A Study of Backporting in Npm, PyPi, and Maven Ecosystems

Elmira Adeeb
Department of Computer Science
Carleton University
Ottawa, Canada
ElmiraAdeeb@cmail.carleton.ca

Samin Azhan
Department of Computer Science
Carleton University
Ottawa, Canada
SaminAzhan@cmail.carleton.ca

***Abstract*—The goal of backporting is to import the benefits of bug or security patches from a higher release of a package to a lower release of the same package. Backporting is a useful practice because it allows packages with outdated dependencies to take advantage of new features and fixes. This fact is especially true in cases where upgrading dependencies is expensive in terms of time and resources, leads to breaking changes, or it is not an option due to backward incompatibility issues. In spite of its advantages, the phenomenon of backporting is still quite unexploited. In the most comprehensive study to date, Decan *et al.* investigated the practice of backporting in *Cargo*, *npm*, *Packagist* and *RubyGems*. Their results suggest that backporting updates are quite rare and are mainly practiced by more mature and active packages. Motivated by the benefits of backporting, in this work, we extend the work of Decan *et al.* and carry out an empirical comparative study to understand the impact of the backporting practice within the *npm*, *PyPi* and *Maven* ecosystems. Our results are in line with those of the original work as far as the lifetime, the release frequency, and the number of dependents of packages with backports are concerned. Our findings also provide unique insights in regards to the role of backporting within the *PyPi* and *Maven* ecosystems.**

***Index Terms*—Software Ecosystems, Software Engineering, Version Control, Reusable software, Reusable libraries, Backporting, npm, PyPi, Maven**

I. Introduction

Relying on existing packages is a necessity of modern software development [1]. Packages in software ecosystems continually evolve to add new features, fix bugs, and apply security patches. To benefit from these improvements and to avoid technical lag, dependent packages must stay up to date with the packages they depend on [2] [3]. Consequently, the effective management of dependencies is critical for re-using existing packages. However, keeping packages up-to-date requires a significant amount of time and effort, especially in cases where changes are backward incompatible [1]. Furthermore, updating packages can lead to breaking changes which result in compilation-time or run-time errors. These errors are in particular a burden on dependent packages since they create a sudden urgency for these packages to fix issues without a genuine cause and can ripple through the entire software ecosystem [4] [5].

Based on the aforementioned elaboration, it is easy to apprehend why it is common for packages to have outdated dependencies [6]. Despite its convenience, relying on outdated packages not only raises serious security concerns, but also makes future upgrades cumbersome to handle [7] [8] [9] [10] [11]. This is when the study of backporting becomes a relevant topic.

Backporting is the process of applying a software update, *i.e.*, a bug or a security fixing patch from a newer release of a software to its lower version [6]. Even though the application of this strategy requires additional effort from the package maintainer side, it provides the dependent packages with an opportunity to benefit from backported fixes without having to upgrade to a higher major or minor version of a package they rely on. Backporting is especially useful in cases where upgrading is not feasible due to known version incompatibilities, security risks, or because it entails too much effort from the developer [6].

In spite of its evident benefits, the practice of backporting has not received adequate research attention in the field of software ecosystems. As a matter of fact, Decan *et al.* [6] are the first to have carried out an extensive investigation to study backporting practices in four ecosystems, namely, *Cargo*, *npm*, *Packagist* and *RubyGems*. Their empirical results show that the majority of security vulnerabilities affect more than one major train of a package but are only fixed in the latest one, leaving thousands of dependent packages exposed to the vulnerability. This suggests that many dependent packages could benefit from backports provided by their dependencies, however, such updates are rare, and mostly practised by more mature and more active packages.

Motivated by the effectiveness of backporting in managing package dependencies, in this work we intend to conduct a differentiated replication study [5] of the work of Decan *et al.* and extend their analysis to explore current backporting practices within the package ecosystems of the following programming languages: *npm* for *JavaScript*, *PyPi* for *Python*, and *Maven* for *Java*. Specifically, we intend to address the following research questions:

RQ*$_1$ *How many packages still depend on older versions? Given that it is frequent for packages to have outdated dependencies while backporting allows dependent packages to rely on lower major trains and benefit from

new patches and security updates, finding the number of dependent packages that rely on older major trains allows us to verify if backporting is a useful solution for the ecosystems under study.

RQ*$_2$** ***How much backporting exists in the selected package distribution systems? This question serves to quantify the number of packages that practice backporting and to understand the extent to which backports are currently being used.

RQ*$_3$** ***What are the distinguishing characteristics of packages with backports? The goal of this question is to assess whether specific packages are more geared towards the use of backporting and if those packages have distinguishing factors.

RQ*$_4$** ***Does backporting affect the upkeep of lower major releases? A significant number of dependents packages still rely on releases from a lower major train. As such, package maintainers continue to support lower major trains by backporting updates to them. Therefore, we want to verify if backporting extends the support for lower major trains and thus serves a meaningful purpose as far as package maintenance is concerned.

Our choice of software ecosystems is motivated by two factors. First, Python, Java, and JavaScript are ranked as the top three most popular programming languages according to a study conducted in December 2021 by Popularity of Programming Language Index (PYPL)[1]. Second, the results from a survey conducted by Bogart *et al.* [4] that was designed to study the values and practices across 18 different software ecosystems and involved more than 2000 developers [4], suggested that backporting is still an underexploited phenomenon. To find out the extent of backporting practices reported for the three package distributions considered in our work, We downloaded and filtered the survey data provided in [12]. The results are shown in Figure 1 and confirm that the use of backporting is also rare in case of the *npm*, *PyPi*, and Maven ecosystems, as the majority of developers for each package ecosystem indicate that they do not use backporting.

Our results can be summarized as follows: Backporting is a useful strategy for package ecosystems to adopt, given that a significant number of packages depend on lower major trains. Packages with backports have a longer lifetime, more frequent releases as well as a higher portion of dependent packages. We observed that backporting does not affect the *npm*, *PyPi*, and *Maven* ecosystems in the same way. In fact, the *Maven* ecosystem relies on the phenomenon of backporting as a way of ensuring stability between required and dependent packages. The primary contributions of our work are as follows:

- To the best of our knowledge, this work is the first attempt to investigate backporting practices in *PyPi*, *Maven* ecosystems and compare the results with the *npm* ecosystem.

[1]https://pypl.github.io/PYPL.html

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

- Our findings provide key insights about current backporting practices within the studied ecosystems.
- Based on our observations, we identify potential research studies worth pursuing as far as the study of backporting is concerned.

This research study is organized as follows. In the next section we will review the relevant work to this project. In section III, we will describe our research methodology including how we extracted and preprocessed the data. In section IV, we will present the results from our study and outline the threats to validity in section V. We will discuss the results of our findings and compare them with the work of Decan *et al.* [6] in Section VI. We present our final remarks In Section VII.

II. Related Work

This project is based on an empirical research conducted in [6]. In that work, Decan *et al.* carried out a comprehensive study on backporting practices and analyzed how those practices were followed in four popular package ecosystems, namely *Cargo*, *npm*, *Packagist* and *RubyGems*. As part of their explorations, these researchers identified the necessity of backporting, quantified the proportion of packages that practice backporting, and estimated the percentage of dependent packages that benefit from this process. Decan *et al.* concluded that even though most dependent packages are up-to-date, a significant share of dependent packages continue to rely on a lower major train and argue that since discovered bugs and security vulnerabilities are usually fixed in the highest major train of a package, developers need to adopt a backporting strategy to enable dependent packages to benefit from the existing patch fixes. While this study performs a quantitatively rigorous analysis, the package distributions under study are not highly popular when the PYPL ranking is taken into consideration. Therefore, in this work we extend the study of Decan *et al.* to analyze and compare the current backporting practices within the *npm*, *PyPi*, and *Maven* ecosystems.

Even though the idea of backporting was originally applied in the Linux operating system [13], there are three different lines of research in the field of software engineering that are related to this practice. The first one is patch transplantation and patch porting. The focus of this area is to port patches that repair software issues or vulnerabilities from a project to forked projects. An example work from this line of study is [14], which proposes an automated technique for avoiding bug propagation between forked projects. Similar automated strategies from this emerging research area can also be implemented for backporting patches. Over time, the idea of backporting has been adopted by many package management systems. Our work is an attempt to investigate backporting practices and verify if the use of this phenomenon can be of benefit to the considered package ecosystems.

The second line of work related to backporting is package outdatedness and focuses on the extent to which a package is not referencing the latest version of a direct or transitive dependency. This inconsistency, also called update delay, makes packages prone to security issues even in cases where there are known fixes for these problems. Backports are particularly useful when package dependencies are not up-to-date, specifically in cases where a dependency is one or more major versions behind [6]. In [15], Wang *et al.* studied the correlation between out of date packages and security risks and quantified such risks in third-party libraries of Java open source projects. Zimmermann *et al.* [16], on the other hand, concentrated on the *npm* ecosystem and discovered evidence of single points of failure in unmaintained packages with security vulnerabilities that would transitively affect a broader number of packages. In [2] [3], Gonzalez-Barahona *et al.* quantified package outdatedness across various dimensions such as time, version, and vulnerability by introducing the concept of technical lag. These researchers extended their work in [17] and applied the technical lag framework as a formal model to *Docker* container images, to calculate the lag between image versions as well as the lag between package releases contained in image versions. Kula and his colleagues [7] conducted a study that covers 4,600 GitHub projects and 2,700 dependencies to explore how developers update their packages. Their experimental results showed that many of the studied systems rely heavily on dependencies, 81.5% of which are outdated. Additionally, 69% of the interviewees claimed that they were unaware of their vulnerable dependencies. Consequently, in the first research question (RQ_1), we calculate the proportion of packages that rely on lower major trains, to quantitatively verify if the practice of backporting can be useful to the ecosystems under study.

The third research direction which is closely related to the study of backporting is semantic versioning (SemVer). SemVer is a multi-component versioning scheme that is used to classify the type of changes that have been made in new package releases as major, minor, or patch to alert developers about potentially breaking changes or backward incompatibility issues [5]. Raemaekers *et al.* [18] explored the link between semantic versioning and the effects of breaking changes in the Maven dependency network of Java packages. This work reports that developers spend little effort to communicate backward incompatibilities or deprecated methods in package releases, which leads to a significant number of breaking changes and compilation errors. In [5], Ochoa *et al.* addressed some of the limitations of the work of Raemaekers *et al.* and expanded their analysis to study seven more years of the Maven Central Repository. Ochoa *et al.* note that their findings are in contrast with the results reported by the original study, in that an increasing number of Maven packages follow SemVer principles and that a small number of dependent packages are affected by breaking changes. Likewise, Decan and Mens [19] studied semantic versioning practices in four ecosystems (*Cargo*, *npm*, *Packagist*, and *Rubygems*) by examining the dependency constraints utilized when defining package dependencies. The experimental results of this work show that a large number of dependency constraints allow patch and minor updates to be installed automatically, and that this trend has increased over time. Opdebeeck *et al.* [20] analyzed the adoption of semantic versioning in Ansible, a popular Infrastructure-as-Code platform, that provides reusable collections of tasks called roles. Their results show that although most Ansible role developers follow the semantic versioning format, they do not always consistently follow the same rules when selecting the version bump to apply. We recognize that semantic versioning is necessary to study backporting. As a matter of fact, backporting takes a minor or patch update of a package and applies it in a higher major train. Therefore, as part of RQ_2, we will use this versioning scheme to identify existing backports and to quantify them within the *npm*, *PyPi* and *Maven* ecosystems.

III. Methodology

A. Terminology

Backporting is to take a minor or patch update which is applied to a higher major train, and to apply it to one or more lower major trains. Figure 2, adapted from [6], visually illustrates the process of backporting. According to this figure the patch update 2.1.1 fixes a security issue that was found in release 2.1.0. However, this vulnerability impacts release 1.3.0. Therefore, the patch is applied to major train 1 through patch update 1.3.1. In future, if the update 3.1.1 fixes a bug that is found in release 3.1.0, and this fix is going to be applied to the two lower major trains 1 and 2 through upcoming updates 1.3.2 and 2.2.1 then that would also be considered as backporting.

Semantic Versioning (SemVer for short), is a multi-component ([major].[minor].[patch]) version numbering template that is used to categorize the changes that have been made in a package update [6]. Backward incompatible changes lead to the increment of the major version component, significant backward compatible changes such as adding new functionality which does not impact existing dependents lead to the increment of the minor component, and backward compatible bug patches lead to the increment of the patch component. The version range of a required package can be restricted to those

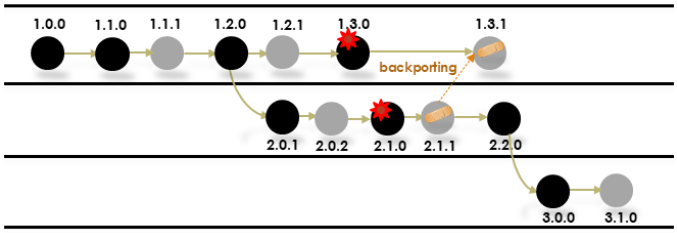

Fig. 2. Backporting security vulnerability patches from the highest major train to lower ones.

TABLE I
INITIAL DATASET OVERVIEW

initial dataset	*npm*	*PyPi*	*Maven*	**total**
packages fol. SemVer	1,217,677	15,171	1,922	1,234,770
active packages	15,661	749	231	16,641

releases that are expected to be backward compatible, when SemVer is combined with dependency constraints [6]. A major train is associated with each major version of a package. A major train is the ordered sequence of all releases of a package that have the same major version component. The major train of a package is considered to be higher than another major train, when the major version component assigned to it, is higher than the major version assigned to the other major train. The same procedure is followed for the minor and patch numbers of the package.

B. Dataset Overview.

To conduct our analysis, we used the libraries.io dataset (version 1.6.0) which contains dependency metadata for 33 different package distribution systems, including all releases of the package, their version numbers, their release dates, and their dependencies. We focused on version numbers and release dates to identify the practice of backporting in packages of the *npm*, *PyPi* and *Maven* ecosystems. Following the work of Decan *et al.*, [6], for each package release, we considered only dependencies to other packages within the same distribution and ignored dependencies that reference external sources such as git repositories. We developed a parser, to ensure that packages being studied were adhering to semantic versioning. Naturally, for each ecosystem, packages that did not have sufficient information or did not follow SemVer guidelines were removed from the dataset. Table I provides an overview of this initial dataset. As shown in the first row of this table, the number of *Maven* packages are far less than those of the *npm* and the *PyPi* ecosystems.

Similar to the work of Decan *et al*, we narrowed our exploration to study **active packages** together with all dependents of these packages. An active package is a package that was updated at least once in the last 12 months and has at least 5 dependents in the latest considered snapshot of each package ecosystem,*i.e.*, *npm*,*PyPi*, and *Maven*. To find the active packages we referred to the dependency metadata

TABLE II
CLEANED DATASET OVERVIEW

cleaned dataset	*npm*	*PyPi*	*Maven*	**total**
act. pkgs.	15,281	704	217	16,202
releases of act. pkgs.	570,676	29,896	11,562	612,134
dependents of act. pkgs.	242,618	14,370	1,859	258,847
depencs. of dependent pkgs.	1,021,843	28,498	3,512	1,053,853

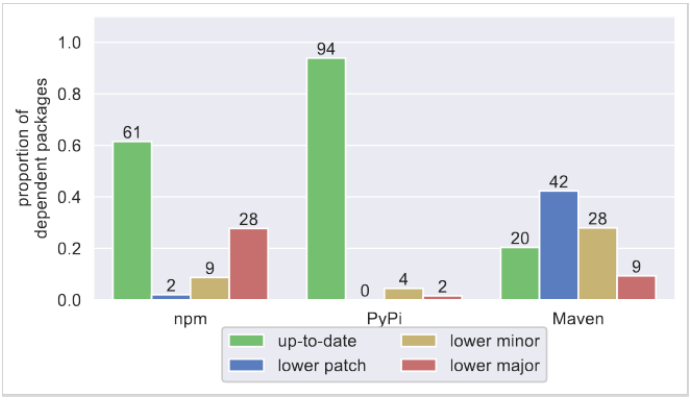

Fig. 3. Dependent packages relying on the highest available release, a lower patch, a lower minor or a lower major train

specified in the latest available release of each package in the dataset. Following the same cleaning process of the original work, we filtered the data as follows:

(i). We excluded those packages for which the release dates of its major versions did not follow a chronological order.
(ii). We excluded those packages for which a release precedes the release date of its corresponding major.

At the end of this cleaning process, we found 380, 45, and 14 of such packages for *npm*, *PyPi*, and *Maven* respectively. The cleaned dataset includes 16,202 required packages (and their entire release history), as well as the latest release of the 258,847 packages that depend on them. Table II provides an overview of the cleaned dataset.

IV. RESULTS

In this section we will present the results of our empirical study. Precisely, we will discuss the answers to the four research questions that were formulated in Section I. The code and results are made available for replication and further analysis on https://github.com/Samin005/COMP5900K-FinalProject

RQ*$_1$** ***How many packages still depend on older versions? The aim of this question is to understand if backporting is a beneficial practice for the *npm*, *PyPi*, and *Maven* ecosystems. Figure 3 shows the percentage of dependent packages which rely on the highest available release, a lower minor, a lower patch or a lower major train. We see in Table III, most dependent packages in the *PyPi* ecosystem (93.9%) are up to date and a smaller portion of packages in this ecosystem (1.6%) still rely on a previous major train. In case of *npm* (61.5%) of packages are up

TABLE III
PROPORTION OF OUTDATED DEPENDENT PACKAGES DEPENDING ON LOWER MAJOR TRAINS.

Ecosystem	(M-1)	(M-2)	(M-3)	(M->= 4)
npm	69.4%	20.0%	5.9%	4.7%
PyPi	57.2%	23.9%	10.6%	8.3%
Maven	90.7%	8.1%	1.1%	0.0%

TABLE IV
PROPORTION OF UP-TO-DATE AND OUTDATED PACKAGES

Ecosystem	(M)	(M-2)	(M-3+)
npm	61.5%	20.0%	10.6%
PyPi	93.9%	23.9%	18.9%
Maven	20.4%	8.1%	1.1%

to date while a more significant portion of packages in this ecosystem (27.7%) continue to depend on previous major trains. We notice a slight shift in case of the *Maven* ecosystem, since a fewer percentage of *Maven* packages (20.4%) rely on the highest major train. These numbers lead us to conclude that since bugs and vulnerabilities are typically fixed in the highest major train, to allow dependents packages to make use of those fixes, adopting a backporting strategy is necessary.

We now shift our focus on the category of dependent packages that rely on lower major trains (red bars in Figure 3) and verify how many packages are not depending on the highest major train. In Table IV, if we consider M to be the highest available major train, then a significant proportion of *npm* and *PyPi* packages (20.0% and 23.9%, respectively) are at least two major trains ($M-2$) behind. This number is significantly lower (8.1%) in case of the *Maven* ecosystem. To understand the reason behind this difference, we analyzed the recent download history of Junit, one of the most popular *Maven* packages. Table V, shows this history for three different releases of this package, including the number of downloads as well as the release dates.This snapshot was captured on April 10, 2022 [2]. It can be seen from this table that even though the latest release of this package, in this table, is version 5.8.2 which was released in November 2021, the majority of developers prefer to download version 4.12.0, which was released in December 2014. Furthermore, it is worth to notice that even though the latest release in the major train 4, is version 4.13.2, the number of downloads in case of version 4.12.0 are seven times higher than those for version 4.13.2 and about thirty three times higher than those of version 5.8.2. This observation suggests that developers prefer version 4.12.0, even if this means that they risk to miss all the bug fixes that were applied to both version 4.12.0 as well as the overall latest version 5.8.2.

RQ*$_2$ *How much backporting exists in the selected package

[2]https://mvnrepository.com/

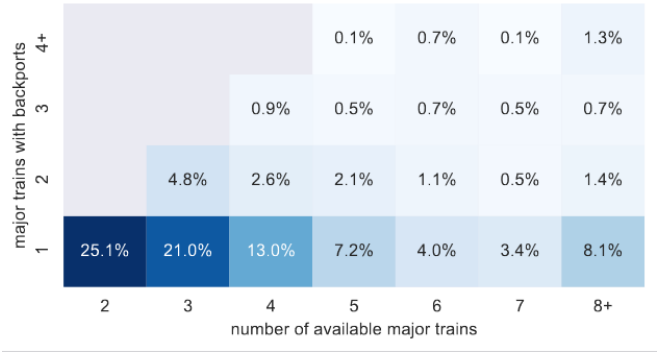

Fig. 4. Proportion of packages with backports based on the number of available major trains and the number of major trains that received a backport. (All ecosystems)

Fig. 5. Proportion of packages with backports based on the number of available major trains and the number of major trains that received a backport. (npm)

Fig. 6. Proportion of packages with backports based on the number of available major trains and the number of major trains that received a backport. (PyPi)

TABLE V
RECENT HISTORY OF MAVEN'S JUNIT DOWNLOADS

Version number	*Total downloads*	*Release Date*
4.12.0	57,214	Dec. 2014
4.13.2	8,024	Feb. 2021
5.8.2	1,777	Nov. 2021

TABLE VI
PROPORTION OF BACKPORTS

Ecosys-tems	Active Pkgs	Pkgs >= 2 Major Trains	Back-ports	Back-ports w.r.t Active Pkgs	Back-ports w.r.t >= 2 Major Trains
npm	15,281	8,870	1,010	6.60%	11.38%
PyPi	704	404	56	7.95%	13.86%
Maven	217	105	47	21.65%	44.76%

distribution systems?

The goal of this question is to explore how much backporting is practiced in the *npm*, *PyPi*, and *Maven* ecosystems and to analyze if this proportion is in line with the results of the survey that was depicted in figure 1. To quantify the number of backports in each ecosystem, we followed the methodology of Decan *et al.* and calculated any new release that was applied to a lower major train. Table VI shows the number and the percentage of packages with backports. The fifth column in this table shows the proportion of backports *w.r.t.* the active packages, while the last column provides a more accurate vision of the packages that have backports as it shows the proportion of backports *w.r.t.* those packages that have at least two major trains and thus are the real candidates for practicing backports. As it can be observed in this table, a small portion of packages in both the *npm* and *PyPi* ecosystems, 11.38% and 13.86% respectively, currently have backports, which is lower than the percentage of developers of each ecosystem (26% for *npm* and *22%* for *PyPi*) that confirmed to use backporting updates.

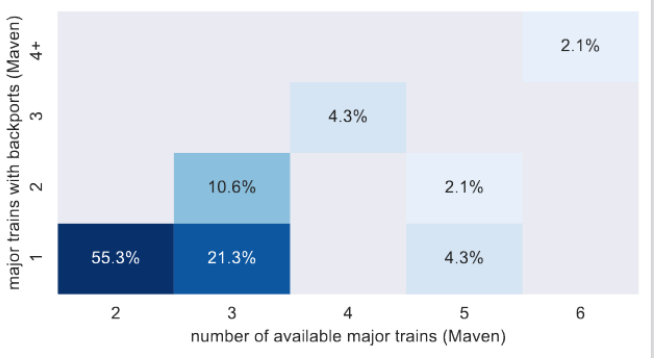

Fig. 7. Proportion of packages with backports based on the number of available major trains and the number of major trains that received a backport. (Maven)

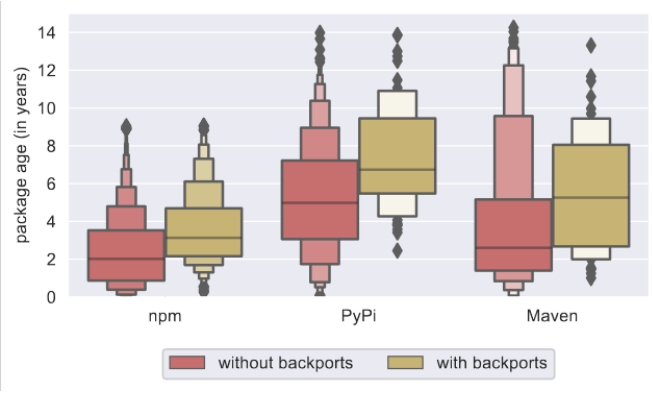

Fig. 8. Distribution of package age for packages with and without backports.

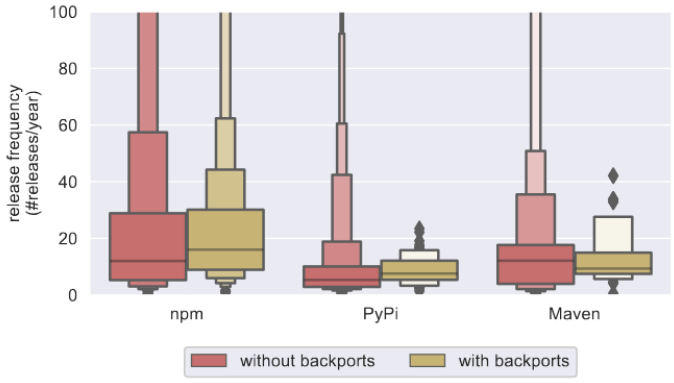

Fig. 9. Distribution of package age for packages with and without backports.

For *Maven*, however, the proportion of packages with backports is higher (44.76%) than that of reported by *Maven* developers (35%).

Since from table III, it can noticed that many packages have a variety major trains that are used by their dependents, we rely on figures 4 to7 to find out which lower trains are mostly targeted by backports. It can be observed that even if there are multiple major trains, most packages are backported to a single major train. For all ecosystems, figure 4, the previous major train (84.6%) and the major train prior to that (10.3%) are the major trains that receive most of the backports, a similar trend can be noticed in case of each individual ecosystem (see: figures 5 to7).

RQ$_3$ What are the distinguishing characteristics of packages with backports?

This question is formulated for us to analyze if packages that have shown distinguishing factors. Following the work of Decan *et el.* and considering the age of a package, since more mature packages have had a longer time to have backports. Figure 8 depicts the distribution of package age for each ecosystem, categorized for packages that have backports and those that do not have them. It can be observed that packages with backports have a longer lifetime. This observation is also confirmed from

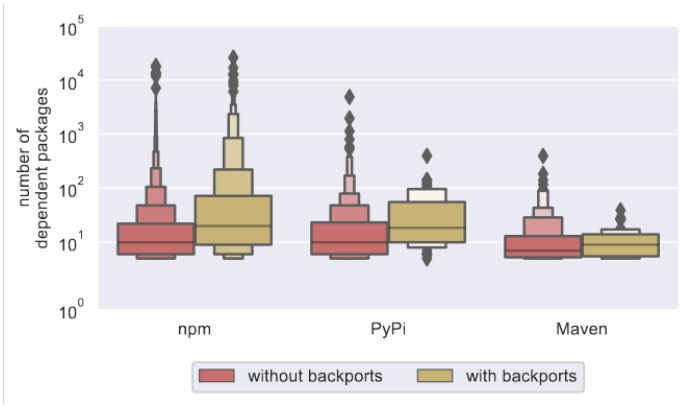

Fig. 10. Distribution of number of dependents for packages with and without backports.

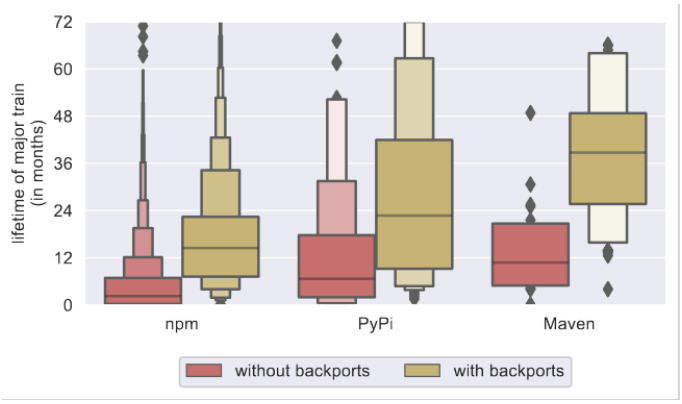

Fig. 11. Lifetime of lower major trains, with and without backports.

a statistical point of view ($p < 0.01$) based on the Mann-Whitney-U tests after using the Bonferroni-Holm method to check for family-wise error rate [6]. Based on the interpretation provided by Romano *et al.*, the effect size (measured using Cliff's delta) is medium for *npm* ($|d| = 0.367$) and *PyPi* ($|d| = 0.408$), and small for *Maven* ($|d| = 0.321$).

We also measure a package's release frequency as Decan *et al.* reported that packages with backports have a higher activity rate. Figure 9 shows the distribution of package release frequency, categorized by packages that have backports and those that do not. The Mann-Whitney-U tests confirm that this frequency is in fact higher for those category of packages that have backports. The effect size is small($0.150 <= |d| <= 0.280$) to negligible for all ecosystems, to be more specific 0.150, 0.226 and 0.014 for *npm*, *PyPi* and *Maven* respectively. In their Decan et al. [6] reported that packages with backports have a higher number of dependents. Figure 10 shows the distribution of the number of dependents of a package, categorized by packages that practice backporting and those that do not practice it. It can be observed that packages with backports have a higher number of dependents, this is also validated statically by Mann-Whitney-U tests, since the effect size is small ($0.182 <= |d| <= 0.382$) to negligible for all package dependencies, to be more precise 0.311, 0.316 and 0.049 for *npm*, *PyPi* and *Maven* respectively.

RQ*$_4$ *Does backporting affect the upkeep of lower major releases?

The purpose of this question is to explore if the phenomenon of backporting extends the support for lower major trains. Following the work of Decan *et al.* we defined the lifetime of a major train as the time between its first and last chronological release. Figure 11 depicts the distributions of the lifetime of lower major trains, categorized by major trains that do receive backports and those that do not. It can be noticed from this figure that lower major trains that are targeted by backports, have a longer life. This fact is also statically confirmed by Mann-Whitney-U tests, since the *PyPi* ecosystem has a medium effect size (0.47) while the *npm*, *Maven* package ecosystems have a large effect size respectively 0.666 and 0.772.

Fig. 12. Proportion of packages based on the number of major trains used by dependents.(all ecosystems)

Figures 12-15 show the proportion of packages relative to the number of major trains used by the dependents of all ecosystems as well as in case of each individual

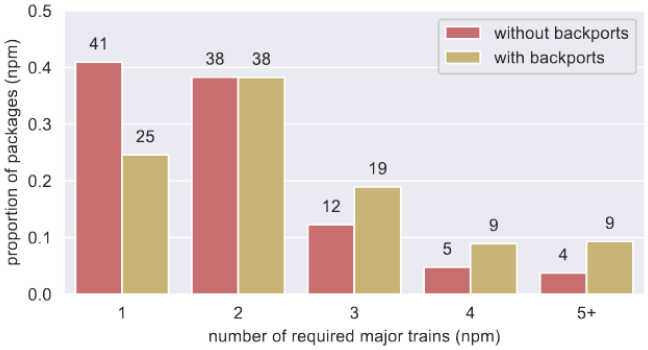

Fig. 13. Proportion of packages based on the number of major trains used by dependents.(npm)

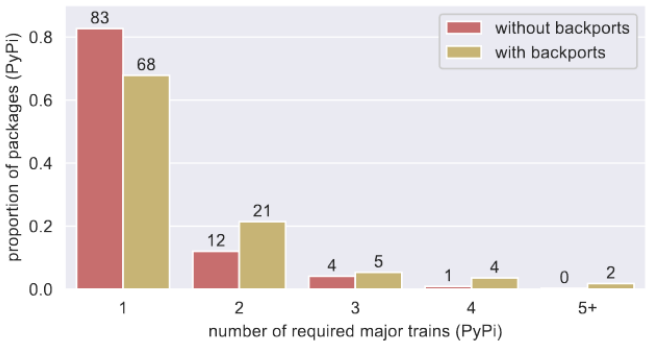

Fig. 14. Proportion of packages based on the number of major trains used by dependents.(PyPi)

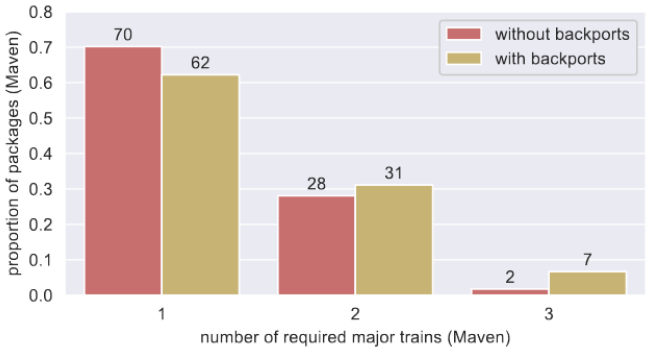

Fig. 15. Proportion of packages based on the number of major trains used by dependents.(Maven)

ecosystem. The figures in case of all ecosystems as well as the individual ones show that packages with backports have a higher number of major trains used by their dependents. We observe that as the number of required major trains increases, the proportion of packages with backports also increases over time. We can observe a similar trend for each individual ecosystem.

V. Threats to Validity

A potential threat to our study is that we use semantic versioning to identify existing backports updates. Since semantic versioning is a principle developed by humans, it is likely prone to human errors. As a result, we acknowledge that some developers may fail to adhere to semantic versioning for their packages but since, to the best of our knowledge, no foolproof methods exist to verify versions assigned to packages, we still made the decision to base our exploration on semantic versioning but filtered out packages that did not abide by this versioning scheme.

Another possible threat to our work is that since a small portion of *Maven* packages adhere to the convention of semantic versioning, a relatively smaller number of *Maven* packages were included in our analysis. This may lead some readers of this work to argue that, due to this shortcoming, our empirical study fails to fully quantify the real extent of backporting practices in case of the *Maven* ecosystem. However, it is worth to remark that many *Maven* packages were created before the adoption of semantic versioning and thus not all of the existing packages in this ecosystem follow this principle. Furthermore, since we relied on semantic versioning to identify backports in the *npm* and *PyPi* ecosystems, to make a fair comparison between the ecosystems being studied, we also restricted our study of the *Maven* packages to the subset that complies to this versioning scheme. In fact Decan et al. [6], excluded the *Maven* ecosystem from their study because many *Maven* packages are known to not follow semantic versioning. However, since recent studies have shown an increased adherence by *Maven* packages to semantic versioning [5], we decided to include the *Maven* ecosystem as part of our studies.

VI. Discussions

One of our major goals from extending the work of Decan *et al.* was to verify if the results of their work could be generalized to the *PyPi* and *Maven* ecosystems. Our observations in this regards are as follows:

1) Similar to the observation made by Decan *et al.*, most packages within the *npm* and *PyPi*, were up-to-date, *i.e.* most dependent packages relied on the highest available major train, but still a considerable number of packages were relying on lower major trains. As concluded by Decan *et al.*, our results also suggest that backporting can be a useful practice to be adopted by packages that rely on lower major trains. However, in case of the *maven* ecosystem, we noticed a different pattern. As a matter of fact, a relatively smaller proportion of *Maven* packages were up-to-date and at the same time a higher proportion of packages in this ecosystem relied on a patch update or lower minor releases, which could suggest that the *Maven* packages have been using backporting packages for a longer period of time.
2) As reported by Decan *et al.* for the ecosystems that they studied, backporting practices in case of the *PyPi* ecosystem are not significant at the time of our study, but this argument does not hold true for the *Maven* ecosystem. In fact, *Maven* packages have a higher proportion of backporting. Decan et al. had also noted that Packagist had a relatively high percentage of backporting when PHP7 was released in 2016. Although *Maven* also has a high percentage of backports, there is no evidence suggesting a sudden spike in the adoption of backporting by the packages in this ecosystem. This observation coupled with the fact that not many *Maven* packages depend on the highest major train, seems to suggest that the *Maven* ecosystem has been benefiting from the phenomenon of backporting as a way of providing stability to the dependent packages of this ecosystem.
3) As far as the unique characteristics of packages with backports are concerned, we noticed a similar pattern as the results reported by Decan *et al.*. More specifically, older and more active packages show a higher proportion of backporting, which makes a logical sense since such

packages have had a longer period of time to take advantage of this phenomenon. Furthermore, packages with a higher release frequency have a higher proportion of backports and this might be due to the fact that their dependent packages are outdated quite often and must adopt backporting more frequently to take advantage of new updates and bug fixes.

4) Our findings align with the observations of the work of Decan *et al.*, in the sense that in case of all ecosystems, namely *npm*, *PyPi*, *Maven*, packages with backports are maintained for a longer period of time. Therefore, we can conclude that backporting can be an efficient way of increasing the lifespan of ecosystem packages. Although we did not fully verify the validity of this claim as part of our studies, an interesting analysis would be to study the effect of backporting in deprecated packages that used to support backporting.
5) Since the phenomenon of backporting seems to play an important role in the stability of the *Maven* ecosystem, we believe a direct connection could be established by future research attempts between the practice of backporting and the survival of package ecosystems.

VII. Conclusion

In this work, we made an attempt to fill the research gap in the study of backporting practices in the *npm*, *PyPi*, and *Maven* ecosystems. Our results show that *npm* and *PyPi* developers have the tendency to create newer releases more often, which leads to an increasing number of packages to rely on outdated dependencies. In contrast, most *Maven* packages do not rely on the latest version of a required package. We also noticed that most packages are backported to a single major train. As a matter of fact, the major train that immediately precedes the highest available one, tends to be the one that is mostly targeted by backports. Finally, packages that support backporting tend to have a longer lifetime, a higher release frequency, and more dependent packages. Although our findings are for the most part in line with the results reported by Decan *et al.*, we expect our unique insights in regards to the practice of backporting within the *npm* and *PyPi* ecosystems can be beneficial to the future studies of this phenomenon. Furthermore, we hope our empirical study helps in creating awareness among developers of package ecosystems about the impacts and benefits of backporting, and invite them to follow this practice in their packages. Last but not least, we encourage ecosystem maintainers to adopt policies and strategies that would lead to a better management of backporting in their ecosystems.

References

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

[2] 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 Conference on Open Source Systems*. Springer, Cham, 2017, pp. 182–192.

[3] A. Zerouali, T. Mens, J. Gonzalez-Barahona, 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*, vol. 31, no. 8, p. e2157, 2019.

[4] 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," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 4, jul 2021. [Online]. Available: https://doi.org/10.1145/3447245

[5] L. Ochoa, T. Degueule, J.-R. Falleri, and J. Vinju, "Breaking bad? semantic versioning and impact of breaking changes in maven central," *arXiv preprint arXiv:2110.07889*, 2021.

[6] A. Decan, T. Mens, A. Zerouali, and C. De Roover, "Back to the past – analysing backporting practices in package dependency networks," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[7] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[8] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," *arXiv preprint arXiv:1811.00918*, 2018.

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

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

[11] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 84–94.

[12] C. Bogart, A. Filippova, J. Herbsleb, and C. Kastner, "Culture and Breaking Change: A Survey of Values and Practices in 18 Open Source Software Ecosystems," 8 2017. [Online]. Available: https://kilthub.cmu.edu/articles/dataset/Culture_and_Breaking_Change_A_Survey_of_Values_and_Practices_in_18_Open_Source_Software_Ecosystems/5108716

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

[14] L. Ren, "Automated patch porting across forked projects," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1199–1201.

[15] 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 (ICSME)*. IEEE, 2020, pp. 35–45.

[16] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Smallworld with high risks: A study of security threats in the npm ecosystem," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 995–1010.

[17] A. Zerouali, T. Mens, A. Decan, J. Gonzalez-Barahona, and G. Robles, "A multi-dimensional analysis of technical lag in debian-based docker images," *Empirical Software Engineering*, vol. 26, no. 2, pp. 1–45, 2021.

[18] 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.

[19] 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, 2021.

[20] R. Opdebeeck, A. Zerouali, C. Velázquez-Rodríguez, and C. De Roover, "Does infrastructure as code adhere to semantic versioning? an analysis of ansible role evolution," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2020, pp. 238–248.