An Empirical Study of Yanked Releases in the Rust Package Registry

Hao Li, Filipe R. Cogo, Cor-Paul Bezemer

Abstract—Cargo, the software packaging manager of Rust, provides a yank mechanism to support release-level deprecation, which can prevent packages from depending on yanked releases. Most prior studies focused on code-level (i.e., deprecated APIs) and package-level deprecation (i.e., deprecated packages). However, few studies have focused on release-level deprecation. In this study, we investigate how often and how the yank mechanism is used, the rationales behind its usage, and the adoption of yanked releases in the Cargo ecosystem. Our study shows that 9.6% of the packages in Cargo have at least one yanked release, and the proportion of yanked releases kept increasing from 2014 to 2020. Package owners yank releases for other reasons than withdrawing a defective release, such as fixing a release that does not follow semantic versioning or indicating a package is removed or replaced. In addition, we found that 46% of the packages directly adopted at least one yanked release and the yanked releases propagated through the dependency network, which leads to 1.4% of the releases in the ecosystem having unresolved dependencies.

Index Terms—Software ecosystems, Release deprecation, Yanking, Rust, Cargo.

1 Introduction

In a software ecosystem, deprecation can happen in APIs, releases, and packages. Usually, the owner of a deprecated API plans to remove this API in the future and attempts to warn the developers who are using these APIs. For example, the owner of a package plans to remove a `foo()` function in one year, and adds a warning message which will be printed when `foo()` is called, giving a developer time to deal with the prospective deprecation. However, deprecation of releases and packages usually takes place unexpectedly. For example, when an owner of a package finds a critical bug in a release which was published a year ago, the owner can immediately deprecate this buggy release (without deleting the data) to make developers aware that the release is buggy, if the package manager supports release-level deprecation.

Prior studies have focused on the deprecation of APIs [26] [29] [32] [33] [36] [37] and packages [1] [2] [6] [7] [18] [20] [24]. In our prior work [28], we studied the release-level deprecation mechanism in `npm`,[1] which has supported this mechanism since 2010.[2] `Cargo`[3] (for `Rust`) is another software packaging ecosystem that has supported release-level deprecation from the very beginning of its creation (since 2014)[4]. We also considered other five ecosystems that contain the largest number of packages,[5] namely `Maven`[6] for `Java`, `PyPI`[7] for `Python`, `Packagist`[8] for `PHP`, `NuGet`[9] for `.NET`, and `RubyGems`[10] for `Ruby`. We observed that `Maven` and `Packagist` do not support release-level deprecation. `RubyGems` introduced release-level deprecation in 2012[11] but changed it to deletion in 2015 due to a too-heavy burden on the support team.[12] `NuGet` and `PyPI` have started to support release-level deprecation in 2019[13] and 2020[14] but there is not much data available.

In this study, we focus on `Cargo`, which implements a forceful release-level deprecation (i.e., yanking). The yank mechanism in `Cargo` will remove yanked releases from the registry index without deleting the data, compared to the deprecation mechanism in `npm` which just provides warning messages for deprecated releases and continues to allow the package installation. This forceful deprecation mechanism in `Cargo` can lead to unresolved dependencies of certain releases of a package. In addition, `Cargo` records the date on which a release was yanked, which allows us to study how the number of deprecated releases evolves (in contrast to our prior work [28] in which we had to estimate the date of deprecation).

Our study mines and analyzes `Rust`'s official package registry (`crates.io`[15]) to improve the understanding of the yank mechanism. In addition, we compare the results in `Cargo` with `npm`. Our results can help researchers to understand release-level deprecation in software packaging ecosystems, help the `Rust` community to improve the yank mechanism, and help other packaging ecosystems to improve their own deprecation mechanism. We collected data of all the 48,823 packages on `crates.io` and used this dataset to answer the following research questions (RQs):

RQ1. How many releases are yanked?
The number of releases has increased steeply from 2014 to 2020 and the proportion of yanked releases also keeps increasing. In addition, we found that `Cargo` has a slightly higher proportion of yanked releases than `npm`.

RQ2. Why do packages use the yank mechanism?
We study five patterns of yanking releases and we found that releases are being yanked for another reason than being defective. However, we noticed that only 5.3% of packages explained why a release was yanked: this makes yanked releases even harder to

- *Hao Li and Cor-Paul Bezemer are with the Analytics of Software, GAmes And Repository Data (ASGAARD) Lab, University of Alberta, Edmonton, AB, Canada. Email: li.hao@ualberta.ca, bezemer@ualberta.ca.*
- *Filipe R. Cogo is with the Centre for Software Excellence at Huawei, Canada. Email: filipe.roseiro.cogo1@huawei.com. This work is not related to his role at Huawei.*

deal with for developers who adopted such releases.

RQ3. How many packages adopt yanked releases? Even though the proportion of yanked releases is small in the ecosystem, a relatively large proportion of packages adopt yanked releases. Also, we found that yanked releases were transitively adopted in the ecosystem and caused unresolved dependencies, which in turn led to 4,158 broken releases currently in the ecosystem.

Paper Organization. The rest of this paper is organized as follows. Section 2 provides background information about the package manager of `Rust` and its yank mechanism. Section 3 discusses related work. Section 4 presents the method that we used in our study. Section 5 presents the findings of our three research questions. Section 6 discusses the implications of our findings. Section 7 discusses the threats to the validity of our study. Section 8 concludes this paper.

TABLE 1: Five types of versioning specifications in Cargo

Types	Statement	Interpretation
Comparison	$=$1.2.3	[1.2.3]
	$>$1.2.3	]1.2.3, $+\infty$[
	$<$1.2.3	[0.0.0, 1.2.3[
	$\geq$1.2.3	[1.2.3, $+\infty$[
Compound	$>$1.2.3, $\leq$ 2.3.4	]1.2.3, 2.3.4]
Caret	$\wedge$1	[1.0.0, 2.0.0[
	$\wedge$1.2	[1.2.0, 2.0.0[
	$\wedge$1.2.3	[1.2.3, 2.0.0[
	$\wedge$0	[0.0.0, 1.0.0[
	$\wedge$0.1	[0.1.0, 0.2.0[
	$\wedge$0.0.1	[0.0.1, 0.0.2[
	$\wedge$0.1.2	[0.1.2, 0.2.0[
Tilde	$\sim$1	[1.0.0, 2.0.0[
	$\sim$1.2	[1.2.0, 1.3.0[
	$\sim$1.2.3	[1.2.3, 1.3.0[
Wildcard	$\star^a$	[0.0.0, $+\infty$[
	1.$\star$	[1.0.0, 2.0.0[
	1.2.$\star$	[1.2.0, 1.3.0[
	1.2.3	[1.2.3, 2.0.0[

[a]: removed in 2016

2 Background

In this section, we describe how `Rust` manages packages, and we discuss the dependency requirements and yank mechanism in `Cargo`.

2.1 Package management in Rust

`Cargo` is the official package manager of `Rust`. Most developers use `Cargo` to compile their packages instead of using the compiler `rustc`[16] directly. Before performing a compilation, `Cargo` will resolve the dependencies and download specific versions of packages to satisfy the dependency requirements. After that, developers can run their package locally (as a standalone application) or publish their package to a package registry (as a library) using `Cargo`.

The `Rust` community's package registry is `crates.io`, which stores the packages online and provides a platform to search and browse the information of uploaded packages. Usually, developers interact with `crates.io` through the command-line interface of `Cargo`. For example, developers can use the `cargo search`[17] command to find packages in `crates.io`. In addition, package owners can publish releases to `crates.io` and manage their packages through the command-line interface. For instance, `cargo publish`[18] will upload the current package to a registry (which is set to `crates.io` by default). In this paper, for simplicity we refer to `crates.io` as `Cargo`.

2.2 Dependencies in Cargo

Dependency requirements in `Cargo` are based on semantic versioning[19] and `Cargo` will determine the version of dependencies when developers build their projects. The semantic versioning specification defines that a version number consists of three parts: major, minor, and patch. For example, version number `1.2.3` has a major number `1`, a minor number `2`, and a patch number `3`. The packages should guarantee that patch updates only introduce "backwards compatible bug fixes", minor updates only add features which are backwards compatible, and only the major updates can introduce breaking changes. We refer to this guarantee as the *semantic versioning guarantee* in our paper.

The semantic versioning specification is used by `Cargo.toml`,[20] a file under the directory of a `Rust` project, to store the dependency requirements. The interpretation of requirement statements is different across software ecosystems [11]. Table 1 shows the versioning specifications which are used in `Cargo`. It is notable that `Cargo` interprets `1.2.3` as a caret requirement ($\wedge$`1.2.3`) and the wildcard requirement statement "$\star$" (i.e., matching any version) was banned in January 2016.[21] `Cargo` searches the registry index to find versions which can satisfy the requirements and downloads dependencies. If there are multiple versions available that satisfy a requirement, `Cargo` will choose the version which has the largest version number. We call the owner of a dependency requirement a *client* and the package to which the dependency requirement points a *provider*. For example, a client package `C` has a dependency requirement `3.0.1` for a release from a provider package `P`. `Cargo` will choose the greatest version `3.5.1` from `P` which satisfies this requirement even though there exists an exactly matched version `3.0.1` (since `Cargo` interprets the requirement `3.0.1` as a caret requirement).

2.3 Yanked releases

`Cargo` provides a command called `cargo yank`[22] to deprecate a published release, which can also be unyanked with the `yank undo` command. After a developer calls the yank command for a certain release, this release will be indicated as yanked and is no longer available from the registry index for `Cargo`. Thus, when `Cargo` is trying to resolve dependencies for a project, it will automatically skip yanked releases and choose the release which has the largest version number that still satisfies the dependency requirement. For example, a client package `C` has a dependency requirement $\sim$`2.5.1` for a package `P`. The latest releases of `P` are `2.5.5` and `2.5.6`. However, the latter was yanked. Hence, `Cargo` will select release `2.5.5` of `P`. Due to the deletion of yanked releases from the registry index, `Cargo` cannot download a

yanked release even if the dependency requirement uses the "=" operator.

Notably, the yank command does not completely delete any data from the package registry, hence the yanked releases can still be downloaded through work arounds. One approach is using the download API which is provided by the package registry,[23] and another approach is through the locking mechanism of Cargo. Cargo will generate a Cargo.lock[24] file if the building process is successful. This file stores the versions of dependencies that were used during the build. When the developer compiles the project a second time, Cargo will reuse the versions of the dependencies that are stored in Cargo.lock (as long as the developers did not change a required version in Cargo.toml) even if the depended versions are yanked or a newer version is available. A standalone application will usually upload both Cargo.toml and Cargo.lock to its repository,[25] which assures the reproducibility of the building process. In contrast, a library will upload Cargo.toml but not Cargo.lock, so Cargo will help the clients of this library to determine a suitable version to use. However, as we show in Section 5.3, this could lead to unresolved dependencies when building a package that depends on a yanked release.

3 Related work

In this section, we discuss related work about software packaging ecosystems and the deprecation of APIs and packages.

3.1 Software packaging ecosystems

Most research on software packaging ecosystems has focused on npm [8] [14] [35] [38] of JavaScript, PyPI [19] [31] [34] of Python, and CRAN [4] [5] [12] [17] of R. In this paper, we study the yank mechanism in the Cargo ecosystem, the packaging system of Rust.

Few studies have focused on the Cargo ecosystem. Evans et al. [16] studied the safety of packages in the Cargo packaging ecosystem of Rust. They found that 29% of the packages directly use the unsafe keyword, which is provided by Rust to avoid safety checking of the compiler. Furthermore, they observed that popular packages use unsafe more frequently.

Many studies include Cargo as a subject when comparing multiple software packaging ecosystems. Decan and Mens [13] investigated pre-releases of three packaging ecosystems and observed that more than 90% of the packages in Cargo published a pre-release as their latest release. In addition, they found that most dependencies that point to pre-releases allow patch updates in Cargo [11], which does not follow the semantic versioning specification. Constantinou et al. [9] studied packages which are distributed across multiple packaging ecosystems and found that these packages in Cargo have more stars on GitHub than in other ecosystems. Like other packaging systems, a relatively small proportion of packages are depended on by most of the packages in Cargo [15].

Many researchers have studied the npm ecosystem. In prior work [8], we investigated dependency downgrades in npm and observed that packages changed their dependency constraints for migrating away from defective dependencies. Decan et al. [14] also found that a proper dependency constraint can help a package migrate away quickly from a vulnerable dependency. In addition, they found that most of the security vulnerabilities are fixed before they are published in npm. Wittern et al. [35] studied dependencies in npm and found that the package dependencies keep increasing. However, most of the dependencies point to a small proportion of packages in the ecosystem. Zerouali et al. [38] analyzed various popularity metrics in npm and found that the results of identifying popular packages can be different based on the metrics used.

Imminni et al. [19] implemented a semantic search engine for the PyPI ecosystem since PyPI has a limited ability to provide quality search results for developers. To detect dependency conflicts in PyPI, Wang et al. [34] developed a tool to monitor the ecosystem. Valiev et al. [31] built survival models for PyPI to analyze the risk of a package become dormant.

German et al. [17] studied the CRAN packaging ecosystem of R. They found that most dependencies point to popular packages and user-contributed packages need more time to grow their community than core packages. Claes et al. [5] observed that the time of fixing errors in CRAN packages differs across operating systems. They also [4] developed a tool to analyze the maintainability of a package in CRAN, which can visualize information such as release history, dependencies and namespace. Decan et al. [12] found that packages in CRAN also manage their repositories on Github, which can influence the dependency management.

3.2 Deprecated APIs and packages

In prior work [28], we studied the deprecation mechanism in npm. To the best of our knowledge, that was the first study that focused on the release-level deprecation mechanism of a software packaging ecosystem. This follow-up paper focuses on the yank mechanism in Cargo and compares it with the deprecation mechanism in npm. The reason is threefold: 1) The yank mechanism in Cargo is more forceful than the deprecation mechanism in npm. 2) Cargo provides the date of yanking which supports a more in-depth analysis. 3) *"Comparative studies can be seen as a prerequisite for designing successful domain-specific ecosystem solutions"* [30]. Hence, the comparisons across these two software ecosystems can help us better understand the design of a release-level deprecation mechanism.

Many researchers have studied deprecated APIs at the code-level. Sawant et al. [29] interviewed Java API producers and surveyed Java developers, and suggested Java to provide a warning mechanism for developers. Wang et al. [33] investigated six popular packages in Python and observed that developers did not have a consistent strategy to deprecate an API, and that about 25% of the deprecated APIs are not documented. Robbes et al. [26] analyzed deprecated functions and classes in Smalltalk, and found that about half of the deprecation messages cannot help the developers to migrate away from the deprecation.

Few studies have focused on the deprecation of web APIs. Yasmin et al. [37] analyzed 1,368 RESTful APIs and

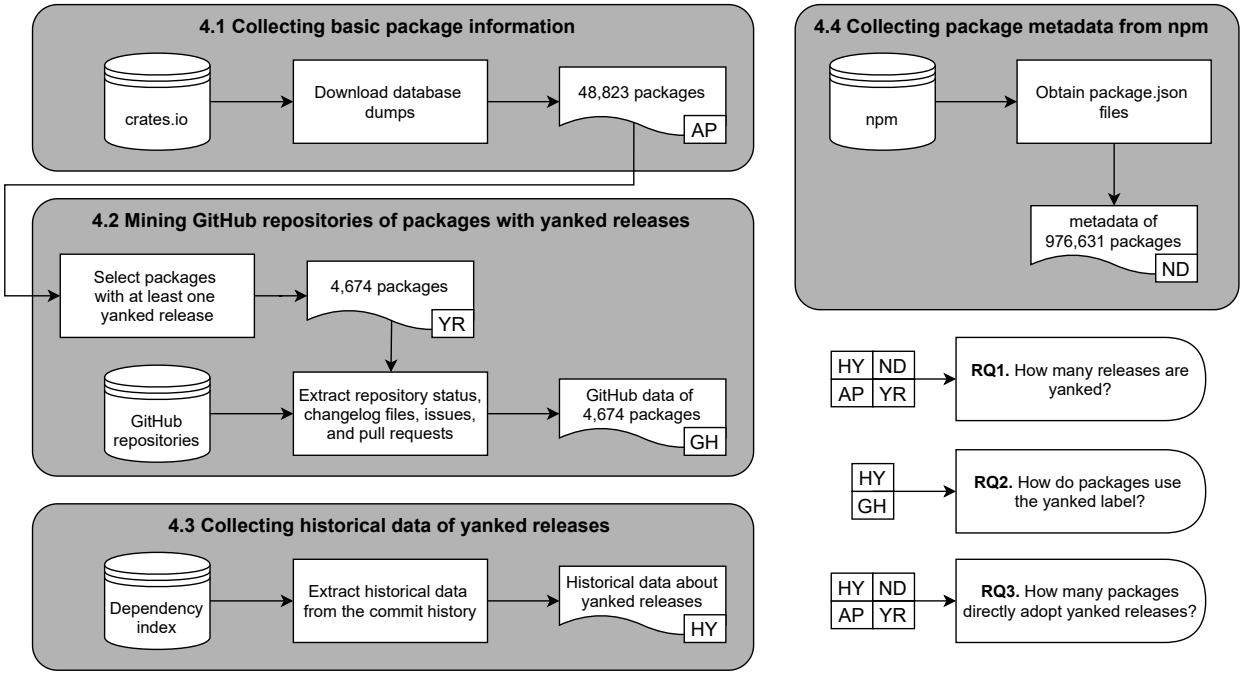

Fig. 1: Overview of our methodology.

found that most of the removed APIs did not deprecate the interface to inform their users before introducing the deletion.

Unlike deprecated APIs, it is not easy to identify whether a package is deprecated because the owner may not indicate deprecation in the documentation. Coelho et al. [7] found that the important features to predict whether a package is deprecated or unmaintained include the number of commits and closed issue reports. Khondhu et al. [20] introduced the maintainability index to identify whether a package is inactive or abandoned on `SourceForge.net`. In contrast, Maqsood et al. [24] implemented eight machine learning algorithms to identify successful projects.

Many researchers conducted studies to understand the reasons behind the deprecated and abandoned packages. Coelho et al. [6] surveyed the owners of 104 deprecated `GitHub` packages, and showed that the reasons include environmental factors, project characteristics, and human factors. Iaffaldano et al. [18] interviewed developers from the open-source software community, and also found the reasons behind abandoned packages include human factors and project characteristics. Avelino et al. [1] found that the loss of core developers can increase the risk of a package becoming abandoned.

4 Methodology

In this section, we introduce the methodology of our study of yanked releases in the `Rust` package registry. Figure 1 gives an overview of our study.

4.1 Collecting basic package information

`Cargo` provides database dumps[26] which contain all the information (e.g., dependencies, downloads, creation date) exposed through the official API. The database dumps are the primary data source of our study and we downloaded the dump that contains the information of 48,823 packages with 294,801 releases on October 29th, 2020. Table 2 shows the database fields which store important information for our study.

TABLE 2: Key information in the database.

Field	Description
versions.id	The identifier of a release.
versions.num	The semantic version number of a release such as `1.2.3` or `0.1.2-alpha`.
versions.created_at	The creation date of a release.
versions.yanked	A flag to indicate whether a release is yanked.
crates.id	The identifier of a package.
crates.readme	The content of the `readme` file in a package.
crates.repository	The link to the repository of a package.
dependencies.version_id	The identifier of the release to which the dependency belongs.
dependencies.crate_id	The identifier of the package to which the dependency points.
dependencies.req	The dependency requirement (e.g., `^1.2.3` or `~1.2.3`).

4.2 Mining GitHub repositories of packages with yanked releases

As Table 2 shows, yanked flags are stored in the `versions` table. We retrieved all entries that are indicated as yanked. Then, we selected the packages which have at least one yanked release and collected the links to their repositories. For the links which direct to a `GitHub` repository, we used the `GitHub` API[27] to extract the issue reports and pull requests of these repositories. In addition, we collected the status of these repositories (active, archived or forked) through the `GitHub` API. If the repository cannot be found, we marked its link as invalid.

Furthermore, we collected the changelogs of the packages which have at least one yanked release from their `readme` file and `GitHub` repository. The `readme` field of the `crates` table in the database contains the content of the `readme` file. We identified whether the `readme` contains a changelog by searching for the keywords "changelog", "change log", "release notes", and "release note" in the content, as well as searching "news" and "history" in the headings. For packages which provide a valid link to their `GitHub` repository, we queried the filenames in the root directory of the repository and collected the file if the filename matches the same keywords which we used above.

4.3 Collecting historical data of yanked releases

`Cargo` determines the dependencies based on the registry index which is managed in a `Git` [3] repository. This index repository[28] contains the information (e.g., dependencies, version numbers, and yanking flags) of all published releases. The data of each package is stored in separate files, and the information is updated automatically whenever a change occurs (e.g., a new release is uploaded, yanked or unyanked). Because all the changes are managed in the `Git` repository, the commit history contains the date of each change. We mined the commit messages to extract when a release was yanked or unyanked.

However, we noticed that the commit history in the main branch is not complete because the maintainer of `Cargo` regularly squashed commits into one to speed up the cloning of the repository.[29] These squashed commits are stored in `snapshot` branches, hence we collected all the commits of these branches to obtain a complete historical overview of (un)yanked releases.

4.4 Collecting package metadata from npm

We reused the dataset from our prior study [28] which contains the metadata of 976,613 packages from `npm` at May 5th, 2019. For each package, we collected the information of all releases and selected the dependencies in their latest releases. Finally, we extracted 7,829,362 releases and 6,178,019 dependencies from 976,613 packages.

5 Results

In this section, we present the motivation, approach, and findings for each of our three research questions (RQs).

5.1 RQ1: How many releases are yanked?

Motivation. Deprecation can happen at the code-level, release-level, and package-level. In our prior work [28], we studied the release-level deprecation mechanism in `npm`. Similarly, `Cargo` has a yank mechanism for release-level deprecation to allow the owner of a package to "remove a previously published crate's version from the server's index".[30] In contrast to `npm`, `Cargo` records the date on which a release was yanked or unyanked. Hence, we can study how often developers use the yank mechanism in the history of `Cargo`. The goal of this research question is to understand the yank mechanism in `Cargo` by investigating the frequency of yanked releases and comparing the result with `npm`.

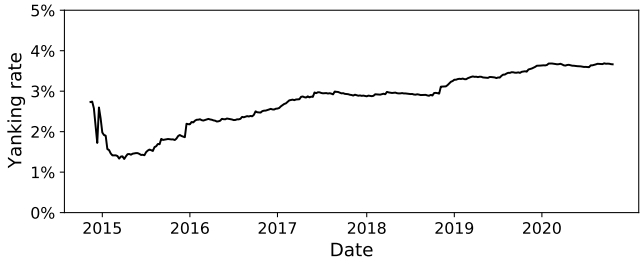

Fig. 2: The percentage of yanked releases in `Cargo` from November 2014 to October 2020.

Approach. We calculated the proportion of yanked releases and packages that have at least one yanked release in `Cargo` to measure how often the yank mechanism is used. To analyze the trend of usage, we investigated the historical information of releases and yanked releases. We collected the date on which a release was published from the `created_at` field of the `versions` table (as shown in Table 2). In addition, to count the yanked releases in a certain period more precisely, we also considered the date of unyanking a release. Finally, we calculated the number of releases and the proportion of yanked releases from November 2014 to October 2020.

Next, we calculated the yanking rate (i.e., the percentage of yanked releases in a package) for every package. For example, the yanking rate is 100% for *fully yanked* packages (i.e., packages of which all releases are yanked), and 0% for packages which do not have any yanked release. Then, we compared our findings between `Cargo` and `npm` by performing the Mann-Whitney U test [23] at a significance level of $\alpha = 0.05$ to determine whether the differences are significant. However, the Mann-Whitney U test only determines whether two distributions are different. Therefore, we computed Cliff's delta d [22] effect size to quantify the difference. To explain the value of d, we used the thresholds which are provided by Romano et al. [27]:

$$\text{Effect size} = \begin{cases} negligible, & \text{if } |d| \leq 0.147 \\ small, & \text{if } 0.147 < |d| \leq 0.33 \\ medium, & \text{if } 0.33 < |d| \leq 0.474 \\ large, & \text{if } 0.474 < |d| \leq 1 \end{cases}$$

Findings. 3.7% of the releases in Cargo are yanked and 9.6% of the packages have at least one yanked release. There are 10,761 yanked releases in `Cargo` and 4,674 packages with at least one yanked release. We found that the proportion of yanked releases in `Cargo` (3.7%) is close to the proportion of deprecated releases (3.2%) in `npm`. In contrast, `Cargo` has a larger proportion of packages which have at least one yanked release (9.6%), compared to the proportion of packages with at least one deprecated release in `npm` (3.7%).

Between 2014 and 2020, the percentage of yanked releases in Cargo has gradually increased from 1.4% to 3.7%. We found that unyanking only happened 725 times in the history, which is relatively uncommon compared to 10,761 yanked releases in `Cargo`. Figure 2 shows that the cumulative number of releases in `Cargo` has increased steeply from 0 to nearly 300,000 in the period 2014 to 2020.

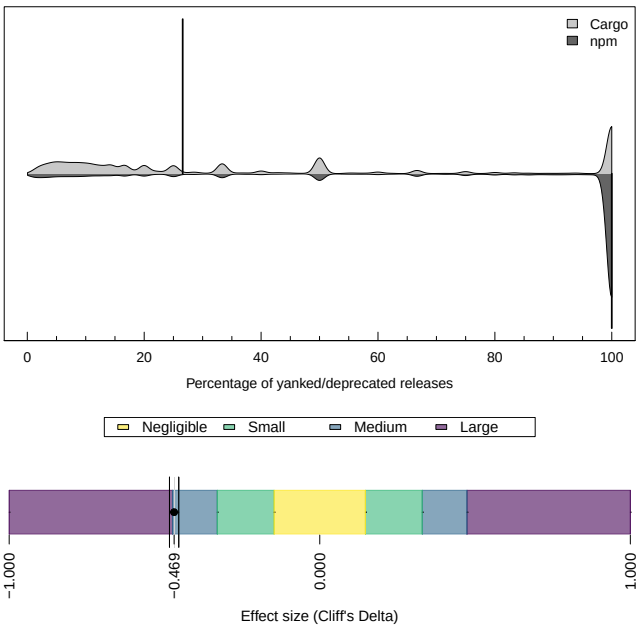

Fig. 3: Distributions of the yanking rate of packages with at least one yanked release in `Cargo` and `npm` and Cliff's Delta d.

The Cox-Stuart test [10] shows that the increasing trend of the number of releases is significant ($p \ll 0.05$). Also, the yanking rate has increased gradually to 3.7% since 2014, and again the Cox-Stuart test shows that the increasing trend is significant ($p \ll 0.05$). We cannot analyze the trend of the yanking rate in `npm` because it does not provide historical information about deprecated releases.

It is more common to partially yank a package in Cargo than in npm. The majority (75%) of the packages with at least one deprecated release are partially yanked packages in `Cargo`. In contrast, partially deprecated packages are the minority (34%) of packages with at least one yanked release in `npm`. Among the partially yanked packages, the median yanking rate is 17% in `Cargo`, which is half of the median number in `npm` (33%). Figure 3 (using the package from Lin et al. [21]) shows the distributions of the yanking rate in the two ecosystems and the value of Cliff's Delta d. The Mann-Whitney U test shows that the distributions of the yanking rate are significantly different in these two software ecosystems. In addition, the value of Cliff's Delta $|d|$ is 0.469, which indicates that the effect size is medium.

> **RQ1 Summary:** The proportions of yanked releases in `Cargo` and `npm` are similar (3.7% vs. 3.2%), but it is much more common in `Cargo` (75% vs. 34%) to yank only a few releases of a package.

5.2 RQ2: Why do packages use the yank mechanism?

Motivation. In our prior work [28], we studied non-forceful release-level deprecation in `npm`. However, in `Cargo`, release-level deprecation (yanking) is forceful, which means that releases are no longer accessible once they are deprecated. In this research question, we take a closer look at packages with at least one yanked release in `Cargo` to investigate why the yanking mechanism is used and to study the rationales for yanking a release. The results can help us to understand the release-level deprecation in `Cargo` from the developers' point of view.

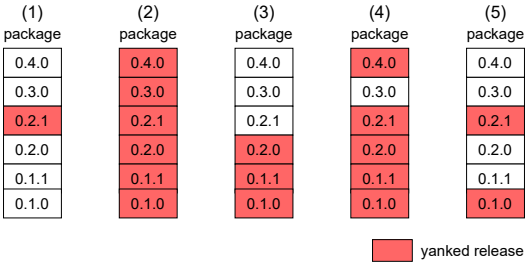

Fig. 4: Five patterns of yanking: (1) A package yanked only one release; (2) A package yanked all releases; (3) A package yanked back-to-back releases; (4) A package yanked all releases except one; (5) A package yanked nonadjacent releases.

Approach. To understand the usage of the yank mechanism, we looked for five possible patterns in which releases are yanked in `Cargo` (as shown in Figure 4). First, we collected the packages which have at least one yanked release and sorted their releases based on the release date. Then, we went through the collected 4,674 packages to identify whether a package belongs to one of the five patterns.

In addition, we investigated the changelogs, issue reports, and pull requests from the 4,674 packages to analyze the rationales behind yanking. We selected the packages which contain the terms "yank" or "deprecate" in their changelogs, issue reports, or pull requests. The first author went through the selected 638 packages and filtered out 380 packages which did not provide information that is related to the yanked release. After that, the first and third author performed open card sorting together to identify the rationales behind the yanked releases of the remaining 258 packages. We could not identify the rationales for 9 out of 258 packages during the card sorting (i.e., 3.5% false positives of the filtering process). Hence, our results cover the remaining 249 packages.

Findings. The usage of the yank mechanism among packages in Cargo follows one of the five patterns in Figure 4. In addition, developers yanked releases for 11 reasons (see Table 3). We summarize below the patterns that we were able to identify, together with some examples and the rationales we categorized in the card sort.

Pattern 1: Yanking only one release (40% of the packages). We found this pattern in 1,879 packages and *Breaking SemVer* is the main rationale behind this pattern (47.5%) in the card sort. One example is `pyo3`[31], an issue report of this project mentions that "[v]ersion 0.5.1 breaks SemVer guarantees"[32] because the owner of `pyo3` accidentally merged a new feature to this patch update for `0.5.0`. It introduced breaking changes and should be a minor or major update since the semantic versioning guarantee requires that patch updates only introduce backwards-compatible bug fixes. Hence, the owner yanked `0.5.1`, backported the changes, and published `0.5.2` which did not include the new feature.

TABLE 3: Identified rationales behind yanked releases in the card sort

Rationale	Description	Pkg	P1	P2	P3	P4	P5
Breaking SemVer	The release introduces breaking changes and therefore does not follow the semantic versioning specification	43.0%	47.5%	25.0%	40.0%	-	42.9%
Defect	Release contains a defect/bug	36.9%	28.0%	-	46.7%	100.0%	50.0%
Fixing "*" dependencies	Release uses the wildcard dependency ("*") which has been prohibited since 2016	7.2%	4.2%	25.0%	8.3%	-	8.9%
Package removed or replaced	The whole package is removed or is replaced by another package	5.2%	5.1%	50.0%	-	33.3%	-
Broken dependencies	The release contains a broken dependency	4.8%	6.8%	-	1.7%	-	5.4%
Bump propagation	A patch release that includes a minor/major update of an existing dependency and therefore should be a minor/major release as well	3.2%	5.1%	-	1.7%	-	1.8%
MSRV policy	Upgrading the minimum supported rust version (which is a breaking change) in a patch update	2.8%	0.8%	-	6.7%	-	3.6%
Yanked dependencies	Dependencies are yanked	2.0%	3.4%	-	1.7%	-	-
Placeholder release	An initial release for holding the name in `Cargo`	1.2%	0.8%	-	1.7%	-	1.8%
License updated	Forgetting to update the license	0.4%	-	-	-	-	1.8%
Unsupported	Releases are no longer supported	0.4%	-	-	1.7%	-	-

Note: one package can have multiple rationales as it can have multiple yanked releases. We identified one rationale per yanked release. The "Pkg" column presents the percentage of packages that contain the corresponding rationales. The "P1" to "P5" columns present the percentage of packages that contain the corresponding rationales under the yanking patterns.

In addition, we noticed that developers yanked a release for *Broken dependencies* (6.8%) or *Yanked dependencies* (3.4%). One example of *Broken dependencies* is `diesel_cli`,[33] which yanked version `0.99.0` to restrict the dependency of `clap` from $\geq$`2.27.0` to $\wedge$`2.27.0` to prevent using `3.x.x` versions of `clap` since major updates could introduce breaking changes. For *Yanked dependencies*, one example is version `0.9.2` of `winping`.[34] This version was yanked "due to a yanked dependency"[35] and `winping` had to update the dependency of the `quote`[36] package from `1.0.2` (yanked) to `1.0.3` in version `0.9.3`.

Bump propagation (5.1%) is a notable rationale, which happens when updating the existing dependencies of a package. For instance, `sdl2`[37] updated the requirement of `sdl2-sys` from $\wedge$`0.7.0` to $\wedge$`0.8.0` and published a patch update `0.12.2` for `0.12.1`. Since another package `sdl2_image` depends on $\wedge$`0.12.1` of `sdl2` and $\wedge$`0.7.0` of `sdl2-sys` "which leads to conflicts",[38] `sdl2_image` was broken and `sdl2` had to yank `0.12.2` and republished it as `0.13.0`. The behaviour of bumping the required version of a dependency (`sdl2-sys`) in `sdl2` broke its dependent (`sdl2_image`) and `sdl2` had to bump its version number as well.

Pattern 2: Yanking all releases (25% of the packages). We observed this pattern in 1,172 packages, with 60% of these packages having only one release. The main rationale behind this pattern (50.0%) in the card sort is *Package removed or replaced* since developers cannot point a new dependency to a fully yanked package. One example is `ncollide`,[39] the owner explained that "the overly generic crate ncollide has been replaced by two distinct crates: ncollide2d and ncollide3d which are dedicated to 2D and 3D respectively."[40] Another example is `c`,[41] which yanked all releases to "kill"[42] this package. We also identified *Breaking SemVer* (25.0%) and *Fixing "*" dependencies* (25.0%) rationales under this pattern, however, the identified rationales are not for the whole package, but for some specific releases.

Pattern 3: Yanking back-to-back releases (17% of the packages). There are 814 packages that followed this pattern and *Defect* (46.7%) is the most common rationale that we identified in the card sort under Pattern 3. We noticed that developers often yanked multiple releases due to the same defect. For example, `clap`[43] yanked the versions from `1.4.0` to `2.21.0` since these versions are all affected by "an erroneous definition of a macro".[44] In fact, one of the maintainers left a comment about `Cargo` not supporting "yank[ing] everything from X.X.X to Y.Y.Y". We also observed packages that yanked older releases for security purposes, such as `untrusted`[45] and `bitvec`.[46] Moreover, the only instance of the *Unsupported* rationale in the card sort was found under this pattern. Old releases of `ring`[47] were yanked because the owner no longer supports these versions, even though they do not have any known vulnerabilities. Instead, the owner recommends that people only use the latest version.[48] In addition, we observed that packages yanked multiple releases due to the *MSRV*[49] *policy* (6.7%). For instance, `block-buffer`[50] yanked versions `0.7.0` to `0.7.2` since these versions used an interface that "was stabilized only in Rust 1.28",[51] hence, the dependents of `block-buffer` will be broken if they used a lower version of `Rust`.

Pattern 4: Yanking all releases except one (11% of the packages). There are 506 packages that follow this pattern and 94% of these packages left their newest release unyanked. We only found three packages that explained the rationales under this pattern in the card sort. All three packages explained that at least one of the yanked releases contained a defect. `battery-cli`[52] is the only package that explained that the rationale for yanking was *Package removed or replaced*. This package left the newest release unyanked to show the `readme` file which mentions that "this crate was yanked and replaced by battop crate" instead of the default page on the website.

Pattern 5: Yanking nonadjacent releases (7% of the packages). There are 383 packages that do not belong to any patterns which we proposed. Similar to Pattern 1 and Pattern 3, the two most common rationales that we identified in the card sort are *Defect* (50.0%) and *Breaking SemVer* (42.9%). In

addition, the only instance of *License updated* we identified in the sort is from this group. Particularly, `sic`[53] yanked version `0.10.0` since the "dependency licenses [were] not updated".[54] `sic` also yanked version `0.7.1` for a defect which "fails to build from crates.io".[55]

5.3% of the packages with at least one yanked release explain the rationales in their changelogs, issue reports, and pull requests. During the pattern analysis, we observed that the proportion of packages which explain the rationales behind yanked releases is relatively small, compared to 64% of the deprecation messages in `npm` which explain the rationales for deprecating a package or release. Hence, it is not possible to identify the rationales behind all yanked releases.

1.5% of the packages with at least one yanked release explain the rationale behind yanking a release in a changelog. There are 21% (994 packages) of the packages with at least one yanked release that have a changelog, but only 1.5% (70 packages) of these changelogs explain the rationale behind yanking a release. In other words, if a developer desired to know the rationale behind a yanked release of a package by searching its `changelog`, only 2% of the packages can provide an answer for the developer. In addition, we noticed that most (93%) of the changelogs in the packages with at least one yanked release are kept in a separate file (e.g., `changelog.md` or `releases.md`). However, 86% of the packages which have a `changelog` file do not mention this file in their `readme`. As a result, developers could overlook the `changelog` file if they did not access the `GitHub` repository of these packages.

4.1% of the packages with at least one yanked release explain the rationale behind yanking a release in the issue reports or pull requests of their GitHub repository. We found that 192 packages explained the rationale behind yanking a release in their issue reports or pull requests (13 of the packages also explained in changelogs). In addition, we noticed that the pull requests were usually created by the owner of the package to explain why a release was yanked. However, the issue reports were mostly created by developers who used the package to ask questions about yanked releases.

> **RQ2 Summary:** In `Cargo`, packages use the yank command for several reasons other than just to indicate a release is defective, but they rarely provide the reason for yanking.

5.3 RQ3: How many packages adopt yanked releases?

Motivation. `Cargo` does not allow the owner of a package to delete their releases, but it allows the owner to remove a release from the registry index by yanking. Hence, packages cannot point a new dependency to a yanked release because `Cargo` cannot find it in the registry index. In this research question, we study how often packages directly adopted yanked releases.

In addition, we study how many releases have *unresolved dependencies* due to yanked releases. The yank mechanism in `Cargo` is more forceful than the deprecation mechanism in `npm`. When `npm` resolves the dependency requirements for a package, it will use a deprecated release if needed and provide a warning message. However, `Cargo` will not choose a yanked release even if only this yanked release can satisfy the dependency requirements, which leads to unresolved dependencies. We call a release with unresolved dependencies an *implicitly yanked release*, because `Cargo` cannot use this release to resolve the dependency requirements of other package (and hence build those packages) even though this release was not yanked explicitly.

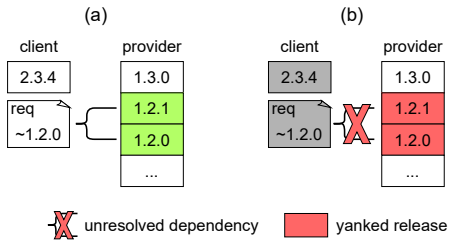

Fig. 5: Two scenarios of resolving dependencies: (a) The dependency requirement can be resolved; (b) The dependency requirement cannot be satisfied because of yanking.

Approach. We analyzed the `dependencies` table in the database (as shown in Table 2) to collect packages which directly adopted at least one yanked release. First, we selected the dependency requirements which can be satisfied by a yanked release. Then, we collected the information about the owners (i.e., releases from different packages) of these requirements. For each yanked release d, we calculated the proportion p_d of direct adoptions of yanked releases [28]. The value of p_d was calculated by:

$$p_d = \frac{a_d}{\sum_{d \in yanked\ releases} a_d}$$

where a_d is the number of times that release d is directly adopted by a package. For example, if packages directly adopted 100,000 yanked releases (sum of a_d) and a yanked release d accounts for 1,000 times of these adoptions (a_d), the value of p_d is 1%.

Next, we study how many releases have unresolved dependencies because of adopting yanked releases. Figure 5 shows two scenarios in which a dependency requirement of a client package points to a provider package. In the first scenario, version `2.3.4` of the client package has a requirement `~1.2.0`, and this requirement can be resolved by `1.2.0` or `1.2.1` of the provider package. In the second scenario, only versions `1.2.0` and `1.2.1` can satisfy the requirement `~1.2.0` but these two versions are both yanked. We investigated all the dependency requirements and collected the releases which have unresolved dependencies.

In addition, we study how yanked releases propagate through the dependencies in the ecosystem. The propagation happens when yanked releases break dependency requirements and cause implicitly yanked releases (i.e., releases with unresolved dependencies). We collected the implicitly yanked releases which directly adopted yanked releases. However, the propagation continues if those implicitly yanked releases cause new unresolved dependencies. Hence, we performed the analysis recursively to collect all implicitly yanked releases.

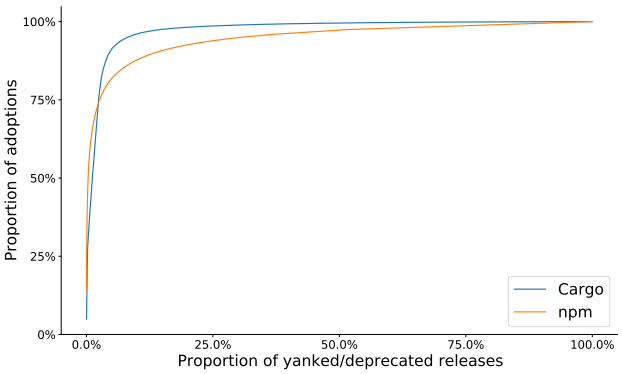

Fig. 6: Cumulative histogram for the proportion of direct adoptions of yanked releases in `Cargo` and `npm`.

Findings. 46% of packages in Cargo directly adopted at least one yanked release and 2.4% of the yanked releases accounted for 75% of these adoptions. There are 22,277 packages that directly adopted at least one yanked release of a partially yanked package and 268 packages directly adopted fully yanked packages in `Cargo`. The proportion of packages which directly adopted yanked releases in `Cargo` is 19% higher than in `npm` (27%). Although such adoptions do not directly put a package at risk (since there could be newer releases that satisfy a requirement), they could contribute to unresolved dependencies if another release is yanked. One reason could be that `Cargo` has a higher proportion of packages that are partially yanked (Section 5.1). In addition, Figure 6 shows that most of the direct adoptions of yanked releases are concentrated on a relatively small proportion of yanked releases in both `Cargo` and `npm`.

1.4% of the releases in `Cargo` have unresolved dependency requirements which are related to yanked releases. 4,158 releases in `Cargo` have unresolved dependency requirements because it directly or transitively adopted a yanked release and these releases became implicitly yanked releases. We found that 65.2% (2,712 releases) of the implicitly yanked releases are caused by packages that follow Pattern 3 (yanking back-to-back releases) in Section 5.2 and 39.2% (1,631 releases) of the implicitly yanked releases are caused by the `ring` package which yanked unsupported old releases. Moreover, we noticed that 15 releases from 5 packages have invalid dependency requirements (i.e., they are unrelated to yanked releases). For example, 10 releases of `leveldb`,[56] version numbers `0.3.4` to `0.5.1`, have a common dependency requirement that indicates that a version of `db-key`[57] should satisfy the constraint `^0.0.4`. However, there is no release (i.e., yanked or unyanked) in `db-key` that can fulfill this requirement.

54% of the implicitly yanked releases are caused by direct adoption of yanked releases. There are 2,266 releases that directly adopted yanked releases and became implicitly yanked releases. For example, version `0.1.2` of `chrono`[58] has a caret dependency constraint for `time`[59] (i.e., `^0.0.3`), but `time` yanked all releases before `0.1.0`. In this case, this dependency constraint cannot be satisfied and version `0.1.2` of `chrono` becomes an implicitly yanked release. We found that 54% (1,411 releases) of these implicitly yanked releases were caused by only 10 packages. For example, `clap`,[60] is a library for parsing command-line arguments which yanked 91 (out of 211) releases. There are 353 releases from 85 packages which directly adopted at least one of these yanked releases from `clap` and became implicitly yanked.

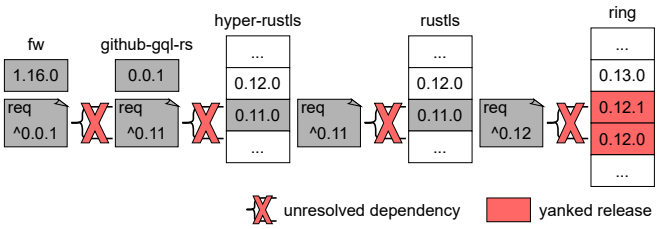

Fig. 7: An example of yanking propagation of `ring`. `0.11.0` of `rustls`, `0.11.0` of `hyper-rustls`, `0.0.1` of `github-gql-rs`, and `1.16.0` of `fw` became implicitly yanked releases.

46% of the implicitly yanked releases are caused by the transitive adoption of a yanked release and 54% of them are propagated from a single package. We found 1,892 implicitly yanked releases that transitively adopted yanked releases, and then themselves became implicitly yanked. Figure 7 shows that version `1.16.0` of `fw`[61] became implicitly yanked since it transitively adopted yanked releases of `ring`.[62] In this case, the implicitly yanked release `0.11.0` of `rustls`[63] was propagated from two yanked releases of `ring`. Then, the propagation happened to `hyper-rustls`,[64] `github-gql-rs`,[65] and `fw`. The depth of this propagation is 4 and we noticed that the maximum depth of yanking propagation in `Cargo` is also 4. However, 1,631 (39%) out of 4,158 implicitly yanked releases were propagated from a common root package `ring`.

> **RQ3 Summary:** 46% of packages directly adopted at least one yanked release. Yanked releases were propagated in `Cargo` and 1.4% of the releases that are currently in `Cargo` have unresolved dependencies due to the yanking propagation.

6 Implications

In this section, we discuss our findings and implications for the maintainers of package managers, the package owners, the maintainers of `Cargo`, and researchers.

6.1 Implications for maintainers of package managers

Package managers should implement a release-level deprecation mechanism. The proportion of deprecated releases in `Cargo` and `npm` are 3.7% and 3.2% respectively, and the percentage of deprecated releases has gradually increased from 2014 to 2020 in Cargo (Section 5.1). Since the release-level deprecation mechanism has seen increased use in these two ecosystems, there is likely a need for it in other ecosystems as well. In recent years, `PyPI` and `NuGet` saw the need and have started to support release-level deprecation in April 2020 and September 2019. We suggest other package managers also implement the release-level deprecation mechanism.

Package managers should provide features to support the various ways in which developers deprecate releases. We observed five patterns of yanking in Cargo (Section 5.2). To support Pattern 2, we suggest that the deprecation mechanism of package managers should support package-level deprecation, which is also used by 80% of the deprecations in npm [28]. However, package managers should notice that deprecating packages can break the dependents of those packages if the managers implement a forceful deprecation mechanism. To support other patterns, we suggest package managers to support deprecating a single release or multiple releases at the same time to offer flexibility for developers.

Package managers should allow package owners to decide whether a deprecation is forceful or not. A forceful deprecation mechanism like yanking in Cargo can cause unresolved dependencies for releases of other packages (Section 5.3). We suggest that package managers leave the choice of forceful deprecation to the package owners. The package owners can deprecate a release forcefully (like in Cargo) when it is necessary, such as if they found security vulnerabilities in a cryptography package. Otherwise, the package owner can decide to deprecate a release non-forcefully (like in npm) and allow developers to decide whether they still want to adopt the deprecated release in their own packages. In addition, we suggest that package managers provide a warning with information about how many packages would break (similar to our analysis in Section 5.3) if a developer decides to deprecate forcefully.

6.2 Implications for package owners

Package owners should explain the rationales behind yanked releases in the documentation. 94.7% of the packages that have at least one yanked release in Cargo never explained the rationale behind yanking (Section 5.2). This percentage is high compared to npm in which for 64% of the deprecated releases a rationale is known [28]. Since Cargo does not support adding a message for a yanked release, we recommend that package owners record the reason for yanking a release in the package's documentation. For example, package owners can create an issue report to track a yanked release and put its link into the readme or changelog. The issue report should contain detailed information about a yanked release, and provide a place for developers to discuss this release, or to give advice on how to deal with the yanking. In addition, we noticed that only 21% of the packages (Section 5.2) have a changelog in Cargo. We suggest package owners maintain a changelog to tell developers about the notable changes in each release, which can also be used to explain the rationale behind yanked releases.

Package owners should avoid yanking old releases which are no longer supported without providing an alternative release or migration guidelines. We found that 39.2% of the implicitly yanked releases in Cargo are caused by the ring package which yanked old unsupported releases (Pattern 3 in Section 5.2), even though these packages had no known vulnerabilities (Section 5.3). For the packages that insist on yanking unsupported releases, we recommend they indicate replacement releases or provide guidelines for developers to migrate away from yanked releases.

TABLE 4: Comparisons of the yanked mechanism in Cargo and the deprecation mechanism in npm.

Package manager	Record deprecation date	Describe rationale	Deprecate a package	Ban yanked releases
Cargo	✓			✓
npm		✓	✓	

6.3 Implications for Cargo maintainers

We compare the yanked mechanism in Cargo with the deprecation mechanism in npm based on our findings. We summarize the difference in Table 4 for Cargo maintainers.

Cargo should allow the owner of a package to add a note to a yanked release and provide a warning for packages that adopted it. We found that the percentage of packages that explain the rationales behind yanked releases is low in Cargo (Section 5.1) compared to npm (5.3% vs. 64%). One reason could be that npm allows the package owners to add a message for deprecated releases while Cargo does not. Moreover, we observed that it is much more common in Cargo to yank only a few releases instead of the whole package (Section 5.2) and the owners of packages yanked releases for various reasons. For fully yanked packages, at least developers know that these packages will probably no longer be maintained. However, developers who depend on a partially yanked package can hardly understand what is happening since there is no mechanism for describing why a release was yanked. Hence, we recommend that Cargo should allow the owner of a package to leave a message when they are yanking a release. There is an issue report[66] asking for the same functionality since April 23, 2016, but it is still not implemented. With the increasing proportion of yanked releases in the ecosystem (Section 5.1), more developers will be affected by this issue. In addition, we recommend that Cargo should provide a warning message for packages that adopted a yanked release.

Cargo should detect implicitly yanked releases and provide a warning for these releases. We found that 1.4% of releases in Cargo are implicitly yanked (Section 5.3). We recommend that Cargo should mention that a release is implicitly yanked on the webpage of a package. For example, there could be an "unresolved" label beside an implicitly yanked release on the webpage, hence developers can avoid using this release. In addition, for a package which adopted an implicitly yanked release, Cargo can show the dependency tree and indicate the release which breaks the dependencies for developers.

Cargo should warn the owner of a package which adopts a yanked release in its lock file. Cargo.lock stores the information about dependencies locally for a project if the project was compiled successfully. However, one of the dependencies can be yanked after the compilation and Cargo does not inform the developer. Since the proportion of packages which directly adopted yanked releases is 46% in Cargo (Section 5.3), we recommend that Cargo should check up the package registry when developers are building their project based on the Cargo.lock. Hence, Cargo

can give a warning message to developers if one of the dependencies was yanked.

6.4 Implications for researchers

Researchers should study automatic semantic versioning guarantee checkers to detect whether a release follows the guarantee. In Section 5.2, we found that *Breaking SemVer* is the most common rationale behind yanking. This finding indicates the difficulty for package owners to decide whether an update follows the guarantee. Automatic semantic versioning guarantee checkers can help package owners by analyzing the code before a release is published. In addition, these checkers can be used to analyze how packages in different software ecosystems follow the semantic versioning guarantee.

7 Threats to validity

In this section, we discuss the threats to the validity of our study about yanked releases in `Cargo`.

Internal validity: We analyzed the percentage of yanked releases from 2014 to 2020 based on the `Git` history of the registry index. Ideally, this index repository is updated automatically by a program. However, we found eight records which show that the maintainer edited the index manually to delete some packages. Since these deletions are not considered, our results will include the yanked releases of these deleted packages.

The identification of changelogs is based on searching keywords in `readme` and matching filenames under the root directory of packages' `GitHub` repository. It is possible that the owner of a package did not use a word in our keyword list to indicate their changelog. We randomly selected 341 readmes which correspond to a 95% confidence level and $\pm 5\%$ confidence interval. The manual analysis shows that our heuristic approach achieves a precision of 91% and a recall of 91%.

In addition, we filter out changelogs, issue reports, and pull requests which do not have "yank" and "deprecate" in the content. This filtering might exclude information that explains why a release was yanked. We randomly selected 100 changelogs and 100 issue reports/pull requests from the excluded samples to manually verify whether they contain the rationales for yanked releases. We found that the keywords searching approach missed 3 of the 100 changelogs (2 for *Package removed or replaced* and 1 for *Defect*) and 0 of the 100 issue reports/pull requests. Since few packages maintained a changelog and most rationales are identified from issue reports/pull requests (Section 5.2), the result of identifying rationales is considered reliable. Besides, we selected 638 packages which have the keywords in changelogs, issue reports, or pull requests and filtered out 380 out of the 638 for card sorting. The author who did not participate in the card sorting independently analyzed 100 samples of the 380 packages and reported the false negative rate is 9%.

External validity: We studied yanked releases in the `Rust` and `npm` package registries, but the findings of our study may not generalize to the package managers of other programming languages. One reason could be that other package managers may not provide a release-level deprecation mechanism for developers, and the identification of whether a release is yanked could be complicated (or not possible). Future studies should further investigate other software ecosystems with release-level deprecation.

We investigated the changelogs, issue reports, and pull requests of packages with at least one yanked release. However, we only looked at packages which provide a link to their `GitHub` repository. There are 15% of these packages that do not have a link to their repository and 214 packages (4%) provide a link to other repository hosting platforms such as `Bitbucket`[67] and `GitLab`.[68] Hence, future studies are necessary to investigate if our results hold for packages that maintain their code outside of `GitHub`.

The results of our study might not apply directly to other software ecosystems, because the community and the development model of the programming language can also affect the results. However, our methodology can be applied to analyze other software ecosystems.

8 Conclusion

We studied 48,823 packages in `Cargo` to understand the yank mechanism. In particular, we studied the frequency in which the yanked mechanism is used, the patterns of yanking in packages and the rationales behind yanking, and the adoption of yanked releases. The most important findings of our study are:

1. The proportion of yanked releases in `Cargo` has increased from 1.4% to 3.7% between 2014 and 2020.
2. Even though the proportions of packages with at least one yanked release are similar in `Cargo` and `npm`, these packages in `Cargo` have a lower yanking rate.
3. We observed 5 yanking patterns in `Cargo` and the rationales include withdrawing a defective release or a release that broke the semantic versioning guarantee, indicating a package is removed or replaced, or fixing dependencies.
4. 46% of packages directly adopt at least one yanked release in `Cargo`. These yanked releases were propagated through the adoption, causing 1.4% of the releases in `Cargo` to have unresolved dependencies and hence become unbuildable.

In this paper, we found that yanked releases cause unresolved dependencies since the yank mechanism is more forceful in `Cargo` than `npm`. Based on our findings, we suggest that `Cargo` should provide a package-level deprecation mechanism and allow package owners to leave a reason for yanking a release, and we recommend that other package managers integrate a release-level deprecation mechanism as well.

Acknowledgments

The work described in this paper has been supported by the ECE-Huawei Research Initiative (HERI) at the University of Alberta.

References

[1] G. Avelino, E. Constantinou, M. T. Valente, and A. Serebrenik, "On the abandonment and survival of open source projects: An empirical investigation," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2019, pp. 1–12.

[2] N. Cerpa and J. M. Verner, "Why did your project fail?" *Commun. ACM*, vol. 52, no. 12, pp. 130–134, Dec. 2009.

[3] S. Chacon and B. Straub, *Pro Git*, 2nd ed. USA: Apress, 2014.

[4] M. Claes, T. Mens, and P. Grosjean, "maintaineR: A web-based dashboard for maintainers of CRAN packages," in *IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 597–600.

[5] ——, "On the maintainability of CRAN packages," in *IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 308–312.

[6] 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 (ESEC/FSE)*. ACM, 2017, pp. 186–196.

[7] 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*, ser. ESEM '18. ACM, 2018.

[8] F. R. Cogo, G. A. Oliva, and A. E. Hassan, "An empirical study of dependency downgrades in the npm ecosystem," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[9] E. Constantinou, A. Decan, and T. Mens, "Breaking the borders: an investigation of cross-ecosystem software packages," *arXiv preprint arXiv:1812.04868*, 2018.

[10] D. R. Cox and A. Stuart, "Some quick sign tests for trend in location and dispersion," *Biometrika*, vol. 42, no. 1/2, pp. 80–95, 1955.

[11] A. Decan and T. Mens, "What do package dependencies tell us about semantic versioning?" *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[12] A. Decan, T. Mens, M. Claes, and P. Grosjean, "When GitHub meets CRAN: An analysis of inter-repository package dependency problems," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, vol. 1, 2016, pp. 493–504.

[13] A. Decan and T. Mens, "Lost in zero space – an empirical comparison of 0.y.z releases in software package distributions," 2021.

[14] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. ACM, 2018, pp. 181–191.

[15] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Softw. Engg.*, vol. 24, no. 1, pp. 381–416, Feb. 2019.

[16] A. N. Evans, B. Campbell, and M. L. Soffa, "Is Rust used safely by software developers?" in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. ACM, 2020, pp. 246–257.

[17] D. M. German, B. Adams, and A. E. Hassan, "The evolution of the R software ecosystem," in *2013 17th European Conference on Software Maintenance and Reengineering*, 2013, pp. 243–252.

[18] G. Iaffaldano, I. Steinmacher, F. Calefato, M. Gerosa, and F. Lanubile, "Why do developers take breaks from contributing to OSS projects? a preliminary analysis," in *Proceedings of the 2nd International Workshop on Software Health*. IEEE Press, 2019, pp. 9–16.

[19] S. K. Imminni, M. A. Hasan, M. Duckett, P. Sachdeva, S. Karmakar, P. Kumar, and S. Haiduc, "SPYSE: A semantic search engine for Python packages and modules," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. ACM, 2016, pp. 625–628.

[20] J. Khondhu, A. Capiluppi, and K.-J. Stol, "Is It All Lost? A Study of Inactive Open Source Projects," in *9th Open Source Software (OSS)*, ser. Open Source Software: Quality Verification, vol. AICT-404. Koper-Capodistria, Slovenia: Springer, Jun. 2013, pp. 61–79.

[21] D. Lin, C.-P. Bezemer, and A. E. Hassan, "An empirical study of early access games on the Steam platform," *Empirical Software Engineering*, vol. 23, 04 2018.

[22] J. D. Long, D. Feng, and N. Cliff, *Ordinal Analysis of Behavioral Data*. American Cancer Society, 2003, ch. 25, pp. 635–661.

[23] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Annals of Mathematical Statistics*, vol. 18, pp. 50–60, 1947.

[24] 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*. ACM, 2017, pp. 145–150.

[25] N. D. Matsakis and F. S. Klock, "The Rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14. ACM, 2014, pp. 103–104.

[26] 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*, ser. FSE '12. ACM, 2012.

[27] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohensd indices the most appropriate choices," in *annual meeting of the Southern Association for Institutional Research*. Citeseer, 2006, pp. 1–51.

[28] F. Roseiro Côgo, G. Oliva, and A. E. Hassan, "Deprecation of packages and releases in software ecosystems: A case study on npm," *IEEE Transactions on Software Engineering*, 01 2021.

[29] A. A. Sawant, M. Aniche, A. van Deursen, and A. Bacchelli, "Understanding developers' needs on deprecation as a language feature," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 561–571.

[30] A. Serebrenik and T. Mens, "Challenges in software ecosystems research," in *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ser. ECSAW '15. ACM, 2015.

[31] 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 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018, pp. 644–655.

[32] R. Štrobl and Z. Troníček, "Migration from deprecated API in Java," in *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '13. ACM, 2013, pp. 85–86.

[33] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated Python library APIs are (not) handled," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2020, pp. 233–244.

[34] 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*, ser. ICSE '20. ACM, 2020, pp. 125–135.

[35] 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 (MSR)*. ACM, 2016, pp. 351–361.

[36] Y. Xi, L. Shen, Y. Gui, and W. Zhao, "Migrating deprecated API to documented replacement: Patterns and tool," in *Proceedings of the 11th Asia-Pacific Symposium on Internetware*. ACM, 2019.

[37] J. Yasmin, Y. Tian, and J. Yang, "A first look at the deprecation of RESTful APIs: An empirical study," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 151–161.

[38] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the diversity of software package popularity metrics: An empirical study of npm," in *IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 589–593.

List of URLs

1. https://www.npmjs.com/
2. https://github.com/npm/cli/blob/v0.2.8/lib/deprecate.js
3. https://doc.rust-lang.org/cargo
4. https://github.com/rust-lang/crates.io/commit/663488fc0a0073d47402e61cf9cb999d054652c6
5. http://www.modulecounts.com/
6. https://maven.apache.org
7. https://pypi.org/
8. https://packagist.org
9. https://www.nuget.org/
10. https://rubygems.org
11. https://github.com/rubygems/rubygems/commit/6f99254adf19a35850b1a9b81eb5356ff45f6090#diff-f1e136837461f7ea89d3c7442de42b0723bc7099fe3d77f76ba96823d8530378
12. https://blog.rubygems.org/2015/04/13/permadelete-on-yank.html
13. https://devblogs.microsoft.com/nuget/deprecating-packages-on-nuget-org/
14. https://discuss.python.org/t/pep-592-support-for-yanked-files-in-the-simple-repository-api/1629
15. https://crates.io
16. https://doc.rust-lang.org/rustc
17. https://doc.rust-lang.org/cargo/commands/cargo-search.html
18. https://doc.rust-lang.org/cargo/commands/cargo-publish.html
19. https://github.com/steveklabnik/semver
20. https://doc.rust-lang.org/cargo/reference/manifest.html
21. https://doc.rust-lang.org/cargo/faq.html
22. https://doc.rust-lang.org/cargo/commands/cargo-yank.html
23. https://doc.rust-lang.org/cargo/reference/registries.html
24. https://doc.rust-lang.org/cargo/guide/cargo-toml-vs-cargo-lock.html
25. https://doc.rust-lang.org/cargo/faq.html#why-do-binaries-have-cargolock-in-version-control-but-not-libraries
26. https://crates.io/data-access
27. https://docs.github.com/en/rest
28. https://github.com/rust-lang/crates.io-index
29. https://internals.rust-lang.org/t/cargos-crate-index-upcoming-squash-into-one-commit
30. https://doc.rust-lang.org/cargo/commands/cargo-yank.html
31. https://crates.io/crates/pyo3
32. https://github.com/PyO3/pyo3/issues/285
33. https://crates.io/crates/diesel_cli
34. https://crates.io/crates/winping
35. https://github.com/TyPR124/winping/blob/master/RELEASES.md
36. https://crates.io/crates/quote
37. https://crates.io/crates/sdl2
38. https://github.com/Rust-SDL2/rust-sdl2/issues/478
39. https://crates.io/crates/ncollide
40. https://github.com/dimforge/ncollide/issues/322
41. https://crates.io/crates/c
42. https://github.com/hilbert-space/c/issues/1
43. https://crates.io/crates/clap
44. https://github.com/clap-rs/clap/issues/2076
45. https://github.com/briansmith/untrusted/issues/29
46. https://github.com/bitvecto-rs/bitvec/issues/59
47. https://crates.io/crates/ring
48. https://github.com/briansmith/ring/issues/774
49. https://github.com/rust-lang/rfcs/pull/2495
50. https://crates.io/crates/block-buffer
51. https://github.com/RustCrypto/utils/issues/22
52. https://crates.io/crates/battery-cli
53. https://crates.io/crates/sic
54. https://github.com/foresterre/sic/issues/193
55. https://github.com/foresterre/sic/issues/50
56. https://crates.io/crates/leveldb
57. https://crates.io/crates/db-key
58. https://crates.io/crates/chrono
59. https://crates.io/crates/time
60. https://crates.io/crates/clap
61. https://crates.io/crates/fw
62. https://crates.io/crates/ring
63. https://crates.io/crates/rustls
64. https://crates.io/crates/hyper-rustls
65. https://crates.io/crates/github-gql-rs
66. https://github.com/rust-lang/cargo/issues/2608
67. https://bitbucket.org/
68. http://gitlab.com/