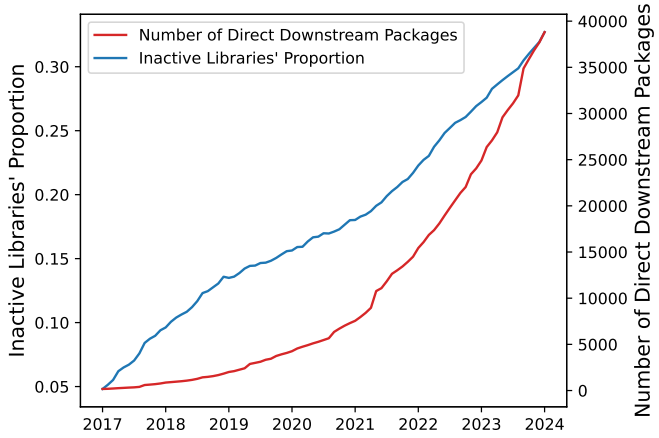

Fig. 3: Trend of Inactive Libraries and Their Direct Downstream Packages

The red line in Figure 3 illustrates the number of direct downstream packages of these inactive libraries from 2017 to 2024. It indicates that over time, more and more packages have incorporated inactive libraries. Specifically, as of January 17, 2024, the number of Rust packages that are dependent on inactive libraries has reached 38,808.

Among the 13,289 inactive libraries, a total of 1,468 libraries declared themselves deprecated using at least one of the seven methods. The specific usage numbers of these seven methods are shown in Figure 4. Among all the deprecation methods, archiving the GitHub repository is the most widely used, with a total of 1,167 (8.8%) libraries using this method to actively deprecate their repositories. Additionally, a library may use multiple methods to declare

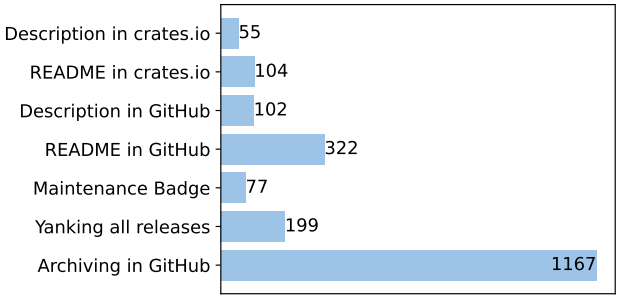

Fig. 4: Number of Libraries Using Each Deprecation Method

TABLE 3: Comparison of Download Count Between Deprecated and Non-Deprecated Libraries Among Inactive Libraries

Metric	Libraries with Deprecation Declarations	Libraries without Deprecation Declarations
mean	3.2×10^5	5.1×10^5
min	7.8×10^1	1.1×10^2
Q1	1.5×10^3	1.3×10^3
Q2	4.3×10^3	3.6×10^3
Q3	2.3×10^4	1.7×10^4
max	2.9×10^7	2.7×10^8

deprecation. For example, a total of 28 libraries have used *Archiving in GitHub*, *README in GitHub*, and *Description in GitHub* to declare their deprecation.

Table 3 presents a comparison of download counts between 1,468 deprecated libraries and 11,821 non-deprecated libraries among a total of 13,289 inactive libraries. We use Download Count as an indicator of a library's popularity, and employ the mean, minimum, maximum, and quartiles (Q1, Q2, Q3) to illustrate the data distribution. We used the Mann-Whitney U Test [46] to compare the statistical differences between deprecated and non-deprecated libraries in terms of download count. We calculated Cliff's delta (d) to illustrate the effect size of the differences between the data. The results show that deprecated and non-deprecated libraries exhibit negligible effect sizes[14] in download count (d = -0.05). This suggests that there is no significant difference in popularity between deprecated and non-deprecated libraries. Despite prolonged inactivity, only 11% of libraries explicitly declare deprecation, leaving the maintenance status of most packages ambiguous to users.

Summary for RQ1: In all the 40,636 open source Rust libraries, we identified 13,289 (32.7%) inactive ones, which are widely utilized by 38,808 packages and play a critical role in the Rust ecosystem. However, only 11% of the inactive libraries have declared their deprecation on crates.io or GitHub. It brings uncertainty to users whether the majority of the remaining inactive libraries are still being maintained.

14. An effect size can tell us how large this difference is. As suggested in [47], we interpret the effect size value to be negligible for $|d| < 0.147$, small for $0.147 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$ and large for $|d| \geq 0.474$.

5 RQ2: Impacts of Deprecation Declaration

5.1 Method

We examined the impact of deprecation declarations on their users by calculating the changes in direct downstream packages of the library.

1. Selection of study subjects. Compared to other deprecation methods, yanking all releases is a more mandatory approach. It renders all library versions unavailable, causing issues during the build process when this library is introduced into a project. In contrast, other deprecation methods only provide warnings, which might not be noticed unless specifically checked in the corresponding locations. Therefore, we divided the seven deprecation methods into two groups for our study. The first group, called **yanked_lib**, includes libraries deprecated through yanking all releases. The second group, called **declared_lib**, includes libraries deprecated using at least one of the other five methods, i.e., REAMDE in crates.io, Description in crates.io, README in GitHub, Maintenance Badge, Archiving in GitHub. We excluded libraries deprecated only through a GitHub description because we could not determine the exact time of deprecation, and there were only 16 such libraries, with an average of one direct downstream package. Therefore, we deemed that excluding them did not significantly impact the results.

To avoid situations where downstream libraries do not have enough time to react due to a short deprecation period, we excluded libraries from the $yanked\_lib$ and $declared\_lib$ sets where the deprecation duration (the time from the deprecation point to the data collection time) is less than 6 months. This led to the exclusion of 2 libraries from $yanked\_lib$ and 150 libraries from $declared\_lib$. Ultimately, $yanked\_lib$ contains 197 libraries, and $declared\_lib$ contains 1,019 libraries.

2. Retrieving the deprecation declaration time. For libraries deprecated through yanking all releases and those with a deprecation notice in the README on crates.io, we obtained the deprecation time from the data exported from crates.io. For the Maintenance Badge and Description in crates.io methods, the corresponding fields were recorded in the $Cargo.toml$ file. We identified the earliest version in the $Cargo.toml$ file where the deprecation notice was added and used the release date of that version as the deprecation date. For libraries that declare deprecation in the README on GitHub, we check the commit history of the *README.md* file to identify when the deprecation declaration was added. Lastly, by scraping the GitHub homepage of archived libraries, we could obtain the time when the archive status was applied. Libraries in the $declared\_lib$ group may use more than one method to declare deprecation. In this case, if the corresponding deprecation times differed, we chose the earliest time as the official deprecation declaration time for that library.

3. Calculation of downstream package changes. There are three types of changes in the set of direct downstream packages of a certain library l between time $t1$ and a subsequent time point $t2$. We introduce each type as follows.

- Packages that were dependent on library l at time $t1$ abandon their dependency by time $t2$. We define the collection of these packages that have abandoned their dependency as del_pkg_{t1-t2}.
- Packages that were dependent on library l at time $t1$ continue to maintain their dependency on library l at time $t2$. We define the collection of these packages that persist in their dependency as $stay\_pkg_{t1-t2}$.
- Packages did not depend on library l at time $t1$ but have established a dependency on library l by time $t2$. We define the collection of these newly dependent packages as add_pkg_{t1-t2}.

We denote the set of all direct downstream packages of a certain library at a specific time t as ddp_set_t. We calculated each deprecated library's set of direct downstream packages at deprecation declaration time, denoted as ddp_set_{stop}. The calculation method for direct downstream packages is described in Section 2.3. Subsequently, we computed the set of direct downstream packages of each deprecated library at the time of data collection (January 17, 2024), labeled as ddp_set_{now}. The calculation methods for each library's del_pkg_{s-n}, $stay\_pkg_{s-n}$, and add_lib_{s-n} (the subscripts s and n represent "stop" and "now" respectively) are as follows (the minus sign denotes the set difference):

$$del\_pkg_{s-n} = ddp\_set_{stop} - ddp\_set_{now}$$

$$stay\_pkg_{s-n} = ddp\_set_{stop} \cap ddp\_set_{now}$$

$$add\_pkg_{s-n} = ddp\_set_{now} - ddp\_set_{stop}$$

We got ddp_set_{stop}, del_pkg_{s-n}, $stay\_pkg_{s-n}$, and add_pkg_{s-n} for each library in $yanked\_lib$ and $declared\_lib$. This allows us to gain insights into how the number of downstream dependencies changed for each library after it declared deprecation.

4. Identifying alive packages. For the set of direct downstream packages at the time of deprecation for each library, we want to determine how many packages are still "alive" in order to calculate the proportion of packages that had the opportunity to replace the deprecated library but did not. From the packages in ddp_set_{stop}, we chose those that released a new version after their direct upstream library was declared deprecated. These packages were then grouped into a new set called $alive\_ddp\_set_{stop}$, which represents the number of direct downstream packages that were still well-maintained when the upstream library was deprecated.

5. Assessing the effectiveness of deprecation declarations. Regardless of the deprecation declaration, direct downstream libraries may stop depending on the deprecated library for various reasons, such as dissatisfaction with its functionality or insufficient maintenance [37]. What we aim to understand here is whether the act of declaring a library as deprecated facilitates developers' decisions to stop using it. Additionally, we also want to examine whether such declarations help prevent new packages from adopting these deprecated libraries. These questions need to be addressed by showing statistical differences in the data before and after the deprecation announcement.

Specifically, we compared the number of newly added downstream packages (add_pkg) and the number of removed downstream packages (del_pkg) in the N months before and after the deprecation declaration. In this experiment, we set N to 2, 4, and 6 months as the time window. We did

TABLE 4: Changes in Direct Downstream Packages of Two Categories[16]

Categories	# libraries	# ddp_set_{stop}	# $alive\_ddp\_set_{stop}$	# del_pkg_{s-n}	# add_pkg_{s-n}	# $stay\_pkg_{s-n}$
$declared\_lib$	1,019	7,432	2,300	1,405	1,783	6,027
$yanked\_lib$	197	242	29	29	0	213

TABLE 5: Mann-Whitney Test (p-value) and Cliff's Delta (d) for N Months Before and After Deprecation[15]

Month	del_pkg		add_pkg	
	p-value	d	p-value	d
2	0.74	-3.8×10^{-3} (negligible)	1.2×10^{-2}	0.025 (negligible)
4	0.36	-1.3×10^{-2} (negligible)	3.3×10^{-5}	0.054 (negligible)
6	0.44	-1.1×10^{-2} (negligible)	2.6×10^{-7}	0.078 (negligible)

not choose overly short observation windows to reduce the impact caused by the direct downstream libraries not having sufficient time to remove deprecated libraries. In this section, we chose deprecated libraries with the same deprecation declaration methods as those in $declared\_lib$ as our research subjects. These libraries form the set $compared\_lib$. For each N, we selected libraries from the $compared\_lib$ set whose creation-to-deprecation time was greater than N months and whose deprecation duration (the time from the deprecation point to the data collection time) was longer than N. We then calculated the ddp_set for these libraries at three time points: N months before the deprecation declaration, N months after the deprecation declaration, and at the time of the deprecation declaration. Subsequently, we calculated the number of direct downstream packages that were removed (del_pkg) and added (add_pkg) for each library, using the same method as described in Step 3 in this section.

We used the Mann-Whitney test [46] to compare the statistical significance of the differences in del_pkg between the N months before and after deprecation. The same test was also applied to add_pkg. We calculated Cliff's delta (d) [14] to illustrate the effect size of the differences between the data. The results of these calculations are shown in Table 5[15].

5.2 Results

We summed the values of ddp_set_{stop}, $alive\_ddp\_set_{stop}$, del_pkg_{s-n}, $stay\_pkg_{s-n}$, and add_pkg_{s-n} for all libraries in the $declared\_lib$ category, and performed the same operation for the $yank\_lib$ category, resulting in the data presented in Table 4[16]. It shows how the direct downstream packages in these two categories changed overall after the libraries were deprecated.

Downstream packages of Libraries in yanked_lib. As shown in Table 4, there are 197 libraries in the $yanked\_lib$ category that have been deprecated by yanking all their releases. These libraries had a total of 242 direct downstream packages, of which only 29 were "alive". After these libraries yanked all their releases, by the time of data collection, all 29 alive downstream packages had removed their dependency on these libraries, and no new packages chose to depend on them. This data demonstrates that yanking all releases is indeed a very drastic deprecation measure. Libraries using this method become unavailable, effectively forcing downstream packages to remove them. This approach is suitable for libraries with significant defects, as it effectively prevents both existing and new users from using them.

We examined the code repositories of these 29 packages that removed their dependencies on deprecated libraries by analyzing the commit history of their Cargo.toml files. We found that all of them identified the deprecated libraries in use and replaced them. For example, *echonet-lite* originally depended on the *bare-io* library. After *bare-io* was deprecated by yanking all releases, *echonet-lite* chose to replace it with *core2*, explicitly mentioning in the commit message: "use core2 because bare-io is yanked."[17]

Existing Cargo projects will generate a $Cargo.lock$ file to save resolved dependencies, so projects that have already been built can still use libraries with all releases yanked, allowing some leeway for developers to replace these deprecated libraries. However, for any new projects that do not have a pre-existing lock file, using libraries with all releases yanked still poses serious issues, leading to dependency resolution failures[18]. These problems also affect the transitive downstream packages of the deprecated libraries, causing broader risks across software supply chains. As shown in Figure 5(b), there are currently 164 packages (after deduplication of the 213 packages in the *stay_pkg* collection) that depend on libraries where all releases have been deprecated.

15. The data under the del_pkg column represent the statistical comparison between direct downstream packages that abandoned the dependency within N months before the deprecation declaration and those that abandoned it within N months after the declaration.

The data under the add_pkg column represent the statistical comparison between direct downstream packages that newly added the dependency within N months before the deprecation declaration and those that newly added it within N months after the declaration.

16. $\#librarie$ represents the total number of libraries in each Category. $\#ddp\_set_{stop}$ denotes the total number of direct downstream packages of the libraries in each Category at the time of deprecation declarations. $\#alive\_ddp_{stop}$ refers to the number of alive direct downstream packages in each Category at the time of deprecation declarations (as defined in Section 5.1.4). $\#del\_pkg_{s-n}$, $\#add\_pkg_{s-n}$, and $\#stay\_pkg_{s-n}$ represent, respectively, the number of direct downstream packages that abandoned the dependency, newly added the dependency, or retained the dependency from the time of deprecation declaration to the time of data collection (as defined in Section 5.1.3).

17. https://github.com/tomoyuki-nakabayashi/echonet-lite-rs/commit/b3e0901b30de74647aa7419d314ab3c2328e6c94

18. https://doc.rust-lang.org/cargo/commands/cargo-yank.html

Additionally, these 164 packages are release-active[19]. These 164 packages have a total of 93 direct downstream packages. Because these downstream packages indirectly depend on libraries with all releases yanked, their builds will encounter issues.

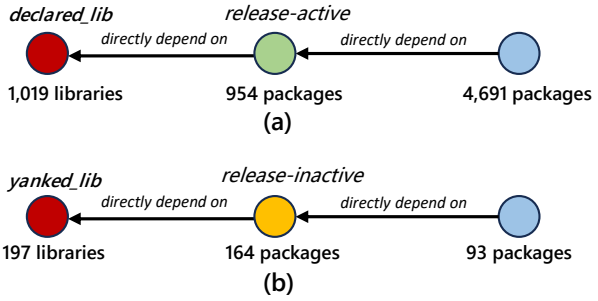

Fig. 5: Dependency Chain of Libraries in the $declared\_lib$ and $yanked\_lib$ sets[19]

Downstream packages of libraries in declared_lib. For the 1,019 libraries in the $declared\_lib$ category, when they were declared deprecated, there were a total of 7,432 direct downstream packages, including 2,300 alive packages. By the time of data collection, 1,405 (61.1%) of these alive packages chose to abandon their dependency on the deprecated libraries, while 38.9% of the alive packages continued to use the deprecated libraries in their new versions. Furthermore, 1,783 new packages have opted to use these libraries even though they have been declared deprecated. This indicates that many package developers either were unaware that they were using deprecated libraries or, for some reason, knew the libraries were deprecated but still chose to continue using them. This issue will be further researched in Section 6.

Figure 6 illustrates the number of newly added direct downstream packages (the red dashed line) and the number of packages that stopped depending on deprecated libraries (the blue solid line) during each of the first six months following the deprecation declaration. The trends of the red and blue lines indicate that both the number of new users and the number of downstream packages discontinuing use of deprecated libraries decline over time. The number of users who chose to remove deprecated libraries was highest in the first month after the deprecation declaration. However, some packages only removed the deprecated libraries after a longer period, such as five to six months later.

To investigate whether specific risks have been identified in these 1,019 already deprecated libraries, we searched the Open Source Vulnerabilities Database (OSV)[20] for unresolved vulnerabilities caused by libraries from the *declared_lib* set. Table 6 shows the number of libraries in the *declared_lib* set of 1,019 that produce vulnerabilities at each severity level[21], as well as the number of Direct Downstream Packages affected

19. In RQ2, "inactive" has a different meaning than in RQ1, as the downstream packages analyzed were drawn from all crates.io packages, many of which lack GitHub links. As a result, we use a simplified classification based on release activity: *release-inactive* packages had no new versions in the year before data collection ended, while *release-active* ones released at least one version during that period.

20. https://osv.dev/list?ecosystem=crates.io

21. The severity level definition of vulnerability comes from the Common Vulnerability Scoring System (CVSS, https://www.first.org/cvss/v3-0/specification-document#Qualitative-Severity-Rating-Scale).

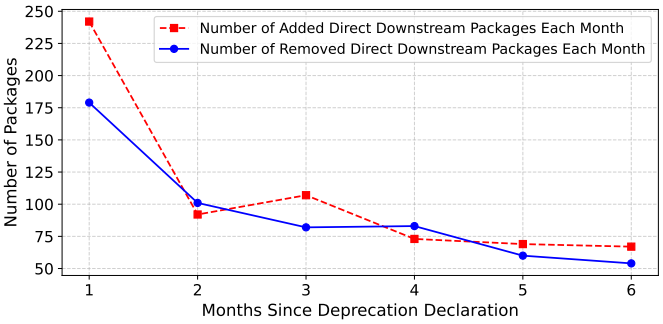

Fig. 6: Number of Direct Downstream Packages in the six months after the Deprecation of Libraries in *declared_lib*.

TABLE 6: Number of Direct Downstream Packages Affected by Deprecated Libraries in *declared_lib*

Severity	# Libraries	# Direct Downstream Packages
critical	2	1,906
high	3	5
undefined	23	3,375

by these libraries. In the OSV database, not all vulnerability records include a severity level. Therefore, the category for vulnerabilities without a specified severity level is labeled as "*undefined*" in Table 6.

As shown in Table 6, although only two deprecated libraries contain critical severity defects, these two libraries affect 1,906 Rust packages. Specifically, these two vulnerabilities are *CVE-2020-25575*[22] and *CVE-2020-35880*[23], which occur in the *failure*[24] and *bigint* [25] libraries, respectively. The critical vulnerability in *failure* introduces potential type confusion flaws during downcasting, as well as compatibility hazards in certain applications.[26] In contrast, the critical vulnerability in *bigint* contains several instances of undefined behavior, including the use of uninitialized memory and out-of-bounds pointer access.[27] The issues present in both libraries can potentially lead to severe runtime problems in their downstream packages. Since these libraries have been deprecated, their publicly known issues cannot be fixed, leaving thousands of dependent packages at risk.

We further studied the 6,088 packages that still rely on these deprecated libraries (i.e., the result of deduplicating the 6,027 packages in the $stay\_pkg$ collection and the 1,783 packages in the add_pkg collection). We found that only 954 (15.7%) of these packages have released new versions in the past year. These 954 packages are marked as release-active[19] in Figure 5(a). This means that most of the packages still relying on deprecated libraries are not actively maintained themselves. Additionally, the total number of direct downstream packages that depend on these 954 packages is 4,691. This indicates that for the developers of these 4,691 packages, the libraries they directly depend on are actively

22. https://nvd.nist.gov/vuln/detail/CVE-2020-25575
23. https://nvd.nist.gov/vuln/detail/CVE-2020-35880
24. https://crates.io/crates/failure
25. https://crates.io/crates/bigint
26. https://rustsec.org/advisories/RUSTSEC-2019-0036.html
27. https://github.com/advisories/GHSA-wgx2-6432-j3fw

maintained, but in reality, their packages indirectly introduce deprecated libraries. Figure 5(a) visually illustrates the above phenomenon, showing that the declaration of deprecated libraries in the supply chain is "hidden" due to the presence of intermediary libraries.

Effectiveness of deprecation declarations. For the case of 2 months (N=2), we selected 1,056 libraries from the $compared\_lib$ set that met the criteria. During the 2 months before their deprecation declaration, a total of 233 direct downstream packages ceased using them. In the 2 months following their deprecation, a total of 280 direct downstream packages stopped using these deprecated libraries. Although there are more direct downstream packages that discontinued their dependency on deprecated libraries in the two months after the deprecation declaration (280 *vs.* 233), there is no significant difference in the overall distribution regarding whether the deprecation declaration led existing users of the library to stop using it (p-value of 0.74, which is greater than 0.05). When N takes other values, similar results are observed. The results suggest that deprecation notices may fail to reach existing users effectively.

Regarding the effect of deprecation declarations on preventing new packages from adopting deprecated libraries, the results in Table 5 indicate a statistically significant difference between deprecated and non-deprecated libraries in terms of new downstream packages (p-values significantly less than 0.05 for all months we measured). For example, in the four months prior to deprecation, 740 new direct downstream packages were added, whereas this number dropped to 514 in the four months after deprecation. The add_pkg count thus shows a decline following the deprecation declaration. However, the effect size is negligible, suggesting that while the signal is not entirely ignored by developers, its practical impact on developer behavior is minimal. One possible explanation for the observed, albeit small effect, is that some potential adopters may check a library's homepage before adoption, notice the deprecation warning, and decide not to adopt the library. Nevertheless, the current measures remain insufficient in effectively notifying developers, highlighting the need for more robust and standardized deprecation communication mechanisms.

Summary for RQ2: Yanking all releases is a coercive deprecation method that effectively forces all alive downstream packages to abandon their dependencies on deprecated libraries. Other deprecation declarations serve merely as warnings and fail to trigger downstream updates effectively, as 61.1% of alive packages continue to rely on the deprecated libraries. More actionable mechanisms for communicating deprecation status are needed.

6 RQ3: Reasons for Developers Using Deprecated Libraries

In RQ2, we found that despite 1,019 libraries being declared deprecated, many active packages still rely on these libraries. To understand the reasons behind this and how Rust developers currently become aware of the presence of deprecated libraries in their projects, we analyzed first-hand information by surveying developers.

6.1 Method

To improve the response rate of the questionnaire, we targeted Rust packages that are still under maintenance and depend on deprecated libraries. The filtering process was as follows: in RQ2, the $declared\_lib$ set contained a total of 1,019 deprecated libraries. We obtained all direct downstream packages of these 1,019 deprecated libraries at the data collection point (following the method described in Section 4.1.4). From these direct dependents, we then filtered out the packages that had released a new version during 2023, resulting in a final set of 1,006 packages.

Among these 1,006 packages, we found public emails of the owners for a total of 546 libraries. We sent our survey invitation via email to the owners of these libraries. In the invitation email, we first revealed that the project *[Project-Name]* they are involved in maintaining, is using a deprecated library *[Library-Name]*, then invited them to answer our survey, which mainly contains the following questions:

1) How long have you been involved in Open Source Software?
2) How long have you been using Rust?
3) In your past development experiences, how have you come to know about the presence of deprecated libraries in the project?
4) Since the project you are highly involved in relies on a deprecated library, could you tell us the reason for this?

We also informed the research aim of this study and the anonymous way of response handling policy in the inviting letter to ease the ethical concern. The survey passed the review by the ethics board at our university. One month after sending the questionnaire to these 546 developers, we received a total of 53 responses. The response rate was 9.7%, higher than the typical 5% response rate for software engineering-related surveys [48]. To protect the anonymity of the survey participants, we used tags P1 to P53 to identify these participants. In the results, we will reference some of the participants' answers, and when referring to specific package names, we will use *[Package-Name]* instead.

Figure 7 shows the duration of respondents' involvement in OSS development and Rust development. 86.8% of the respondents have over three years of experience in open source development, and 60.4% have over 3 years of experience as Rust developers. This means most of the respondents have extensive experience in Rust and open source development.

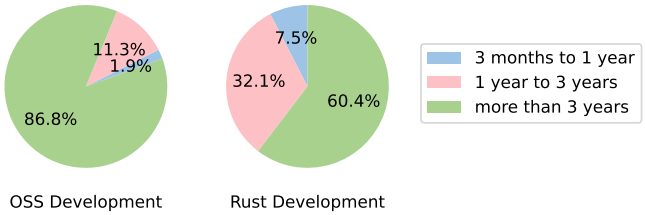

Fig. 7: Participants' experience in OSS and Rust Development, respectively.

We organized and analyzed the responses to the open questions in the survey using thematic analysis [7, 49]. Two

authors of this paper independently assigned an initial code to each response and identified themes within the assigned codes. For example, we extract the code *"didn't receive any deprecation warning"* from a participant's response, which generates the corresponding theme *"unawareness of deprecation"*. After generating corresponding themes for all responses, we merge similar themes to obtain the final results. For the analysis of the third and fourth questions, the Cohen's kappa coefficient of agreement between the two authors was 0.82. In cases where the first two authors have different perspectives, these differences are resolved through a face-to-face meeting.

6.2 Results

Ways developers have historically discovered deprecated libraries. Among the 53 respondents, a total of 41 developers indicated that they have experience in discovering deprecated libraries.[28] Table 7 shows how these 41 participants found deprecated libraries in projects during their past development experiences. Note that a participant can mention multiple methods, and 12 participants indicated that they had no relevant experience. Based on the responses of these experienced developers, it can be observed that within the current open source ecosystem, Rust developers can actively or passively become aware of the presence of deprecated libraries in projects through various channels of information.

Table 7 indicates that in addition to checking the library's GitHub and crates.io pages, developers can also use the "*cargo audit*" command and *Dependabot* to identify deprecated libraries within their projects. These two tools use the RustSec Advisory Database (RAD) [50] and the GitHub Advisory Database (GAD) [51] as their respective data sources. We downloaded the data as of January 17, 2024 to examine how many deprecated libraries are recorded in these two databases. Similar to the approach in RQ1, we utilized keywords related to deprecation to filter the "summary" and "details" fields of each risk record in these two databases. Subsequently, two authors of this paper individually examined each filtered record to determine if it documented deprecated libraries. Discrepancies in analysis results were resolved through discussion to reach a consensus (Cohen's kappa coefficient of agreement between them was 0.84). Before the data collection date, RAD and GAD only documented 85 and 4 deprecated Rust packages, respectively. Therefore, using the *"cargo audit"* command or Dependabot can only uncover a small fraction of deprecated libraries, rendering their actual effectiveness quite limited.

Reasons developers use abandoned libraries. Table 8 displays four major categories of reasons why developers continue to use deprecated libraries. The last column shows the number of responses for each major category. Among the major categories of reasons, the insightful specific reasons are listed in the "Specific Reasons Mentioned in Responses" column. The number in parentheses following each specific reason indicates how many developers mentioned it. A participant's response may mention multiple specific reasons within its associated category. It's also possible that no specific reasons are mentioned. Therefore, for a given category, the sum of the numbers in parentheses in the *Specific Reasons Mentioned* column may be smaller than the value in the last column. For example, P15 simply responded, *"I learned only now the lib was deprecated."* This indicates that the developer was previously unaware of the deprecation (and thus the response falls under Unawareness of deprecation), but it does not specify the concrete reason why the developer was unaware. Consequently, P15's response does not belong to any specific item in the Specific Reasons Mentioned column.

The detailed descriptions of these four categories are listed below based on their popularity:

R1. Unawareness of library deprecation (29, 54.7%). The most commonly cited reason for using deprecated libraries is that the package developers are unaware that the package uses deprecated libraries. Despite these libraries being declared deprecated, using them has not caused any issues. Additionally, users of these libraries have not received any notifications from Cargo or GitHub. For example, P15 mentioned, *"Deprecation warnings aren't emitted by any tool I use there ..."* and P27 wrote, *"I don't recall any notification of this library being deprecated."* Two respondents indicated that they were unaware of existing automated methods for detecting deprecated libraries. For example, P41 stated, "*I'm not aware of any automated way to detect unmaintained/deprecated crates/dependencies.*"

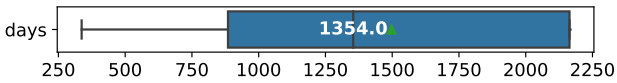

Fig. 8: Number of Days Since Deprecation for Libraries Used by the Respondents' Package.

We calculated the duration of deprecation for each deprecated library mentioned in our emails that was used by the respondents (measured as the number of days from the deprecation declaration to the data collection date). The results are presented in the form of a boxplot in Figure 8. The shortest deprecation period for the libraries used by these respondents is 337 days, indicating that the deprecated libraries used by the respondents have been obsolete for a significant amount of time.

We examined the projects maintained by the respondents and confirmed that all of these projects had released new versions after the libraries they used were declared deprecated. This indicates that the projects remained active after the deprecation declarations, and the developers had opportunities to replace the deprecated libraries. However, the results show that despite the deprecation being declared for a long time, 54.7% of the developers were still unaware that their projects contained deprecated libraries. This suggests that existing deprecation declaration methods are still insufficient for effectively notifying users of deprecated libraries.

Among the 29 respondents who were unaware of the presence of deprecated libraries in their projects, 3 explicitly stated in their responses that they would remove the deprecated libraries. For example, P51 mentioned, "*I will fix it as soon as I find the time for it.*" This indicates that if

28. This investigation solely focuses on whether developers have had similar experiences, which is independent of whether they discovered the particular deprecated libraries mentioned in the email. A total of 41 developers shared their experiences of discovering deprecated libraries, while the other 12 developers indicated that they had no such experience.

TABLE 7: Ways to Discover Deprecated Libraries in Projects

Methods	Description	# Response
Checking the library's homepage on crates.io	Developers determine if the library is deprecated by examining its homepage on crates.io.	27
Checking the library's homepage on code hosting platforms	Developers determine if the library is deprecated by examining its homepage on its code hosting platforms (such as GitHub).	21
Using "*cargo audit*" command	Cargo provides the *"cargo audit"* command to check a project's dependencies for known security vulnerabilities. This command can detect deprecated libraries in a project based on the RustSec Advisory Database [50].	18
Other developers submitted Issues or Pull Requests to notify	For open source projects on platforms like GitHub, when users of a project discover that deprecated libraries are being used, they can alert the project maintainers through Issues or Pull Requests.	15
Configuring dependabot	By configuring dependabot for projects on GitHub, it can check for deprecated libraries in projects based on the GitHub Advisory Database [51].	2

TABLE 8: Reasons for Developers Using Deprecated Libraries

Categories	Specific Reasons Mentioned in Responses	# Response
Unawareness of deprecation	1) The maintainer has not received any relevant warnings from Cargo or GitHub. (3) 2) The maintainers are unaware of the methods for automatically detecting deprecated libraries. (2)	29
Insufficient time and resources for library replacement	1) Replacing the library requires considerable work. (3) 2) Replacing the library is not the top priority in the current project. (1)	12
No need for replacement	1) Using deprecated libraries has not caused any problems. (5) 2) The deprecated library is used for testing. (1) 3) The deprecated library is used only for a very minor functionality. (1) 4) The mechanism on which the deprecated library depends is relatively simple. (2)	10
No appropriate alternative	1) Similar libraries are also not actively maintained. (1) 2) Similar libraries can't fully replace current ones. (1)	2

deprecation declarations can more effectively notify users of deprecated libraries, they could better encourage developers to stop using them.

R2. Insufficient time and resources for library replacement (12, 22.6%). Three developers indicate that even though they are aware of deprecated libraries in their projects, replacing them would require a significant amount of time and effort, potentially introducing a series of new issues to the project. For example, P4 mentioned, *"[...] changing it would further require possible debugging and new issues that we do not have the bandwidth to handle right now."* P24 also states that replacing deprecated libraries is not the most critical task at the moment, as the package has more urgent tasks to complete. P24's response mentioned, *"The most important thing about this project at the moment is not about it, but it may be in the future."*

R3. No need for replacement (10, 18.9%). Five developers believe that even though a library is no longer maintained, using the deprecated library has not caused any issues so far, so there is currently no need to replace it. They prefer to wait until a real problem arises before removing the deprecated library from the project. For example, P12 stated, *"Deprecated does not mean it's not working. It does what I need, I'll change it if/when it breaks."* P21 considers it acceptable to use deprecated libraries for testing purposes. P21's response mentioned, *"It is only used in the test suite. [...] Even if an attacker got access through the dependency, it won't be able to do anything."* Some developers believe it's acceptable to use deprecated libraries in projects when these libraries play a minor role or if the design mechanisms of the library are relatively simple. For example, P48 mentioned: *"I use the [Package-Name] whose underlying mechanisms are simple and unlikely to need maintenance updates."*

R4. No appropriate alternative library (2, 3.8%). Some developers are unable to find an appropriate library to replace the deprecated library they are currently using, which reduces their willingness to remove the deprecated library. For example, P34 mentioned, *"Out of the 2 libraries currently reported as 'maintained' replacements, one is just as much unmaintained [...]. The other one is not a 1:1 replacement and requires additional work without any clear benefits."*

The results in Table 7 and Table 8 provide a strong explanation for our findings in RQ2, namely that most of the maintained packages continue to rely on upstream libraries even after those libraries have been formally deprecated. Currently, Rust developers still largely depend on actively searching for information to discover deprecated libraries, while existing automated detection mechanisms remain very limited in effectiveness. As a result, many users of deprecated libraries fail to identify and remove them in a timely manner. Even when some developers are aware that they are using deprecated libraries, they may not remove

them due to practical reasons such as a lack of time or suitable alternatives. Taken together, these findings indicate that continued downstream reliance on deprecated libraries is not only a technical phenomenon but also a socio-technical one, driven by awareness gaps, prioritization trade-offs, and ecosystem limitations.

Summary for RQ3: Rust developers can check the homepage of libraries on crates.io and GitHub, or utilize tools provided by Cargo and GitHub to determine whether deprecated libraries exist in their projects. Nevertheless, unawareness of library deprecation remains the most cited reason why many Rust developers use deprecated libraries. These findings provide an indicative understanding of why deprecated libraries persist in Rust projects.

7 Implications

Building upon our findings, in the remaining part of this section, we will elaborate on the insights and implications of our experimental results from the perspectives of package managers, open source code hosting platforms, and future research.

7.1 Implications for Maintainers of Package Managers

Improving the effectiveness of existing deprecated library detection tools. Our findings diagnose a major breakdown in information dissemination: unawareness about deprecated libraries is the primary reason for their continued use. This indicates that, within the current ecosystem, the information published by libraries does not effectively reach their users. Although it is possible to receive deprecation alerts by configuring Dependabot on GitHub or by using the *cargo audit* command to check for deprecated libraries in a project, the results from RQ3 show that the number of deprecated libraries recorded in the databases used by these tools is very limited. As a result, most deprecated libraries cannot be detected. Therefore, detecting deprecated libraries based on the seven deprecation methods proposed in our paper could more effectively expand the range of deprecated libraries that these tools are able to identify.

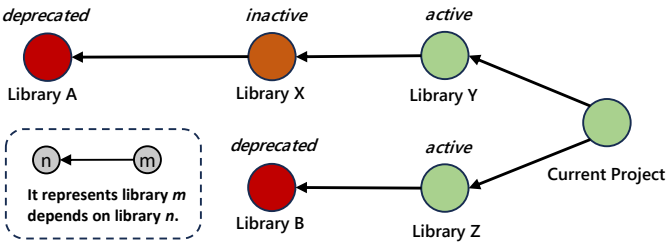

Fig. 9: Library Dependency Chain with Maintenance Status

Displaying deprecated libraries in dependency chains. Many developers may be unaware of the existence of current deprecated library detection tools. For example, P42 mentioned in their response: *"I wasn't aware of cargo audit before this survey"*. Cargo should integrate proactive deprecation alerts into the build or publish process, as suggested by 15 respondents, to eliminate the need for manual discovery tools. For example, P21 mentioned:" *Would be nice if Cargo would give a warning when publishing if one of the dependencies has an issue [...] I do not want any increased burden on library maintainers but it would be nice if you could mark a crate as deprecated in Cargo.*" NPM seems to have achieved this, but it simply lists deprecation information for both direct upstream libraries and transitive upstream libraries. This will confuse developers who may not know how deprecated transitive upstream libraries were introduced into their projects. As demonstrated in Figure 5, while numerous packages directly depend on well-maintained libraries, they ultimately inherit deprecated dependencies indirectly. The complex network of intermediate nodes in the software supply chain creates intricate propagation paths for deprecated libraries, significantly increasing the effort required for developers to eliminate these deprecated dependencies from their projects. Therefore, it would be better if the package manager could present to developers the nodes and edges in the dependency chain of the project that have maintenance issues, allowing developers to intuitively understand the position of these nodes with maintenance issues in the project.

In Table 7, 15 developers mentioned that they became aware of deprecated libraries in their projects through Issues or Pull Requests submitted by other developers. By understanding how deprecated libraries were introduced into their projects, these developers can then take more effective measures, such as providing Issues and Pull Requests to upstream libraries that use deprecated libraries, to mitigate risks at an earlier stage. For example, in Figure 9, project developers can see that libraries A and B are deprecated. Therefore, developers can send Issues or Pull Requests to the repository of Library Z to make them aware of the risks in their dependencies, prompting the maintainers of Library Z to replace Library B more quickly.

Simplifying and standardizing the deprecation mechanism of Rust libraries. Currently, Rust developers can declare deprecation in various ways, such as adding a deprecation notice in the README file or archiving the corresponding GitHub repository. However, these approaches are not consistently adopted and often require developers to check multiple sources to fully understand a library's status. To address the diagnosed limitations of current signals, establishing a standardized and more streamlined process—such as a unified deprecation flag that automatically propagates across crates.io, GitHub, and documentation—could reduce ambiguity and ensure that downstream users receive clear and consistent signals. Such standardization would not only help new users avoid adopting deprecated libraries but also improve overall ecosystem transparency and maintainability.

7.2 Implications for OSS Hosting Platforms

Displaying maintenance status of open source repositories. Although GitHub and Cargo currently provide some ways to indicate the maintenance status of a library, these methods heavily rely on the self-awareness and responsibility of library maintainers. In our work, a lot of libraries that have not been maintained for a long time do not indicate whether they are deprecated. This can confuse users of these libraries, leading them to spend a lot of time confirming

the maintenance status of the libraries, as reflected in the repository of *ansi_term* [29]. In this regard, code hosting platforms can proactively assess the health status of open source projects [52–56], helping users of projects better understand the real status of the current libraries. Further, libraries do not necessarily need to be constantly updated; they may be in a state of development completion, especially for trivial packages that provide fewer features [57]. Therefore, code hosting platforms are best able to provide various types of maintenance-related labels to reflect the current stage of development of the project. For example, providing a *completed* label indicates whether the current library is in a completed state and only accepts bug-fix commits.

Suggesting alternative repositories. Based on the results presented in Table 8, the inability to find suitable replacement libraries also hinders the removal of deprecated libraries within a project. Code hosting platforms can provide mechanisms to help developers discover appropriate alternatives. For example, if other developers have taken over a deprecated library through forking, the forked repository can be recommended to users of the deprecated library. Code hosting platforms can also recommend libraries most similar to the current one based on library migration history. For example, He et al. [58] have studied methods for library migration recommendations within the Java ecosystem, while Mujahid et al. [59] have explored the automatic identification of alternative libraries in the NPM ecosystem. These existing studies may provide some insights for library recommendations in the Rust ecosystem. In future work, researchers can focus on deprecated libraries in the Rust ecosystem to explore better methods for recommending alternatives, thereby helping developers remove deprecated libraries from their projects more efficiently.

Providing reasons for deprecation. In RQ3, many developers believe that deprecated libraries do not need to be removed as long as they do not cause problems. However, most libraries are deprecated only through archiving, so no description of the deprecation reason is provided. If a library is archived due to its defects, users of these deprecated libraries do not know the risks involved. Furthermore, because the repository is in an archived state, new Issues and PRs cannot be submitted, resulting in users of the library being unaware of defects discovered by others in the library. Therefore, maintenance status labels should be accompanied by corresponding reason descriptions. For example, if developers want to abandon a project, they can mark the project as "deprecated" on the code hosting platform and provide the reason for deprecation.

Displaying maintenance status in dependence chains. The maintenance status of projects on code hosting platforms can be obtained through public APIs, which can be well integrated with package managers to help developers understand the maintenance status of upstream libraries. As shown in Figure 9, Library A has been deprecated. Submitting an Issue or Pull Request in the repository of the inactive Library X may not be the most effective approach. Therefore, when project developers become aware of the inactivity of Library X, they may choose to alert the dependencies of Library Y, thus more effectively avoiding reliance on deprecated libraries.

29. https://github.com/ogham/rust-ansi-term/issues/72

7.3 Implications for Future Research

Two prior studies have investigated library deprecation in the NPM [13] and PyPI [12] ecosystems. Our study on Rust enriches this line of research by extending the scope to a new ecosystem, thereby serving as a complementary contribution and offering inspiration for future work. Across the three ecosystems, several commonalities emerge. For instance, both the PyPI study and ours found that only a small proportion of maintainers of inactive libraries explicitly declare deprecation. Moreover, all three studies observed that an explicit notice of abandonment facilitates clients' decision to stop using deprecated libraries, although the effect appears weaker in our Rust study.

Despite these similarities, each study exhibits distinct emphases, which make the results unique. The PyPI study [12] defined libraries with explicit deprecation notices as deprecated (equivalent to what we call advisory deprecation), and focused on upstream behavior—such as why some maintainers are reluctant to declare deprecation, and how they might better inform downstream users. The NPM study [13] considered both inactive packages (no commits for over two years) and those with explicit deprecation notices as deprecated, and primarily investigated what characteristics of packages make them more likely to depend on deprecated libraries.

However, these prior studies overlooked ecosystem-specific automated detection tools or deprecation mechanisms, and paid little attention to developers' attitudes toward deprecated libraries. For example, some developers are not opposed to using deprecated libraries, or may be forced to use them because of external constraints. In contrast, our study emphasizes understanding downstream users of deprecated libraries, capturing both the subjective reasons (e.g., developers' own circumstances or perceptions of deprecation) and the objective reasons (e.g., the limited effectiveness of detection mechanisms) behind their continued use. This perspective extends prior work by providing a more comprehensive view of deprecation practices.

Besides, the multiple deprecation methods identified in the Rust ecosystem may also directly exist (such as 'Description in GitHub' and 'README in GitHub') or have variants in other ecosystems (such as 'Maintenance Badge'). Future research on deprecated libraries should therefore pay closer attention to ecosystem-specific deprecation and detection mechanisms, while also considering the perspectives of both upstream and downstream developers, to build a more holistic understanding of deprecation and its impacts.

8 THREATS TO VALIDITY

In this section, we discuss the threats to the validity of our study from the perspectives of internal validity and external validity.

Internal Validity. Although our data comes from the official data export of crates.io, the links to the code repositories in crates.io are set by the maintainers themselves. This may result in some GitHub repository links for Rust

packages pointing to packages that are not the original ones. To mitigate this risk, we conducted a manual validation of a statistically representative subset of our data. Specifically, from the 40,636 libraries selected as study subjects in RQ1, we randomly sampled 381 libraries based on a 95% confidence level and a 5% margin of error. We manually inspected each sampled library to ensure that the metadata and GitHub repository linkage were correct. No inconsistencies were found during this process. Furthermore, we manually verified 1,468 deprecated libraries found in inactive libraries to ensure that the Rust packages and GitHub repositories have the correct correspondence.

To answer RQ1, we identified inactive libraries using a one-year threshold along with their version release frequency. To validate the accuracy of the identified inactive libraries, we randomly selected 374 inactive libraries from the 13,289 identified inactive libraries based on a 95% confidence level and a 5% margin of error. The data collection for the earlier version of the paper was finalized on January 17, 2024. We collected commit information for these 374 libraries from January 17, 2024, to March 20, 2025. We found that 329 libraries (88%) had no commits at all during this period of over one year. Among the remaining 45 libraries that have commits, only two were relatively active, with commits on more than 30 days. The other 43 libraries had an average of 2.4 days with commits, and 21 of them had commits on only one day. Overall, these results suggest that our method is generally effective: the vast majority of libraries remained inactive after being classified as such, with only a few showing signs of renewed activity.

To determine whether the text indicates deprecation of a library, we used keywords for filtering, and the final filtering results are heavily dependent on the number and quality of keywords. In this experiment, we selected as many deprecation-related keywords as possible and used the presence of keyword stems as a criterion for filtering to reduce the possibility of libraries that have already declared deprecation in the text being missed.

For the exploration of reasons why developers use deprecated libraries, our analysis is based on responses from 53 developers to a questionnaire. Different samples may lead to different results. To minimize the situation where projects still depend on deprecated libraries due to being unmaintained, we selected projects that have had new versions released within the past year. We examined the projects maintained by the respondents and confirmed that all of these projects had released new versions after the libraries they used were declared deprecated. This indicates that the projects maintained by these respondents remained active after the deprecation declarations. Furthermore, among the 53 responses we received, no developer indicated that they had suspended maintenance of their respective projects. These results demonstrate the effectiveness of our selection criteria for survey recipients.

In Section 5.1.1, we chose to observe deprecated libraries that had been declared deprecated for more than six months. This is because if the threshold is set too short, for example, one month, many downstream packages might not have had enough time to react and complete the removal of deprecated libraries within such a short period, thus affecting the validity of some conclusions in RQ2. Selecting a threshold that is too long might significantly reduce the number of libraries included in the study. For instance, if a one-year threshold were used, the number of libraries in the *declared_lib* set would drop to 90% of those included with the 6-month threshold. By choosing a six-month threshold, we excluded 150 libraries from the declared_lib set. These excluded libraries had an average Historical Dependent Count of 4.3, suggesting that their exclusion had a relatively small impact. Therefore, we believe that six months is a relatively reasonable threshold. Future work can explore more reasonable observation windows to investigate the impact of the deprecation declaration.

In the paper, we employed seven methods to identify deprecated libraries. These methods were not chosen arbitrarily; rather, they were derived from a review of Rust's official documentation, common practices, and related literature. The deprecation identification methods used in prior studies on the NPM [13] and PyPI [12] ecosystems are included within the seven methods we adopted in this paper. Taken together, these mechanisms represent the most common and widely accepted ways of declaring deprecation in practice. Even though we cannot guarantee that there will be no other ways. It's a promising avenue for future work to explore more deprecation ways.

Another minor limitation concerns our focus on inactive libraries as the primary source for identifying deprecated ones. This approach may exclude libraries that have been recently marked as deprecated. We made this choice to strike a balance between the substantial manual effort required and the likelihood of identifying true deprecations, given the scale of the full dataset. To assess the potential impact of this decision, we randomly sampled 100 active libraries and found that only two were deprecated, one via yanking all releases and the other via archiving the repository and adding a deprecation notice in the GitHub README. Consequently, the omission of active libraries poses minimal impact on the validity of our findings.

External Validity. Our work focuses on libraries within the Rust ecosystem. Rust's package manager, Cargo, does not provide a formal package-deprecation mechanism, and the locations where developers mark the status, such as description and badges, may not exist in other language ecosystems. Therefore, some of our conclusions may not be easily generalized to other systems. Different language ecosystems have their unique characteristics and design differences in package management. Existing literature lacks research on the activity and abandonment of Rust ecosystem packages, and our work addresses this gap.

Our study exclusively selected Rust libraries hosted on GitHub. While some libraries may be hosted on other code hosting platforms, exporting data from those platforms may be limited, and their functionalities may not be as comprehensive as GitHub's. As the world's largest open-source code hosting platform[30], GitHub offers a richer set of features and functionalities (such as Dependabot), making it sufficiently representative for our research purposes.

Our survey sample in RQ3 is skewed toward experienced Rust developers, which may limit the generalizability of the RQ3 results; therefore, the findings should be interpreted as

30. https://en.wikipedia.org/wiki/GitHub

indicative of trends among experienced contributors rather than as definitive conclusions for the broader developer population.

9 Conclusion

In this paper, we empirically study packages in the Rust ecosystem to explore the ways libraries declare deprecation, the effects of library deprecation declarations, and the reasons developers use deprecated libraries. We analyze seven methods of library deprecation declarations and find that only 11% of inactive libraries indicate that they are no longer maintained. These deprecation declarations provide limited practical influence on preventing the adoption of deprecated libraries. And many downstream packages still rely on deprecated libraries due to their lack of active maintenance or unawareness of their usage of deprecated libraries in projects.

Additionally, due to dependency chains, the issues in deprecated libraries can propagate to a broader range of downstream libraries. Based on these findings, we propose some views and suggestions regarding the current situation of deprecation declarations to facilitate better dissemination of information about deprecated libraries in the software supply chain and reduce associated risks stemming from deprecated libraries.

10 Data Availability

To facilitate the reproducibility of our results or to expand on our findings, the data and scripts used in this study are available at https://doi.org/10.5281/zenodo.15266092.

11 Acknowledgments

This work is supported by the National Natural Science Foundation of China (Grant No. 62572048 and 62232003).

References

[1] B. Farbey and A. Finkelstein, "Exploiting software supply chain business architecture: a research agenda," 1999.

[2] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: a review of industrial studies," *Empirical Software Engineering*, vol. 12, pp. 471–516, 2007.

[3] Synopsys, "About us synopsys," https://www.synopsys.com/company.html, 2023.

[4] D.-L. Vu, Z. Newman, and J. S. Meyers, "Bad snakes: Understanding and improving python package index malware scanning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 499–511.

[5] J. Sun, Z. Xing, Q. Lu, X. Xu, L. Zhu, T. Hoang, and D. Zhao, "Silent vulnerable dependency alert prediction with vulnerability key aspect explanation," *arXiv preprint arXiv:2302.07445*, 2023.

[6] M. Fan, Y. Zhang, K.-J. Stol, and H. Liu, "Core developer turnover in the rust package ecosystem: Prevalence, impact, and awareness," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3729392

[7] J. Coelho and M. T. Valente, "Why modern open source projects fail," in *Proceedings of the 2017 11th Joint meeting on foundations of software engineering*, 2017, pp. 186–196.

[8] Y. Yu, A. Benlian, and T. Hess, "An empirical study of volunteer members' perceived turnover in open source software projects," in *2012 45th Hawaii International Conference on System Sciences*. IEEE, 2012, pp. 3396–3405.

[9] Y. Wang, P. Sun, L. Pei, Y. Yu, C. Xu, S.-C. Cheung, H. Yu, and Z. Zhu, "Plumber: Boosting the propagation of vulnerability fixes in the npm ecosystem," *IEEE Transactions on Software Engineering*, 2023.

[10] npm Docs, "Deprecating and undeprecating packages or package versions," https://docs.npmjs.com/deprecating-and-undeprecating-packages-or-package-versions.

[11] S. O. Community, "2022 developer survey," https://survey.stackoverflow.co/2022#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages, 2022.

[12] Z. Zhong, S. He, H. Wang, B. Yu, H. Yang, and P. He, "An empirical study on package-level deprecation in python ecosystem," *arXiv preprint arXiv:2408.10327*, 2024.

[13] C. Miller, M. Jahanshahi, A. Mockus, B. Vasilescu, and C. Kästner, "Understanding the response to open-source dependency abandonment in the npm ecosystem," in *Int'l Conf. Software Engineering (ICSE), IEEE/ACM*, 2025.

[14] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 351–361.

[15] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 102–112.

[16] X. Tan, K. Gao, M. Zhou, and L. Zhang, "An exploratory study of deep learning supply chain," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 86–98.

[17] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu, and Z. Zhu, "Watchman: Monitoring dependency conflicts for python library ecosystem," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 125–135.

[18] Y. Wang, M. Wen, R. Wu, Z. Liu, S. H. Tan, Z. Zhu, H. Yu, and S.-C. Cheung, "Could i have a stack trace to examine the dependency conflict issue?" in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 572–583.

[19] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the dependency conflicts in my project matter?" in *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 319–330.

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

[21] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, pp. 381–416, 2019.

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

[23] A. Zerouali, T. Mens, A. Decan, and C. De Roover, "On the impact of security vulnerabilities in the npm and rubygems dependency networks," *Empirical Software Engineering*, vol. 27, no. 5, p. 107, 2022.

[24] B. Chinthanet, R. G. Kula, S. McIntosh, T. Ishio, A. Ihara, and K. Matsumoto, "Lags in the release, adoption, and propagation of npm vulnerability fixes," *Empirical Software Engineering*, vol. 26, pp. 1–28, 2021.

[25] G. A. A. Prana, A. Sharma, L. K. Shar, D. Foo, A. E. Santosa, A. Sharma, and D. Lo, "Out of sight, out of mind? how vulnerable dependencies affect open-source projects," *Empirical Software Engineering*, vol. 26, pp. 1–34, 2021.

[26] M. Alfadel, D. E. Costa, and E. Shihab, "Empirical analysis of security vulnerabilities in python packages," *Empirical Software Engineering*, vol. 28, no. 3, p. 59, 2023.

[27] C. Teyton, J.-R. Falleri, and X. Blanc, "Mining library migration graphs," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 289–298.

[28] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc, "A study of library migrations in java," *Journal of Software: Evolution and Process*, vol. 26, no. 11, pp. 1030–1052, 2014.

[29] H. Alrubaye, M. W. Mkaouer, and A. Ouni, "On the use of information retrieval to automate the detection of third-party java library migration at the method level," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 347–357.

[30] "Rust api deprecation," https://doc.rust-lang.org/reference/attributes/diagnostics.html#the-deprecated-attribute.

[31] "Cargo yank," https://doc.rust-lang.org/cargo/commands/cargo-yank.html.

[32] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to api deprecation? the case of a smalltalk ecosystem," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[33] A. A. Sawant, M. Aniche, A. van Deursen, and A. Bacchelli, "Understanding developers' needs on deprecation as a language feature," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 561–571.

[34] F. R. Cogo, G. A. Oliva, and A. E. Hassan, "Deprecation of packages and releases in software ecosystems: A case study on npm," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2208–2223, 2021.

[35] H. Li, F. R. Cogo, and C.-P. Bezemer, "An empirical study of yanked releases in the rust package registry," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 437–449, 2022.

[36] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration," *Empirical Software Engineering*, vol. 23, pp. 384–417, 2018.

[37] X. Xia, S. Zhao, X. Zhang, Z. Lou, W. Wang, and F. Bi, "Understanding the archived projects on github," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 13–24.

[38] J. Coelho, M. T. Valente, L. Milen, and L. L. Silva, "Is this github project maintained? measuring the level of maintenance activity of open-source projects," *Information and Software Technology*, vol. 122, p. 106274, 2020.

[39] J. Coelho, M. T. Valente, L. L. Silva, and E. Shihab, "Identifying unmaintained projects in github," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018, pp. 1–10.

[40] J. Maqsood, I. Eshraghi, and S. S. Ali, "Success or failure identification for github's open source projects," in *Proceedings of the 2017 International Conference on Management Engineering, Software Engineering and Service Sciences*, 2017, pp. 145–150.

[41] "Semantic versioning 2.0.0," https://semver.org/.

[42] J. Khondhu, A. Capiluppi, and K.-J. Stol, "Is it all lost? a study of inactive open source projects," in *Open Source Software: Quality Verification: 9th IFIP WG 2.13 International Conference, OSS 2013, Koper-Capodistria, Slovenia, June 25-28, 2013. Proceedings 9*. Springer, 2013, pp. 61–79.

[43] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration," *Empirical Software Engineering*, vol. 23, pp. 384–417, 2018.

[44] J. Zhang, Q. Gong, Y. Chen, Y. Xiao, X. Wang, and A. Y. Ding, "Understanding work rhythms in software development and their effects on technical performance," *IET Software*, vol. 2024, no. 1, p. 8846233, 2024.

[45] J. Wu, W. Xu, K. Gao, J. Li, and M. Zhou, "Characterize software release notes of github projects: Structure, writing style, and content," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023, pp. 473–484.

[46] N. Nachar *et al.*, "The mann-whitney u: A test for assessing whether two independent samples come from the same distribution," *Tutorials in quantitative Methods for Psychology*, vol. 4, no. 1, pp. 13–20, 2008.

[47] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.

[48] J. Singer, S. E. Sim, and T. C. Lethbridge, "Software engineering data collection for field studies," *Guide to advanced empirical software engineering*, pp. 9–34, 2008.

[49] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *2011 international symposium on empirical software engineering and measurement*. IEEE, 2011, pp. 275–284.

[50] R. S. C. W. Group, "Rustsec advisory database," https://rustsec.org/.

[51] Github, "Github advisory database," https://github.com/github/advisory-database/tree/main/advisories/github-reviewed.

[52] S. P. Goggins, M. Germonprez, and K. Lumbard, "Making open source project health transparent," *Computer*, vol. 54, no. 8, pp. 104–111, 2021.

[53] Z. Liao, F. Fu, Y. Zhao, S. Tan, Z. Yu, and Y. Zhang, "Hspm: A better model to effectively preventing open-source projects from dying." *Computer Systems Science & Engineering*, vol. 47, no. 1, 2023.

[54] R. Yang, Y. Yang, Y. Shen, and H. Sun, "An approach to assessing the health of opensource software ecosystems," in *CCF Conference on Computer Supported Cooperative Work and Social Computing*. Springer, 2022, pp. 465–480.

[55] M. Valiev, B. Vasilescu, and J. Herbsleb, "Ecosystem-level determinants of sustained activity in open-source projects: A case study of the pypi ecosystem," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 644–655.

[56] H. Hata, T. Todo, S. Onoue, and K. Matsumoto, "Characteristics of sustainable oss projects: A theoretical and empirical study," in *2015 IEEE/ACM 8th International Workshop on Cooperative and Human Aspects of Software Engineering*. IEEE, 2015, pp. 15–21.

[57] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? an empirical case study on npm," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 385–395.

[58] H. He, Y. Xu, Y. Ma, Y. Xu, G. Liang, and M. Zhou, "A multi-metric ranking approach for library migration recommendations," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 72–83.

[59] S. Mujahid, D. E. Costa, R. Abdalkareem, and E. Shihab, "Where to go now? finding alternatives for declining packages in the npm ecosystem," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1628–1639.