Back to the Past – Analysing Backporting Practices in Package Dependency Networks

Alexandre Decan, *Member, IEEE,* Tom Mens, *Senior Member, IEEE,*
Ahmed Zerouali, Coen De Roover, *Member, IEEE*

Abstract—The practice of backporting aims to bring the benefits of a bug or vulnerability fix from a higher to a lower release of a software package. When such a package adheres to semantic versioning, backports can be recognised as new releases in a lower major train. This is particularly useful in case a substantial number of software packages continues to depend on that lower major train. In this article, we study the backporting practices in four popular package distributions, namely *Cargo*, *npm*, *Packagist* and *RubyGems*. We observe that many dependent packages could benefit from backports provided by their dependencies. In particular, we find that a majority of security vulnerabilities affect more than one major train but are only fixed in the highest one, letting thousands of dependent packages exposed to the vulnerability. Despite that, we find that backporting updates is quite infrequent, and mostly practised by long-lived and more active packages for a variety of reasons.

Index Terms—D.2 Software Engineering; D.2.7 Distribution, Maintenance, and Enhancement; D.2.7.g Maintainability; D.2.7.n Version control; D.2.8 Metrics/Measurement; D.2.13 Reusable Software; D.2.13.b Reusable libraries; D.2.16 Configuration Management; D.2.16.f Software release management and delivery

1 Introduction

Backporting refers to the process of applying a software update, typically a patch that fixes a bug or security vulnerability, to a lower version of the software. As stated by Bogart et al. [1], "*making bug fixes to outdated versions of code, or even backporting new features, can be helpful for users who cannot update to the cutting-edge versions for some reason.*" Backporting patches therefore enables users of lower releases to benefit from more recent fixes and features.

This article studies the practice of backporting in the context of distributions of reusable software libraries that form large and evolving package dependency networks [2]. In this context, backports become relevant for packages that depend on other packages, especially when those dependent packages do not refer to the highest available version of the required package. Depending on outdated packages has been shown to be a prevalent and well-studied problem, as it brings about an increased risk of bugs and security vulnerabilities [3], [4], [5], [6], [7], [8].

Because it is so common for packages to have outdated dependencies, we aim to gain empirical insights into how backporting is used in package dependency networks. Backporting requires additional effort from the package producer, as it entails transplanting important bug and vulnerability fixes to earlier versions. In contrast, dependent packages can benefit from backported fixes without having to upgrade to a higher major or minor version. This is especially useful when upgrading is not feasible due to known version incompatibilities or because it would incur risks or involve too much effort.

We are the first to carry out an extensive empirical study into backporting practices and compare these practices across four popular software package distributions that are known to adhere to semantic versioning: *Cargo* for the *Rust* programming language, *npm* for *JavaScript*, *Packagist* for *PHP* and *RubyGems* for the *Ruby* programming language. More concretely, we study the following research questions for the dependency networks of those distributions:

- RQ_1 *How outdated are dependent packages?* This initial question allows us to understand if there is a real need for backporting. This would not be the case if all dependent packages are relying on the highest version of their required packages.
- RQ_2 *How many major versions of a package are used by its dependents?* If lower major versions of a required package are still being used actively, the use of backporting becomes more relevant for package producers, in order to continue to support dependents relying on those lower major versions.
- RQ_3 *How prevalent are backports?* This question aims to understand how many required packages practice backporting and how many of their dependents benefit from this practice.
- RQ_4 *Do packages with backports exhibit different characteristics?* Since not all packages practice backporting, this question aims to characterise those packages that do.
- RQ_5 *How long do lower major releases continue to be supported?* This question aims to assess whether lower major releases continue to benefit from backporting practices for extended periods of time.
- RQ_6 *To which extent are security fixes being backported?* This question aims to quantify one of the main goals of

- *A. Decan and T. Mens are with the Software Engineering Lab, University of Mons, Avenue Maistriau 15, B-7000 Mons, Belgium. E-mail: alexandre.decan@umons.ac.be and tom.mens@umons.ac.be*
- *A. Zerouali and C. De Roover are with the Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussel, Belgium. E-mail: ahmed.zerouali@vub.be and coen.de.roover@vub.be*

backporting, namely to allow outdated packages to continue to benefit from (backported) security updates.

The remainder of this article is structured as follows. Section 2 presents related work. Section 3 describes the research method followed, including the terminology that will be used, and the data extraction and preprocessing phase. Section 4 addresses each of the research questions. Section 5 presents the threats to validity of our study. Section 6 discusses the consequences of our research findings and the recommendations that can be made based on these findings. Finally, Section 7 concludes the article.

2 Related Work

Backporting has primarily been studied in the context of the Linux operating system. Tian et al. [9] proposed to automatically identify bug fixing patches in the Linux kernel in order to apply them to older long term releases. Ray et al. [10] proposed to detect and characterize porting errors to help developers avoid them. Rodriguez et al. [11] studied backporting of device drivers for older OS versions, and presented a backporting strategy involving the use of a backport library and of the program transformation tool Coccinelle to automatically generate code to be backported. Thung et al. [12] proposed an automatic recommendation system to guide the selection of candidate code changes to be backported. Shariffdeen et al. [13] proposed a patch backporting tool called FixMorph to automate the transfer of patches from the mainline version of the Linux kernel into older stable versions.

In 2017, Christopher Bogart et al. reported on a survey with more than 2,000 developers to study the values and practices across 18 different software ecosystems [14], [1]. One of these questions was "*When working on (my package), I spend extra time backporting changes, i.e., making similar fixes to prior releases of the code, for backward compatibility.*" The results that they reported for the 4 package distributions considered in this paper are shown in Figure 1. For each distribution a majority of developers reports that they do not spend extra time to backport changes in their packages, suggesting that backporting practices are still underexploited.

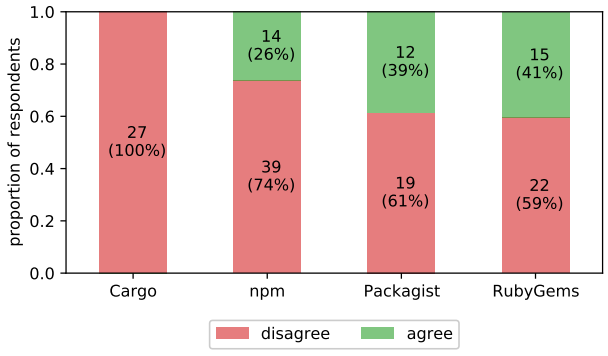

Fig. 1: Do developers spend time on backporting changes?

Also related is the work on patch transplantation and patch porting, which consists of taking patches that repair software defects or software vulnerabilities in one project, and transplanting or porting this patch to forked projects [15] or other similar projects [16]. This line of research is very interesting and relevant in its own right, and automated patch transplantation techniques are likely to be useful for backporting patches as well.

Another important and related line of research focuses on package outdatedness, referring to the extent to which a package is not using the most up-to-date versions of the packages on which it depends (either directly or transitively). Backporting only becomes relevant if package dependencies are outdated, and especially if the declared dependency is one or more major versions behind. Outdatedness makes packages more vulnerable to security issues, even if there are known patches for these issues. Cox et al. [17] introduced a metric of dependency freshness to study the relationship between outdated dependencies and security vulnerabilities. Mirhosseini [18] studied the use of automated pull requests to update outdated software dependencies. Gonzalez-Barahona et al. [19], [4], [7], [8] introduced the notion of technical lag to quantify the degree of outdatedness of packages, along different dimensions, including time lag, version lag and vulnerability lag. Wang et al. [6] studied the relation between outdatedness (update delay) and security risks in 13,565 third-party libraries of 806 *Java* projects. Lauinger et al. [3] studied the use of vulnerable and outdated *JavaScript* libraries in 133k websites, and observed that transitive libraries are more likely to be vulnerable. Zimmermann et al. [5] found evidence of single points of failure in the *npm* dependency network of *JavaScript* packages, containing unmaintained packages with security vulnerabilities that transitively impact a large number of packages.

Research on semantic versioning is also related. Semantic versioning introduces a set of simple rules that suggest how to assign version numbers to package updates to inform developers about potentially breaking changes. Raemaekers et al. [20] studied the relation between semantic versioning and the impact of breaking changes in the *Maven* dependency network of *Java* packages. Similarly, Decan et al. [21] studied semantic versioning practices in 4 other package dependency networks, by analysing the dependency constraints used when declaring package dependencies. Opdebeeck et al. [22] studied the practice of semantic versioning in the Ansible Galaxy ecosystem of Infrastructure-as-Code libraries.

3 Research Method

3.1 Terminology

This section introduces the terminology used throughout this article. All main terms are highlighted in **boldface**.

Package distributions (such as *Cargo*, *npm*, *Packagist* and *RubyGems*) are collections of (typically open source) software **packages**, distributed through some package manager. Each package is developed and maintained by a (team of) developer(s) whom we refer to as the **package producer**. Each package has one or more **releases** that are denoted by a unique **version number**. New releases of a package are called **package updates**. The version number reflects the sequential order of all releases of a package.

Package releases can declare **dependencies** on other packages. In this way, the collection of all package releases

contained in a package distribution forms a **package dependency network**. If package release R depends on package P, R is called a **dependent**, while P is called a **required package**. In the context of this article, we only consider a single **snapshot** of each package distribution, containing all packages and their entire release history available until the snapshot date. By abuse of terminology we declare a *package* to be **dependent** on P if its latest available release in this snapshot depends on P.

A dependent can specify a **dependency constraint** to describe which releases of P are allowed to be selected for installation. Such constraints express a **version range**. For example, constraint $< 2.0.0$ defines the version range $[0.0.0, 2.0.0[$, signifying that *any* release below version 2.0.0 of the required package is allowed to be installed. In order to benefit from bug fixes and newly added functionalities, producers of dependent packages need to keep their dependencies up to date. This may require significant effort, especially in case the newer releases of a dependency include backward incompatible changes [23].

Semantic versioning, hereafter abbreviated as semver[1], proposes a multi-component version numbering scheme major.minor.patch[-tag] to specify the type of changes that have been made in a package update. Backward incompatible changes require an increment of the **major** version component, important backward compatible changes (e.g., adding new functionality that does not affect existing dependents) require an increment of the **minor** component, and backward compatible bug fixes require an increment of the **patch** component. Combining semver with dependency constraints enables producers of dependent packages to restrict the version range of a required package to those releases that are expected to be backward compatible [21].

To each major version of a package we can associate a corresponding **major train**, being the ordered sequence of all releases of that package that have the same major version component.[2] We will consider a major train of a package to be higher (respectively, lower) than another major train if the corresponding major version component is higher (respectively, lower).

Backporting refers to the process of taking a minor or patch update applied in a higher major train, and applying this update (possibly after some necessary rework) to one or more lower major trains. The original update will be called the **backported update**, and the one applied to the lower train will be called a **backport**. Figure 2 visually illustrates this backporting process: patch update 2.1.1 fixes a vulnerability discovered in release 2.1.0. Since the vulnerability also affects release 1.3.0, the fix is backported to major train 1 through patch update 1.3.1. Similarly, patch update 3.1.1 fixes a vulnerability discovered in release 3.1.0, and this fix is backported to the two lower major trains 1 and 2 through patch updates 1.3.2 and 2.2.1, respectively.

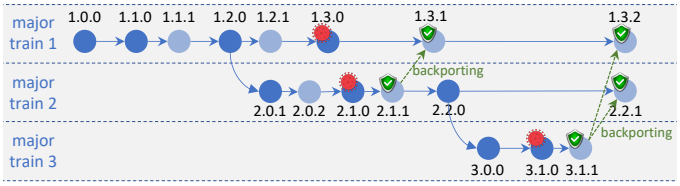

Fig. 2: Examples of backporting security vulnerability patches from the highest major train to lower ones.

3.2 Data Extraction

For our empirical analysis we rely on version 1.6.0 of the libraries.io dataset, released in January 2020, which contains dependency metadata for 33 distinct open source package distributions [24]. We focus on four of the largest package distributions for which we already studied the practice of semantic versioning and dependency constraints in earlier work [21]: *Cargo* for the *Rust* programming language, *npm* for *JavaScript*, *Packagist* for *PHP* and *RubyGems* for *Ruby*.

The dataset contains package metadata, including all releases of the package, their version number, their release date, and their dependencies. For each dependency, there is information about the required package, the dependency constraint, and the scope of the dependency (e.g., whether it is needed to execute, develop or test the package). We excluded dependencies that are only needed to test or develop a package because not all considered package distributions make use of them, and not every package declares a complete and reliable list of such dependencies. We therefore consider only those dependencies that are required to execute the package, and that hence more accurately reflect what is needed to actually use the package. For each package release, we consider only dependencies to other packages within the same distribution, i.e., we ignore dependencies targeting external sources (e.g., websites or git repositories).

The four considered package distributions rely on a different syntax for expressing dependency constraints on the range of releases that can be accepted by a declared package dependency. We converted those constraints to a uniform version range notation [25]. To do so, we wrote a parser for each package distribution. Those parsers were able to cope with 99.4% of all considered dependency constraints: 100% for *Cargo*, 99.6% for *npm*, 97.7% for *Packagist*, and 94.3% for *RubyGems*. The remaining dependencies, i.e., those that were not syntactically correct, were excluded from the dataset.

Since our goal is to study backports, we focus on *active packages* by only considering required and dependent packages that have been updated at least once in the last 12 months. In order to have sufficient information to study the impact of backported updates on dependents, we only consider required packages having *at least 5 dependents* in the latest considered snapshot of each package distribution, together with all dependents of these packages. To do so, we rely on the dependency metadata specified in the latest available release of each package in the dataset. We exclude packages for which the release dates of its major versions do not follow a chronological order (e.g., a major version x released before major version $y < x$). This usually corresponds to packages whose releases were imported all at once in the package distribution, leading their release dates to correspond to the import date and not to their actual release date. We found respectively 5, 363, 88 and 63 of such

1. See https://semver.org

2. We use the term "major train" by analogy with the notion of "release train". Some authors have used "major branch" instead, which could be confused more easily with the notion of git branches.

packages for *Cargo*, *npm*, *Packagist* and *RubyGems*.

At the end of this data filtering phase, the curated dataset consists of 19,365 required packages (and their entire release history), and of the latest release of the 299,164 packages that depend on them. Table 1 compares the initial dataset with the curated one, broken down per considered package distribution. From this table, one can derive that ≥ 5 dependents of active required packages roughly corresponds to 20% packages in each package distribution (to be precise: 19% for *Cargo*, 20% for *npm*, 17% for *Packagist* and 22% for *RubyGems*).

4 Research Questions

This section aims to answer the research questions that were introduced in Section 1. The code to replicate the analysis is available on https://doi.org/10.5281/zenodo.5055500.

4.1 How outdated are dependent packages?

In previous work [21] we studied the adherence to semver in the four considered package distributions, by analysing the dependency constraints used by package releases in their respective dependency networks. We observed that the large majority of dependency constraints allow patch and minor updates to be installed automatically, and that this trend has increased over time.

While the above observation implies that most of the dependents rely on the highest patch and minor updates *within* a major train, this does not mean that they make use of the highest major train. Therefore we identified, for each dependent, the highest release of the required package that could be accepted by the dependency constraint. We verified if this release corresponds to the highest available release in the highest available major train.

Figure 3 shows the proportion of dependents that are up-to-date (i.e., they rely on the highest available release); that depend on a lower patch or a lower minor release within the highest major train; or that depend on a lower major train. While most dependents are up-to-date (ranging from 61.5% for *npm* to 81.8% for *RubyGems*), there remains a non-negligible proportion (ranging from 5.7% for *Cargo* to 27.7% for *npm*) of dependents that still rely on a lower major train. This is the most striking for *Packagist* and *npm*, with respectively 24.6% and 27.7% of their dependents relying on a lower major train. This can be an issue since discovered bugs and security issues are typically fixed in the highest major train. In order to allow dependents to benefit from those fixes, one would need to adopt a backporting strategy, which will be the focus of RQ_3 and RQ_4.

Focusing on those dependent packages that rely on some lower major train (corresponding to the red bars in Figure 3), we analysed the proportion of dependents in function of the number of major trains they missed. Table 2 reports on these proportions. Assuming that M is the highest available major train, in the large majority of cases (69.4% to 93.7%), dependents rely on releases belonging to the immediately preceding major train M-1. Still, a non-negligible proportion of packages (up to as many as 20% for *npm*) depend on releases that are two major trains behind (M-2). Dependencies are rarely three or more major trains behind (between 1.5%

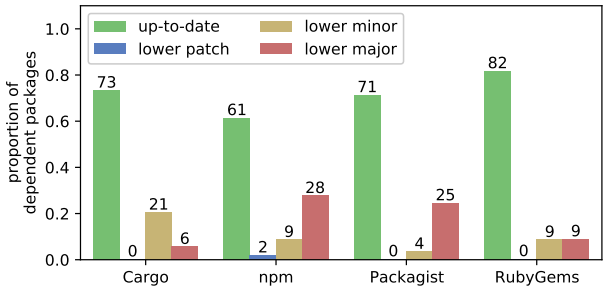

Fig. 3: Proportion of dependent packages relying on the highest available release, a lower patch, a lower minor or a lower major train.

for *Packagist* up to 10.6% for *npm*). The deviating behaviour for *npm* can be explained by the fact that many *npm* package producers have a tendency to create new major releases often. This might be due to a stricter adherence to semantic versioning [21].[3]

> A majority of dependent packages are up-to-date in the considered package distributions. Still, a non-negligible proportion depends on a lower major train. In *Packagist* and *npm* this happens in roughly 1 out of 4 cases. In *npm*, more than 3 out of 10 dependents on a lower major train are even several major trains behind.

4.2 How many major versions of a package are used by its dependents?

From RQ_1 we observed that many dependents still rely on a lower major train of the required package. In order to assess the potential and actual use of backporting practices, we first need to quantify, for each required package, how many major trains exist and how many of those are still used by its dependents. The more major releases a dependent is behind, the more likely that it will be unable to benefit from bug and security fixes, and the harder it will become to upgrade to the most recent major train. Conversely, from the required package point of view, the more of its dependents still rely on a lower major train, the more it becomes important to adopt a backporting strategy or to incentivise its dependents to upgrade to the highest major train.

Figure 4 shows the proportion of required packages in function of their number of available major trains, regardless of whether these major trains are actually used by some dependents. Except for *Cargo*, we observe that more than half of all required packages in each considered distribution have more than one major train, and between 31% and 36% even have 3 or more available major trains. In *Cargo*, the large majority of all required packages (85%) have a single major train, which can be explained by the fact that the overwhelming majority of *Cargo* packages (more than 9 out of 10) only have releases with major version 0 [26].

3. By adhering to semantic versioning, any backward incompatible change imposes the creation of a new major version.

TABLE 1: Dataset characteristics

initial dataset	*Cargo*	*npm*	*Packagist*	*RubyGems*	**total**
all packages	34,769	1,217,677	180,094	154,998	1,587,538
active packages	19,433	430,976	58,620	18,791	527,820
active packages with ≥ 1 dependent	6,248	78,395	13,949	3,313	101,905
active packages with ≥ 5 dependents	1,190	15,644	2,335	715	19,884
curated dataset	*Cargo*	*npm*	*Packagist*	*RubyGems*	**total**
active packages with ≥ 5 dependents	1,185	15,281	2,247	652	19,365
releases of these required packages	26,185	570,676	100,870	35,264	732,995
dependents of these required packages	13,143	242,618	35,488	7,915	299,164
dependencies for these dependent packages	59,814	1,021,843	89,667	15,066	1,186,390

TABLE 2: Proportion of outdated dependent packages relying on some lower major train. Notation M-n indicates how many major trains n the dependent is behind.

distribution	M-1	M-2	M-3	M-≥ 4
Cargo	93.7%	4%	1.1%	1.2%
npm	69.4%	20%	5.9%	4.7%
Packagist	89.1%	9.5%	1.3%	0.2%
RubyGems	82.4%	12.8%	3.5%	1.3%

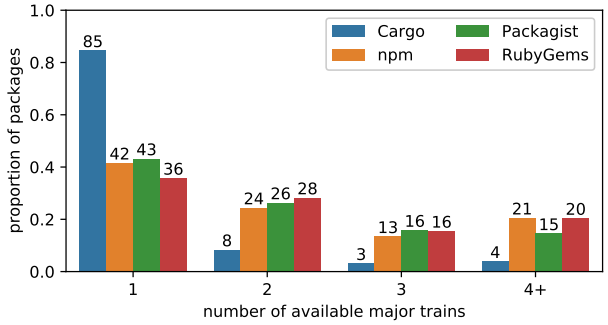

Fig. 4: Proportion of required packages in function of their number of major trains.

When restricting ourselves to only those major trains of required packages that are actually used by dependent packages, we obtain the proportions shown in Figure 5. To prevent the analysis to be biased by major trains used by a negligible fraction of the dependents only, we counted for each required package the minimum number of major trains needed to capture 90% of its dependents. We also limited the analysis to required packages having *at least two major trains* since dependent packages can only use a single major train if it is the only available one. We observe that between 37% (for *RubyGems*) and 51% (for *npm*) of the required packages with multiple major trains have more than one of these trains being used by most of their dependent packages.

> Except for *Cargo*, the majority of required packages in all package distributions have several major trains. Dependents frequently use releases (of required packages) that do not belong to the highest major train. This highlights the need for producers of required packages to support earlier major trains, since they continue to remain actively used.

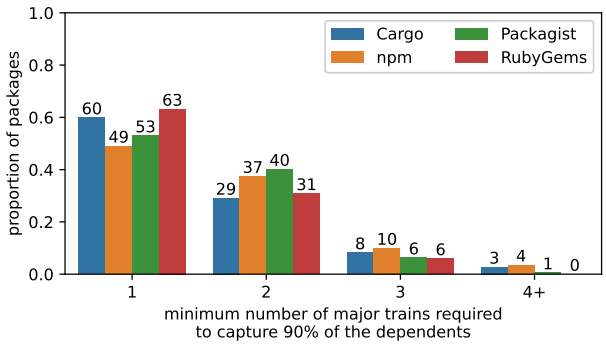

Fig. 5: Proportion of required packages (having at least two major trains) in function of the minimum number of major trains required to capture 90% of their dependent packages.

4.3 How prevalent are backports?

Given the need to support dependents on lower major trains (cf. RQ_2), RQ_3 aims to analyse the prevalence of **backports** (as defined in Section 3.1), by computing as a proxy *any new release that is applied to a lower major train*. This proxy quantifies the extent to which package producers are still actively maintaining lower major trains, and it can be seen as an upper bound on the actual number of backports one could expect to find.[4] To compute this we compared the chronological order of releases with the ordering induced by their version numbering. To illustrate the idea, reconsider the example of Figure 2: release 1.3.1 is a patch of 1.3.0. Since it was released after patch release 2.1.1 belonging to a higher major train, we consider 1.3.1 to be a backport.

TABLE 3: Backports and packages with backports.

distribution	**backports**	**packages**	
Cargo	149	21	1.7%
npm	6,425	1,010	6.6%
Packagist	10,886	457	20.3%
RubyGems	2,036	83	12.7%
total	19,496	1,571	8.1%

Table 3 reports on the number and proportion of packages with backports, and the number of such backports in the considered snapshot of the four package distributions. Overall, we found nearly 20K backports in around 1.5K distinct packages. We observe that only a small proportion of packages in *Cargo* (1.7%) and, to a lesser extent, in *npm* (6.6%) have backports. On the other hand, one out of five

4. The threats to construct validity of using this proxy are discussed in Section 5.

packages in *Packagist* have backports. These proportions are considerably lower than the proportions of developers that agree that time is being spent on backporting changes, as shown in Figure 1. Relative to the number of packages that have the opportunity for backporting (i.e., those with at least two major trains), the proportions of packages with backports are 11.5%, 11.4%, 35.9% and 20%, respectively for *Cargo*, *npm*, *Packagist* and *RubyGems*. This shows that lower major trains continue to remain maintained in all four distributions, but only for a minority of packages. RQ_4 will focus on the characteristics of those packages.

Compared to the three other package distributions, *Packagist* has a high number of backports, both proportionally and in absolute numbers. We discovered an important increase in the number of backports in early 2016. This was an immediate consequence of the release of PHP 7 in December 2015, which is noted for its backward incompatibility. It forced popular packages to maintain two major release trains at the same time, in order to facilitate the migration process towards PHP 7, without forcing dependents to move to PHP 7 from one day to another.

We observed in Table 2 (see RQ_2) that many required packages have multiple major trains being used by their dependents. If a required package backports some of its updates, the question arises which of these lower trains are targeted by the backport. Is it only the immediately preceding one, or will multiple lower trains continue to be maintained? Consider for example Figure 2, where package updates in the highest major train 3 have the potential to be backported into lower major trains 1 and 2. This is indeed the case for release 3.1.1, which is effectively backported to releases 1.3.2 and 2.2.1, respectively.

We quantified this phenomenon by identifying, for each required package that backported some of its updates, how many major trains were available, and which major train(s) received the backport. Figure 6 shows the proportion of required packages with backports, in function of the number of available major trains and the number of major trains that actually received a backport.

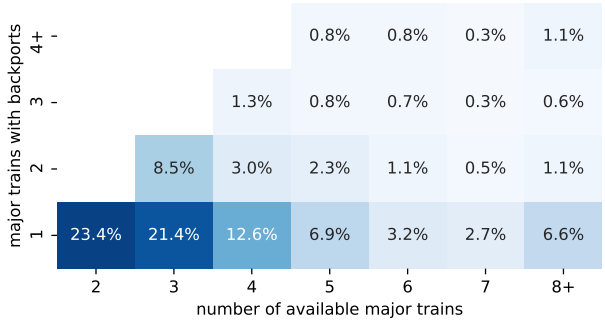

Fig. 6: Proportion of packages with backports in function of the number of available major trains and the number of major trains that received a backport.

We observe that most packages are backported to a single major train, even if multiple lower trains are available. The next question that naturally arises is which major train tends to be targeted by backports. Intuitively, we would expect this to be the major train that immediately precedes the highest available one. We could confirm this intuition, since we found that 86% of all backports target the previous major train, and 11.9% of all backports target the major train before that one.

Incidentally, it is interesting to note that major train 0 still receives 4.2% of all backports, even though packages with major version 0 are meant to be for initial development according to the semver policy. This phenomenon can be attributed almost exclusively to *Cargo*, which has many more `0.y.z` releases than any of the other studied distributions [26]. As a consequence 83.9% of all backports in *Cargo* are actually from major train 1 to major train 0 (compared to only 8.3% in *npm*, 1.2% in *Packagist* and 1.5% in *RubyGems*).

Even if a required package practices backporting, there is no guarantee that its dependents actually benefit from these backports. While it is likely that they do, it could be the case that the dependency constraints declared by the dependents prevent them from accepting the backported patch or minor release (e.g., in the case of a strict constraint). We checked this for all dependents relying on a major train that received a backport. Fortunately a large majority of these dependent packages, ranging from 81.9% in *RubyGems* to 96.4% in *Cargo*, do accept the backports to be installed.

> Backporting is infrequently practised. Only a minority of package producers continues to maintain lower major trains. Those that do, tend to maintain only the immediately preceding major train. The large majority of dependents that can potentially benefit from backporting actually do. *Packagist* has considerably more backports than the other studied distributions. The backward incompatible release of PHP 7 seems to have played an important role in this observed difference. *Cargo* has the specificity of backporting updates to major train 0.

4.4 Do packages with backports exhibit different characteristics?

RQ_3 revealed that backporting is practised by only a minority of packages. RQ_4 aims to assess if those packages exhibit characteristics that differ from the other packages in each considered distribution. We expect backporting to be mainly practised by more mature, active and popular packages (i.e., used by more dependents).

As a first characteristic we considered the *package age*, computed as the number of days between its first and latest available releases. The age of a package might be correlated with the practice of backporting, simply because longer-lived packages had more time to seize the opportunity to backport. Figure 7 shows the distributions of package age, grouped by package distribution, and distinguishing between packages with and without backports. We observe that packages with backports tend to be longer-lived. This observation is statistically confirmed ($p < 0.01$) by Mann-Whitney-U tests [27] after controlling for family-wise error rate with the Bonferroni-Holm method [28]. Following the interpretation by Romano et al. [29], the effect size (mea-

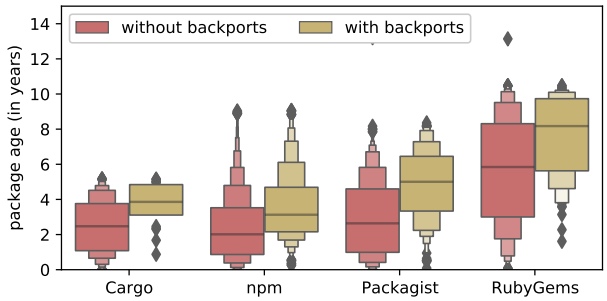

Fig. 7: Distribution of *package age* for packages with and without backports.

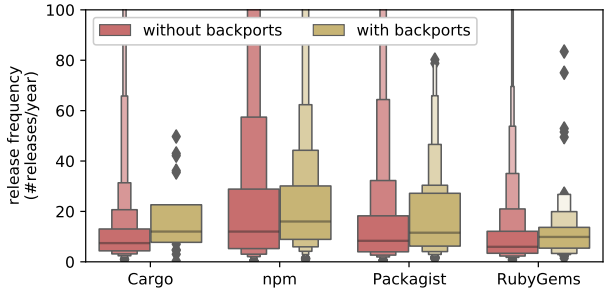

Fig. 8: Distribution of *package release frequency* for packages with and without backports.

sured using Cliff's delta [30]) is *medium* ($0.366 \leq |d| \leq 0.466$), and even *large* for *Packagist* ($|d| = 0.487$).

The *number of releases* of a package is a related characteristic for which we expect to see a difference, as it provides another means to increase the number of opportunities to practice backporting. One might intuitively expect the age and number of releases of a package to be correlated. To verify this intuition, we computed Pearson's r and Spearman's ρ correlation coefficients between these two characteristics. We obtained $r = 0.18$ indicating there is no linear correlation, and $\rho = 0.41$ indicating a moderate monotonic correlation. This observation implies that, in general, longer-lived packages do not necessarily release more versions. For this metric, we could also statistically confirm by Mann-Whitney-U tests, that packages with backports tend to have a higher number of releases. The effect size was *large* for all package distributions ($0.547 \leq |d| \leq 0.659$).

We also hypothesised that packages with backports have a higher activity rate, measured as *package release frequency* (i.e., the ratio of number of releases against package age). To prevent packages from having their release frequency "artificially" increased by the presence of backports, we counted a backport and the corresponding backported release as a single release. Figure 8 shows the distributions of package release frequency, and Mann-Whitney-U tests indeed confirmed that this frequency is statistically higher for packages with backports. The effect size is *small* ($0.150 \leq |d| \leq 0.280$).

As another distinguishing characteristic, we posit that packages with more dependents are more likely to practice backporting since backports benefit a larger audience. Conversely, packages with backports are more likely to have more dependents since depending on them allows to benefit from bug or security fixes even in lower major trains. Figure 9 shows the distributions of *number of dependents* of a package, grouped by package distribution, and distinguishing between packages with and without backports. We visually observe that packages with backports have a higher number of dependents. This observation is statistically confirmed by Mann-Whitney-U tests. The effect size is *small* for all package dependency networks ($0.182 \leq |d| \leq 0.328$).

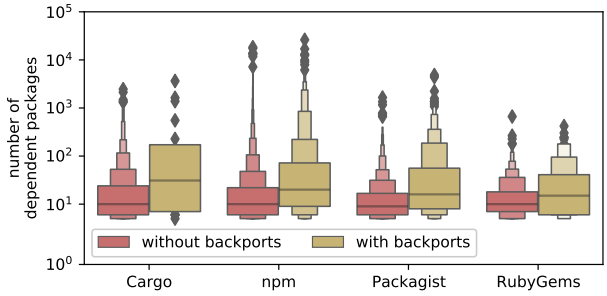

Fig. 9: Distribution of *number of dependents* for packages with and without backports.

> The characteristics of packages with backports are different from those without backports. In particular, packages with backports tend to have a longer lifetime, more releases, a higher release frequency and more dependents.

4.5 How long do lower major releases continue to be supported?

In an ideal world, whenever a new major release is created, all dependents would upgrade immediately to this major release, so that only the highest major train needs to be maintained. In practice, however, many dependents continue to rely on releases from a lower major train for a variety of reasons. Because of this, package producers may continue to support lower major trains by backporting updates to them.

We posit that backporting extends the lifetime of lower major trains, where we define the *lifetime* of a major train as the time between its first and last chronological release. For example, in Figure 2 the lifetime of major train 1 would be the time difference between the release date of 1.3.2 and 1.0.0. By backporting release updates to a lower major train, it continues to be maintained over extended periods of time. However, that does not automatically imply that major trains with backports have a longer lifetime than the ones without.

Figure 10 shows the distributions of the lifetime of lower major trains, distinguishing between those major trains that received backports and those that did not. Since we found in Section 4.4 that packages without backport exhibit a different release frequency than packages with, and to avoid the analysis to be biased by their different release frequencies, only major trains of packages that backported at least one

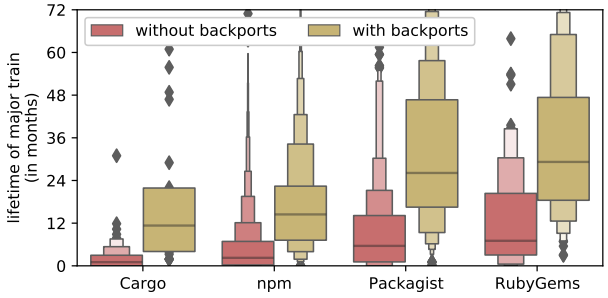

Fig. 10: Distributions of the lifetime of lower major trains, distinguishing between those that receive backports and those that do not.

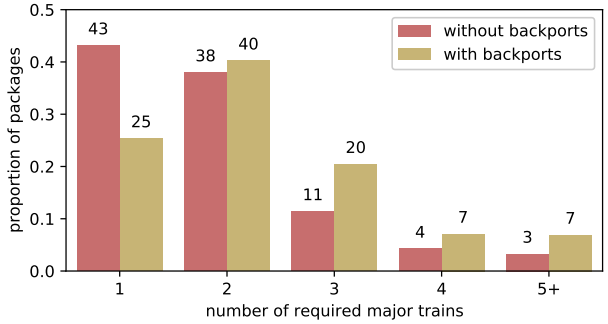

Fig. 11: Proportion of packages in function of the number of major trains used by dependents, distinguishing between packages with and without backports.

update are considered for the analysis. [5]

We observe that lower major trains that received backports indeed have a longer lifetime. This observation was statistically confirmed by Mann-Whitney-U tests, with a *large* effect size ($0.660 \leq |d| \leq 0.783$) for all package distributions. Comparing the median values in Figure 10 we observe that major trains that receive backports have their lifetime extended by a factor 11 for *Cargo* (11.3 vs 1 month), a factor 6.4 for *npm* (14.5 vs 2.3 months), a factor 4.7 for *Packagist* (26.1 vs 5.6 months), and a factor 4.2 for *RubyGems* (29.2 vs 7 months).

Thanks to this extended maintenance support of previous major trains, dependents can continue to rely on those trains and still benefit from the latest bug and security fixes. As a consequence, we expect packages with backports to have more major trains in use by their dependents.

Figure 11 shows the proportion of packages in function of the number of major trains used by the population of their dependents, distinguishing between packages with and without backports. To make a fair comparison, required packages having only one major train were excluded from this analysis. The figure indeed reveals that packages with backports have a higher number of major trains used by their dependents: 75% of packages with backports have two major trains being used by their dependents while this proportion is only 57% for packages without backports. This observation was again statistically confirmed using Mann-Whitney-U tests. The effect was *small* for *npm* and *RubyGems* ($0.226 \leq |d| \leq 0.237$), and *medium* for *Cargo* and *Packagist* ($0.340 \leq |d| \leq 0.428$).

> Major trains that receive backports are maintained for longer periods of time (ranging from a factor 4.2 for *RubyGems* to as high as a factor 11 for *Cargo*). Packages with backports have more major trains used by their dependents.

4.6 To which extent are security fixes being backported?

One of the benefits of doing backports is to fix as early as possible security vulnerabilities that can be exploited to abuse systems and environments in which these systems are deployed. Indeed, dependents relying on a lower major train can benefit from these security fixes through backports, without having to go through the potentially time-consuming adoption of a newer major release.

RQ_6 therefore aims to identify the extent to which security vulnerabilities affect several major trains of the packages in which they occur, the extent to which lower major trains are receiving security fixes, and the extent to which dependents can benefit from these fixes. To motivate this question, consider two concrete vulnerabilities[6] in two very popular *npm* packages, bootstrap and lodash:

1) A medium severity cross-site scripting vulnerability in bootstrap affects versions `<3.4.0 || >=4.0.0 <4.1.2`. The vulnerability was fixed in patch update `4.1.2` and the fix was cherry-picked and backported into minor release `3.4.0` belonging to major train 3. In the considered snapshot, bootstrap is used by 2,384 dependents, of which 372 rely on major train 3 to which the vulnerability fix was backported. As a consequence, even outdated dependents benefitted from the fix by allowing backward compatible minor updates for their affected dependencies, without the need to upgrade to the new, backward incompatible, major version 4.
2) A high severity vulnerability of type prototype pollution was reported for lodash. The vulnerability was fixed in patch `4.17.12` and was reported to affect all previous releases, hence leaving the four lower major trains vulnerable.[7] In the considered snapshot, lodash is used by 26,073 dependents, and 401 of them still rely on lower major trains that do not have any fix for the vulnerability. Those outdated dependents would need to upgrade their dependencies to a new major version, and are likely to encounter backward incompatible changes that may require significant effort to deal with. For instance, the changelog of version 4.0.0 lists over 50 incompatible changes.

5. When redoing the same analysis by also including packages with only one major train, we reached similar conclusions.

6. See https://snyk.io/vuln/SNYK-JS-BOOTSTRAP-11109 and https://snyk.io/vuln/SNYK-JS-LODASH-450202

7. The vulnerability database does not report a lower bound on the affected versions, so this is likely to be an over-approximation to remain on the safe side.

To study RQ_6 we rely on a dataset of vulnerability reports kindly provided by Snyk.io on 12 April 2020.[8] Since the provided dataset does not contain vulnerability reports for *Cargo* and *Packagist*, the analysis for RQ_6 necessarily restricts itself to *npm* and *RubyGems* only. This dataset contains 2,874 vulnerability reports in total (of which 2,188 for *npm* and 686 for *RubyGems*) affecting 2,034 distinct packages (1,700 from *npm* and 334 from *RubyGems*). Each vulnerability report contains information about the affected package, the releases being affected by the vulnerability, and the releases in which the vulnerability was fixed.

Since the focus is on backporting security fixes, we restrict the list of vulnerabilities to those having been fixed. This results in 834 vulnerabilities with known fixes (541 for *npm* and 293 for *RubyGems*) affecting 383 distinct packages (292 for *npm* and 91 for *RubyGems*). All of these packages have dependents that can potentially be affected by the vulnerability. This concerns 108,008 dependents for *npm* (out of the 242,618 dependent *npm* packages in our curated dataset) and 7,234 dependents for *RubyGems* (out of the 7,915 dependent *RubyGems* packages in our curated dataset).

A given vulnerability may affect one or several major trains. When a vulnerability affects the highest major train, it can be fixed either by releasing a new patch or a minor release, or by releasing a new major version including the fix (i.e., starting a new major train). When a vulnerability also affects lower major trains, the producers of the affected package can decide to backport the fix to these lower major trains as well.

Based on the information provided in the vulnerability reports, we counted how many major trains were affected by the vulnerability, and how many major trains benefited from a security fix. Figure 12 shows the proportion of vulnerabilities in function of the number of affected major trains and the number of fixed major trains.

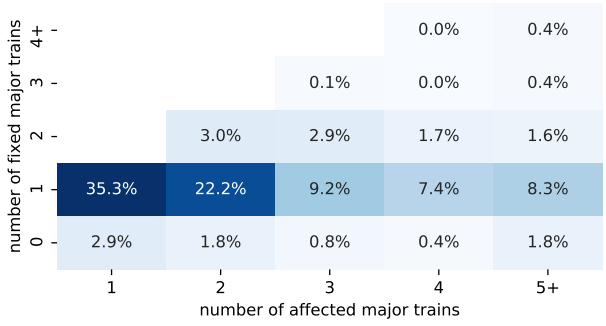

Fig. 12: Proportion of vulnerabilities in function of the number of affected and number of fixed major trains.

We observe that 318 vulnerabilities (213 for *npm* and 105 for *RubyGems*) affect *a single major train only*. This corresponds to 38.2% of all vulnerabilities, of which 35.3% have a fix available. The large majority of those fixes (90.9% of them, accounting for 289 vulnerabilities) were released as a patch or a minor release in the current major train. The remaining 29 vulnerabilities were fixed by creating a new major release (e.g., a vulnerability affecting 1.9.0 that is fixed in 2.0.0). Amongst them, only 5 fixes were backported to the previous affected major train while in the remaining 24 cases, the previous major train did not benefit from a fix, leading their number of fixed major trains to be zero.

Let us now focus on the remaining 516 vulnerabilities (61.8% of which 328 for *npm* an 188 for *RubyGems*) that affect *more than one major train*. Surprisingly, 83.9% of them (427) were fixed in only one major train, leaving the other affected major trains vulnerable. On average, three major trains were affected while the security fix was deployed in only one major train. All of these vulnerabilities were fixed in the highest major train.

Given that not all affected lower major trains actually benefit from a fix, it may be risky for dependents to rely on these major trains. To quantify this risk, we identified for each dependent whether it depends on a major train that was affected by one or more vulnerabilities, and whether it has benefited from a security fix applied to that major train.

As we want to understand the practice and effect of backporting security fixes, we only consider dependents relying on one of the 261 affected lower major trains (i.e., we excluded dependents relying on the highest major train). We found 6,274 of such dependents out of a total of 97,402 (i.e., 6.4%). These dependents have a total of 7,219 dependencies on one of the 150 packages affected by a vulnerability. They rely on a total of 438 affected major trains (if a major train is affected by multiple vulnerabilities, it is counted more then once, explaining why this number exceeds 261), and most of these major trains (84%) did not benefit from a backported fix. As a consequence, only 2,023 dependents (32.4%) rely on a lower major train that benefited from at least one backported fix, while 5,056 dependents (80.6%) are still exposed to the vulnerability and would benefit from a such a backport.[9]

> A majority of the security vulnerabilities for *npm* and *RubyGems* packages affect more than one major train but are only fixed in the highest major train, even if the lower trains are also marked as vulnerable. This is worrisome, because thousands of dependents still rely on a lower major train that is known to be exposed to a vulnerability even when a fix is available in a higher major version.
>
> This calls for an increased awareness of the need for backporting in package dependency networks, especially for producers of popular packages, whose vulnerabilities may potentially affect thousands of dependents.

5 Threats to Validity

We follow the structure recommended by Wohlin et al. [31] to discuss the main threats to validity of our research, and we provide some suggestions for future work based on this discussion.

Threats to *construct validity* concern the relation between the theory behind the experiment and the observed

8. The dataset being subject to a *non-disclosure agreement*, it is not included in the replication package.

9. The two aforementioned percentages add up to more than 100% since the same dependent can be exposed to multiple vulnerabilities.

findings. They can be mainly due to imprecisions in the measurements we performed. We relied on the libraries.io dataset of software library dependencies [24] and a dataset of vulnerabilities provided by Snyk.io. To convince us of the correctness of these datasets, we manually inspected random samples and we cross-checked the data with other metadata we used in previous research [32], [2], [21]. We found and excluded 305 packages with a *dirty* release history, i.e., packages for which the chronological order of their major trains is inconsistent. This is generally a consequence of a large-scale import of their releases within the package distribution, leading the release dates to correspond to the date of the import and not to the actual release date.

Another source of imprecision stems from our proxified definition of backports. As explained in Section 4.3, we relied on a comparison between the chronological order of the releases and their version number order to identify backports. We may have missed some backports in a few edge cases. For example, when a 1.6.1 version actually contains backported changes from 2.0.0 but nevertheless is released a few minutes before 2.0.0. However, we are confident of our approach since it was able to capture all backported security fixes reported in Section 4.6. The proxy for detecting backports may also lead to false positives. For example, a new release deployed in a lower major train is considered as a backport even if it does not actually contain changes backported from a higher major train. A detailed comparison of the changes made in each release would enable confirming whether they are actual backports or not. We tried to do so by studying the changelog files (e.g., CHANGELOG.md) of all packages for which we identified backports. Only 575 out of the 1,571 identified packages actually had a changelog file available in their GitHub repository. Moreover, only 139 of these 575 files were sufficiently complete, in the sense that they explicitly reported changes related to the specific releases that we identified as backports. The reason for this is that most changelog files list only the changes for major or (to a lesser extent) minor releases, or for releases belonging to the highest available major train. They tend to ignore patch releases in lower major trains, that represent 91.8% of the backports we identified using our proxy. A manual inspection of the 139 useful changelog files allowed us to confirm the presence of backports in 125 out of 139 cases (i.e., 89.9%)[10]. It is not possible to know whether these results generalise to the set of all packages.

There are also construct validity threats related to package dependencies. For example, packages may contain *bloated dependencies* that are declared but not really needed when executing the application. As such, the potential impact and need for backporting can be lower in practice. Soto-Valero et al. [33], [34] studied the phenomenon of bloated dependencies for the *Maven Central distribution* of *Java* packages. In a similar vein, not all vulnerabilities necessarily affect all dependents of the vulnerable package. If the vulnerable functionality of the required package is not used by the dependent, the latter will not be affected. Still, it may be wise to adopt vulnerability fixes nevertheless, since it is very difficult and computationally intensive to analyse when a vulnerability is harmless to dependents.

Threats to *internal validity* concern choices and factors internal to the study that could influence the observations we made. One such choice was our decision to study only *active packages* that have been updated in the last 12 months preceding the considered snapshot of the package distributions. This choice was motivated by earlier work in which we found that more than 90% of all package updates (for required packages) are made within 12 months after a previous update [2].

Another choice that might have influenced our observations was the decision to consider only packages that have *at least 5 direct dependents*. We do not think this has affected our findings much, as 5 is a rather low number, and it is reasonable to believe that package producers will not bother spending effort to backport release updates if there are too few packages relying on it. A possible threat is that we attributed the same weight to all dependents, considering all of them as being equally important. This is not necessarily the case, since some direct dependents may themselves be used by many direct and indirect dependents. In future work, we will consider the effect of transitive dependents on the decisions of package producers related to backporting.

In RQ5, we compared the major trains from potentially heterogeneous packages. As packages have different characteristics, a paired comparison (i.e., per package) between major trains with and without backports would be more appropriate. However, the number of major releases per package is usually small, and such a paired analysis would not yield any statically significant result. For instance, only 125 out of 1,570 packages with backports have at least 2 major trains with a backport and at least 2 major trains without. We actually conducted such a comparison on this small subset of packages. Despite the too small size of this comparison to be statistically relevant, the findings were in accordance with the results reported in the paper based on the alternative analysis.

Threats to *conclusion validity* concern the degree to which the conclusions derived from our analysis are reasonable. Since our conclusions are mostly based on quantitative observations, and supported by statistical tests with high confidence (even after correcting for family-wise errors), they are unlikely to be affected by such threats.

The threats to *external validity* concern whether the results can be generalized outside the scope of this study. While our analyses can be applied to other software package distributions, those distributions should also adhere to semantic versioning practices in order to be able to compare their results with the package distributions that we considered. We already observed some differences between the considered package distributions, and we expect to see more of such differences in other package distributions. For example, backporting practices would be totally absent in the official *CRAN* distribution of *R* packages, since *CRAN* adopts a *rolling release* policy, requiring all dependents to rely on the highest releases of the packages they depend upon.[11] Earlier work has shown that this makes the CRAN package dependency network an outlier compared to other

10. The Appendix provides some isolated concrete examples of what we found in changelog files

11. https://cran.r-project.org/web/packages/policies.html

package distributions [2].
Since our dataset only considers packages and their dependents within a package distribution, we cannot generalise the findings to dependents that live outside of the package distribution. In particular, we did not study the effect of backporting on external projects (e.g., repositories on GitHub) depending on the considered packages. Studying this effect would constitute an interesting topic as future work.
A final external validity threat is that we only considered backporting from major trains to lower major trains because, if semantic versioning is being practised, dependents using semver-compliant dependency constraints automatically benefit from bug fixes and security patches within a major train. As such, there is no need to backport updates from minor trains to lower minor trains.

6 Discussion

6.1 Continued maintenance of lower major trains

Producers of required packages that have many dependents face a difficult choice whenever they create a new major release. In order to urge dependents to move to the highest major release, they could decide to stop maintaining lower major trains. This might induce problems for those dependents for which the effort to upgrade to the new major release is too high (e.g., because of backward incompatibilities, lack of manpower, or the need to carefully plan and test major upgrades). This may explain the observation in RQ_1 that many dependents still depend on a lower major train. The alternative is that package producers continue to support lower major trains, in order to allow outdated dependents to continue to benefit from bug fixes, security patches and possibly other relevant backported changes.

Continued support of lower major trains might have the adverse effect of delaying the migration of dependents to the highest major release: they might prefer to stick to lower major trains because it requires less effort from their part. This is just a respite, however, since lower major trains are unlikely to be maintained forever even when they benefit from backports, as shown in RQ_5. For these reasons, package producers should devise and advertise a clear phase-out strategy of lower major trains, to avoid giving dependents a false sense of security. For example, they could decide to continue supporting lower major releases for a fixed amount of time, after which the dependents will be left on their own. The advantage of such a delay is to give dependents the necessary time to upgrade, without running an increased security risk during this transition period. Publishing prereleases and release candidates of an upcoming new major release may also help dependents by giving them time to prepare the upgrade to the next major release before it is officially published. This allows them to adopt the new major release as soon as it is officially published, rather than having to start integrating changes only from that point onwards.

RQ_2 highlighted the need to support earlier major trains, since they continue to remain actively used. This implies that package producers also need to decide whether they should continue to support multiple lower major trains or only the previous one. The more major trains that require backports, the more effort is needed from package producers. Especially if a *rapid release* policy is being followed, with new major versions being released frequently, one cannot reasonably assume all dependents to keep up with this rapid pace. In that case, package producers could opt for distinguishing between *short-term support* (STS) and *long-term support* (LTS) major releases. The latter are guaranteed to be supported by bug fixes and security patches over extended periods of time. An example is Angular's support policy.[12] Another strategy is used by Electron, allowing so-called stabilization branches of major version trains to exist simultaneously.[13] At least two of such branches continue to receive backports of security updates and cherry-picked bug fixes as necessary.

6.2 Tool support

Tools that help in creating backports can reduce the burden for *producers of required packages*. An example is the backport tool on *npm* that automates the process of backporting commits on a GitHub repository. Backporting tools like FixMorph [13] (that was proposed for transferring patches in the Linux kernel) and other automated patch transplantation techniques [16] could be adapted for backporting patches of bugs and vulnerabilities in package dependency networks as well.

Producers of dependent packages can also benefit from tools that automate some or all of the changes required to adopt a new incompatible version. Examples of such tools are ng update for angular and lodash-migrate. Package producers may also provide migration guides to make it easier for dependents to upgrade to newer major trains. These guides provide a detailed list of incompatible changes and instructions on how to incorporate them in a dependent package. Some examples of migration guides are https://expressjs.com/en/guide/migrating-4 and https://vuejs.org/v2/guide/migration.

At the level of the package distribution itself, tools and dashboards could be provided to highlight which major trains of distributed packages are still in active use, and by how many dependents. This will help package producers to make more informed choices about which major trains to support. npm-deprecate could be considered as a lightweight example of such a tool. Its purpose is to signal the deprecation of a particular version (or range of versions) of a package by providing warnings to anyone who attempts to install this version. Such deprecations can be used to communicate to dependents for which major trains no new releases should be expected.

Distribution-level tools should also integrate information about security vulnerabilities in distributed packages, for example to highlight how many and which outdated dependents are still affected by vulnerabilities for which fixes are available in higher major trains. From the point of view of dependents, automated dependency analysis tools (such as Dependabot) could be extended, for example by

12. https://angular.io/guide/releases#support-policy-and-schedule
13. https://www.electronjs.org/docs/tutorial/electron-versioning#stabilization-branches

warning dependents that the period of support of a lower major train is nearing its end.

The package distributions could also provide concrete guidelines and policies for package producers willing to backport updates to lower major trains. This is quite common in Linux distributions such as RedHat[14], Ubuntu[15] and Debian[16]. For example, Debian maintains packages for several simultaneous release lines. The most important ones are Testing, Stable and Oldstable. In Testing, packages are updated frequently, so that users benefit from new functionality but are also expected to report potential bugs. Stable and Oldstable, in contrast, include only the most important fixes and security updates. These fixes received in Stable and Oldstable are in fact backports.

6.3 Differences between package distributions

While the decision to backport lies in the hands of a package producer, there may be community values and expectations from the package distribution to be considered. For example, some package distributions are known to practice *semantic versioning* more strictly than others [21]. In presence of semver, backporting may be more relevant, since new major releases are known to be backwards incompatible, making it harder for dependents to upgrade to the highest major train. Yet, even in absence of semver, it is still useful to backport bug and vulnerability fixes to lower release trains when higher release trains are known to be incompatible.

During our analysis for RQ_3 we observed an important effect of backward incompatibilities on backporting practices in the *Packagist* distribution of *PHP* packages. This was caused by the introduction of *PHP* 7, a notoriously backward incompatible[17] successor of *PHP* 5. It caused many packages to start maintaining two parallel branches: one for *PHP* 5 (so that dependents are not forced to migrate to *PHP* 7 to continue to use the package), and one for *PHP* 7 (to allow dependents that already migrated to that version to use the package). Incidentally, the same kind of issue also arose for the *Python* language during the transition from *Python* 2 to *Python* 3. However, to facilitate the migration, a compatibility layer for *Python* 2 was published, supporting some of the incompatible changes of *Python* 3 to allow package producers to release new versions that worked on both *Python* 2 and *Python* 3, obviating the need to maintain two parallel branches. Similarly, the Symfony Polyfill project for *PHP* enables backporting features of the highest *PHP* versions to increase portability across *PHP* versions.

Another factor may be the extent to which dependents use strict dependency constraints. By doing so, they may prevent automatic installation of new patches and minor upgrades, regardless of the major train they are depending on. While RQ_1 revealed that a majority of dependents rely on the highest available major train, we also observed that not all of them are relying on (and thus benefit from) the highest minor or patch releases within the highest major train. If the community of a package distribution has the habit of using strict constraints, then backporting becomes less useful and hence less practised, as there are less packages that can benefit automatically from backported updates. It may also result in a higher risk of unfixed security vulnerabilities in such package distributions.

For some of the analyses reported in this article, we observed that the *Cargo* package distribution behaved differently from the other considered package distributions. This was caused by a very high proportion of *Cargo* packages that continue to stay in the major 0 version space [26]. As such, only a minority of packages can actually practice backporting (from one major release to a lower one). This may explain why RQ_3 reported a lower proportion of packages with backports in *Cargo*, and why most backports in *Cargo* are actually to major train 0. It is possible that *Cargo* packages practice backporting at the level of minor releases (i.e., patches to minor releases are backported to previous minor releases). We did not study this phenomenon in the context of this article because it goes against the semantic versioning policy, but it is an interesting topic to study in future work. Indeed, we found anecdotal evidence of such minor backporting practices being applied occasionally in other package distributions as well.[18]

6.4 On the *why* and *how* of backports

This article is the first in its kind to empirically study backporting in package dependency networks. The presented results were essentially quantitative in nature, focusing on a coarse-grained analysis based on the release metadata and dependency metadata of the considered packages. In order to gain a better insight into *why* and *how* backports are being used, one would need to delve deeper along two different dimensions.

Concerning the *why* question, the reported quantitative insights need to be complemented by qualitative analyses in order to understand the rationale behind why package producers create backports, which types of changes are being backported, and which major release trains are targeted by these backports. Surveys and in-depth interviews with package producers that practice backporting, as well as producers of outdated dependents, will allow us to better understand the internal and external factors that influence their decisions. Once we have gained more qualitative evidence we could exploit it to come up with prediction models for backporting practices, which could be useful for both required and dependent package producers.

Concerning the *how* question, in addition to the aforementioned qualitative insights, more fine-grained quantitative analyses should be conducted. These analyses could involve studying the impact of backports on transitive dependents, but also the types of changes that are being backported. RQ_6 focused on a specific type of changes, namely security fixes, but other important changes such as bug fixes might be backported as well. Based on our manual inspection of those changelogs that actually report about backports we observed that backports tend to fix security issues and bugs that were discovered while working on a higher major train, but they are also used to resolve compatibility issues (e.g., a breaking change in a dependency)

14. https://access.redhat.com/security/updates/backporting
15. https://wiki.ubuntu.com/UbuntuBackports
16. https://backports.debian.org
17. https://www.php.net/manual/migration70.incompatible.php
18. For example: https://github.com/electron/electron/releases/tag/v1.7.16

and even to port relevant new functionality to a lower major train.

In this article we used a proxy that considers as backport any new release that is applied to a lower major train. A manual inspection of changelog files, for those packages that have such files, revealed that many releases (especially patch releases from lower major trains) do not even have their changes documented in the changelog file. Even when they do, changes corresponding to a backport might not be signaled explicitly as such. This makes it very difficult to carry out automated analyses, and calls for a more widespread, systematic and disciplined way of documenting changes between package releases. Managers of package distributions should encourage this, and could provide tools to facilitate the creation, verification and maintenance of changelogs of packages.

It would also be of interest to carry out fine-grained quantitative analyses of the actual source code changes to come to a more detailed assessment of the nature of backports. Doing so would require analysing the code commits stored in the packages' version control repositories. Such analysis would be quite challenging and time-consuming because package releases may have a high number of commits, because the history of git repositories can change over time [35], because it is not easy to map commits to package releases, and because it is unclear how to identify whether a given (set of) commit(s) actually corresponds to a backport.

7 Conclusion

This article focused on the phenomenon of backporting in package dependency networks. Changes applied to a package's major train (i.e., a sequence of releases with the same major version number) may be backported by the package producer to lower major trains. Keeping lower major trains stable and secure over extended periods of time is especially beneficial to the many dependent packages that lag behind.

We empirically studied the practice of backporting in the package dependency networks of four large package distributions: *Cargo*, *npm*, *Packagist* and *RubyGems*. We observed that a non-negligible proportion of dependent packages still depend on lower major trains. In *npm*, it is even common to lag multiple major trains behind. From the perspective of required packages, we observed that their dependents frequently rely on lower major trains, highlighting the need to continue maintaining those releases to avoid dependents becoming too vulnerable. Despite this need, only a minority of required packages continues to maintain such lower releases. If they do, most of the dependents relying on such releases benefit from the backporting. Those packages that do practice backporting tend to have a longer lifetime, and also tend to have more releases and more dependents. Major trains that receive backports continue to receive updates much longer than major trains that do not.

Manual inspection of changelogs revealed that backporting is practised for a wide variety of reasons: to fix security issues, to port bug fixes from a higher to a lower major train, to resolve backward compatibility issues, and even to port relevant new functionality to a lower major train. Focusing specifically on security vulnerabilities in the *npm* and *RubyGems* dependency networks, we found that a majority of them affect several major trains but are only fixed in a single one. We found thousands of dependents that still rely on a lower major train that is exposed to a vulnerability for which a fix is available in a higher major train. This calls for action, such as backporting those fixes so the dependents can adopt them automatically.

Packages producers and package distributions themselves should adopt explicit policies about which major trains will remain supported by backports and for how long. Automated tools and dashboards could help to raise awareness and to reduce the burden of backporting for producers of required packages. Similarly, tools and guides for upgrading can be beneficial for producers of dependent packages.

Acknowledgments

This research is supported by the Fonds de la Recherche Scientifique – FNRS under Grants number O.0157.18F-RG43 (Excellence of Science project *SECO-ASSIST*) and T.0017.18.

References

[1] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "When and how to make breaking changes: Policies and practices in 18 open source software ecosystems," *Trans. Softw. Eng. Methodol.*, vol. 30, no. 4, Jul. 2021. [Online]. Available: https://doi.org/10.1145/3447245

[2] 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.

[3] T. Lauinger, A. Chaabane, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web," in *ISOC Network and Distributed System Security Symposium*, February 2017.

[4] A. Decan, T. Mens, and E. Constantinou, "On the evolution of technical lag in the npm package dependency network," in *IEEE International Conference on Software Maintenance and Evolution*, September 2018.

[5] 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 *USENIX Security Symposium*, 2019, pp. 995–1010.

[6] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An empirical study of usages, updates and risks of third-party libraries in Java projects," in *2020 IEEE International Conference on Software Maintenance and Evolution*, 2020, pp. 35–45.

[7] A. Zerouali, J. M. González-Barahona, T. Mens, A. Decan, E. Constantinou, and G. Robles, "A formal framework for measuring technical lag in component repositories – and its application to npm," *Journal of Software: Evolution and Process*, 2019.

[8] 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, no. 3, p. 47, 2021.

[9] Y. Tian, J. Lawall, and D. Lo, "Identifying Linux bug fixing patches," in *International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 386–396.

[10] B. Ray, M. Kim, S. Person, and N. Rungta, "Detecting and characterizing semantic inconsistencies in ported code," in *International Conference on Automated Software Engineering (ASE)*, 2013, pp. 367–377.

[11] L. R. Rodriguez and J. Lawall, "Increasing automation in the backporting of Linux drivers using Coccinelle," in *European Dependable Computing Conference*, 2015, pp. 132–143.

[12] F. Thung, X.-B. D. Le, D. Lo, and J. Lawall, "Recommending code changes for automatic backporting of Linux device drivers," in *International Conference on Software Maintenance and Evolution*. IEEE, 2016, pp. 222–232.

[13] R. Shariffdeen, X. Gao, G. J. Duck, S. H. Tan, J. Lawall, and A. Roychoudhury, "Automated patch backporting in Linux (experience paper)," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. ACM, 2021, pp. 633–645. [Online]. Available: https://doi.org/10.1145/3460319.3464821

[14] C. Bogart, A. Filippova, C. Kästner, J. Herbsleb, and F. Thung. (2017) Values and practices in 18 software ecosystems. [Online]. Available: http://breakingapis.org/survey/

[15] L. Ren, "Automated patch porting across forked projects," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019, pp. 1199–1201.

[16] R. S. Shariffdeen, S. H. Tan, M. Gao, and A. Roychoudhury, "Automated patch transplantation," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 1, Dec. 2021.

[17] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *Int'l Conf. Software Engineering*. IEEE Press, 2015, pp. 109–118.

[18] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" in *International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 84–94.

[19] J. M. Gonzalez-Barahona, P. Sherwood, G. Robles, and D. Izquierdo, "Technical lag in software compilations: Measuring how outdated a software deployment is," in *IFIP International Conf. on Open Source Systems*, 2017, pp. 182–192.

[20] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the Maven repository," *Journal of Systems and Software*, vol. 129, pp. 140 – 158, 2017.

[21] A. Decan and T. Mens, "What do package dependencies tell us about semantic versioning?" *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1226–1240, June 2021.

[22] R. Opdebeeck, A. Zerouali, C. Velazquez Rodriguez, and C. De Roover, "Does infrastructure as code adhere to semantic versioning? an analysis of Ansible role evolution," in *International Working Conference on Source Code Analysis and Manipulation*, 2020, pp. 238–248.

[23] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an API: cost negotiation and community values in three software ecosystems," in *International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 109–120.

[24] J. Katz, "Libraries.io open source repository and dependency metadata (version 1.6.0)," 2020.

[25] A. Decan, "portion – python data structure and operations for intervals," 2018. [Online]. Available: https://github.com/AlexandreDecan/portion

[26] A. Decan and T. Mens, "Lost in zero space: An empirical comparison of 0.y.z releases in software package distributions," *Science of Computer Programming*, vol. 208, no. 1, August 2021.

[27] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Statist.*, vol. 18, no. 1, pp. 50–60, 03 1947.

[28] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65–70, 1979. [Online]. Available: http://www.jstor.org/stable/4615733

[29] 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 Cohen's d indices the most appropriate choices?" in *Annual Meeting of the Southern Association for Institutional Research*, 2006.

[30] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," *Psychological bulletin*, vol. 114, no. 3, p. 494, 1993.

[31] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[32] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *International Conference on Mining Software Repositories*, May 2018.

[33] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the Maven ecosystem," *Empirical Software Engineering*, vol. 26, no. 3, 2021.

[34] C. Soto-Valero, T. Durieux, and B. Baudry, "A longitudinal analysis of bloated java dependencies," in *Int'l Conf. ESEC/FSE*. ACM, 2021.

[35] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. Germán, and P. T. Devanbu, "The promises and perils of mining Git," in *Int'l Conf. Mining Software Repositories*, 2009, pp. 1–10.