UMONS
Université de Mons

Faculté
des Sciences

A Measurement Framework for Analyzing Technical Lag in Open-Source Software Ecosystems

Ahmed Zerouali

A dissertation submitted in fulfillment of the requirements of the degree of *Docteur en Sciences*

Advisor:
Dr. Tom Mens
Université de Mons - Belgium

Jury:
Dr. Olivier Delgrange
Université de Mons - Belgium

Dr. Alexandre Decan
Université de Mons - Belgium

Dr. Jesus Gonzalez-Barahona
Universidad Rey Juan Carlos de Madrid - Spain

Dr. Alexander Serebrenik
Eindhoven University of Technology - The Netherlands

September 2019

Acknowledgment

This work could have never been completed without the support of many people. Thus, I apologize in advance if I forgot to mention someone important; if we have met and talked, you are also a part of this.

First of all, I would like to express my sincere gratitude to my advisor Dr. Tom Mens for his help throughout the years. He is a great professor who has provided me with a lot of guidance and knowledge, and I am very privileged to have been his PhD student. He encouraged me to surpass myself and I would like to thank him warmly for that. I address a special thank to Bitergia for hosting me during a year and half; and in particular to the co-advisors Dr. Jesus Gonzalez-Barahona and Dr. Gregorio Robles for their guidance during this period of my stay in Madrid and after.

I would like to thank the members of my jury: Dr. Alexandre Decan for his advices and help to improve my work and for following my thesis progress during these three years; Dr. Olivier Delgrange and Dr. Alexander Serebrenik, to accept being part of my jury. I am also grateful to my colleagues and co-authors Dr. Valerio Cosentino and Dr. Eleni Constantinou for their precious advice.

I wish to thank my friends who offered me much joy and who regularly proved me that there is a life apart from the thesis. Finally, my deepest gratitude goes to my family for their unflagging love and encouragement throughout my life, and without whom, this thesis would not have been possible.

Abstract

Software development practices have evolved quite a lot since the early days of programming. Most software projects today, especially in the open source software community, are using distributed versioning development practices. In addition, they heavily rely on reusing external software packages, to realize part of their functionality, rather than needing to implement these functionalities themselves.

Frequently, reusable Open Source Software (OSS) components for major programming languages and operating systems are available in public package repositories where they are developed and evolved together within the same environment. Developers rely on package management tools to automate deployments, specifying which package releases satisfy the needs of their applications. However, these specifications may lead to deploying package releases that are outdated or otherwise undesirable because they do not include bug fixes, security fixes, or new functionality. In contrast, automatically updating to a more recent release may introduce incompatibility issues. Moreover, while this delicate problem is important at the level of individual packages, it becomes even more relevant at the level of large distributions of software packages where packages depend, directly or indirectly, on a large number of other packages.

The goal of this thesis is to capture and study this delicate balance between the need of updating to the ideal release and the risk of having breaking changes. We formalize a generic model of *technical lag*, a concept that quantifies to which extent a deployed collection of components is outdated with respect to the ideal deployment. Then, we show how to operationalise this model for different case studies and we empirically analyze its evolution in npm, Debian and Docker ecosystems. Finally, we develop a tool to support Docker deployers in assessing the health of the software included in their containers.

Contents

Chapter 1

Introduction

1.1 Context

Software components are being created and reused on a regular basis. Over the past years, depending on external software components has become a common software development practice, especially in the free, Open Source community [1]. This practice can lead to a significant gain in productivity, due to the ability to reuse complex functionality, rather than implementing it from scratch [2].

Because these components are usually evolving to avoid becoming obsolete [3, 4, 5], many versions of them are being created and distributed via online package managers and repositories everyday (e.g., *npm, Maven, Debian*, etc). Usually, new versions include new features, changed requirements, improved performance, fixed bugs, etc. In general, these changes are seen as a good sign of a well maintained software component [6]. On the other hand, major changes may require breaking changes.

While the availability and abundance of reusable components facilitates building software, it can also cause problems in maintenance and evolution. For example, a recent version of an application may be outdated although not because of its own code, but due to depending on components that were not updated to their latest versions. If this happens, there is a higher risk of having bugs and security issues that may have been already fixed in newly released versions [7]. On the other hand, updating to more recent releases of reusable components is not *for free*, since it might lead to a risk of facing backward incompatible changes [8], which cause conflict and problems for developers using these components. In many cases, these problems may eventually lead to ripples through software ecosystems [9, 1].

For individual developers, there is a balance between benefits (e.g., new functionality, bug and vulnerability fixes, etc) and cost of updating a dependency (e.g., risk of having

breaking changes). To represent this balance, we introduce the *technical lag* concept as a measurement to capture the difference between the reusable software component version that we *want* to update to and the deployed version of the same software component that we rely on.

1.1.1 Empirical Software Engineering

Empirical software engineering is a research area concerned with the empirical observation of software engineering artifacts and the empirical validation of software engineering tools, theories and assumptions. Sub-fields of software engineering that are related to empirical research include software maintenance, software evolution and software repository mining.

Two decades ago, it was rare to see a conference or journal article about a software development tool or process that had empirical data to back up the claims. Today, in contrast, it is common that conferences and journals publish articles where the scope is a description of a study or evaluation. For this reason, a very successful conference (International Symposium on Empirical Software Engineering and Measurement), a journal (Empirical Software Engineering), and organization (International Software Engineering Research Network) have emerged and evolved in the last two decades to focus only on the empirical software engineering research area. Moreover, it is nowadays rare for software engineering research works not to include some empirical studies. In fact, empirical studies are among the top most popular topics in conferences like the ACM/IEEE International Conference on Software Engineering.

Using several research methods and techniques, empirical software engineering has produced a steady stream of evidence-based results concerning the factors that affect important outcomes such as software cost and quality. The most popular methods are surveys to collect data on some phenomenon, controlled experiments to measure the correlation between variables and case studies to investigate contemporary phenomena.

Our research includes creation of models, tools and assumptions. For this reason, we will rely on mixed-methods approach of empirical software engineering methods for cross-validation. Mixed method research employs data collection and analysis techniques associated with both quantitative and qualitative data [10]. More specifically, this thesis makes use of case studies and surveys as two complementary methods to quantitatively and qualitatively validate our research results.

1.1.2 Free and Open-Source Software

Free and open source software contain source code that anyone can inspect, modify, and improve. Because of this characteristic, collaborative development of free software has

witnessed an exponential increase in the last three decades. It represents a successful example of software development where communities of developers work together on an often voluntary basis, while users of the free software can generally submit bug reports and requests for change or improvement.

The word "free" in "free open source software" refers to freedom, not monetary cost. Though most free open source software are indeed free in price, the word "free" is referring to the freedom to use software and edit their source code, as long as their copyright is attributed to the individuals or organizations that created the software. Moreover, it stays free and open source when it is distributed to others.

In fact, when the *Free Software Foundation (FSF)* was founded in 1985 to support the free software movement, it was founded with the goal of promoting the universal freedom to study, distribute, create, and modify computer software [11]. According to the FSF, the word "free" software refers to four stages of freedom:

1. freedom to execute a software program, for any (private or commercial) usage;
2. freedom to study the software functionalities and edit its source code to adapt it to one's own needs;
3. freedom to redistribute copies of the program;
4. freedom to distribute the modifications made to the program.

The last freedom, which necessitates access and modifications to the source code, gives to the whole community the opportunity to use the software and make changes to further improve it.

Later on, in 1998, the *Open Source Initiative (OSI)* has been created to inform and promote the benefits of open source. According to OSI, the promise of open source is better quality and higher reliability, more flexibility, lower cost, and less dependence on commercial software. However, it has been acknowledged that in many cases, definitions of both organizations on free and open source software are equivalent [12]. Thus, the term Open Source Software (OSS) is conventionally used to describe a software system developed based on these notions.

In the context of this thesis, we focus on open source software systems for different reasons: (1) the growing interest of OSS development in industry [13], government, and academia [14]; (2) the abundance and accessibility of software projects for which historical data is freely available; (3) the ability to publish scientific results about these systems without breaking confidentiality agreements; (4) the ability to allow other researchers to reproduce and verify the obtained results.

1.1.3 Software Ecosystems

The widespread use of online collaborative development solutions surrounding distributed version control tools (such as Git and GitHub) has led to a growing popularity of so-called *software ecosystems*.

The term ecosystem is borrowed from biology where it represents a complex community of organisms and its environment functioning as an ecological unit. *Software ecosystem* is a term that appeared during the last decade in the fields of software evolution and software repository mining. Since no clear agreement existed on the definition of a software ecosystem [15, 16], many researchers started using software ecosystems as what Lungu et al. have proposed [17, 18], i.e., *"A software ecosystem is a collection of software projects which are developed and co-evolve in the same environment".*

According to Lungu [17], a software ecosystem should have an environment that can host it. This environment can be physical, like in the case of a company or a research group that has a geographical address, but can also be virtual, like the projects that are part of an open-source community. In the context of this thesis, we focus on the virtual environments, more specifically on software components that are surrounded by open source software communities.

There are multiple well-known examples of open source software communities that can be referred to as software ecosystems. For example:

- The *GNU* project aims at providing a fully Open Source operating system and it consists of many software projects and packages, each providing a piece of that operating system.
- The Debian project is an association of individuals who have defined as a common cause to create a free operating system. Debian-based operating systems use the Linux kernel or the *FreeBSD* kernel. Moreover, a large part of the basic tools that construct the operating system come from the *GNU* project.

Software ecosystems tend to be very large, containing from tens to millions of software components[1] [1], that are in many cases interconnected, in the sense that they form a dependency network. In the case of so large ecosystems, the dependency network can be so big and complicated that it can cause problems and conflicts between dependents and dependencies [19]. So, while assessing the problem of technical lag is important at the level of individual components, it becomes more relevant and problematic at the level of software ecosystems where packages depend directly and indirectly on each other.

For this reason, we consider different software ecosystems as case studies in this thesis:

[1] http://www.modulecounts.com/

- GitHub Java projects that make use of popular Java testing-related libraries.
- npm, the default package repository of *JavaScript* packages and also the largest package repository in the internet [1].
- Debian, one of the most popular operating systems that are based on the Linux kernel, forming the basis for many other Linux-based distributions.
- Docker Hub, the most popular repository for the Docker containerization technology [20].

1.1.4 Technical Lag

To quantify how outdated a deployed software package release is w.r.t. the "ideal" situation, we present the concept of "technical lag" as *the increasing lag between upstream development and the deployed system when no corrective actions are taken.* Its goal is to enable developers to decide on a more informed basis whether or not to seize the opportunity of relying on more recent component releases, while taking into account the increased risks that may result from keeping one's dependencies outdated [21].

When measuring technical lag, the first problem is to decide what is the ideal component or version of the software we deploy. Then, we compare this ideal version with the current used version of the same deployed software. Depending on requirements and needs, the comparison may focus on stability, functionality, performance, or something else.

Once the ideal version of the deployed software is defined, we still need to choose the metric or unit of measurement to use in order to represent the lag between the ideal and the deployed components. For example, if the focus is on security, the measurement unit could be the number of security issues fixed in the ideal software component which have not been fixed in the deployed system. If the focus is functionality, the measurement unit could be the number of features implemented in the ideal component which have not been implemented in the deployed system. Some other interesting measurement units could be the difference in time between the ideal and deployed component release dates, or the number of updates and version numbers changed between them.

1.2 Goals and Contributions of the Thesis

In the light of all the above, the goal of this thesis is to empirically study how software components are used in projects and other components of open source software ecosystems and provide support to developers of such projects by creating a technical lag framework that can assess the health of their software. More specifically, the main contributions of

this thesis are the following:

- ✓ We provide evidence that developers do not often switch between software testing-related libraries that provide the same functionality, they tend to stick to one library (Chapter 3).
- ✓ Qualitative analysis: We assess the usefulness of the technical lag concept by carrying out surveys and interviews with software practitioners (Chapter 4).
- ✓ We define an extensible framework of technical lag in order to be able to quantify to which extent a deployed (collection of) component(s) is outdated with respect to the ideal deployment (Chapter 4).
- ✓ Quantitative analysis: We operationalize the technical lag framework and analyze its evolution in three different ecosystems: npm, Debian and Docker containers. Then, we offer actionable results and lessons learned in order to support developers in reducing their technical lag (Chapter 5, 6).
- ✓ We provide the ConPan tool to inspect Docker containers and extract their installed system and third-party packages and analyze how outdated they are in terms of missing updates, vulnerabilities and bugs (Chapter 7).

1.3 Structure of the Dissertation

This introduction has provided the necessary context and described the objectives of this thesis.

The remainder of this dissertation starts by presenting a state of the art of the research in Chapter 2. It presents previous research results found in the scientific literature on measuring software outdatedness in software ecosystems.

Chapter 3 presents the results of an exploratory analysis on the use of testing libraries in Java projects. This first analysis allows us to understand some of the common practices of library usage, which will pave the way for more extensive research.

Chapter 4 presents the concept of technical lag in more detail and introduces a general framework for measuring the technical lag. This chapter also includes interviews and surveys carried out with software practitioners for the sake of assessing the usefulness of the technical lag concept.

Chapter 5 operationalizes the generic framework of technical lag for the npm packaging ecosystem and then measures and analyzes it for the whole package dependency network of npm.

Chapter 6 operationalizes the generic framework of technical lag for Docker containers and evaluates how outdated, vulnerable and buggy packages in containers are.

Chapter 7 presents the ConPan tool that inspects Docker containers for installed system and third-party packages. This tool provides relevant information about packages: their technical lag, outdatedness, vulnerabilities and bugs.

Finally, Chapter 8 concludes by summarizing the main contributions, limitations, and future work.

Chapter 2

State of the Art

Software component reuse has been an important topic of software engineering research for several decades [22]. Around the 2000s, with the emergence of COTS (components-off-the-shelf), many researchers investigated how to manage dependencies and evolution [23, 24], and advocated the need to have software ecosystems with powerful package managers and repositories (often referred to as *configuration management tools*) that allow to improve software reuse [25]. Nowadays, such package managers have become commonplace due to the rising popularity of Open Source Software repositories, accessible through online collaborative platforms.

In this chapter, we present a non-exhaustive summary of the related research to this thesis. The chapter is divided into six sections. The first section presents all terms used in this chapter. The second and third sections discuss general studies around version and dependency management in software ecosystems. The forth and fifth sections review previous research about software vulnerability and outdatedness management, while the final section presents the work in which the notion of technical lag has been first introduced.

2.1 Terminology

In this chapter, different terms are used to explain the related work. Inspired by [1], Table 2.1 introduces the main terminology used in this article. The parts of each term that are indicated between parentheses in the first column of the table will be implicitly assumed if they are clear from the context.

Term	Meaning
(Packaging) Ecosystem	The collection and history of all software artefacts and community members contained a particular package repository.
Package Manager	A collection of software tools that automates the process of installing, configuring, upgrading or removing software packages on a computer's operating system in a consistent manner.
Package	A software program providing specific functionalities. A package usually exists in many versions which are called releases. By abuse of language, a package at time t denotes its latest available release at time t.
(Package) Release	A specific version of a package that can be accessed from a package repository and installed through the package manager. It includes what is needed to build, configure and deploy the package version, as well as a manifest containing important metadata such as its owner, name, description, timestamp, etc.
(Package) Update	A new release of a package, provided by the package manager, that succeeds (i.e., corresponds to a higher version number or timestamp) a previous release of the same package.
(Package) Dependency Network (at time t)	A graph structure in which the nodes represent all the packages made available by the package manager at time t, and the directed edges represent direct dependencies between the latest available releases at time t.
Dependency	An explicitly documented reference (in the manifest of a release) to another package that is required for its proper functioning. Dependencies that are explicitly documented in the release manifest, are called direct dependencies. Those that are part of the transitive closure of the dependency network are called transitive dependencies. Transitive dependencies that are not direct are called indirect dependencies
(Dependency) Constraint	A condition that is used by a dependency to restrict the supported releases of the target package.
Required package	A package that is the target of at least one dependency from another package. Similarly we define transitively required.
Reverse Dependency	Reverse dependencies are obtained by following the edges of the dependency network in the opposite direction. As for normal dependencies, they can be direct, transitive or indirect.
Dependent (package)	A package that is the target of at least one reverse dependency from another package. In a similar vein, we define transitively dependent.

Table 2.1: Meaning of terms used in this chapter

2.2 Version Management in Software Ecosystems

Semantic Versioning (henceforth referred to as *semver*[1]) has become a popular policy to recommend how to manage, assign and increment version numbers of new component releases. It provides a simple set of rules and requirements to communicate the type of changes made when releasing a new version of a software component. This allows dependent software components to be informed about possible "breaking changes" [9].

A syntactic *semver*-compatible release uses a version number composed of a *major*, *minor* and *patch* number. This format allows to order releases and indicates the importance of each new release. For example, *1.2.3* occurs before *1.2.10* (higher patch number), which occurs before *1.3.0* (higher minor number), which occurs before *2.1.0* (higher major number). Backward incompatible updates should increment the *major* number, backward compatible updates that may add new functionalities should increment the *minor* number, while simple bug fixes or security patches should increment the *patch* number. Additional labels or tags, for example for specifying pre-releases and build metadata, may be appended to the MAJOR.MINOR.PATCH format (e.g., *1.2.3-pre1, 1.2.10-beta2*, etc).

A component can restrict the releases of other components on which it depends by specifying version constraints in the specification of the dependencies. Upon installation of the component, the installation manager will consider these constraints to install the most "appropriate release" for each dependency. The meaning of "appropriate release" may vary from one package repository to another. For example, the npm package manager will select for installation the highest available version satisfying the dependency constraints, while in NuGet (a package manager for the .NET environment), the first version satisfying the dependency constraints will be selected, starting from the oldest to the new released versions.

Version constraints can be quite diverse. For example, Table 2.2 summarizes the types of version constraints that can be used for specifying npm package dependencies, together with the interpretation of each constraint type. Tilde ($\sim$) and caret ($\hat{}$) are the most common version constraints. Tilde allows only new patch releases to be installed while caret allows new patch or minor releases to be installed. In addition to these two constraints, other constraints and logical operators can be used to specify which package version to be used (e.g., *latest*, x, $>$, $=$, etc). Note that these constraints may have a different interpretation when they are used with the unstable major version zero (*0.x.y*) that is typically used for initial development.

Besides the fact that different packaging ecosystems have different variants of version constraints and syntactic notations, they may interpret the same notation in a different way. For example, in npm, Packagist and RubyGems, the version constraint 1.0.0 means that 1.0.0 is the only allowed release, while in Cargo this constraint means that the whole

[1] `https://semver.org`

Constraint	**Interpretation**	**Notation**	**Satisfied versions**
strict	use exactly this version	2.0.0	2.0.0
tilde (∼)	the latest *patch* update	∼ 2.3.0	≥ 2.3.0 ∧ <2.4.0
caret (^)	the latest minor or patch update	^2.3.0	≥2.3.0 ∧ <3.0.0
latest	the latest available release	latest, x, x.x.x, * or *.*.*	≥0.0.0
minimal	the latest release above this version	>= 2.3.0	≥ 2.3.0
maximal	the latest release below this version	< 2.3.0	< 2.3.0
version ranges	the latest release in the specified version interval	1.2.3 - 2.3.4	≥1.2.3 ∧ ≤2.3.4
logic operators	any logic combination of constraints	2.5.3 || >2.8.1	2.5.3 ∨ >2.8.1
wildcard	allow updates to a release compatible with the wildcard	1.2.x	≥1.2.0 ∧ <1.3.0
		2.*	≥2.0.0 ∧ <3.0.0

Table 2.2: Types of dependency constraints for npm package dependencies.

range between $[1.0.0, 2.0.0[$ is allowed to be installed.

Because of its importance, semantic versioning has been subject to many research studies. Raemaekers et al. [26] investigated the usage of semantic versioning by Java packages in *Maven* over a seven-year period. They observed that package maintainers do not respect the semantic versioning syntax (e.g., one third of all minor releases introduce at least one API breaking change), and that the adherence to semantic versioning only marginally increases over time. Macho et al. [27] proposed the *BuildMedic* tool to automatically repair Maven builds that break due to dependency-related issues.

In order to study how developers declare dependencies, Decan et al. [28] empirically compared semver compliance of four software packaging ecosystems (Cargo, npm, Packagist and Rubygems), and studied how this compliance evolves over time. They also explored to what extent do ecosystem-specific characteristics or policies influence the degree of compliance. Their general conclusion was that the proportion of compliant constraints increases over time for all ecosystems, while ecosystem-specific notations, characteristics, maturity and policy changes play an important role in the degree of such compliance.

2.3 Dependency Management in Software Ecosystems

Today's software systems are increasingly depending on reusable libraries and packages stored in online package distributions for specific programming languages (e.g., npm, RubyGems, Maven) or operating systems (i.e., Debian and Ubuntu). The availability of such reusable packages and releases in package repositories and managers facilitates software development and evolution. For this reason, software dependency management is of great importance and it has been subject to many research studies in different software ecosystems.

2.3.1 Maven

Several related studies have focused on the Maven ecosystem of Java packages. In order to help developers to decide when to use which library version of a software dependency, Mileva et al. [29] proposed an approach and an associated tool based on concept of the "wisdom of the crowds". If a library version is used by more developers, it is more likely to be recommended. The authors acknowledge that other context-specific factors need to be taken into account to recommend the most appropriate version of a library, and that a cost-benefit analysis needs to be made before deciding to switch to a new version of a software library.

Software library migrations have also been subject to many studies. Teyton et al. [30] proposed an approach to identify sets of similar libraries that might produce library migrations. The results can be used for suggesting alternative libraries to developers who want to migrate from a library to another one. In an extended work [31], they analyzed how and why library migrations occur. They found that library migrations are relatively rare, and only few exceptional projects have witnessed more than one migration.

Benelallam et al. [32] presented the Maven Dependency Graph, an open-source dataset that aims at enabling the Software Engineering community to conduct large-scale empirical studies on Maven Central. Later on, they performed a quantitative empirical analysis on more than 1.4M artifacts that represent the versions of 73.653 Maven libraries [33]. They found that 30% of the libraries have multiple releases that are actively used by latest artifacts. This proportion increases in the case of popular libraries to 50%. They also found that more than 90% of the most popular versions are not the latest releases, and that both active and significantly popular versions are distributed across the history of library versions.

Sulir et al. [34] studied the *buildability* of Java software projects that are based on Maven and Gradle. Using a virtual environment, they tried to fully and automatically build target archives from the source code of over 7,200 open source Java projects. They found that more than 38% of these projects failed to build, mainly because of dependency

related issues, followed by Java compilation and documentation generation. They also found that larger, older and less recently updated projects fail more often.

2.3.2 npm

Dependency-related problems have also been investigated in the default package manager and repository for *JavaScript* packages, npm. Wittern et al. [35] examined the npm ecosystem in an extensive study that covers package descriptions, the dependencies among them, download metrics, and the use of npm packages in publicly available repositories hosted on GitHub. One of their findings is that the number of npm packages and their updates is growing super-linearly. They also observed that packages are depending more and more on each other. More than 80% of npm packages have at least one direct dependency.

Abdalkareem et al. [36] focused on potential problems caused by the huge number of dependencies on "trivial" packages in npm. While interviewed developers did not consider those packages as harmful, they were found to be less tested than other packages. Kula et al. [37] studied the impact of the same category of "micro-packages" in the npm *JavaScript* ecosystem. They found that some micro-packages have long dependency chains and incur just as much usage costs as other npm packages.

Mezzetti et al. [38] presented a novel technique, *type regression testing*, that automatically detects whether an update of an npm package contains type-related breaking changes in the API. These changes are modifications of a library that affect the presence or types of functions or other properties in the library interface, including renaming a public function, moving it to another location, or changing its type signature. To validate their technique, they conducted an experiment on 12 widely used packages. The experiment showed that their technique can identify type-related breaking changes with high accuracy. They observed that their evaluation was capable of detecting 26 breaking changes in 167 minor and patch updates of 5 high quality npm packages, and most of those breaking changes could not have been detected by existing techniques.

Most of the works for the npm ecosystem or for the *JavaScript* packages in general rely on the high-level metadata of software packages and their dependencies and not on a static or dynamic source code analysis of the package content. The latter is a very valuable fine-grained analysis but it is time-consuming. It cannot scale up to the ecosystem-level because of the massive amount of packages and their dependencies, and because of the large amount of newly created package versions. Zapata et al. [39] performed a manual inspection on a total of 60 *JavaScript* client projects from three cases of high severity vulnerabilities. They carried out an exploratory study at the function level and access of the library's API. Surprisingly, they found evidence that up to 73.3% of outdated clients were actually safe from the threat of the high severity vulnerabilities, which means that

analysis, at dependency level, is an overestimation.

2.3.3 Debian

The study of software dependencies has been deeply investigated in Linux-based software component distributions [40]. In this regard, researchers have found different types of dependencies that may arise between software components [41], and have proposed solutions for managing dependencies in evolving software component ecosystems and distributions [42]. In particular, Abate et al. [43] provide a formal framework for analyzing the future of software component repositories. They applied this framework to detect future problems related to challenging upgrades and outdated packages, and validated it on the Debian distribution. This approach is quite complementary to the work presented in the current dissertation, which provides a formal framework and associated metrics for studying the temporal evolution of outdated package dependencies.

Dependency conflicts can lead to co-installability issues when multiple versions of the same package or different packages are not allowed to be installed at the same time. Claes et al. [44] performed an extensive analysis on the evolution of package incompatibilities (strong conflicts) in the Debian stable and testing distributions. One of their findings is that packages that are always in strong conflict have a smaller survival probability than those who are not. Moreover, using different metrics related to the duration and presence of conflicts, they could identify several packages that have been reported as problematic by the Debian community in the past.

2.3.4 Docker

In recent years, the way of developing software has significantly changed to cope with the continuous changes in the product development cycle and the need to accelerate the time to market. In this evolving scenario, containerized applications, and in particular Docker images, play a key role, improving portability, reliability and deployment [45].

A Docker image is a lightweight, stand-alone executable piece of software, that includes an entire runtime environment [46]. Thus, an image contains an application, plus all its dependencies, such as system and third-party packages, libraries, binaries, and configuration files. The build configuration of an image is declared using a *Dockerfile* and consists of a list of commands grouped into hierarchical layers, each one identified by a unique hash signature. An image can be built upon another image, automatically inheriting its layers, and consequently its dependencies. By containerizing the application and its dependencies, differences in operating system distributions and underlying infrastructure are abstracted away. This promotes modularity and eases using and building new software. Docker images are freely available on registries such as Docker Hub, one of the largest

registries providing a common place to build, update and share images among users. In Docker Hub, images are distributed using repositories, that allow users to develop and maintain several versions of different images (e.g., for different architectures), where each image can be tagged with different names (e.g., debian:stretch and debian:buster) to ease search and use.

Because of their importance, different studies have been conducted for Docker containers. Cito et al. [47] conducted an empirical study on a dataset of 70,000 Dockerfiles, and contrasted this general population with samplings containing the top 100 and top 1,000 most popular projects using Docker. Their goal was to characterize the Docker ecosystem, discover prevalent quality issues, and study the evolution of Docker images. Among other results, they found that the most popular projects change more often than the rest of the Docker population, with an average of 5.81 revisions per year and 5 lines of code changed. Furthermore, they found that, from a representative sample of 560 projects, 34% of all Docker images could not be built from their Dockerfiles.

In the Docker image building process, only the top layer is read-write while the bottom layers are all read-only. However, temporary Dockerfiles are often used in the image building process. Nevertheless, if a temporary file is imported and removed in different layers by a careless developer, it will lead to a file redundancy, which will eventually lead to larger-size images. This restricts the efficiency of image distribution and thus affects the scalability of services. To address this problem, Lu et al. [48] termed it *temporary file smell* and conducted an empirical case study to the real-world Dockerfiles on Docker Hub. As a conclusion, they reported that this problem exists in a wide range of Dockerfiles.

Currently, a large number of reusable Docker images and repositories are available online as open source, through services such as Docker Hub and *Docker Store*. Effectively reusing these artefacts requires a good understanding of them, and semantic tags facilitate this understanding. However, the online communities do not support tag recommendation, and little training data is available. To address this problem, Zhou et al. [49] proposed a semi-supervised learning based tag recommendation approach, *SemiTagRec*, for Docker repositories.

2.3.5 Cross Ecosystem Comparison

Decan et al. [50] studied the evolution of huge package dependency networks in three ecosystems, npm, CRAN, RubyGems. Among others, they observed that, in combination with semantic versioning, dependency constraints can prevent packages from breaking due to dependency updates. In more recent work, they compared the evolution of the npm package dependency network with six other packaging ecosystems, and compared their growth, changeability, reusability and fragility over time [1] . The high and increasing number of transitive dependencies was found to be a major cause of fragility, suggesting

the need for better dependency management tools and policies.

Kula et al. [51] proposed a model named the Software Universe Graph (SUG) Model as a structured abstraction of the evolution of software systems and their library dependencies over time. To show the usefulness of their model, they performed an empirical study using 6,374 Maven artifacts and over 6,509 CRAN packages mined from their real-world ecosystems. Their visualizations of the model on different aspects (e.g., *library coexistence pairings* and *dependents diffusion*) showed popularity, adoption and diffusion patterns within each software ecosystem. Their results also showed that the Maven ecosystem is taking a more conservative approach to dependency updating than the CRAN ecosystem.

Vaidya et al. [52] presented a systematic study of security issues that plague open source language-based ecosystems such as the npm and PyPI ecosystems. Focusing mainly on malicious packages and how they are impacted based on the unique characteristics of each programming language ecosystem, they found that fully automated detection of malicious packages is likely to be unfeasible. However, tools and metrics that help developers assess the risk of including external dependencies would go a long way toward preventing attacks.

Bogart et al. [9] performed multiple case studies of three software ecosystems with different tooling and philosophies toward change, Eclipse, R/CRAN, and Node.js/npm, to understand how developers make decisions about change and change-related costs and what practices, tooling, and policies are used. They found that the three ecosystems differ significantly in their practices, policies, etc. As a summary, they reported that in Eclipse, you should not break an API, in R/CRAN you reach out to affected downstream developers, while in Node.js/npm, you increase the major version number if your update has breaking changes.

Later on, the same researchers conducted a survey about shared values and practices among over 2,000 developers in 18 ecosystems [6]. One of their observations was that maintainers generally try to perform breaking changes only rarely. Most developers, across all ecosystems, report less than one breaking change a year. It is also generally common to bundle multiple changes together to avoid disruptions for their users. They also observed that the frequency of breaking changes is higher in some ecosystems (npm, Rust) than others (Perl, CRAN, Eclipse).

2.4 Software Vulnerability Management

Decan et al. [53] carried out an empirical analysis of security vulnerabilities in the npm ecosystem by analyzing how and when these vulnerabilities are discovered and fixed, and to which extent they affect other package releases in the ecosystem in presence of dependencies. They observed that it often takes a long time to discover vulnerabilities since

their introduction. A non-negligible proportion of vulnerabilities (15%) are considered to be risky since they are either fixed after public announcement of the vulnerability, or not fixed at all. They found that the presence of package dependency constraints plays an important role in not fixing vulnerabilities, mainly because the imposed dependency constraints prevent fixes to be installed.

Zimmermann et al. [54] studied security risks for users of npm by systematically analyzing dependencies between packages, the maintainers responsible for these packages, and publicly reported security issues. They found that individual packages could impact large parts of the entire ecosystem. They also observed that a very small number of maintainer accounts could be used to inject malicious code into thousands of npm packages, a problem that has been increasing over time. To reduce the risk of having more and more vulnerable packages, they reported that solutions include trusted maintainers and code vetting process for selected packages.

Shu et al. [55] performed a generic large scale study on the state of security vulnerabilities in both community and official Docker Hub repositories. They proposed the *Docker Image Vulnerability Analysis (DIVA)* framework to automatically discover, download, and analyze Docker images for security vulnerabilities. They studied a set of 356,218 images and observed that both official and community repositories contain on average more than 180 vulnerabilities; many images had not been updated for hundreds of days, demonstrating a strong need for more analysis and systematic methods of studying the content of Docker containers.

Cadariu et al. [56] presented the Vulnerability Alert Service (VAS), a tool-supported process to track known vulnerabilities in software systems throughout their life cycle. To demonstrate its usefulness, they evaluated it in the context of external software product quality monitoring provided by the Software Improvement Group, a software advisory company based in Amsterdam, Netherlands. Besides showing the usefulness of the tool, they empirically reported that using components with known security vulnerabilities is common in practice in proprietary software systems. They also provided a description of a process to handle alerts addressing security concerns.

2.5 Software Outdatedness Management

With the presence of large numbers of software packages and versions in package repositories, developers are often ignoring or delaying to update their software dependencies, including tools and other packages that they depend on. The consequences of neglecting to update outdated dependencies can be risky, as dependencies can suffer from numerous security issues and bugs. For this reason, researchers have conducted multiple studies on software outdatedness.

Kula et al. [7] investigated the latency in adopting the latest library release for thousands of Java libraries. They found that maintainers are less likely to adopt the latest releases at the beginning of a project; and Maven libraries are becoming more inclined to adopt the latest releases when introducing new libraries. In a more recent work, the same researchers [57] empirically studied library migration for more than 4,600 GitHub projects and 2,700 library dependencies. They observed that four out of five of the studied projects keep their outdated dependencies. Through a survey with project maintainers, they discovered that the large majority of them were unaware of such outdated dependencies.

Several researchers found that outdated dependencies are a potential source of security vulnerabilities. Cox et al. [21] analyzed 75 Java projects that manage their dependencies through Maven and introduced different metrics to quantify their use of recent versions of dependencies. They observed that systems using outdated dependencies were four times more likely to have security issues and backward incompatibilities than systems that are up-to-date.

Lauinger et al. [58] studied the client-side use of JavaScript libraries. They found that "the time lag behind the newest release of a library is measured in the order of years" and that this is a major source of known vulnerabilities in websites using these libraries. They also observed that *"libraries included transitively [...] are more likely to be vulnerable"*.

Mirhosseini et al. [59] evaluated two mechanisms, automated pull requests and GitHub badges generated by dependency watchers like David[2], in order to see if they can encourage and motivate developers to upgrade software dependencies. To do so, they introduced a metric to quantify the lag before npm packages update their dependency constraints and studied how this metric changes in projects that make use of automated pull requests and badges. They also surveyed developers about their preferences of similar tool support. They found that both mechanisms can help in reducing such lag, however automated pull requests can encourage developers to update dependencies quicker and at a higher rate than badges.

2.6 Towards a Notion of Technical Lag

All of the above works illustrate that component reuse can suffer from outdated dependencies that may have important consequences such as backward incompatibilities and security vulnerabilities, often without even being aware of them. This highlights the need for a measure that quantifies how outdated reused software is. González-Barahona et al. [60] introduced the concept of *technical lag*, to measure how outdated deployed software systems are. They suggested many ways in which technical lag can be implemented or

[2]https://david-dm.org/

computed and could be used when deciding about upgrading software in production. For example, lines of source code between ideal and deployed components, or the number of commits of difference between them. They also illustrated the evolution of technical lag for some specific packages in the Debian Linux distribution.

This thesis is inspired by, and builds further on, the work of González-Barahona et al. [60]. More specifically, we extend the concept of the technical lag and formalise it in a generic framework. To validate the framework, we carry out interviews and surveys with software practitioners and we perform empirical studies to analyze the technical lag for multiple case studies. Then, in order to support developers, we develop tools that compute their software project's technical lag.

Chapter 3

A Preliminary Analysis of Software Library Usage and Evolution

---

Software systems are commonly implemented on top of frameworks and rely on external libraries to reuse complex functionality and increase productivity [61]. In order to understand how a particular library is used, software practitioners can rely on how the library has been used in other projects, how the library evolves over time, and when and why other projects have updated to a new library version.

This chapter presents an exploratory study on the use of a popular category of Java libraries. Such analysis helps us to figure out the common practices of library use, which will pave the way for us to do more extensive research.

The content of this chapter is mainly based on our previous publications in the SANER 2017 and SATToSE 2017 proceedings [62, 63].

3.1 Introduction

Software development practices have evolved quite a lot since the early days of programming. Most software projects today, especially in the open source software community, are using distributed and agile development practices. In addition, they heavily rely on reusing external software libraries to realize part of their functionality, rather than needing to implement these functionalities themselves.

A concrete example of a very frequently used category of software libraries is testing-related libraries. According to a blog post [64], testing libraries (including, JUnit, TestNG and Spring), matching libraries (Hamcrest and the more recent AssertJ) and to a lesser extent mocking libraries (e.g., Mockito, EasyMock and their PowerMock extension) are among the most popular Java libraries on GitHub.

Software development projects frequently rely on testing-related libraries to test the functionality of the software product automatically and efficiently. Many such libraries are available in different packaging ecosystems for different programming languages, and developers face a hard time deciding which (versions of) libraries are most appropriate for their project, when to update or migrate to a competing library.

In this chapter, we present our first exploration on the use of testing-related software libraries. Such analysis will pave the way for us to understand how libraries and their versions are used and how they are being replaced by other competing libraries or new released versions. To do so, we empirically analyzed the use of twelve popular testing-related libraries in 4,532 open source Java projects hosted on GitHub and use Maven for their automatic build. Later on, we focus on eight of these libraries, namely those that were found to be the most popular. We studied how frequently specific libraries are used over time. We also identified if and when library usages are updated or replaced by competing ones during a project's lifetime.

Our results showed that only a small proportion of software projects have migrated from the library they use to another competing library, while most of the projects tend to stick to only one library, and update its version from time to time.

3.2 Method

We studied open source projects extracted from GitHub. We selected Java projects because Java is the most popular programming language according to the TIOBE index[1]. We selected GitHub as a data source because we need full access to the source code history in order to carry out our analysis, and because it's the largest host of Java source code (containing over $2.4M$ Java repositories).

[1] `http://www.tiobe.com/tiobe-index/` (January 2017)

We decided to study the most popular testing libraries based on their number of dependents in the Maven Central Repository[2]: JUnit, TestNG, Spring test, Arquillian and Spock framework. We added the Hamcrest and AssertJ libraries that are often used as matching frameworks to facilitate writing more complex tests by creating customized assertions. We also considered the most used mocking libraries in Maven[3]: Mockito, EasyMock, PowerMock, JMock and JMockit. Libraries for non-functional testing (such as UI testing, performance testing, acceptance testing, load testing, etc.) were excluded, but could be considered very easily in a similar study.

We began with the GitHub Java Corpus of 14,807 open source Java projects extracted by Allamanis and Sutton [65] and obtained from Github Archive[4]. We removed 1,748 projects that were no longer available on GitHub. Based on popularity measured by number of stars, we added 7,629 popular GitHub projects satisfying the same filters as those applied by the GitHub Java Corpus [65], in order to have a larger corpus of projects that make use of testing-related libraries. More specifically, we ignored projects that were never forked or that were forks of other projects. This filtering reduces the risk of obtaining results statistically biased by groups of strongly related individuals in the considered project population.

This led us to 20,688 projects as potential candidates for our empirical analysis. In September 2016, we created a local clone of the GitHub repository for each of them. We further restricted ourselves to those projects relying on Maven, in order to be able to identify project dependencies defined in Project Object Model files (pom.xml). We also restricted ourselves to projects with an active lifetime of at least two years, which seems an acceptable minimal duration for detecting potential migrations of library usage during the projects' lifetimes. This lead us to 6,424 remaining projects.

We analyzed monthly snapshots of each considered project, by looking at the import statements in each Java file of the project. We found 4,532 Java projects that used at least one of the considered Java libraries. For these projects, we analysed in total 125,580 commits, 10,033,726 Java source code files and 31,264,586 import statements related to testing-related libraries. Table 3.1 presents descriptive statistics of the final corpus of projects we used.

3.3 Research Questions

The research methodology and questions in this analysis are similar to the ones presented by Goeminne et al. [66, 67], who focused on the evolution of Java database access libraries. More specifically, we focus on the following research questions:

[2] `https://mvnrepository.com/open-source/testing-frameworks`
[3] `https://mvnrepository.com/open-source/mocking`
[4] `https://www.githubarchive.org/`

Table 3.1: Descriptive statistics of the considered project corpus

Number of	Value
Projects	4,532
Commits	125K
Source files	10M
Import statements	31M

RQ_1: Which are the most frequently used testing-related libraries? With this research question, we count and compare the number of projects that make use of each one of the considered libraries.

RQ_2: When are libraries introduced in a project's lifetime? With this research question, we compute the time usually needed between the first commit and the commit in which the testing library is introduced.

RQ_3: Which libraries are used simultaneously in projects? Projects from the same category of libraries are not usually used together since they might provide the same functionality. With this research question, we identify the software libraries that have been used together within the same software project.

RQ_4: How frequently are libraries used over time? With time passing, good software libraries gain more visibility and popularity. With this research question, we aim to know which testing-related libraries are getting more dependents by time.

RQ_5: Do projects migrate to competing libraries? Projects from the same category could be considered as competitors. With this research question, we investigate on the software libraries that have been replaced by other libraries from the same category.

RQ_6: How long does it take before projects update their used libraries? With time passing, new versions of software libraries are being released. With this research question, we compute and analyze the time needed (delay) before a software library is updated to a new released version.

3.4 Empirical Evaluation

RQ_1: Which are the most frequently used testing-related libraries?

Figure 3.1 shows the relative use frequency of each of the 4,532 Java projects in our corpus that used at least one of the considered libraries at least once during its lifetime (i.e., at least one Java file imported that library in at least one commit). We observe a high imbalance. JUnit was by far the most popular: of all considered libraries, projects are very likely (97%) to use JUnit. In comparison, the competing TestNG library is used in

only 11.9% of the projects. Of the considered mocking libraries, Mockito was by far the most used library, being used by 31.3% of all projects. EasyMock and PowerMock are considerably less popular, corresponding to 9.3% and 4.7% of all projects, respectively. The matching library Hamcrest (22.5%) is considerably more popular than its competitor AssertJ, which can be explained by the fact that AssertJ is much more recent. However, since its first release in March 2013 its popularity is increasing.

Overall, our findings partially agree with the previous research [64], in the sense that testing libraries are dominant but mocking libraries are still more popular than matching libraries.

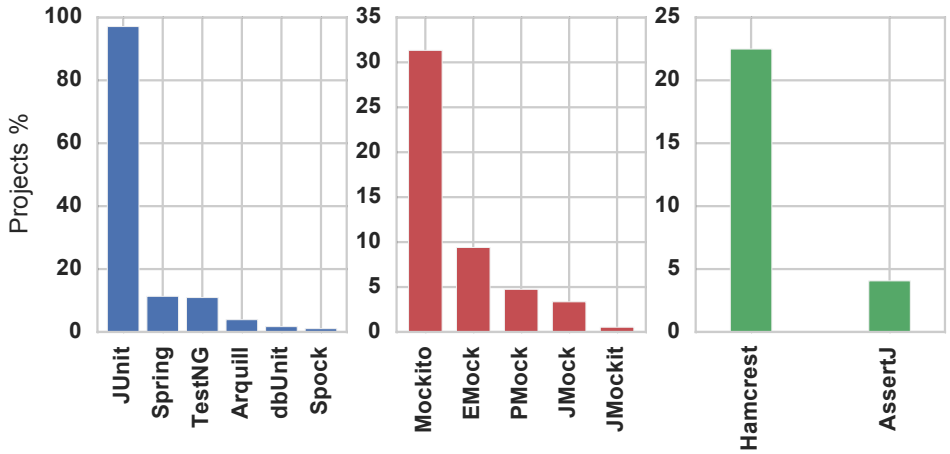

Figure 3.1: Percentage of projects in which a given library is used at least once during its lifetime. Testing libraries are shown in blue, matching libraries in green, and mocking libraries in red.

We decided to focus on those eight libraries that are used by a sufficient number of projects in our corpus. Therefore, we excluded the 5 least frequent libraries in Figure 3.1: Arquillian and Spock, two testing frameworks for writing integration tests; dbUnit, a unit testing framework for database-driven projects; and two less popular mocking libraries JMock and JMockit.

RQ_2: When are libraries introduced in a project's lifetime?

For each project, we analysed after how long each used library got introduced (i.e., the time interval between the first project commit and the first commit where at least one Java project file imported the considered library). Figure 3.2 shows that the considered libraries are introduced early. 56% of all projects started using at least one of these libraries as early as their first commit on GitHub. This could be explained by the fact that these projects were already in development before coming to GitHub, or because they follow a test-driven development process, implying that tests are introduced very early in the

project's lifetime.

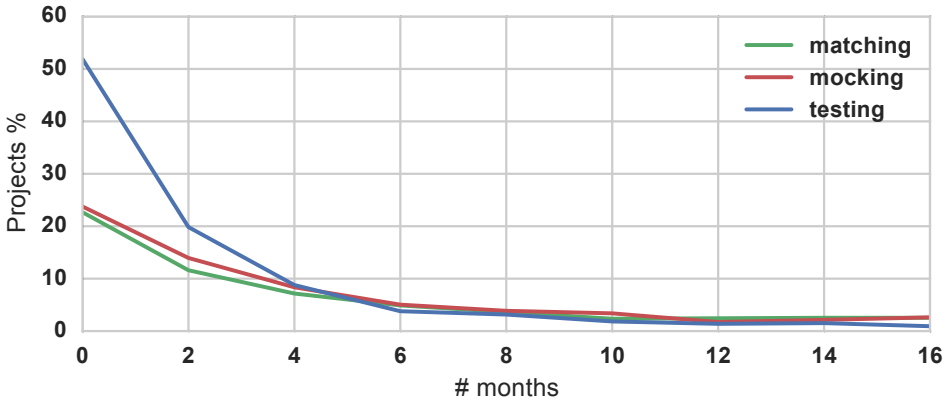

Figure 3.2: Number of months between the first commit and the commit in which one of the testing-related libraries was introduced in the project.

Unsurprisingly, we observed that JUnit and TestNG are the first libraries to be introduced in respectively 88% and 57% of the projects in which they occur with other libraries. AssertJ was never found to be introduced first, probably because it is a much more recent library.

Table 3.2 compares the relative order of introduction of the different testing-related library categories. Because of their role as the core of the testing process, testing libraries tend to be introduced before matching (71.6%) and before mocking (57.9%) libraries that are usually used around the testing libraries. In 41.4% cases, mocking libraries are introduced together (within a deviation of one month) with testing libraries, and in 50.3% cases they are introduced before matching libraries.

Table 3.2: Introduction order of testing-related library categories

A→B	#projects	A before B	A after B	A = B
testing→matching	983	**71.6%**	1%	27.4%
testing→mocking	1443	**57.9%**	0.7%	41.4%
mocking→matching	523	**50.3%**	21.8%	27.9%

RQ_3: Which libraries are used simultaneously in projects?

We analyzed if projects use different libraries over their lifetime. The results are shown using Venn diagrams in Figure 3.3. JUnit occurs as the *only* testing library in 61.3% of all projects (2,777 in total) and 97% of all considered projects have used JUnit at least once. TestNG is used as the *only* testing library much less frequently, in 2.3% (106) of all projects using at least one of the considered libraries, while 11% of all projects have used TestNG

at least once in their lifetime. All projects that used either Hamcrest, Spring or AssertJ also used at least one other library during their lifetime. In the overwhelming majority of the cases, Hamcrest and AssertJ are used in projects that have used JUnit in their lifetime.

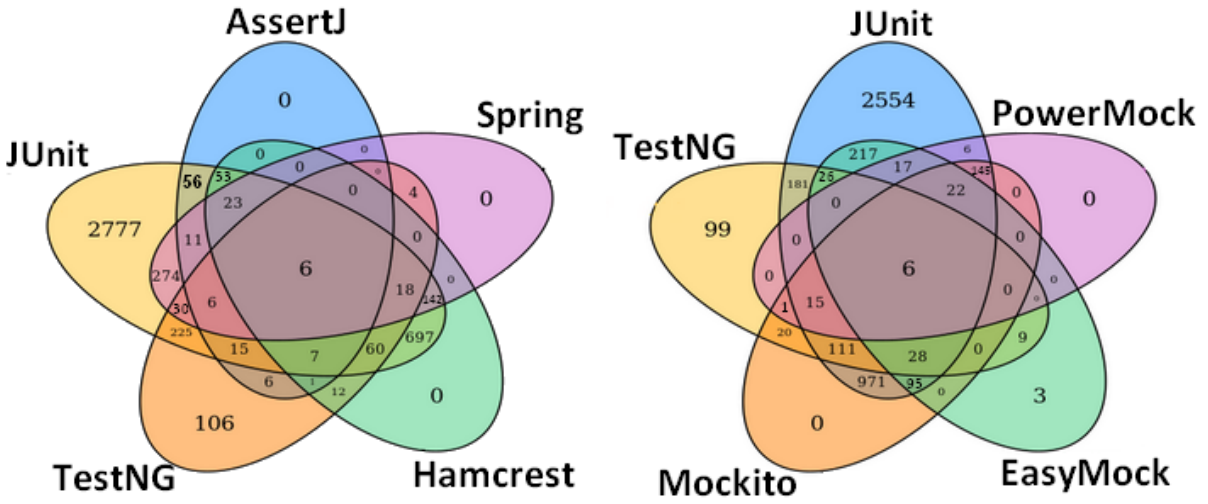

Figure 3.3: Number of projects using different testing-related libraries at least once during their lifetime (not necessarily simultaneously).

Of all projects that used at least two of the considered libraries during their lifetime, we computed which pairs of libraries were actually being used *simultaneously* in a specific project snapshot (not necessarily within the same Java files). Table 3.3 shows the percentage of projects using a library A that also used library B simultaneously at least once (i.e., in at least one commit) during their lifetime.

Nearly all projects that use Hamcrest, AssertJ, Spring, Mockito or PowerMock also use JUnit (values >94%). Those that don't, tend to use TestNG instead. Unsurprisingly, JUnit is used much less frequently with its competitor TestNG: projects that use JUnit rarely use TestNG (7%). However, more than 3 out of 5 projects that use TestNG also used JUnit simultaneously at some point (64%). This may indicate that projects using TestNG actually migrated away from JUnit somewhere during their lifetime. RQ_5 studies this library migration phenomenon in more detail.

Projects using Hamcrest rarely use AssertJ (7.36%). The other way round, more than 2 out of 5 projects that use AssertJ also use Hamcrest simultaneously (40.8%). This may be a sign that many projects that use Hamcrest are in the process of migrating to AssertJ.

For mocking libraries, PowerMock is mostly used as extension of Mockito (86.5%), and much less as an extension of EasyMock (19.1%). As expected, projects that use Mockito rarely use EasyMock (8.23%). However, more than one out of four projects that use EasyMock also use Mockito (27.5%). This is relatively a high percentage if we consider that EasyMock and Mockito are competitors.

Table 3.3: The percentage of projects using library A (rows) that also use library B (columns) simultaneously at least once during their lifetime.

	JUnit	TestNG	Spring	Hamcrest	AssertJ	Mockito	EasyMock	PowerMock
JUnit	100%	7%	12%	23%	4%	32%	9%	5%
TestNG	64%	100%	13%	18%	6%	33%	12%	4%
Spring	99%	12%	100%	37%	8%	53%	18%	10%
Hamcrest	99%	9%	18%	100%	7%	59%	10%	10%
AssertJ	95%	16%	23%	41%	100%	61%	11%	15%
Mockito	98%	11%	19%	42%	8%	100%	8%	13%
EasyMock	77%	14%	22%	24%	5%	28%	100%	10%
PowerMock	99%	9%	23%	47%	13%	87%	19%	100%

RQ_4: How frequently are libraries used over time?

Figure 3.4 (top) shows the monthly evolution over time of the usage of testing and matching libraries. The proportion of projects using JUnit is decreasing (but remains very high), while TestNG and Spring have a stable (but low) proportion of projects using them. The proportional usage of Hamcrest and AssertJ is increasing over time.

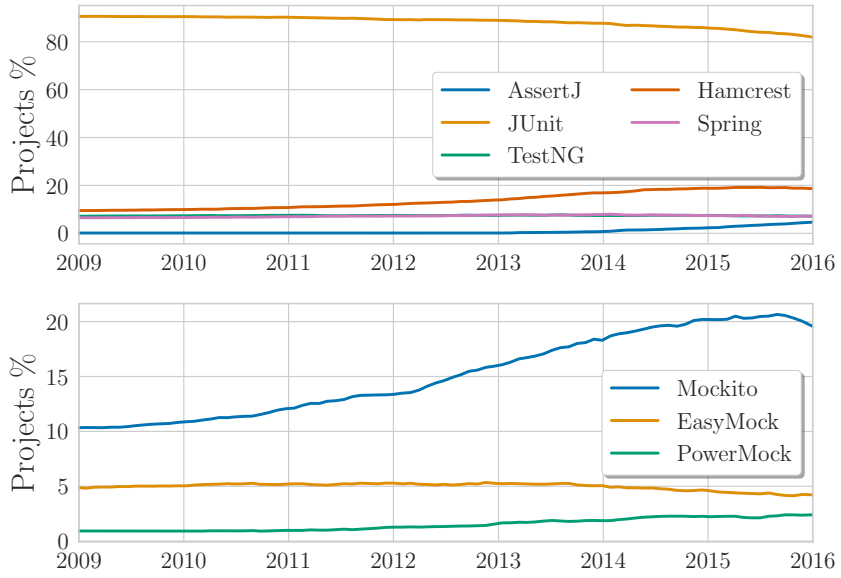

Figure 3.4: Monthly evolution of the (proportion of) Java projects using testing-related libraries.

For mocking libraries, Figure 3.4 (bottom) shows that the proportional usage of Mockito and PowerMock has remarkably increased, whereas the usage of EasyMock is slightly declining over time.

RQ_5: Do projects migrate to competing libraries?

We analyzed all considered libraries used by each project's Java files once every month, to determine if a project p permanently switches from library l_1 to library l_2 during its observed lifetime. We define a ***permanent*** *library migration* from l_1 to l_2 in p, if $\exists$ time t_1 where project p uses library l_1 and does not use l_2, while $\exists\ t_2 > t_1$ such that $\forall\ t \geq t_2$ project p uses l_2 but does not use l_1.

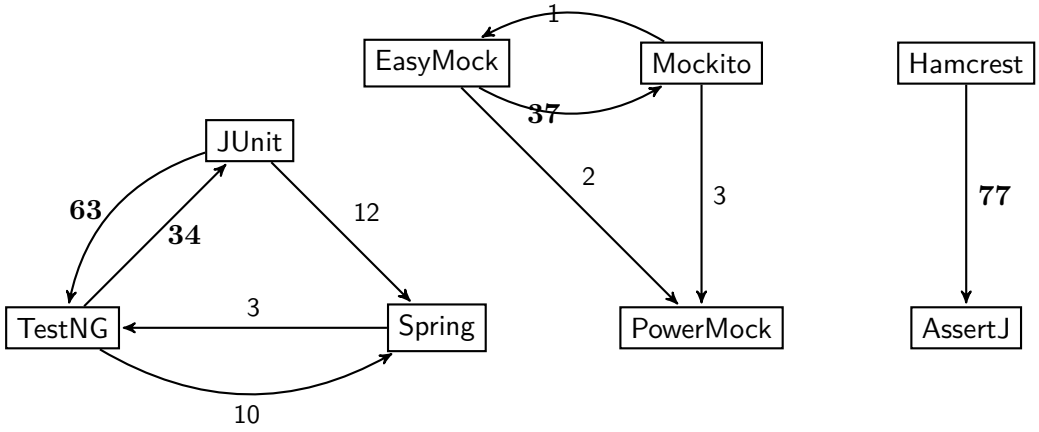

Figure 3.5: Number of migrations observed between testing libraries.

The migration graph of Figure 3.5 visualizes all permanent migrations for the considered libraries, grouped by category. We observed a high number of permanent migrations (63) from JUnit to TestNG, while only 34 projects permanently migrated from TestNG to JUnit. We also observed that 50 out of these 97 projects didn't involve a transition phase (during which both libraries are used simultaneously) for the migration. Only 3 migrations were performed from Spring to TestNG and it is probably because projects that use the Spring test module are using other modules of Spring framework that provide different functionalities, so they are kind of "forced" to stick to this framework.

We observed the highest number of permanent migrations from Hamcrest to AssertJ (77) even if these two libraries were used together in only 90 (i.e., 1.98%) of all considered projects. No permanent migration was observed from AssertJ to Hamcrest. For the mocking libraries, we observe most migrations (37) from EasyMock to Mockito, most likely because it offers more functionality.

We also found cases (not shown in the graph) of ***temporary*** *library migrations*. A temporary library migration from l_1 to l_2 is a transition of migrations from l_1 to l_2 and then from l_2 to l_1. Nine projects temporarily migrated from JUnit to TestNG and returned to JUnit after some time. Four other projects performed the opposite temporary migration. Four projects migrated from Hamcrest to AssertJ and returned to using Hamcrest.

RQ_6: How long does it take before projects update their used libraries?

To answer this question, for each project, for the first snapshot of each month of its lifetime, we extracted the metadata available on the Maven POM file and we identified the used versions of the considered testing-related dependencies.
Figure 3.6 shows the latency to upgrade to a new released library version. We observe that projects that make use of TestNG, PowerMock and AssertJ update them faster. More than 20% of these projects upgraded their testing-related dependency in the first months, where the other ones took longer before updating to a new released version.

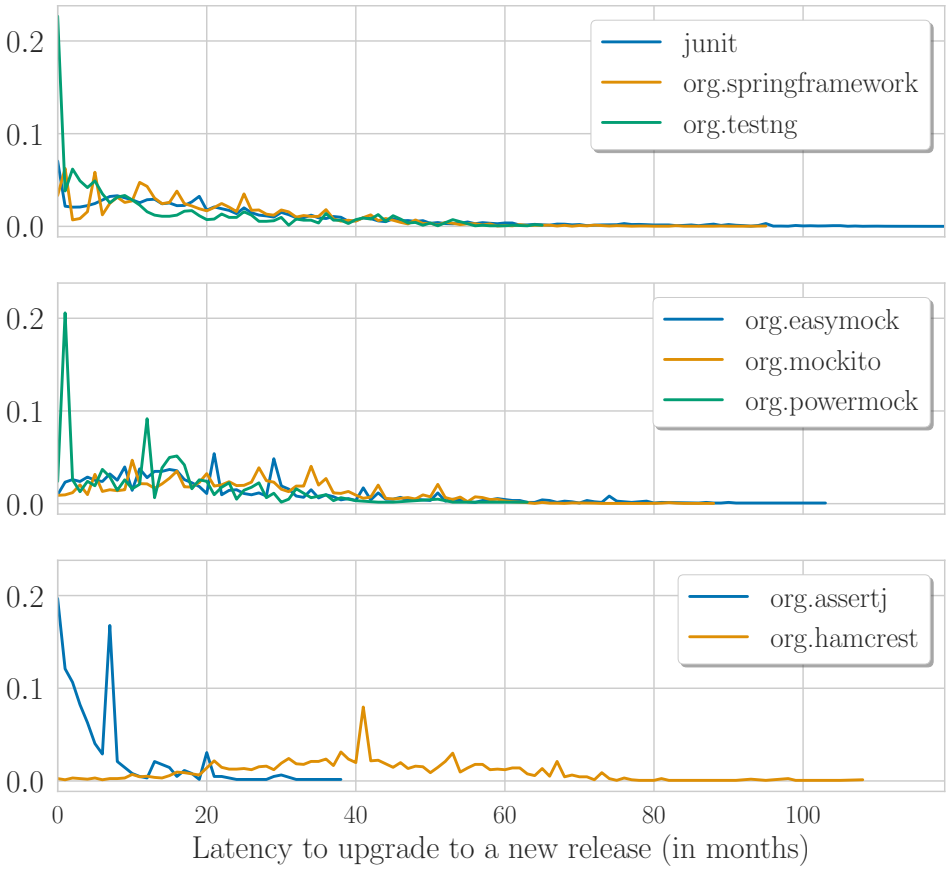

Figure 3.6: The proportion of projections that had a latency to upgrade to a new library release.

Moreover, we found that some versions of the same library are quickly adopted than others. For example, in Figure 3.7, we observe that projects took more time to adopt the first releases of the major version JUnit 4.0 than adopting the latest minor releases of the same major version. This could be explained by the fear of the incompatible API changes that are made on the new major release. We also observed that most of the versions of testing libraries tend to be used for less than two years before being upgraded or before

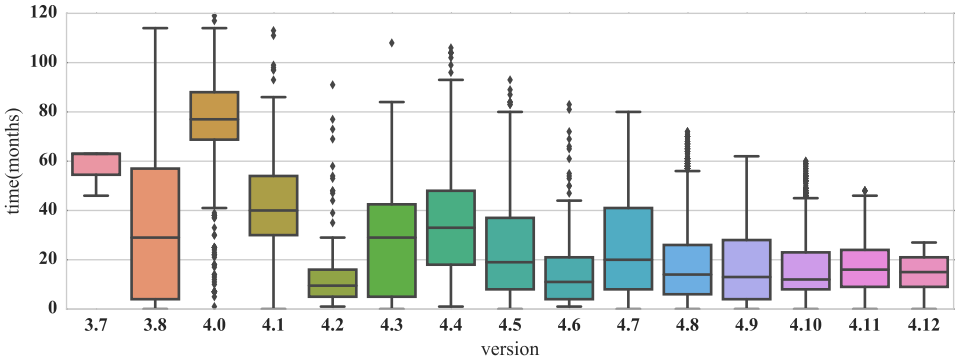

Figure 3.7: Latency to upgrade to a new released JUnit version in months.

switching to another library version.

3.5 Discussion and Limitations

With this first study, we aimed to explore the evolution of how testing-related libraries are used within software projects. We observed that library usage is imbalanced between the set of libraries that we considered for the study. We also found that more than half of the projects started using the testing-related libraries within their first commit. This shows the importance of external libraries in the development of different software projects.

We observed that testing-related competing libraries are not usually used together within the same project. In fact, we found that only a small proportion (5%) of library migrations occurred in the development lifetime of the analyzed Java projects. This suggests that developers do not often use or switch between different testing-related libraries that provide the same functionality, instead they only stick to one library. For this reason, we studied how long it takes before a used library is updated.

Focusing on the most used testing-related libraries, we noticed that major version releases take much more time to be adopted. This is not surprising since it may require more effort and cost to update to new major releases that are supposed to have incompatible break changes by default [26].

Our research suffers from the same threats as other research relying on GitHub [68]. Our results may not be generalisable to non-Java projects or to closed-source industrial projects that are typically subject to more restricted development rules. While we studied the usage of eight testing-related Java libraries, the proposed methodology is applicable to other categories of libraries as well. Our results may, however, be biased by the fact that we have excluded projects with a lifetime of less than two years, as well as projects that are no longer available in GitHub.

In our approach we assume that a library is being used by a Java project if one of the project files contains specific import statements pertaining to that library. This approach may lead to false positives, since imported classes and interfaces are not necessarily used in the source code.

3.6 Conclusion

In this chapter, we carried out a preliminary analysis on the use of software libraries. More specifically, we studied the use of eight popular testing, matching and mocking libraries in a large corpus of GitHub-hosted Java projects. We observed that some libraries were found to complement or reinforce one another (e.g., PowerMock which extends either Mockito or EasyMock) while others are in competition (e.g., JUnit versus TestNG, Mockito versus EasyMock, Hamcrest versus AssertJ).

We found that 5% of the considered projects are subject to library migrations, in which a project replaces one of its used libraries by another. These migrations were mainly permanent. For example, projects tend to migrate from EasyMock to Mockito and from Hamcrest to AssertJ, but not the other way round. In a limited number of cases, the library migrations were temporary, with the opposite migration being observed later on in the project's lifetime. We also analyzed the delay of the adoption of new available library releases. For the case of JUnit, we noticed that it takes more than one year to update to newer versions, and in some cases it may take years.

Our observations about when and how projects perform library updating is promising, but requires a more in-depth analysis. Updating to a new major library release may imply significant changes, potentially leading to an increased migration away from this particular library (or version). In fact, we observed similar cases, e.g., the project livetribe-slp first used JUnit 3, then migrated to TestNG, and then returned to using JUnit 4.

Chapter 4

A Framework for Technical Lag

In the previous chapter, we showed that developers of software projects have different practices regarding the use of software library versions. One of these practices is that developers tend not to update the version of their libraries. This can be harmful for the whole software project, since outdated libraries may contain unfixed bugs and vulnerabilities. Moreover, they may miss new functionalities and features that are included in the recent library releases.

This chapter presents a generic model of technical lag to compute how outdated deployed software components are compared to their ideal available versions. The content of this chapter is mainly based on our previous publications in the ICSR 2018 proceedings and Journal of Software Evolution and Process 2019 [69, 70].

4.1 Introduction

To facilitate software reuse, many online platforms have been created for distributions of operating systems (e.g., Linux distributions such as Debian and Ubuntu) and the most prominent programming languages (e.g., npm for *JavaScript*, Maven for Java, RubyGems for *Ruby*), totaling millions of reusable software component releases.

The mechanism of semantic versioning allows package maintainers to attach some semantics to their package releases (e.g., whether it is a major release, minor release or patch). By doing so, developers that depend on such packages can better assess the risk of facing backward incompatible changes when upgrading their dependency to a new release. Unfortunately, even if semantic versioning is recommended by many package management tools, this policy is not always well-respected by package maintainers [26].

Assessing the problem of technical lag is important at the level of individual components. It becomes even more relevant and problematic when large collections of components are involved, such as when an application depending on many components is deployed. For example, if a component imposes version constraints that are too strict on its direct dependencies, it may suffer from higher technical lag. The impact of this lag becomes even more important if transitive dependencies (a component depending on another component, itself depending on other components) are taken into account. However, developers can only act on how they specify direct dependencies for their components, which means that indirect dependencies are, to some extent, beyond their control.

Therefore, a generic model of the technical lag concept is needed to capture all cases where a deployed software component may be outdated. Since technical lag can be measured in different ways, the mechanism of semantic versioning can help us in defining a metric that captures the difference between two version releases, based on their ordered version numbers.

In this chapter, we propose a *formal framework for measuring technical lag*. This framework can be instantiated to specific reusable component repositories, using different ways of measuring lag, in order to study how dependencies and other characteristics of reusable components affect how far away deployments of applications are from the "ideal" deployment.

4.2 Technical Lag Explained

Ideally, software systems would depend on the most "ideal" available version of their dependencies, thus benefiting from the new features and bug fixes. In practice, however, software projects and packages have a certain lag because many developers choose not to update certain dependencies ("if it ain't broke, don't fix it"). Moreover, in many cases

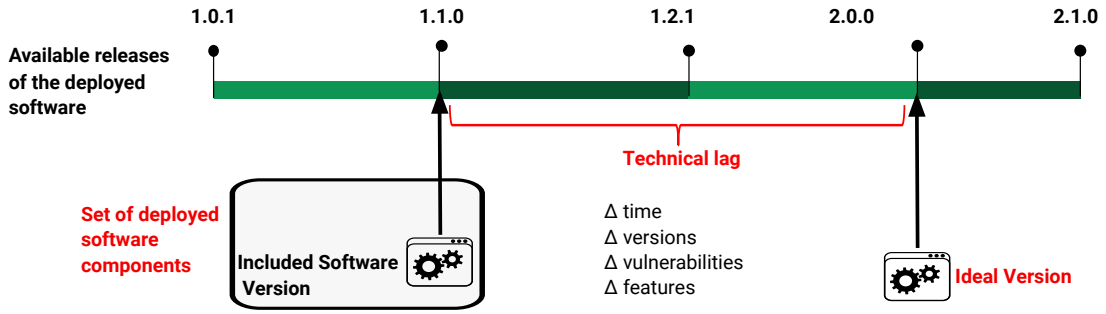

Figure 4.1: Illustration of the concept of technical lag

developers choose not to update because new major releases may include new functionality that is not needed. Dietrich et al. [71] found that 75% of all version upgrades in 109 Java programs are not backwards compatible, but only few are actually affected by the incompatible changes.

The concept of technical lag is related to, but different from, the metaphor of technical debt [72, 73]. Technical debt refers to the qualitative difference between code "as it should be" and code "as it is". Technical lag refers to the increasing lag between the ideal available upstream versions of packages used by a software system and those actually used in the deployed system. Technical lag can be expressed in many ways. Figure 4.1 illustrates the concept of technical lag in an imaginary case. Suppose that there is a software system that deploys a set of software components. For one of this components, the system deploys its version 1.1.0 while there is an available "ideal" version 2.0.0 (it could be the most secure, the most stable, etc). The technical lag is the difference between these two versions (1.1.0 and 2.0.0), and it could be measured in many ways, e.g., in terms of a difference in time, versions, vulnerabilities, features, etc.
Note that the technical lag could be aggregated for the whole set of deployed components.

4.3 Technical Lag Example

The main purpose of the example is to illustrate how and why component releases can become outdated over time, and to illustrate different ways to quantify the extent to which component releases can be outdated.

In the npm ecosystem, developers are invited to follow the semantic versioning specifications by increasing their library version number each time they release a new library version. Each version increase corresponds to patch fixes, added functionalities or incompatible changes. Thus, in npm, the latest available version is supposed to be the highest version and the closest one to the ideal library version that can be used.

Consider the actual software package **youtube-player** taken from the npm repository.

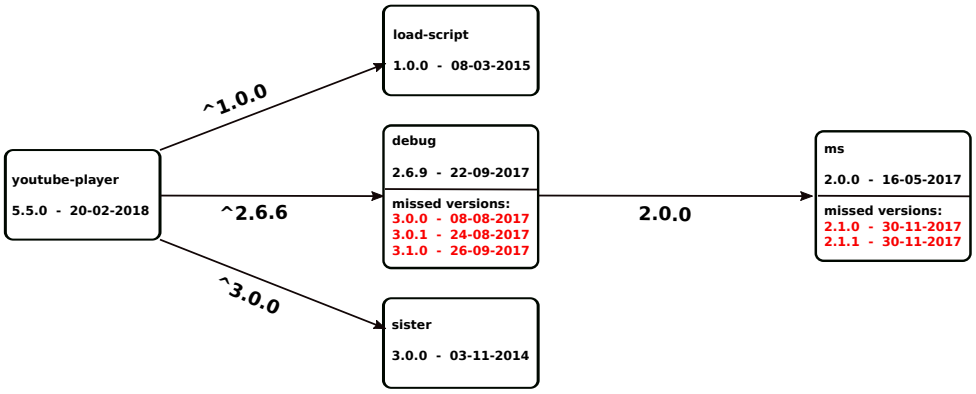

Figure 4.2: Transitive dependencies of version **5.5.0** of the **youtube-player** package at its release date of 20-02-2018.

Figure 4.2 shows the dependency tree of the package release **youtube-player 5.5.0** at its release date. Its direct and transitive dependencies are shown in black, and correspond to all other packages that will have a release installed when the user decides to run **npm install** for **youtube-player 5.5.0** at its release date. The exact release that will be installed for each dependency is determined by the dependency constraint, shown in Figure 4.2 on the edges of the tree. For example, the constraint ^**2.6.6** on **debug** specifies that the latest minor or patch release above version 2.6.6 will be selected upon installation. At the release date of **youtube-player 5.5.0**, constraint ^**2.6.6** on **debug** will select **debug 2.6.9** for installation. Figure 4.2 also shows some more recent releases in red. Even if they are more recent, these releases will not be selected for installation because they do not satisfy the dependency constraints. For example, **debug 3.0.0**, **3.0.1** and **3.1.0** are not accepted by the constraint ^**2.6.6**. It is because of such non-installable, more recent releases that **youtube-player 5.5.0** could be considered as outdated with respect to a more "ideal" situation were the highest release of each dependency is installed.

Technical lag as a time difference

The notion of technical lag aims to capture the difference (or "delta") between the current situation and an "ideal" one. For example, assuming that we measure the technical lag as a time difference, the lag induced by not using the latest and highest version of **debug** would be **4 days** (namely the delta between the release date of **debug 2.6.9** and **debug 3.1.0**). If we assume that the aggregated time lag of **youtube-player 5.5.0** is computed as the maximum of the lags induced by all its direct dependencies, the lag will remain 4 days, because the two other dependencies on packages (**sister** and **load-script**) are up-to-date. If we also take transitive dependencies into account, we need to consider the lag induced by package releases at deeper levels of the dependency tree as well. Package release **ms 2.0.0** induces a technical lag because the latest available release **ms 2.1.1** is

not accepted by the strict dependency constraint **2.0.0**. Because of this, **ms 2.0.0** has a technical lag of **167 days**, corresponding to the time delta between the release date of **ms 2.0.0** and **ms 2.1.1**. Consequently, the aggregated (maximum) time lag of the entire (transitive) dependency tree for **youtube-player 5.5.0** is **167 days**.

Technical lag as a version difference

Although the semver policy is not always respected by package maintainers [26], it provides a way of telling apart releases, and provides relevant information on which upgrades of dependent releases are more likely to cause breaking changes. We will therefore rely on semantic version numbers to measure the version lag between npm package releases.

Reconsidering our motivating example, the sequence of available package releases (at the release date of **youtube-player 5.5.0**) since **debug 2.6.9** is $[2.6.9, 3.0.0, 3.0.1, 3.1.0]$. This sequence can be used to count the number of missed updates between the selected one (**2.6.9**) and the highest available one (**3.1.0**), considering each version component separately. For this particular example, the version delta will be 1 major number (the change from version 2.6.9 to 3.0.0), 1 minor number (the change from version 3.0.1 to 3.1.0) and 1 patch number (the change from version 3.0.0 to 3.0.1). In a similar way, the version delta between the selected package release of **ms** (**2.0.0**) and the highest available one (**2.1.1**), based on the sequence $[2.0.0, 2.1.0, 2.1.1]$, will be 0 major, 1 minor (the change from 2.0.0 to 2.1.0) and 1 patch number (the change from 2.1.0 to 2.1.1). Based on the above, if we would compute the aggregated version lag of **youtube-player 5.5.0** as the sum of the version lags induced by all its transitive dependencies, we would obtain a lag of : (1 major, 1 minor, 1 patch) + (0 major, 1 minor, 1 patch) = **(1 major, 2 minors, 2 patches)**.

Note that we could alternatively compute the version lag by ignoring the patches and minors that are included in a missed major version. However, considering all updates can tell us more about how much effort has been done to maintain the package we use.

4.4 Qualitative Analysis

In order to assess the usefulness of the technical lag concept, we performed structured interviews with 5 software practitioners. Based on the interviews and on some informal discussions with software developers, we carried out an online survey with 17 software developers with the aim of receiving opinions about the characteristics of the most desirable software versions. This helped us to define the notion of "ideal" version that was required for carrying out a quantitative analysis on technical lag.

4.4.1 Semi-structured Interviews

Five semi-structured interviews were conducted with five software practitioners that we encountered during the *FOSDEM 2019*[1] event in Brussels, Belgium. All interviews, requiring approximately 20 minutes each, were recorded and transcribed for analysis. Transcription of all interviews can be found in Appendix-A.

Before starting the questionnaire, we explained the purpose of the interviews and our research goals, disclosed the public funding sources of our research and ensured that the information provided would be treated in a confidential way. In addition, we informed the participants about the estimated time required to complete the interview, and obtained their informed consent. Each interview was structured into four parts: 1) profile, 2) software characteristics, 3) updating process and 4) technical lag.

Each interview started with general information in order to profile the interviewee. All interviewees were highly educated (Master degree or higher) and experienced software engineers (e.g., the average number of years of experience was 10 years).

The second part was related to the software projects in which the participants were involved. Three of the participants were involved in the development of popular open source projects, one participant worked on an internal tool for a big IT company. The last participant worked as a development coach. All participants agreed that their project dependency management is important and a critical task. *"The dependency management is the hottest spot in our project"*, was noted by P5.

In the third part, we asked the participants about why and how they update their dependencies. We found that participants deal with their updating process in different ways. Sometimes they do the updating automatically by relying on tools that provide such service, sometimes manually, depending on the dependency they want to update. We also found that, depending on the importance of the dependency, they decide to update it or not. All participants agreed that not so important dependencies are pinned to "a fixed version", lagging behind with years sometimes. When asked about the reason behind updating dependencies, they responded that features and security are their first concerns. One participant (P1) elaborated saying: *"For security reasons, performance, novelty, just not to have all tools which are not used by anyone and not maintained anymore. In fact, I think that we should use a dependency that have many people working on it, and that has people enjoying working on it"*. We also noticed that participants working on industry or open source projects keep an eye on the outdatedness of their software. One participant that works on an internal tool said that he does not do that, simply because the tool is working.

[1]FOSDEM is a free event for software developers to meet, share ideas and collaborate. https://fosdem.org/2019/

We also asked the interviewees about the most important characteristics of the ideal software version they would like to use, and whether there is a guideline that helps them to decide whether to update or not. None of the interviewees said that there is such guideline. All of them agreed that the ideal software version would have a balance between level of security and new features. Up to this point, the interviewees were not aware of the meaning of the technical lag concept. We did not talk about it to avoid any bias in their responses.

After these questions, we explained the technical lag concept and asked the interviewees about their opinion about it. All interviewees were favorable towards the usefulness of this concept. We asked them about the version that they would consider as the ideal. In general, the answers that we got were a combination between the "bleeding edge" version (i.e., the latest version) in order to benefit from the latest features, and the "most secure" version in order to have as few vulnerabilities as possible. With respect to the most appropriate measurement to compute the technical lag, the responses were a combination between the number of missed versions, features and fixed vulnerabilities. P1 elaborated on how we can use the *changelog* file[2] as a good source to compute the difference between the used and the ideal software versions. Similarly, P5 noted *"I think there are two important metrics, features and vulnerabilities. They can sum up everything, they are almost like a reverse changlog"*.

Throughout the interviews, the importance of updating was stressed. However, participants acknowledged that a mix between what is missing (i.e., benefits) and the effort (i.e., cost) needed to update would be even better than only knowing what is missing. *"But also I think what would be interesting is to know how many people had problems in order to have the ideal update."*, was noted by P5.

4.4.2 Online Surveys

We carried out an online survey to complement the semi-structured interviews and to have a qualitative analysis about the characteristics of the ideal software version, which will be used to define the ideal version when quantitatively analyzing the technical lag.

The survey was designed using Google forms and shared on software developer user groups in Facebook. The survey followed established best practices [74]: prior to asking questions, we explained the purpose of the survey and we informed the participants about the estimated time required to complete the survey, which was 3 minutes. The list of questions started with demographic questions related to years of experience, technologies used and the participant roles in the software project that they are involved in. The list of the survey questions can be found in Appendix-B. We received 17 valid responses. The

[2]A changelog is a record of all notable changes made to a software project. Changelog usually includes records of changes such as bug and vulnerability fixes, new features, etc.

mean number of years of professional experience of the participants was 3 years, and all participants were highly educated (Master degree or higher).

For the question *"What could be the most appropriate (ideal) version of a software library to use?"* we gave the participants a list of ideal software library versions that we previously obtained through informal discussions and interviews with software practitioners and researchers (e.g., latest version, most secure, most tested, etc). The participants had the possibility to select multiple choices. Figure 4.3 shows the total number of answers for each choice. 14 respondents chose the *most stable* version as the ideal version of a software library that they would like to use. 9 participants chose the *latest version*, while only 4 respondents chose the *most secure*. Surprisingly, the *most documented* version received more (7) answers than the *most secure* software version (4 answers), possibly because the question related to choosing a release in general rather than selecting a more recent release for a package that is already in use.

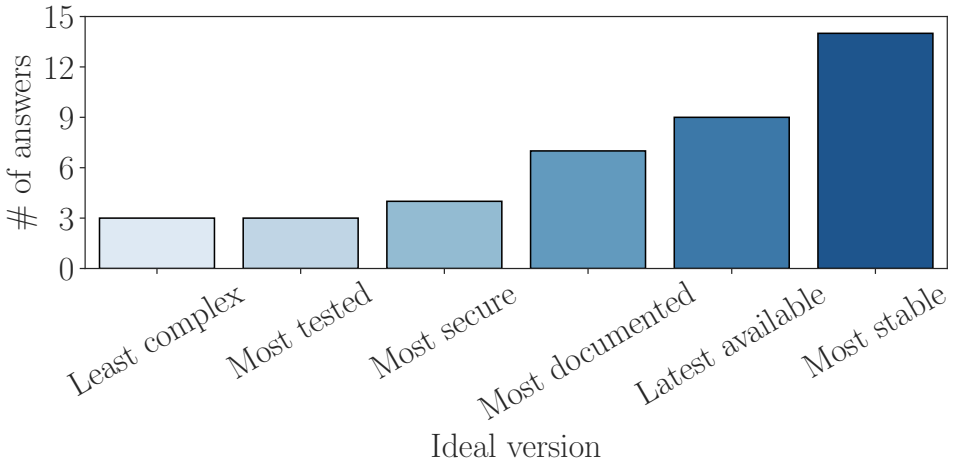

Figure 4.3: The list of the most desirable software versions ordered by number of participants that chose them.

From the interviews, surveys and informal discussions with software practitioners, we noticed that developers that work on code only, care less about their software project security, while it is the inverse for deployers. We also noticed that, in general, there is no unique ideal software version. Software developers or deployers prefer a combination of different characteristics, e.g., features and security. Thus, a technical lag framework should consider and support different measurement units.

4.5 A Formal Framework for Technical Lag

In sections 4.2 and 4.3, we have given examples to illustrate how to compute and aggregate technical lag, reflecting the extent to which a component release is outdated, in many

different ways. We have also shown in section 4.4 how software practitioners think about the concept of technical lag and its variants. It is the purpose of the technical lag framework, proposed in this section, to capture and formalize all these variations and future variants.

In order to measure technical lag in any given repository of reusable software components, we need a formal framework that abstracts the specifications of the various repositories. Refining the definition of Gonzalez-Barahona et al. [60], we define technical lag for a certain component release as the *difference* between that release and the *ideal* release, where *ideal* could be interpreted in different ways: most recent, most compatible, most stable, most secure, etc. Then, we can apply the concept to a collection of component releases, as the aggregated lag for all of them with respect to the ideal releases for those components. A specific situation where it is useful to apply this definition for a collection of releases is when we install a certain component release together with the collection of all its direct or transitive dependencies.

To formally capture the notion of technical lag in a generic way, we define a parameterised technical lag framework:

Definition 1. Technical lag framework

A *technical lag framework* is a tuple $\mathcal{F} = (\mathcal{C}, \mathcal{L}, \textbf{ideal}, \textbf{delta}, \textbf{agg})$ where

- $\mathcal{C}$ is a set of component releases.
- $\mathcal{L}$ is a set of possible lag values.
- $\textbf{ideal} : \mathcal{C} \to \mathcal{C}$ is a function returning the "most preferred" component release (according to the user desiring to deploy a component) over a given one.
- $\textbf{delta} : \mathcal{C} \times \mathcal{C} \to \mathcal{L}$ is a function computing the difference (in terms of lag induced) between a first component release and a second one.
- $\textbf{agg} : \mathbb{P}(\mathcal{L}) \to \mathcal{L}$ is a function aggregating the results of a set of lag values[3]. Typical examples of **agg** functions would be the sum, maximum, mean or median of a set of values.

Given a technical lag framework $\mathcal{F}$, we can formally define the *technical lag* induced by choosing a component release instead of the **ideal** (i.e., most preferred) release for that component. What this means may differ depending on the considered scenario (e.g., we may wish to create the most stable, most recent, or most secure deployment). If we use a component release instead of the ideal one, the induced technical lag (**techlag**) is the difference (**delta**) between both releases.

[3]The notation $\mathbb{P}(\mathcal{L})$, which can alternative be written as $2^{\mathcal{L}}$, represents the powerset of $\mathcal{L}$, i.e., the set of all possible subsets of $\mathcal{L}$.

Definition 2. Technical lag

$$\mathbf{techlag}_{\mathcal{F}} : \mathcal{C} \rightarrow \mathcal{L} : c \rightarrow \mathbf{delta}(c, \mathbf{ideal}(c))$$

Similarly, we can define the *aggregated technical lag* induced by a set of component releases $D \subseteq \mathcal{C}$.

Definition 3. Aggregated technical lag

$$\mathbf{agglag}_{\mathcal{F}} : \mathbb{P}(\mathcal{C}) \rightarrow \mathcal{L} : D \rightarrow \mathbf{agg}(\{\mathbf{techlag}_{\mathcal{F}}(c) \mid \forall c \in D\})$$

The above technical lag framework definition is now ready to be operationalised to concrete use cases. In the next chapters, we will instantiate this framework for multiple software ecosystems and carry out empirical evaluations in order to assess if and how software ecosystems suffer from technical lag.

4.6 Conclusion

Developers and maintainers face many limitations when they need to make decisions about whether and how to upgrade outdated reusable software components, during development or in production, in a certain software deployment. In many cases, those deployments are built automatically with package managers, selecting specific component releases from software component repositories, such as package management systems for the major programming languages, or Linux-based software distributions. Developers are confronted with a difficult choice. Either they stick to outdated software component versions, preventing them to benefit from bug and security fixes and new available functionality, or they decide to upgrade to a newer version, incurring the risk of backward incompatibilities or other technical problems. The fact that many components have explicit or transitive dependencies on many others, makes the risk assessment of upgrading components even more complex – and taking rational decisions becomes even more difficult.

In this chapter, we carried out surveys and interviews with developers to ask them about this updating problem and assess the usefulness of the technical lag concept. We provided a necessary foundation to address this problem by defining a generic formal framework for measuring how outdated (i.e., how far away from a certain "ideal" state) is a given software component or its deployment. This framework is based on the notion of technical lag for a component release in a given component repository, which intends to measure the difference that deploying that release would cause with respect to some "ideal" release. We formally defined the technical lag concept through a parameterised framework that can be instantiated to a wide variety of component distributions or package managers. In the next chapters, we will operationalize this framework and instantiate it to some specific case studies.

Chapter 5

Technical Lag in npm Packages

In the previous chapter, we have shown the opinions of software practitioners about the usefulness of the technical lag concept. We formalised this into generic framework that can capture technical lag for software component repositories.

In this chapter, we instantiate the technical lag framework for the npm ecosystem and we empirically compute and analyze it for the whole registry of npm packages and their dependencies. The content of this chapter is mainly based on our previous publications in the ICSR 2018 proceedings and Journal of Software Evolution and Process 2019 [69, 70].

5.1 Introduction

Every major programming language comes with one or more package repositories and package managers that allow developers to store, contribute, reuse and deploy reusable software packages for this programming language. A package repository allows developers to store and reuse packages. A package manager allows developers to access the repository to find, install and update packages and their dependencies. With the help of package managers (e.g., Maven for Java, pip for Python, and npm for JavaScript[1]), developers can access these repositories and easily find and deploy these packages, and upgrade to newer releases of already deployed packages.

npm is the most used package manager for reusing JavaScript components, along with its "official" package repository with a large and active developer community [75]. JavaScript is one of the most popular programming languages nowadays. According to the Octoverse study[2], *JavaScript* was reported as the most popular programming language on GitHub in 2018. According to Tiobe's programming language index[3] (February 2019), JavaScript is the 6^{th} most popular programming language. npm and JavaScript have also been used as case studies by many other empirical software engineering researchers [1, 36, 58, 76, 37].

In the current chapter, we take a step further and we operationalize the technical lag model by applying it on the npm package repository. We instantiate the technical lag framework to the npm case study and then we empirically analyze the history of package update practices and technical lag evolution for more than $500K$ packages with about $4M$ package releases over a seven-year period. We consider both development and runtime dependencies, and study both direct and transitive dependencies, taking into account the release type and the use of version dependency constraints. We also analyze the technical lag of external GitHub applications depending on npm packages.

5.2 Characteristics of the npm case study

To study npm we relied on the libraries.io[4] dataset. This service monitors several parameters for all (i.e., more than $3M$) packages from 36 different package repositories, including npm[5]. For our empirical study we used version 1.2.0 of the Libraries.io Open Source Repository and Dependency Metadata [77], available as open access under the *CC Share-Alike 4.0* license. The considered timeframe for the analysis was from *2010-11-09* (the date of the first known npm package release) to *2018-03-13* (the date of publication of the dataset).

[1]Note that for npm, the name of the package manager and package repository are the same.

[2]`octoverse.github.com`

[3]`https://www.tiobe.com/tiobe-index/`

[4]https://libraries.io/about

[5]These numbers correspond to the state of libraries.io as of March 2018.

The dataset comprises $698K$ npm packages, $4.76M$ package releases and $52.8M$ npm dependencies.

Relevant information for each package release includes the package name, version number, release date, and information for each dependency of the package release, such as the name of the required package, a dependency constraint and a dependency type. The metadata of npm package releases is stored in a `package.json` file[6] that is available for each release. Figure 5.1 provides an excerpt of relevant information stored in such a file.

```
{"name": "foo",
"version": "1.2.3",
...
"dependencies": {"bar" : ">=1.0.2 <2.1.2",
                 "baz" : ">1.0.2 <=2.3.4"},
"devDependencies": {"boo" : "2.0.1"},
...}
```

Figure 5.1: Excerpt of relevant metadata stored in a hypothetical *package.json* file for package release foo 1.2.3.

npm considers different dependency types. The ones that npm packages make us of are: i) *Runtime* dependencies are required to install and execute the package; ii) *Development* dependencies are used during package development (e.g., for testing); and iii) *Optional* dependencies will not hamper the package from being installed if the dependency is not found or cannot be installed. The rest are *Peer* and *Bundled* dependencies. The dataset is composed of 42.6% of runtime dependencies, 57.3% of development dependencies, and less than 1% (355 in total) of optional dependencies. Because of this very low number of *Optional* dependencies, and because we do not know which of them are actually installed in practice (since they are optional), they are excluded from our analysis.

Through a careful manual inspection of the dataset, we found a number of packages that should be ignored for the analysis, because they would introduce bias in the results. We are primarily interested in packages that can be considered as reusable libraries, i.e., packages that have been stored on npm for the purpose of being reused by other npm packages or external applications. For this reason, we excluded a huge set of packages that were created automatically[7] on 30$^{\text{th}}$ of March 2016 with the only purpose of depending on a large set of npm runtime dependencies. Examples are the "wowdude-x" packages[8], "neat-x" packages[9] and "all-packages-x" packages. We found that a large number of packages

[6]The structure of this file is defined in `https://docs.npmjs.com/files/package.json`, visited on Feb. 13$^{\text{th}}$ 2019.

[7]https://github.com/ell/npm-gen-all

[8]https://libraries.io/search?q=wowdude

[9]https://libraries.io/npm/neat-106

(corresponding to more than 20K package releases) were released in May 2017 with a name ending with "-cdn", (e.g., react-native-cdn[10], webpack-cdn[11], etc). All of these packages were removed from later versions of npm, because they were considered as "spam".

We also removed all package pre-releases (i.e., 8% of all npm package releases) from the dataset (e.g., releases with version numbers of the form *1.0.0-alpha*, *1.0.0-alpha.1*, *2.1.0-beta*, *2.0.0-rc1* and so on). The reason for this exclusion is that prereleases are not recommended to be reused by a dependency because such releases are unstable and might not satisfy the intended compatibility requirements as denoted by its associated normal version [12].

After filtering out the above packages and releases, we obtained a sanitized dataset containing $520K$ packages, $3.6M$ package releases, and $46M$ dependencies for these releases. 57.9% of these dependencies were *runtime dependencies*, while 42.1% were *development dependencies*. We could parse and identify 98.1% of the packages expressed by these dependencies. The other 1.9% are dependencies with local files, git URLs, unknown (e.g., misspelled), etc.

5.3 Instantiating the Technical Lag Framework to npm

The technical lag framework has been designed to be generally applicable to different types of component distributions, and even in different ways for a given component distribution. We illustrate this by instantiating the framework to the case study of the npm repository of JavaScript package releases. We take into account the semver mechanism and the use of version constraints on package dependencies specified by npm package releases.

Definition 4. Package releases

Let $\mathcal{P} \subset \mathcal{N} \times \mathcal{V} \times \mathcal{T}$ be the set of all package releases available in npm, where $\mathcal{N}$ is the set of all possible package names, $\mathcal{V} \subset \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ the set of all possible version numbers, and $\mathcal{T}$ the set of all possible time points. Each package release $p = (p_{name}, p_{version}, p_{time}) \in \mathcal{P}$ has an associated package name $p_{name} \in \mathcal{N}$, a version number $p_{version} \in \mathcal{V}$ and a release date $p_{time} \in \mathcal{T}$. We assume a total order on $\mathcal{V}$ and $\mathcal{T}$.

We propose multiple instantiations (i.e., scenarios of use) of the framework that mainly differ in how technical lag is computed. In a first scenario, technical lag is computed as a time difference between two releases. In a second scenario, technical lag is computed based on a difference between version numbers of two releases. Other scenarios could be envisaged to explore a wider spectrum of ways to compute technical lag. This will be

[10] https://libraries.io/npm/react-native-cdn
[11] https://libraries.io/npm/webpack-cdn
[12] https://semver.org/#spec-item-9

included in the scope of the next study in Chapter 6.

Definition 5. Time-based instantiation of the technical lag framework

We define $\mathcal{F}^{npm}_{time}$ as the function that instantiates the technical lag framework $\mathcal{F}$ at time $t \in \mathcal{T}$, as follows:

$$\mathcal{F}^{npm}_{time}(t) = (\mathcal{P}_t, \mathbb{N}, \mathbf{ideal}^{npm}, \mathbf{delta}^{npm}_{time}, \mathbf{agg}_{time})$$

where:

- $\mathcal{P}_t = \{p \in \mathcal{P} \mid p_{time} \leq t\}$ is the set of npm package releases that are available at time t.
- $\mathbb{N}$ is the set of possible time lag values. Intuitively, each element corresponds to a number of *days*.
- $\mathbf{ideal}^{npm}(p) = \max_{p'_{version}} \{p' \in \mathcal{P}_t \mid p'_{name} = p_{name}\}$. Intuitively, for a given package release p, the function $\mathbf{ideal}^{npm}$ returns the *highest available version*[13] of a package release with the same name as p.

  (An alternative variant of $\mathbf{ideal}^{npm}$ is to select the *highest backward compatible version* for a given package release, restricting the selection of higher releases to those corresponding to the same major release number only.)
- $\mathbf{delta}^{npm}_{time}(p, q) = \max(0, q_{time} - p_{time})$ computes the positive difference in number of days between the release dates of two package releases p and q.
- $\mathbf{agg}_{time}(L) = \max(L)$, with $L \subseteq \mathbb{N}$ computes the maximum of a set of lags. The chosen aggregation function (maximum) is useful to ascertain that the lag of each release in the collection remains below a certain threshold.

  (An alternative would be to use a summation to assess the total lag for the collection as a whole, as an estimation of the effort that would be needed to reduce the lag in all releases of the collection.)

We can directly compute $\mathbf{techlag}_{\mathcal{F}^{npm}_{time}(t)}(p)$ in terms of the above framework instantiation. This definition formalizes the time lag that we informally explained with the motivating example in Chapter 4.

We now define a version-based instantiation of the technical lag framework for npm, corresponding to the second scenario:

[13] assuming that the user does not specify an additional upper bound version constraint upon installation of p

Definition 6. Version-based instantiation of the technical lag framework

We define $\mathcal{F}^{npm}_{version}$ as the function that instantiates the technical lag framework $\mathcal{F}$ at time $t \in \mathcal{T}$, as follows:

$$\mathcal{F}^{npm}_{version}(t) = (\mathcal{P}_t, \mathcal{V}, \mathbf{ideal}^{npm}, \mathbf{delta}^{npm}_{version}, \mathbf{agg}_{version})$$

where:

- $\mathcal{P}_t = \{p \in \mathcal{P} \mid p_{time} \leq t\}$ is the set of npm package releases that are available at time t.
- $\mathcal{V}$ is the set of possible version lag values. Intuitively, each element corresponds to a version number.
- $\mathbf{ideal}^{npm}(p) = \max_{p'_{version}} \{p' \in \mathcal{P}_t \mid p'_{name} = p_{name}\}$.
- $\mathbf{delta}^{npm}_{version}(p, q) = (major_{lag}, minor_{lag}, patch_{lag})$ where
  $coll = \{r \in \mathcal{P}_t \mid p_{version} \leq r_{version} \leq q_{version} \wedge p_{name} = r_{name} = q_{name}\}$
  $major_{lag} = |\{major \mid r_{version} = (major, minor, patch), \forall r \in coll\}| - 1$
  $minor_{lag} = |\{(major, minor) \mid r_{version} = (major, minor, patch), \forall r \in coll\}| - major_{lag} - 1$
  $patch_{lag} = |\{(major, minor, patch) \mid r_{version} = (major, minor, patch), \forall r \in coll\}| - major_{lag} - minor_{lag} - 1$
  In this definition, *coll* is the set of all releases between p and q. We first find the number of releases that increase the major number, then the number of releases that increase the minor number while having the same major number, and finally the number of patch releases that increase the patch number while having the same major and minor numbers. From that, we derive the number of distinct major, minor and patch releases that would be needed to upgrade from r_1 to r_2.
- $\mathbf{agg}_{version}(L) = \sum_{v \in L} v$, with $L \subseteq \mathcal{V}$ computes the sum over a set of versions. To do so, addition has to be defined on version numbers, e.g., as follows.

  Let $v, w \in \mathcal{V}$ be two version numbers.

  Assume $v = (v_{major}, v_{minor}, v_{patch})$ and $w = (w_{major}, w_{minor}, w_{patch})$

  Then $v + w = (v_{major} + w_{major}, v_{minor} + w_{minor}, v_{patch} + w_{patch})$

We can directly compute $\mathbf{techlag}_{\mathcal{F}^{npm}_{version}(t)}(p)$ in terms of the above definitions. This formalizes the version lag that we informally explained with the motivating example in Chapter 4.

Notation 1. In the remainder of this section we use the shortcut notations $\mathbf{techlag}_{time}(p,t)$ for $\mathbf{techlag}_{\mathcal{F}^{npm}_{time}(t)}(p)$ and $\mathbf{agglag}_{time}(P,t)$ for $\mathbf{agglag}_{\mathcal{F}^{npm}_{time}(t)}(P)$, and similarly for the version-based variants $\mathbf{techlag}_{version}$ and $\mathbf{agglag}_{version}$, if it is clear from the context that we are referring to the npm instantiation of the formal framework.

In practice, when installing an npm package release, *dependency relationships* impose the installation of all required package releases as well. These package releases are selected by the `npm install` tool upon installation of a given package.

Definition 7. Direct and transitive dependencies

Let $t \in \mathcal{T}$ be a point in time. Let $\mathcal{P}_t$ be the set of npm package releases available at time t. We define the two following functions:

$\mathbf{deps}_t : \mathcal{P}_t \to \mathbb{P}(\mathcal{P}_t)$ such that $\mathbf{deps}_t(p)$ returns all npm package releases satisfying the direct dependencies of p, as selected by `npm install`.

$\mathbf{deps}^+{}_t : \mathcal{P}_t \to \mathbb{P}(\mathcal{P}_t)$ such that $\mathbf{deps}^+{}_t(p)$ returns all npm package releases satisfying the *transitive* (i.e., direct and indirect) dependencies of p, as selected by `npm install`. Alternatively, we can define $\mathbf{deps}^+{}_t(p)$ as the minimal fix point such that:

$$\mathbf{deps}_t(p) \subseteq \mathbf{deps}^+{}_t(p) \quad \text{and} \quad \forall p' \in \mathbf{deps}^+{}_t(p) : \mathbf{deps}_t(p') \subseteq \mathbf{deps}^+{}_t(p)$$

Based on these two functions and on the definitions of $\mathbf{agglag}_\alpha$ for $\alpha \in \{\text{time}, \text{version}\}$ in the technical lag framework, we can define the technical lag of a deployment of a package.

Definition 8. Technical lag of a package deployment

Let $t \in \mathcal{T}$ be a point in time. Let $\mathcal{P}_t$ be the set of npm package releases available at time t. For $\alpha \in \{\text{time}, \text{version}\}$, we define:

- $\mathbf{deplag}_\alpha(p,t) = \mathbf{agglag}_\alpha(\mathbf{deps}_t(p), t)$ for the direct dependencies
- $\mathbf{deplag}^+_\alpha(p,t) = \mathbf{agglag}_\alpha(\mathbf{deps}^+{}_t(p), t)$ for the transitive dependencies

For example, $\mathbf{deplag}_{\mathbf{time}}(p,t)$ computes the *time lag* of all *direct dependencies* of package release p at time t as the maximum time lag of any of these direct dependencies. Similarly, $\mathbf{deplag}^+_{\mathbf{version}}(p,t)$ computes the *version lag* of all *transitive dependencies* of package release p at time t as the sum of all version lags of all these transitive dependencies.

5.4 Empirical Evaluation

Research Questions

Based on the formal framework for technical lag measurement and its instantiation for the npm case study (presented in Section 5.3), we will empirically study five research questions for the npm package repository. In particular, we address the following research questions:

- RQ_0*: Which operators are most frequently used in dependency constraints?* With this preliminary research question, we aim to identify the operators and constraints that are the most used with the npm dependencies.
- RQ_1*: How much technical lag is induced by direct dependencies?* For all npm package releases, we identify the package versions that are required to depend on after the use of dependency constraints. Then, we measure over time how outdated they are in terms of time and versions and w.r.t their highest available versions.
- RQ_2*: How do constraint operators impact technical lag?* With this research question, we aim to understand the relationship over time between dependency constraints and the technical lag they induce.
- RQ_3*: How do external applications suffer from technical lag?* We measure how outdated external applications that are distributed via GitHub but not via npm are. Then, we analyze the relationship between their technical lag and the dependency constraints they use.
- RQ_4*: How does technical lag propagate over transitive runtime dependencies?* We quantify how outdated transitive npm dependencies are, and how this outdatedness evolve over time and over their dependency trees.

RQ_0: Which operators are most frequently used in dependency constraints?

Although the npm community encourages developers to use the semver standard in order to have more reliable and predictable updates [14], it is not clear to which extent they are following this recommendation. Therefore, we studied how npm package maintainers are using version numbers and constraints for their dependencies.

We quantified the usage of the different types of dependency constraints used by package dependencies defined in npm package releases. If a constraint uses a combination of different constraint types, we consider the type of the least permissive one. This occurred in only 0.0005% of all dependencies. Table 5.1 shows the proportion of the types of dependency constraints used for all npm package releases during the considered observation

[14] https://docs.npmjs.com/getting-started/semantic-versioning

Dependencies (total)	**caret**	**strict**	**tilde**	**latest**	**other**
Runtime (42,3%)	67.5%	16.0%	8.2%	4.0%	4.3%
Development (57,7%)	74.0%	13.4%	6.9%	3.5%	2.2%
All dataset (100%)	71.2%	14.5%	7.5%	3.7%	3.1%

Table 5.1: Proportion of dependency constraints used, grouped by operator for all npm package releases over the considered period.

period. We observe that caret is most frequently used, covering over 71.2% (32M) of all dependency constraints. This is in line with the fact that backward incompatible changes are undesirable (major updates).

14.5% (6.5M) of all dependencies were specified with a strict version number, signifying that package maintainers prefer a possibly older but fixed version of a dependency, rather than benefiting from automatic updates. Anecdotal evidence suggests that this is often the case: *"I personally don't care about bug fixes. I like of course using a bug-free software but I prefer to rely on stable libraries and if I hit a bug I'll check if it is fixed, in which versions and what this version comes up with. Many times I got a new release downloaded fixing a bug in a feature that I never used. So, I prefer strict versioning where I install the same version every time. Version that I know it works."*[15]

7.5% (3.3M) of all dependencies used a tilde constraint to allow patch updates, and 3.7% (1.7M) of the dependencies are very permissive, allowing to use the latest available version of their dependency. All other types of dependency constraints combined (e.g., comparison operators, wildcards like 1.*.*, etc) represent only 3.1% (1.4M) of all dependencies.

The findings above are mixed. The majority of developers are interested in keeping some backward compatibility, benefiting from minor and patch releases, but do not want to worry about new major releases with incompatible changes, requiring a higher maintenance effort. However, many developers still seem to prefer to keep all control, by imposing a strict version constraint (14.5% of the dependencies). This will likely cause a higher technical lag for many releases in npm, as will be shown later.

From Table 5.1 we also observe that, while the caret constraint is more widely used for development dependencies, tilde, strict and latest constraints are more widely used for runtime dependencies. This could be a consequence of the fact that development dependencies are not needed to run packages but only to develop them. Hence, developers might care less about managing them and more often use the default caret constraint provided by npm.

[15]http://krasimirtsonev.com/blog/article/thoughts-on-semantic-versioning-npm-and-JavaScript-ecosystem

We also analyzed the dynamics of npm dependencies, observing that new releases tend not to remove or add dependencies, but to adapt the version constraints of existing ones. We found that adding or removing dependencies is mostly done during major releases. In order to gain more insight about the kind of releases that change dependency constraints, we quantified when and how version constraints are changed to another type (e.g., from ∼1.0.0 to ^1.0.0) or updated to another constraint of the same type (e.g., from ^1.0.0 to ^2.0.0).

Note that the behavior of tilde is different when used with the unstable major version zero (0.x.y) that is dedicated for initial development. When this operator is assigned to a 0.x.y version, it will have the same behavior as the operator caret, which means that new patch or minor releases will be allowed to be installed, as long as the version number is still $< 1.0.0$.

Figure 5.2 shows box plots of the distribution of the number of dependency constraints that are changed in new major, minor or patch releases. We observe that the third quartile of the distribution is highest for major and minor, and lowest for patch releases. We observe a median value of 1, 2 and 2 changed version constraints for patch, minor and major releases, respectively. This suggests that there is higher chance to update dependencies during minor and major releases, when a feature is added or an incompatible change happened. This corresponds to what is advised by semver [16].

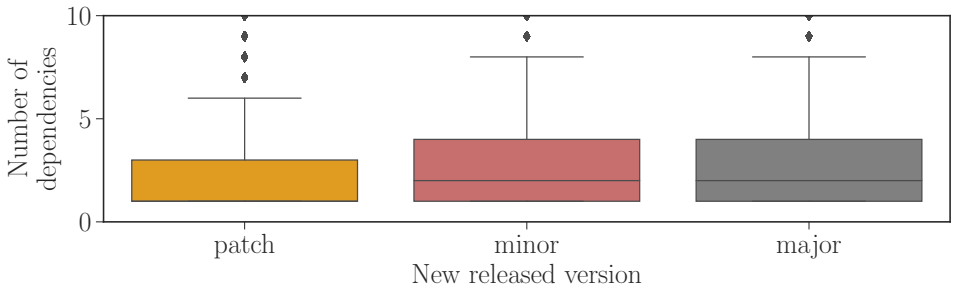

Figure 5.2: Distribution of the number of dependency constraints that are changed in new major, minor or patch releases.

To study the technical lag, we should also know how often packages release new versions. A package with frequent releases may be a real issue for other packages that rely on it, since it could result in a higher than average technical lag for the dependent packages (since they are required to update frequently if they want to keep pace with this release rate). On the other hand, releasing new package versions is important and breaking changes are generally expected for progress because they will enable new features,

[16] https://semver.org/#what-should-i-do-if-i-update-my-own-dependencies-without-changing-the-public-api

new requirements, less technical debt, improved performance, and fixed bugs [6]. Such backward incompatible changes are generally considered acceptable if they are clearly signaled as such, for example by increasing the major release number of the updated package as stated by the semver policy [78, 9].

Releasing a new package version requires time and effort. Therefore, we studied the time needed to release a new chronological, but not necessarily successive, version of a given type. For example, considering the series of versions (1.0.0 (initial), 1.0.1 (patch), 1.0.2 (patch), 1.1.0 (minor), 1.1.1 (patch)), and starting with the (initial → patch) case, we calculate the time between (1.0.0, 1.0.1). For the case of (initial → minor), we calculate the time between (1.0.0, 1.1.0). In a similar way we proceed for the (patch → patch) case, we calculate the time between (1.0.1, 1.0.2) and (1.0.2, 1.1.1). For the case of (patch → minor) we calculate the time between (1.0.2, 1.1.0), etc.

Figure 5.3 shows the distribution of days until the next chronological version is released. As can be seen, it takes more time to release a new major version than a minor or patch one. More specifically, on average, it takes about 19 days to release a patch version, 57 days to release a minor version and 120 days to release a major one.

Considering all types of releases of all packages, we found that 14.7% (0.7M) are *initial* releases, i.e., packages released in npm for the first time. 68.8% (3.3M) are *patch* releases, 13% (0.62M) are *minor* releases and only 3.5% (0.17M) are *major* releases. The total is 4.8M releases at the dataset creation date.

The fact that major releases are rare complies with the results of a recent survey among 2,000 developers coming from different ecosystems, where 48% of npm developers said that they release less than one breaking change per year [6]. However, a recent study [38] showed that breaking changes still occur in minor and patch updates of npm packages and that the majority of the breaking changes are type-related. These are modifications of a library that affect the presence or types of functions or other properties in the library interface, including renaming a public function, moving it to another location, or changing its type signature.

Summary: The way in which npm package releases use dependency constraints suggest that package maintainers are concerned about their dependencies. We also observe, rather unsurprisingly, that the time needed to release a new package version is related to its release type (i.e., major, minor or patch).

RQ_1: How much technical lag is induced by direct dependencies?

This research question focuses on the evolution of the technical lag of package releases induced by their *direct* dependencies. To do so, we compute and analyse $\mathbf{deplag_{time}}(p, t)$ and $\mathbf{deplag_{version}}(p, t)$ at the release dates $t \in \mathcal{T}$ of all analyzed npm package releases p.

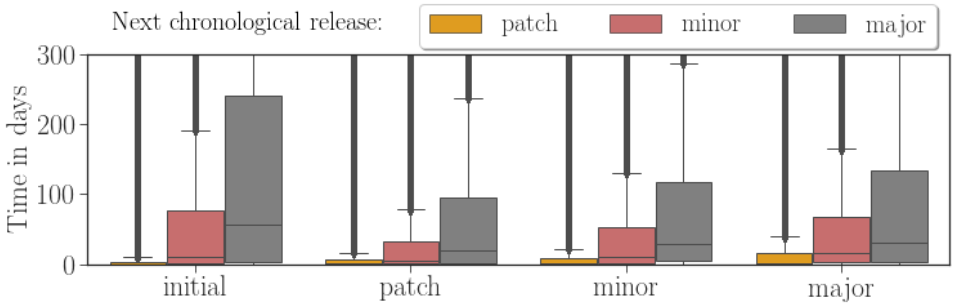

Figure 5.3: Distribution of the time until the next chronological version of npm package releases.

In RQ_4 we will study the same but for *transitive* dependencies.

We found only $11.66M$ (i.e., 25.8%) outdated dependencies (i.e., package releases required by a dependency, for which $\textbf{techlag}_{time} > 0$), $7.94M$ (i.e., 68%) of them are development dependencies and $3.72M$ (i.e., 32%) are runtime dependencies. On the other hand, we found that 54.1% of the up-to-date dependencies are development dependencies, against 45.9% of the runtime dependencies. Given that the dataset contains 42.3% runtime dependencies and 57.7% development dependencies, the relative proportion of outdated dependencies is higher for development dependencies. This was expected, since the presence of outdated dependencies is more problematic for runtime dependencies, required to install and execute the package release, than for development dependencies, used only in a local development environment with little to no impact on the production environment.

Figure 5.4 visualizes the evolution, on a monthly basis, of the distribution of $\textbf{deplag}_{\textbf{time}}$ for all package versions released during that month, grouped by *development* and *runtime* dependencies. We notice that $\textbf{deplag}_{\textbf{time}}$ tends to increase over time for both types of dependencies. This is likely because of the fact that more new releases become available over time. We also observe that the median value of the distribution never exceeded 270 days in the observation period (2011-2018). Considering the whole observation period, we found that the median value for runtime dependencies (160 days) is lower than for development dependencies (195 days). This corroborates our previous observations about the difference between outdated development and runtime dependencies.

To compare the distribution of $\textbf{deplag}_{\textbf{time}}$ for runtime dependencies to the one for development dependencies, we split the considered observation period into seven one-year periods (from 2011 until 2017)[17], each year includes all package versions that were released in it. Using the *Mann-Whitney U* test we found a statistically significant difference (p-value $< .001$) for all years except 2012 and 2016. Using *Cliff's delta* we found a small effect size that increases over time: from $|d| = 0$ for the first year (2011) to $|d| = 0.12$ for the last

[17]The last 3 months in 2018 were excluded from this analysis since it does not cover a full year.

year (2017). In summary, development dependencies tend to be slightly more outdated than runtime dependencies, especially during the last years.

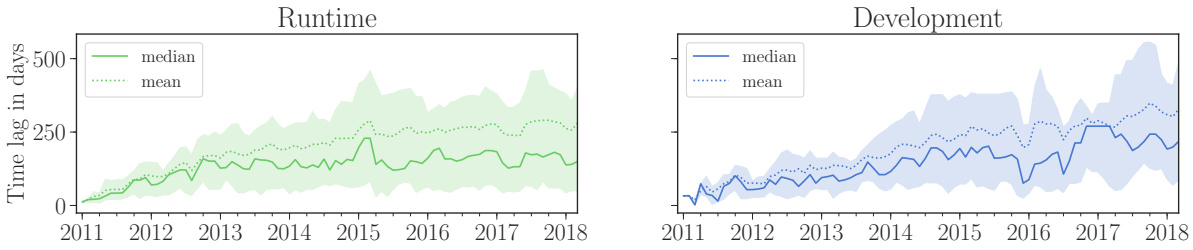

Figure 5.4: Monthly evolution of the distribution of $\mathbf{deplag_{time}}$ for all package releases, grouped by runtime and development dependencies. The shaded areas correspond to the interval between the 25$^{\text{th}}$ and 75$^{\text{th}}$ percentile.

While measuring time lag provides a good first estimate of how outdated a package release is, it is not sufficiently precise to assess the underlying source of the lag. Package releases (or package dependencies) with an identical time lag could have a different evolutionary behaviour. For example, some packages have frequent updates while others are infrequently updated; some packages may have many dependencies while others have only a few ones; some packages primarily provide patch updates while others regularly provide new major releases. Hence, comparing package releases only on the basis of time-based technical lag is not sufficient.

Because of this, we also analyze the temporal evolution of $\mathbf{deplag_{version}}$ of each package release p, computed in function of its three version components (major, minor, patch). This version-based technical lag measurement can help to provide more insights about the type of changes missed by a package release because of its outdated dependencies.

Considering all package releases over the entire observation period, we found a median $\mathbf{deplag_{version}}$ of (1,1,4) for runtime dependencies and a median $\mathbf{deplag_{version}}$ of (2,2,9) for development dependencies. Figure 5.5 visualizes the evolution, on a monthly basis, of the distribution of $\mathbf{deplag_{version}}(p,t) = (Major, Minor, Patch)$ for all package releases available during that month, grouped by runtime and development dependencies, and split per version component. Like for the time lag, we notice that for both types of dependencies the version lag is slightly increasing over time. We also clearly observe that version lag is higher for development dependencies than for runtime dependencies. In particular, in recent months (starting from first quarter of 2017), the version lag has increased a lot for development dependencies. This suggests that the used development dependencies (and their constraints, especially the caret) are less frequently updated than runtime ones. To confirm this hypothesis, we carried out a *Mann-Whitney U* test between the time needed before updating a version constraint within development and runtime dependencies. We found a statistically significant difference (p-value $< .001$). Using *Cliff's*

delta we found a small effect size with $|d| = 0.29$, indicating that the time needed before updating a version constraint is slightly higher within development dependencies.

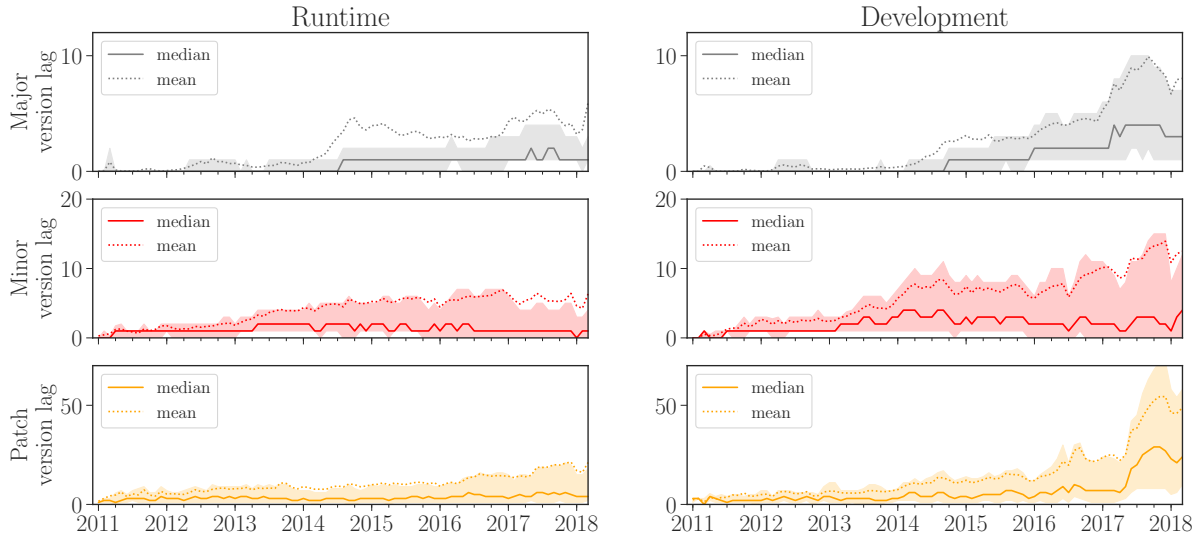

Figure 5.5: Monthly evolution of the distribution of $\mathbf{deplag}_{\mathbf{version}}(p)$ = (Major, Minor, Patch) for all package releases, grouped by runtime and development dependencies, and split per version component. The shaded areas correspond to the interval between the 25$^{\text{th}}$ and 75$^{\text{th}}$ percentile.

Summary: Technical lag induced by direct dependencies in npm package releases is increasing over time due to many missed updates, including major ones. Technical lag is higher for development dependencies: 2 out of 3 outdated dependencies in npm are development dependencies.

RQ_2: How do constraint operators impact technical lag?

In this research question, we explore to which extent the use of the different types of dependency constraint operators is related to the presence of technical lag. For each outdated dependency (i.e., each package release required by a dependency, for which $\mathbf{techlag}_{time} > 0$), we identified the operator used by the corresponding dependency constraint. Results are presented in Figure 5.6, showing the proportion of outdated dependencies w.r.t. the type of constraint operator being used. We observe that the majority of outdated dependencies (15.7% and 43.5%) use the caret constraint, 26.1% (i.e., 10.6% and 15.5%) use the strict constraint and around 11.5% (i.e., 4.1% and 7.4%) of all outdated dependencies rely on the tilde constraint.

We expected such a high proportion of caret constraints because we observed in Table 5.1 that they account for more than 71% of all used constraints. Because of this,

we also computed the proportion of outdated dependencies relative to all dependencies for each kind of constraint. We found that only 21.4% of the package releases using caret constraints are outdated, while 40% using tilde constraints and 46.5% using strict constraints are outdated. These findings confirm the hypothesis that the use of more permissive constraints (caret is more permissive than tilde, which is more permissive than strict) lowers the risk of having outdated dependencies.

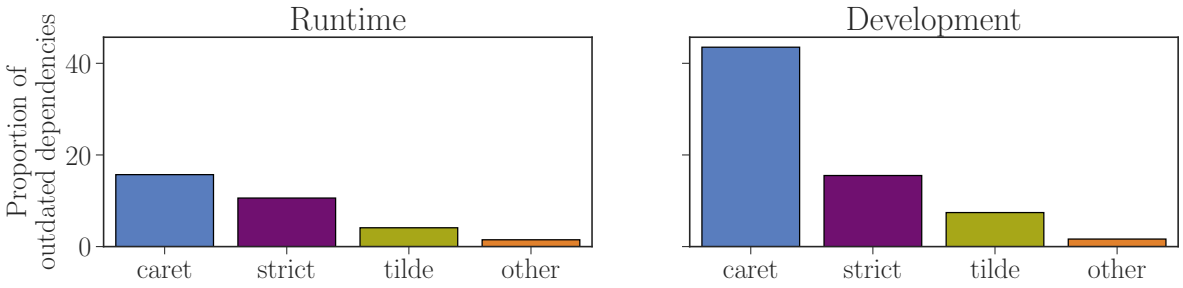

Figure 5.6: Proportion of outdated npm dependencies per constraint type, for runtime dependencies and development dependencies respectively.

To have more insights about the use of dependency constraints in npm, we analyzed the evolution over time of version constraint types used by dependencies at the release date of each package release. Figure 5.7 shows this historical evolution of the constraint type usage, for both runtime and development npm dependencies. The *other* constraint type is gradually overtaken by the *tilde* constraint type until early 2014, suggesting that developers are becoming increasingly aware of the backward incompatible changes that might come with new released versions. Starting from February 2014, when the *caret* constraint was introduced as the default constraint in npm[18], the use of the *tilde* constraint starts to be replaced by the *caret* constraint. The use of the *strict* constraint type tends to remain more or less stable over time, representing about 20% of all constraint types. The use of the most permissive *latest* constraint only represents a small proportion, and is decreasing over time.

We observe from Figure 5.7 that the tilde constraint is progressively replaced by the caret constraint, which implies that the constraint is becoming less restrictive, not more restrictive. We repeated the same analysis by only considering the evolution over time of version constraint usage by *outdated* dependencies in Figure 5.8. We observe that before the introduction of the caret constraint, outdated (runtime or development) dependencies were mainly used with the strict and tilde constraints, with tilde gradually taking over the usage of strict constraints until 2014. With the introduction of the caret constraint, the usage of the strict constraint stabilised over time, while caret was taking over the tilde constraint usage. This could be the reason why we observed an increasing version lag at

[18]See, e.g., `http://fredkschott.com/post/2014/02/npm-no-longer-defaults-to-tildes/`

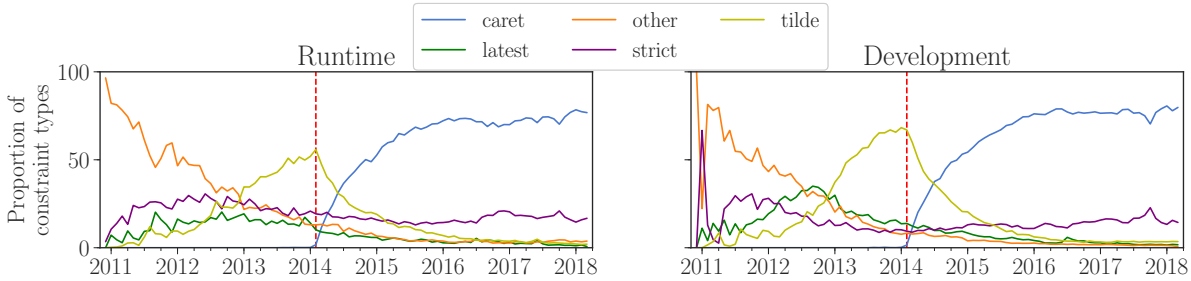

Figure 5.7: Monthly evolution of version constraint usage by **all** package dependencies.

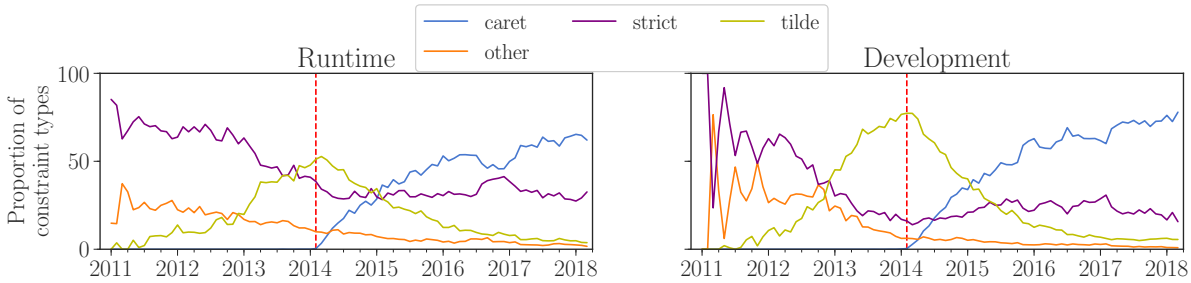

Figure 5.8: Monthly evolution of version constraint usage by **outdated** package dependencies.

the same time in Figure 5.5. To confirm this, we filtered out all dependencies using a caret constraint, and still observed an increasing major lag at the first quarter of 2014 for both runtime and development dependencies. It should not be surprising that nearly no major version lag could be observed before 2014: a limited number of package releases at that time had a version number greater than or equal to *1.0.0*. The proportion of such releases went from 20% in January 2014 to 42% in January 2015. This increase coincides with the resolution of many issues related to the use of constraints with pre-*1.0.0* version numbers[19].

Summary: npm package releases are increasingly using the caret constraint over time. Initially, the least permissive *strict* constraints were gradually being replaced by more permissive *tilde* constraints and, since 2014 these *tilde* constraints were being replaced by even more permissive *caret* constraints. Nevertheless, strict constraints continue to represent a considerable proportion of about 20% of all dependencies.

[19]https://github.com/npm/node-semver/issues/79

RQ_3: How do external applications suffer from technical lag?

The main purpose of the npm package manager is to distribute packages, to facilitate end-users to deploy them, and to allow external applications (that are not intended to be distributed through npm) to depend on them. It is interesting to measure the technical lag of such external applications depending on npm package releases, and compare their technical lag with the one of the npm package releases themselves. Since npm packages are supposed to be reused as libraries, their package maintainers are supposed to be more careful than developers of external applications that just depend on these libraries, without necessarily needing to care about other people depending on their own applications. We therefore expect to find more technical lag in external applications than in (reusable) package releases distributed via npm.

Definition 9. Technical lag framework instantiation for external applications depending on npm

To allow the npm instantiation of the technical lag framework to include external applications, we extend the set $\mathcal{P}_t$ (all npm package releases available at time t) to $\mathcal{E}_t \cup \mathcal{P}_t$, where $\mathcal{E}_t$ is the set of all external applications that are developed and distributed through GitHub[20] at time t, and that depend on at least one npm package release. All other definitions remain unchanged.

The libraries.io dataset contains references to GitHub repositories that are known to host JavaScript applications. Based on this dataset, we identified $480K$ GitHub repositories, focusing only on repositories that are not forks so that applications with many forks will not be considered many times in the analysis. The main purpose of forks is to propose changes to someone else's application, thus forked repositories are not meant to be considered as different applications[21]. For each of these repositories, we extracted the content of the `package.json` file for their last known commit. This file contains the list of dependencies, including the ones that target packages hosted on npm. These dependencies account for around $6.2M$ dependencies.

As for the previous research questions, we distinguish between runtime and development dependencies from external applications to npm package releases. We found that 53.2% of these dependencies are runtime dependencies, while 46.8% are development dependencies. 39.5% ($2.5M$) of these dependencies from external applications were outdated (i.e., they referred to an npm package release q for which $\mathbf{techlag}_{time}(q) > 0$). Proportionally, 48.2% of these outdated dependencies were runtime dependencies and 51.8% were development dependencies.

For all these outdated dependencies originating from external applications, we

[20] GitHub is the main Open Source platform for JavaScript applications [79].
[21] https://help.github.com/articles/fork-a-repo/

identified the operators used in their constraints. Table 5.2 shows the proportion of these constraint types, and compares them to the proportion observed for npm package releases. Unlike what we observed for npm package releases, where caret was the prominent dependency constraint, external applications mostly rely on strict constraints (nearly half of their runtime dependencies, 47.4%), while only around one quarter (25.2%) rely on the caret constraint. A possible explanation is that external applications are not aimed to be distributed for reuse by other packages, but mainly aim to be used in production. In this context, pinning versions of the dependencies facilitates the replication of an environment in which the application is known to work. This explains the higher use of strict constraints. In fact, easing replicating environments is now directly supported by npm through the "package-lock" file[22].

Dependencies	**External applications**				**npm package releases**			
	caret	tilde	strict	other	caret	tilde	strict	other
Runtime	25.2%	21.2%	47.4%	6.2%	49.2%	12.9%	33.2%	4.7%
Development	53.5%	29.3%	16.0%	1.2%	63.9%	10.9%	22.8%	2.4%
Both	39.8%	25.4%	31.1%	3.7%	59.2%	11.6%	26.1%	3.1%

Table 5.2: Proportion of constraint types used by outdated dependencies from external applications to npm package releases, compared to the proportion of constraint types used by outdated dependencies from npm package releases.

Let us now study the time-based technical lag $\mathbf{deplag_{time}}(p)$ of external applications $p \in \mathcal{E}_t$ depending on npm package releases at different time points $t \in \mathcal{T}$. Figure 5.9 shows the evolution, on a monthly basis, of the distribution of $\mathbf{deplag_{time}}$ for all external applications having their last known commit during that month, grouped by runtime and development dependencies. We observe a median value of 218 days for runtime dependencies, and 275 days for development dependencies. This is several months longer than the median values we observed for $\mathbf{deplag_{time}}$ of npm package releases in Figure 5.4 (160 days and 195 days for runtime and development dependencies, respectively). We can also notice that there are many applications that have not been updated in several years (e.g., since 2011), which means that if we compute their technical lag of today, we will find very high values.

We also observe that $\mathbf{deplag_{time}}$ is increasing over time for both runtime and development dependencies of external applications, especially after the first quarter of 2017. This means that external applications whose last known commit is in 2017 still had a high technical lag because of their dependencies. The figure also shows that there was no time lag for development dependencies until mid 2011. We investigated this observation and we found that all development dependencies before July 2011 were up-to-date (zero technical lag).

[22]https://docs.npmjs.com/files/package-lock.json

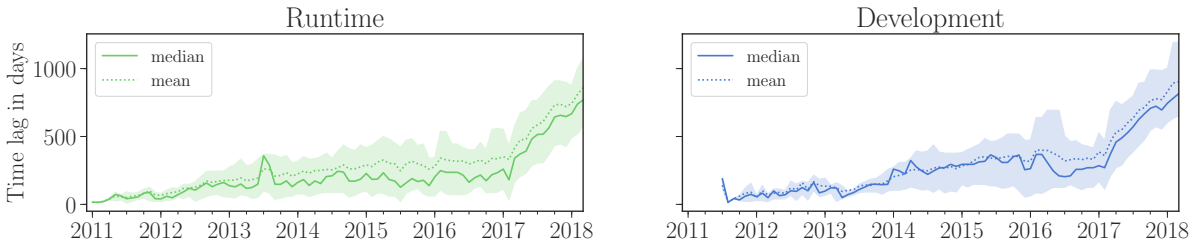

Figure 5.9: Monthly evolution of the distribution of $\mathbf{deplag_{time}}$ for all external applications, grouped by runtime and development dependencies. The shaded areas correspond to the interval between the 25$^{\text{th}}$ and 75$^{\text{th}}$ percentile.

Next, we compute the version-based technical lag $\mathbf{deplag_{version}}(p)$ of external applications $p \in \mathcal{E}_t$ at different time points $t \in \mathcal{T}$. Figure 5.10 visualizes the evolution, on a monthly basis, of the distribution of $\mathbf{deplag_{version}}(p,t) = (Major, Minor, Patch)$ for all external applications during that month, grouped by *runtime* and *development* dependencies, and split per version component. As was the case for the time-based lag, the version-based lag is slightly increasing over time, especially since the beginning of 2017. Focusing on the Major version component, we see that it started increasing by mid 2014, just like what we observed in Figure 5.5. However, it is less than what we observe for Minor or Patch version components.

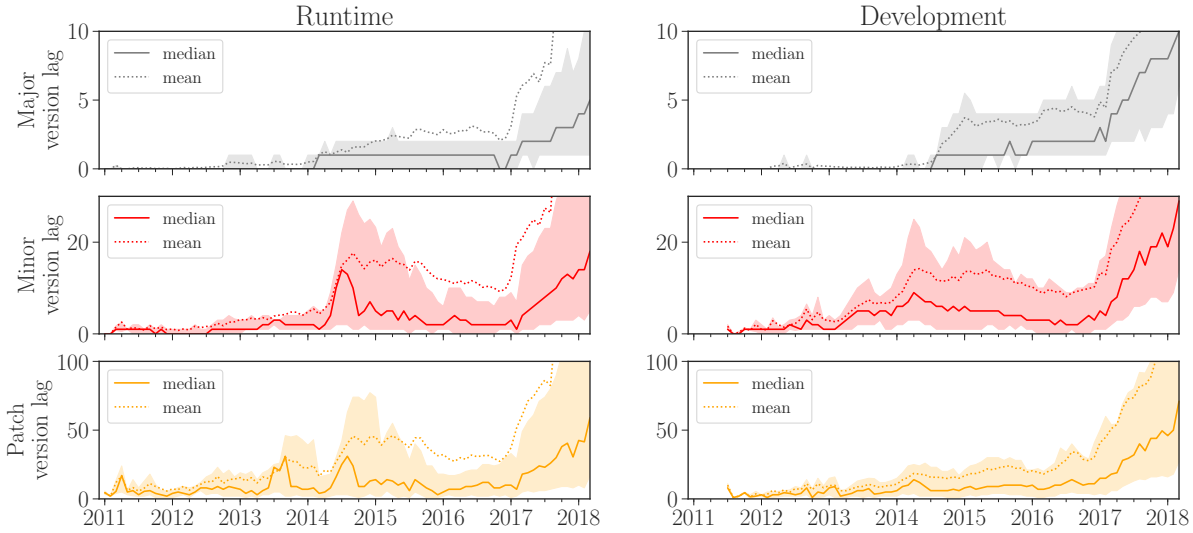

Figure 5.10: Monthly evolution of the distribution of $\mathbf{deplag_{version}}(p)$ =(Major, Minor, Patch) for all external applications, grouped by runtime and development dependencies, and split per version component. The shaded areas correspond to the interval between the 25$^{\text{th}}$ and 75$^{\text{th}}$ percentile.

When we compare $\mathbf{deplag}_{\mathbf{version}}(p,t) = (Major, Minor, Patch)$ between external applications ($p \in \mathcal{E}_t$, Figure 5.10) and npm package releases ($p \in \mathcal{P}_t$, Figure 5.5), we observe a difference, especially at the level of the minor and patch component. For external applications we found a median value of (1,3,9) for runtime dependencies and a median value of (2,4,10) for development dependencies (compared to (1,1,4) and (2,2,9) for $\mathbf{deplag}_{\mathbf{version}}$ of runtime and development dependencies of npm package releases). We hypothesize that this is due to the higher proportion of strict constraints used by external applications that depend on npm packages.

Figure 5.11 shows the evolution of the proportion of constraint types used by external applications for their runtime dependencies to npm packages. Compared to Figure 5.7, we observe a much higher proportion of strict constraints, and this proportion tends to increase over time. While the proportion of the caret constraint increased since 2014, it "stabilised" at a high percentage of strict constraint, which was not the case for the npm package releases. This explains the high technical lag found for external applications relying in npm packages. We also see a tendency of a decreasing use of caret and an increasing use of strict starting in 2017. This might explain the increased technical lag that we observed since the first quarter of 2017.

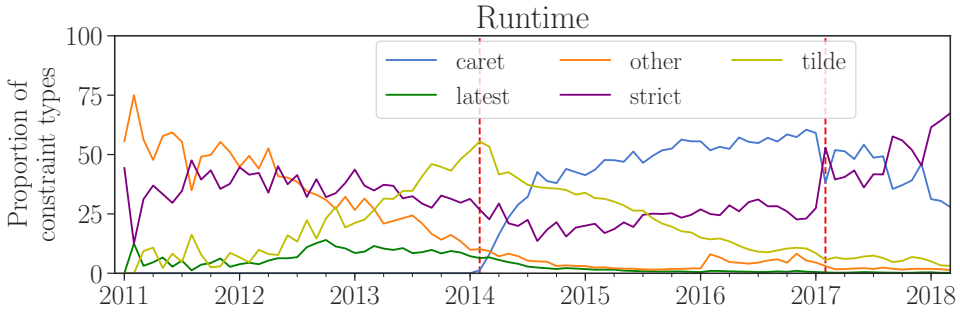

Figure 5.11: Monthly evolution of the proportion of constraint types used by runtime dependencies in external applications depending on npm packages.

Summary: The median technical lag in external GitHub applications that depend on npm packages is much higher than the technical lag found in npm package releases, and is increasing over time. This seems to be the consequence of the more frequent and increasing use of strict constraints by external applications.

RQ_4: How does technical lag propagate over transitive runtime dependencies?

All previous research questions have focused on the technical lag induced by *direct dependencies* (i.e., $\mathbf{deplag}_{\mathbf{time}}$ or $\mathbf{deplag}_{\mathbf{version}}$) because these dependencies are the ones

that are explicitly declared by a package maintainer, and over which they have direct control by means of dependency constraints. However, when a package release is deployed, not only its direct dependencies need to be installed but also all their dependencies, and the dependencies of these dependencies, and so on. The npm package manager installs these *transitive dependencies* in distinct subfolders, leading to a dependency graph that is a tree, even when multiple distinct releases of a same package are transitively required[23]. These transitive dependencies may induce additional technical lag on the package release being installed since the technical lag can be accumulated in the dependency tree from a level to an other (deeper level).

In this research question, we therefore quantify and analyze the *transitive* technical lag of a package release induced by all its transitive dependencies. We will do so for the time-based transitive lag $\mathbf{deplag}^{+}_{time}$ and the version-based transitive lag $\mathbf{deplag}^{+}_{version}$. While deploying a package release requires all transitive (direct and indirect) runtime dependencies to be installed, indirect development dependencies do not need to be installed during development. Indeed, to install development dependencies, only direct development dependencies and their own (transitive) runtime dependencies are required. For this reason, we restrict the analysis of RQ_4 to transitive runtime dependencies only.

We computed the transitive runtime dependency tree of each available npm package release at the beginning of each period of 4 months (quadrimester) starting from December 2010. For each quadrimester, we considered the latest available release p of each package, and computed its time-based transitive lag $\mathbf{deplag}^{+}_{time}$ and its version-based transitive lag $\mathbf{deplag}^{+}_{version}$. For the whole considered observation period, we analyzed the technical lag of $672,906$ package releases of $399,522$ packages having runtime dependencies. These package releases have $163.8M$ transitive runtime dependencies on $308,243$ distinct package releases, representing 45% of the whole ecosystem of npm packages.

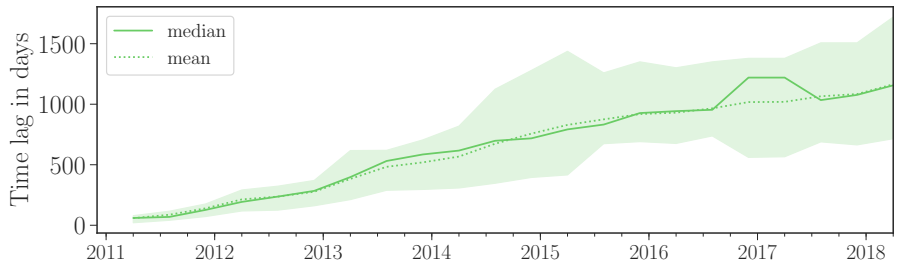

Figure 5.12: Quadrimestrial evolution of the distribution of $\mathbf{deplag}^{+}_{time}$ for runtime dependencies of all npm package releases. The shaded areas correspond to the interval between the 25^{th} and 75^{th} percentile.

When analyzing time-based transitive lag $\mathbf{deplag}^{+}_{time}$ we found 38.4% of all runtime dependencies to be outdated. Figure 5.12 shows the quadrimestrial evolution of the

[23]see https://medium.com/learnwithrahul/understanding-npm-dependency-resolution-84a24180901b

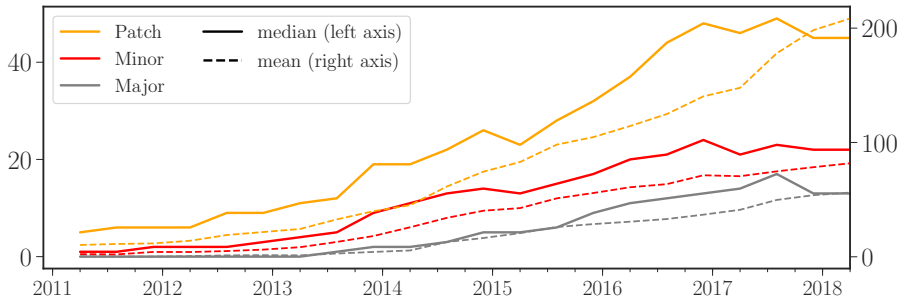

Figure 5.13: Quadrimestrial evolution of the the distribution of $\mathbf{deplag}^{+}_{version}$, split per version component for runtime dependencies of all npm package releases.

distribution of $\mathbf{deplag}^{+}_{time}$ from December 2010 until March 2018. The graph starts in 2011 because we could not find any technical lag in December 2010 (i.e., all runtime dependencies were up-to-date). We observe that $\mathbf{deplag}^{+}_{time}$ is increasing over time. The increase is expected, since $\mathbf{deplag}_{time}$ for direct dependencies was also found to be increasing over time (cf. Figure 5.4). Compared to the median value of time lag of $\mathbf{deplag}_{time}$ for direct runtime dependencies (160 days, see Section 5.4), the median value of $\mathbf{deplag}^{+}_{time}$ for transitive runtime dependencies is much higher and it exceeded 500 days in April 2013. This should not come as a surprise, given that many package releases have a deep transitive dependency tree [1], and that lag accumulates from one level to another one. Next, we computed $\mathbf{deplag}^{+}_{version}$, the version-based lag induced by runtime transitive dependencies of all package releases. Figure 5.13 shows, for each version component of $\mathbf{deplag}^{+}_{version}(p,t) = (Major, Minor, Patch)$, the evolution of the mean and median value on a quadrimestrial basis from December 2010 until March 2018. We observe that the transitive version-based lag is increasing over time.

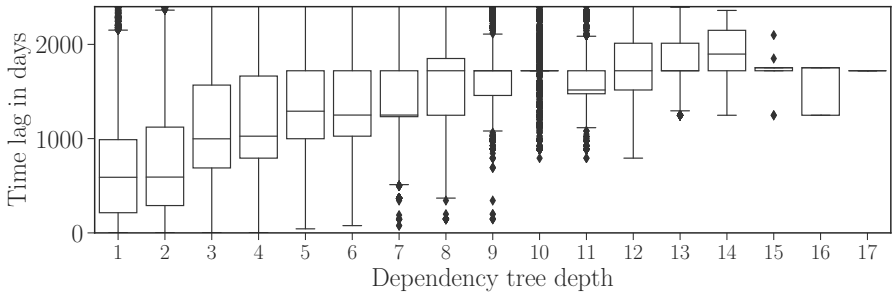

Figure 5.14: Distribution of $\mathbf{deplag}^{+}_{time}$ of the latest releases of all npm packages on 13 March 2018, grouped by transitive dependency tree depth.

We also correlated the transitive time lag $\mathbf{deplag}^{+}_{time}$ of package releases to their transitive runtime dependency tree depth. On the last considered snapshot (13 March 2018) we measured $\mathbf{deplag}^{+}_{time}$ for the latest available package release. Figure 5.14 shows

the distribution of these values, grouped by dependency tree depth. We observe that package releases with deeper dependency trees tend to have a higher transitive time lag. This is due to the fact that releases with deep dependency trees are more likely to have more outdated dependencies, and consequently they are more likely to have higher technical lag. Also, we observed that technical lag somehow "accumulates" from one level of the dependency tree to another one, implying that the more levels a dependency tree has, the higher the technical lag could be.

Summary: Technical lag induced by transitive runtime dependencies is increasing over time and is related to the depth of the transitive dependency tree. Deeper dependency trees have more outdated dependencies, inducing a higher technical lag.

5.5 Discussion

The instantiation of the formal technical lag model to the case study of the npm package manager allowed us to empirically analyze how the technical lag of JavaScript package releases evolves over time. At the same time, the formalization of the concept in general will enable comparison of this analysis with case studies of other package managers, and other types of component-based software distributions that aim to deploy systems from large collections of components.

Our findings show concrete evidence of technical lag induced by direct and indirect dependencies of both npm package releases and external GitHub applications that rely on them. This reflects the known tension between two forces that any maintainer of some software deployment faces. On the one hand, deployments would ideally use the most recent releases of their dependencies as soon as these become available, in order to benefit from the latest functionality and bug fixes. On the other hand, deployments suffer from technical lag in practice, because some dependency constraints do not allow for updating. Moreover, maintainers sometimes consciously choose not to update because they feel they do not need the new functionality provided by updates. In the specific case of security vulnerabilities, it would be acceptable to keep using an older major version of a package as long as security patches are being backported to it. In some cases, depending on the specificities of the package manager, developers don't even have a choice, as they need to rely on older releases, either because upgrading would cause co-installability conflicts [80], or because of the cost and effort required to update. Therefore, we expected to find technical lag in npm. We quantified it in terms of aggregated time lag and version lag at package release time, and revealed the omnipresence of technical lag.

We studied how version constraint operators are used in npm package dependencies, and found that it had an important effect on the technical lag of package releases. By definition, the induced technical lag is determined by the constraints imposed on the

dependencies, as such constraints allow package maintainers to balance between dependency freshness and incompatibility. Only the use of the most permissive constraint operators (e.g. `latest` or `*`) do not increase technical lag. But this goes against the semantic versioning principles, as it may lead to backward incompatible updates when a new major version is released. Therefore, the use of the caret constraint (that allows for minor updates) and the less permissive tilde constraint (that only allows for patch updates) should be more frequently used for specifying dependency constraints. This was confirmed in previous work [81] where it has been shown that, if dependency constraints would rely on semantic versioning rules that enable automatic updates of backward compatible changes, nearly one out of five package releases would no longer suffer from technical lag. Therefore, package maintainers remain the ultimate responsible for the technical lag incurred by their package releases.

For external GitHub applications relying on npm packages, we saw an important use of strict constraints that could be explained by the need to pin dependency versions to facilitate the replication of environments, but necessarily lead to a higher technical lag. The type of dependency constraint being specified represents a package maintainer's conscious choice between "preferring to miss some updates and needing to update constraints manually" versus "preferring to benefit from dependency updates, even if that implies needing to adapt your own software to make it compatible".

5.6 Limitations

Our empirical analysis of npm relied on the libraries.io dataset. There is no guarantee that this dataset is complete (e.g., there may be missing package releases), but we did not observe any missing data based on a manual inspection of the dataset. Our analysis excluded some dependencies with packages we were unable to identify on npm, but this only represented a small fraction of all dependencies (<3%) and is hence unlikely to affect the results. Considering the full set of available npm packages also constitutes a threat to validity. As explained in the data extraction in Section 5.2 we had to sanitise the dataset by excluding a number of packages before starting the analysis. Since there is no automatic way of identifying all of them, we may have missed some that should have been not considered.

The way in which we processed version numbers and dependency constraints may have influenced our results. Each dependency constraint was classified in a specific category (e.g., caret, strict, latest) depending on the syntax used by the constraint. This is, however, an oversimplification. For example, the following three constraints have the same meaning, even though they use a different notation: $\sim$ 1.2.0, 1.2.x, and 1.2. We also did not consider the difference between package releases with major version number 0 (e.g., 0.1.1) and those with a major version of 1 or above. npm treats these cases differently. For example *caret*

and *tilde* have exactly the same meaning when evaluated against 0.y.z while this is not the case for higher major version numbers. A large number of npm package releases (50%) have major release number 0, and we found that 17% of the dependencies in our dataset specified a *caret* or *tilde* version constraint to a 0.y.z version. This may have affected some of our findings.

A similar threat relates to the use of prerelease tags on version numbers in npm package releases. We have excluded such releases from our analysis because they should not be treated in the same way as regular releases as they are considered to be unstable according to the npm semver "A pre-release version indicates that the release is unstable and might not satisfy the intended compatibility requirements as denoted by its associated normal version".

5.7 Conclusion

To validate the potential of the proposed technical lag measurement framework, we have instantiated it for the case of npm, one of the largest and most used package managers. We studied the technical lag of npm package releases, taking into account their direct and transitive dependencies. We considered a time-based and a version-based notion of technical lag. This required us to take into account the specificities of the npm package manager, such as how it supports the semantic versioning policy, and how it allows to specify version constraints on package dependencies. To evaluate how the framework can assist in decision making, we have used it to track the evolution of technical lag of all package releases in npm over a period of more than seven years. We also studied how different types of dependency constraints affect the overall lag, and how developers could reason about such information to decide which constraints they should use for their releases. Finally, we have identified a large collection of *JavaScript* applications in GitHub using npm package releases, and have studied the technical lag for them.

From this empirical study of npm, we found that technical lag induced by direct dependencies is increasing over time, and development dependencies tend to induce more technical lag than runtime dependencies. The main source of technical lag appears to be the type of constraints used by the dependencies. The technical lag is increasing over time because more dependencies are used with too strict constraints, in both npm packages and external applications. Due to their different purpose, the technical lag of external applications hosted on GitHub is higher than the technical lag of npm packages. We also found that the technical lag induced by transitive dependencies can be very high and is increasing over time. Moreover, the transitive technical lag is correlated with the dependency tree depth of package releases. Finally, we observed some important changes in the technical lag over time due to decisions and policy changes made in the npm package manager.

Chapter 6

Technical Lag in Docker Containers

Software container images usually include a collection of software packages corresponding to the used operating system, plus other third-party packages. Once an image is built, its packages remain *frozen* until the image is updated. In many cases, such packages will become outdated because of the availability of new versions that are not installed in the containers.

In this chapter, we create different instantiations of the technical lag framework for the Docker container ecosystem and we empirically compute and analyze it using vulnerabilities and bugs at the level of system and third-party packages that are included in Docker images hosted on Docker Hub.

The content of this chapter is mainly based on our previous publications in the SANER 2019 proceedings [82, 83]. The instantiation of the technical lag framework for Docker containers presents new, unpublished work.

6.1 Introduction

Packaging software into containers has become a common practice during the last years [84]. In particular, Docker containers are a popular schema to provision multiple software applications on a single host. A container is a running image, which includes its own system libraries, configuration files, and software [20], providing support for both Linux-based and other operating systems [45, 85]. Docker allows for the creation of registries, providing a common place to share Docker images. With more than $2.1M$ images (April 2019), Docker Hub is one of the largest of such registries.

Images in Docker Hub are organized in *repositories*, each one providing a set of versioned Docker images. Repositories can be private or public, which in turn are split into official and community repositories. An official repository contains public and certified images from recognized vendors (e.g., ElasticSearch, Debian, Alpine). Images in official repositories are frequently used as the base for other Docker images, since they are supposed to be secure and well maintained. Community repositories can be created by any user or organization [86].

Usually, when Docker images are built with Linux-based operating systems, they follow the packaging model for their Linux distribution of choice and include system packages that correspond to the used distribution (e.g., Alpine or Debian). In some cases they also include a collection of third-party packages that come from popular package managers like *npm*, *PyPI* or *CRAN* (the default package managers for *JavaScript*, *Python* and *R*). Once the image is built, packages remain *frozen* (for a certain version of that image). From time to time, a new version of the image is built, using a newer version of the included packages. However, the old image may continue to be in use, as it may be deployed as a container in production. Those deployed containers corresponding to less recent images may include outdated packages with known security vulnerabilities and bugs, some of which are known to be fixed in newer package versions. Since the containers may run in production, they can be exposed to exploits of those known vulnerabilities, and problems due to those known bugs.

Nevertheless, deployers of containers may prefer to stick to older image versions, because they are known to work well and have been tested in production for a long time. In fact, *reproducibility* is one of the main characteristics of Docker containers, in the sense that using container images provides isolation from evolving dependencies and changes in packages that may break working systems. This is a strong incentive to stick to an older image version because it "just works", and upgrading to a new container image version always involves some risk. Thus, deployers always need to carefully balance the need to update to new image versions containing fixes of known vulnerabilities and bugs, and the risk of breaking a working system due to unexpected or incompatible changes in upgraded versions of the packages contained in the image.

This compromise has been widely reported in literature. According to a 2015 survey by Red Hat and Forrester [87], security is a top concern when deciding whether to deploy containers. A 2016 survey by *DevOps.com* and *RedMonk* [88] revealed that users who are more concerned by image security focused on scanning simple *Common Vulnerabilities and Exposures* (CVE) on the operating system. A 2017 survey by *Anchore.io* focused on the landscape of practices being deployed by container users [89]. One of the questions was: *"Other than security, what are the other checks that you perform before running application containers?"* The top answers related to software packages were: required packages ($\sim 40\%$ of the answers); presence of bugs in major third-party software ($\sim 33\%$); and verifying whether third-party software versions are up-to-date ($\sim 27\%$).

To support deployers of container images in this everyday compromise, we propose a method to assess how *outdated*, *vulnerable*, and *buggy* Docker images are with respect to the latest available releases of the system packages and third-party packages they include. The method is based on the concept of *technical lag*, which we use to estimate the difference of releases between the image deployed in production and the most ideal version of this image. In this chapter, we instantiate the technical lag framework of Chapter 4 to the Docker case study, and we conduct an empirical study to measure technical lag in terms of updates, security vulnerabilities and bugs for a large set of Docker images hosted on Docker Hub. As a case study, we focus on the Debian system and include npm third-party packages. This choice is motivated by their popularity in Docker containers.

6.2 Debian Packages in Docker Containers

6.2.1 Method and Data Extraction

Our study is based on pulling Docker images from Docker Hub, identifying which packages are installed in them, and computing the technical lag for the image by aggregating the technical lag of its packages. We will measure the technical lag of individual packages in terms of version updates, vulnerabilities, and bugs. Our initial sample is composed of all official images in Docker Hub that are based on Debian, and the most pulled community images based on Debian. Therefore, we only need to compute technical lag for Debian packages.

The overall process is composed of four tasks: (1) identification of Docker Hub base images for Debian, defining our base set; (2) identification of Docker Hub images in our dataset, including those derived from the base set; (3) analysis of all those images, matching their packages to a historical archive of all Debian packages; and (4) identification of bug and vulnerability reports for those packages, based on a historical database with those details for Debian packages.

Figure 6.1 shows how we used the main data sources for our study. The next subsections explain in detail how we gathered the used datasets.

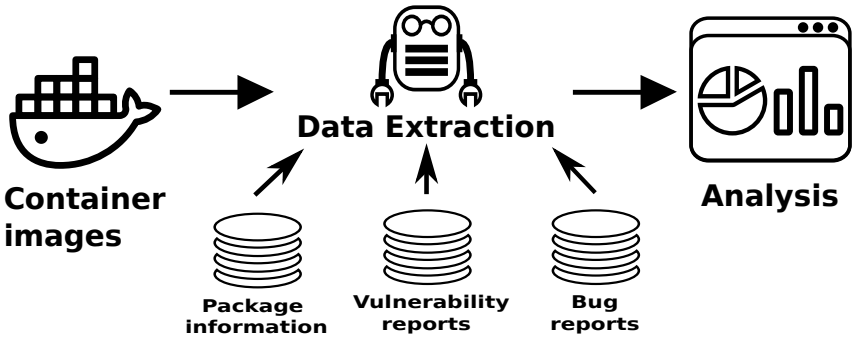

Figure 6.1: Process of the Docker container package analysis.

6.2.1.1 Base Images for Debian

We decided to work with Docker images based on a Linux-distribution, because packages in such images are usually well maintained. We selected the Debian distribution of Linux because of its maturity and widespread use[1] in Docker Hub. On October 1st 2018, the Debian repository on Docker Hub had more than 125M pulls[2].

While it is possible to create Docker images from scratch, most of them are based on others, which in the end are built on *base images* that do not rely on any other image except for the Docker-reserved minimal image named "scratch"[3]. Since we want to deal with images based on Debian, we first identified Debian base images[4].

The Debian project maintains packages for several simultaneous release lines (Debian distributions) [40]. The most important distributions are *Testing, Stable* and *Oldstable*. In *Testing*, packages are updated frequently, when new releases have been inspected and validated (e.g., lack of critical bugs, successful compilation, etc). At some points in time, when *Testing* as a whole reaches a certain level of quality and stability, it is "frozen", and their packages are used to produce a new *Stable* distribution. Upon release of a *Stable* version, the former one becomes *Oldstable*, and the *Oldstable* becomes *Oldoldstable*. While updates in *Testing* usually come with new functionality, updates in *Stable* and *Oldstable* include only the most important fixes or security updates. Currently, there is no security support for *Oldoldstable* and older distributions. Thus, we chose to analyze Debian base images in Docker Hub only for *Testing, Stable* and *Oldstable*. Table 6.1 shows general information about the Debian versions considered for this work.

[1] https://www.ctl.io/developers/blog/post/docker-hub-top-10/
[2] `https://registry.hub.docker.com/v2/repositories/library/debian/`
[3] https://docs.docker.com/develop/developimages/baseimages/
[4] https://hub.docker.com/_/debian/

Table 6.1: General information about the considered Debian versions.

Version name	Version	Distribution type	Release date as stable
Buster	Debian 10	Testing	-
Stretch	Debian 9	Stable	2017-06-17
Jessie	Debian 8	Oldstable	2013-04-25

6.2.1.2 Identifying Analyzed Images

Images in Docker Hub are named with the name of the repository, followed by a colon and a tag (e.g., *imageRepo:Tag*). Any image can be tagged more than once, and therefore may have more than one name (e.g., *debian:testing* and *debian:testing-20181011*). In the case of community images, the name of the repository usually starts with the name of the organization producing the images (e.g., *organizationName/ImageName*). Therefore, full image names tend to have the form *organizationName/ImageName:Tag*.

Each image is composed of one or many *intermediate images* called *layers*. Each layer is related to a change caused by commands that happened in the Dockerfile[5] used to produce the image, and has a unique hash signature.

For example, the Dockerfile of the *debian:stretch* image is:

```
FROM scratch
ADD rootfs.tar.xz /
CMD ["bash"]
```

When building the image with this Dockerfile, a single layer is produced:

```
debian:stretch Layers: [ "sha256:e1df5dc88d2cc2cd9a1b1680ec3cb" ]
```

This image can be used by other Dockerfiles as their base image using *"FROM debian:stretch"*. Each image produced from those Dockerfiles will contain the layer(s) of the base image. For example, *debian:stretch-backports* is produced with this Dockerfile:

```
FROM debian:stretch
RUN echo 'deb http://deb.debian.org/debian stretch-backports main'
> /etc/apt/sources.list.d/backports.list
```

The resulting *debian:stretch-backports* image includes the layer found in *debian:stretch*, its base image:

[5]https://docs.docker.com/engine/reference/builder/

```
debian:stretch-backports Layers: [
"sha256:e1df5dc88d2cc2cd9a1b1680ec3cb",
"sha256:4c1b8d4c6076530dcd195cbc309e5" ]
```

Therefore, we can identify Docker images derived from Debian base images by checking if they contain their layers. Using the Docker Hub API we extracted in March 2018 all available image names from the 124 official repositories, and those with at least 500 pulls from the top 30,000 community repositories (by number of pulls). Using the skopeo tool[6] we inspected images corresponding to all those image names, identifying unique images, and retrieving those that included layers from the set of Debian base images. From 14,653 image names in official repositories, we found 2,453 unique images (i.e., 4,769 names) based on our set of Debian images. From 30,000 community repositories, we found 4,927 unique images derived from our Debian set. All of them together compose our dataset of 7,380 images. Table 6.2 shows the number of images found for each Debian version.

Table 6.2: Number of Docker images per Debian distribution.

Containers	Buster / Testing	Stretch / Stable	Jessie / Oldstable	Total
Official	150	620	1,683	2,453
Community	86	1,248	3,593	4,927
Total	236	1,868	5,276	7,380

6.2.1.3 Identifying Installed Packages

Docker containers based on Debian include specific versions of Debian binary packages. Binary packages are produced from source packages. They are needed to find vulnerabilities and bug reports.

For tracking binary packages and finding their metadata (including the name and version of the source package from which they were produced), we extracted daily snapshots of all *amd64* binary packages for *Oldstable* (Jessie), *Stable* (Stretch) and *Testing* (Buster) distributions from the official and security Debian Snapshot repositories[7]. Then, we pulled each Docker image in our image dataset, and identified their packages using regular Debian tools (`dpkg -l`). We matched them to our dataset obtained from Debian Snapshot, finding in it more than 99% of the packages in our image dataset (1,379,163 package versions in official images, and 561,982 in community images). We found a median number of 190 and 261 installed packages in official and community images, respectively.

[6] skopeo is a utility which inspects a repository on a Docker registry: `https://github.com/containers/skopeo`

[7] `snapshot.debian.org/archive/debian/` and `snapshot.debian.org/archive/debian-security/`

6.2.1.4 Vulnerability Reports

To find out known security vulnerabilities for the Debian packages in our image dataset, we used the Debian Security Bug Tracker[8] as of 2018-03-18. For each package, the status of known vulnerabilities is maintained by the Debian Security Team, using data from different data sources (CVE database[9], National Vulnerability Database "NVD"[10], etc).

The Debian tracker maintains information about vulnerabilities at the source package level. A Debian vulnerability report contains information about affected source packages, severity, status, Debian bug id (if available), affected distributions, fixed version (if available), etc. Using it we can link vulnerabilities to source packages, and from there (using the Debian Snapshot dataset) to binary packages in our container images of interest. For each reported vulnerability for a package version present in one of the analyzed image versions, we say that the corresponding package version is *vulnerable* if the vulnerability is still open, or if the vulnerability has been fixed in a more recent package version than the one installed in the analyzed image version.

6.2.1.5 Bug Reports

For bug reports we relied on the *Ultimate Debian Database*[11] to query for known bugs in the package versions installed in our container image versions. *UDD* is a continuously updated system that gathers a variety of Debian data in the same SQL database [90]: bugs, packages, upload history, maintainers, etc.

UDD contains information about all bug reports, including those that were archived. To identify if a package version is "buggy", we queried UDD for all bug reports for that package. For each reported bug we checked if the specific package version was higher or equal to the version where the bug was first found. In case the bug report is resolved, we also verified if the package version is lower than the one fixing the bug.

6.2.2 Instantiating the Technical Lag Framework to Debian packages used in Docker containers

In Chapter 5, we applied the technical lag concept to npm dependencies used in npm packages. In this chapter, the analysis focuses on package releases that are used in Docker containers. To do so, we instantiate the technical lag framework to Debian package releases installed in Docker container images.

[8] `security-tracker.debian.org/tracker/data/json`
[9] `cve.mitre.org/cve/`
[10] `nvd.nist.gov`
[11] `udd.debian.org/bugs/`

As explained in 6.2.1.1, the Debian project maintains packages for several simultaneous distribution lines. Therefore, each package version is released within a specific Debian distribution. With time passing, these package releases encounter different kind of issues (e.g., vulnerabilities, stability bugs, performance bugs, etc).

Definition 10. Package releases

Let $\mathcal{P} \subset \mathcal{N} \times \mathcal{V} \times \mathcal{T} \times \mathcal{D}$ be the set of all package releases available in Debian, where $\mathcal{N}$ is the set of all possible package names, $\mathcal{V}$ is the set of all possible version numbers, $\mathcal{T}$ is the set of all possible time points and $\mathcal{D}$ is the set of all available Debian distributions (e.g., Buster, Stretch, Jessie, etc). Each package release $p = (p_{name}, p_{version}, p_{time}, p_{distro}) \in \mathcal{P}$ has an associated package name $p_{name} \in \mathcal{N}$, a version number $p_{version} \in \mathcal{V}$, a release date $p_{time} \in \mathcal{T}$ and a Debian distribution $p_{distro} \in \mathcal{D}$. Based on Debian specifications[12], we assume a total order on $\mathcal{T}$ and $\mathcal{V}$.

$\forall d \in \mathcal{D}$ and $\forall t \in \mathcal{T}$, we will denote by $\mathcal{P}_d = \{p \in \mathcal{P} \mid p_{distro} = d\}$ the set of all package releases available in a given Debian distribution $d \in \mathcal{D}$, and by $\mathcal{P}_t = \{p \in \mathcal{P} \mid p_{time} \leq t\}$ the set of all package releases available at a given time $t \in \mathcal{T}$.

Definition 11. Package release issues

security : $\mathcal{P} \times \mathcal{T} \rightarrow \mathbb{N}$ such that **security**(p, t) corresponds to the number of reported vulnerabilities of p at time t. This information can be extracted from the Debian Security Tracker.

stability : $\mathcal{P} \times \mathcal{T} \rightarrow \mathbb{N}$ such that **stability**(p, t) corresponds to the number of reported bugs of p at time t. This information can be extracted from the Ultimate Debian Database.

Note that a package release can have issues (i.e., bugs or vulnerabilities) with different severity levels. We could take into account the severity level in the definitions above, by using a weighted sum that assigns different weights to different severity levels. For simplicity, we have ignored this severity, attaching an equal weight to each bug or to each vulnerability.

Definition 12. Packages installed in a container

In practice, software containers include a set of installed system package releases. In the case of Docker containers that are based on Debian, they include Debian package releases.

Let $\mathcal{C}$ be the set of all possible Debian-based software containers, where each container $c \in \mathcal{C}$ has an associated build date $c_{time} \in \mathcal{T}$. We define:

installed : $\mathcal{C} \rightarrow \mathbb{P}(\mathcal{P})$ such that **installed**(c) corresponds to all Debian package

[12] `https://www.debian.org/doc/debian-policy/ch-controlfields.html#version`

releases installed in the container c.

available : $\mathcal{C} \rightarrow \mathbb{P}(\mathcal{P})$ such that **available**$(c) = \{p \in \mathcal{P} \mid \exists p' \in$ **installed**(c), $p_{name} = p'_{name} \wedge p_{distro} = p'_{distro} \wedge p_{time} \leq c_{time}\}$ corresponds to all available releases of the installed package versions **installed**(c) at the container's build date c_{time}, within the same Debian distribution.

We aim to empirically study how outdated Debian package releases in Docker containers are with respect to the latest available, the most secure and the most stable package releases. To do so, we need to define three different instantiations of the technical lag framework: version-based, security-based and stability-based.

Definition 13. Version-based instantiation of the technical lag framework.

We define $\mathcal{F}^{Debian}_{version}$ as the function that instantiates the technical lag framework $\mathcal{F}$ for a given container $c \in \mathcal{C}$:

$$\mathcal{F}^{Debian}_{version}(c) = (\mathcal{A}, \mathbb{N}, \mathbf{ideal}_{version}, \mathbf{delta}_{version}, \mathbf{agg}_{version})$$

where:

- $\mathcal{A} =$ **available**(c), is the set of all available package releases of the container c.
- $\mathbb{N}$ is the set of possible version lag values. Intuitively, each element corresponds to a number of *missed* releases.
- $\mathbf{ideal}_{version}(p) = \max_{p'_{version}} \{p' \in \mathcal{A} \mid p'_{name} = p_{name}\}$.

  Intuitively, the function $\mathbf{ideal}_{version}$ mimics the choice of the Debian package manager *APT*, i.e., it returns the *highest available version* in $\mathcal{A}$ of a given package.
- $\mathbf{delta}_{version}(p, q) =\mid coll \mid$, where:

  $coll = \{r \in \mathcal{A} \mid p_{version} < r_{version} \leq q_{version} \wedge p_{name} = r_{name} = q_{name}\}$
  In this definition, the lag is the number of all releases between p and q. For the sake of simplicity, we do not distinguish between version types [13].
- $\mathbf{agg}_{version}(L) =\mid \{x \in L | x > 0\} \mid$, where $L \subseteq \mathbb{N}$ is a set of version lags. Intuitively, the aggregation corresponds to the number of outdated package versions.

Definition 14. Security-based instantiation of the technical lag framework.

We define $\mathcal{F}^{Debian}_{security}$ as the function that instantiates the technical lag framework $\mathcal{F}$ for a given container $c \in \mathcal{C}$:

[13] `https://www.debian.org/doc/debian-policy/ch-controlfields.html#version`

$$\mathcal{F}^{Debian}_{security}(c) = \Big(\mathcal{A}, \mathbb{N}, \mathbf{ideal}_{security}, \mathbf{delta}_{security}, \mathbf{agg}_{security}\Big)$$

where:

- $\mathcal{A} = \mathbf{available}(c)$, is the set of all available package releases of the container c.
- $\mathbb{N}$ is the set of possible security lag values. Intuitively, each element corresponds to a number of reported vulnerabilities.
- $\mathbf{ideal}_{security}(p) = \max_{p'_{version}}\{p' \in \mathcal{A} \mid p_{name} = p'_{name} \wedge \forall r \in \mathcal{A}, r_{name} = p'_{name} \implies \mathbf{security}(p', c_{time}) \leq \mathbf{security}(r, c_{time})\}$

  Intuitively, for a given package release p, the function $\mathbf{ideal}_{security}$ returns the package release with the highest version (within the same Debian distribution as p) having the *lowest number of vulnerabilities* .
- $\mathbf{delta}_{security}(p, q) = \mathbf{security}(p, c_{time}) - \mathbf{security}(q, c_{time})$ computes the difference between the number of vulnerabilities of package releases p and q.
- $\mathbf{agg}_{security}(L) = \sum_{x \in L} x,$ with $L \subseteq \mathbb{N}$, computes the sum over a set of lags.

Definition 15. Stability-based instantiation of the technical lag framework.

We define $\mathcal{F}^{Debian}_{stability}$ as the function that instantiates the technical lag framework $\mathcal{F}$ for a given container $c \in \mathcal{C}$:

$$\mathcal{F}^{Debian}_{stability}(c) = \Big(\mathcal{A}, \mathbb{N}, \mathbf{ideal}_{stability}, \mathbf{delta}_{stability}, \mathbf{agg}_{stability}\Big)$$

where:

- $\mathcal{A} = \mathbf{available}(c)$, is the set of all available package releases of the container c.
- $\mathbb{N}$ is the set of possible stability lag values. Intuitively, each element corresponds to a number of reported bugs.
- $\mathbf{ideal}_{stability}(p) = \max_{p'_{version}}\{p' \in \mathcal{A} \mid p_{name} = p'_{name} \wedge \forall r \in \mathcal{A}, r_{name} = p'_{name} \implies \mathbf{stability}(p', c_{time}) \leq \mathbf{stability}(r, c_{time})\}$

  Intuitively, for a given package release p, the function $\mathbf{ideal}_{stability}$ returns the package release with the highest version (within the same Debian distribution as p) with the *lowest number of bugs*.
- $\mathbf{delta}_{stability}(p, q) = \mathbf{stability}(p, c_{time}) - \mathbf{stability}(q, c_{time})$ computes the difference between the number of bugs of package releases p and q.

- $\mathbf{agg}_{stability}(L) = \sum_{x \in L} x,$ with $L \subseteq \mathbb{N}$, computes the sum over a set of lags.

Now, we can directly compute $\mathbf{techlag}_{\mathcal{F}_\alpha^{Debian}(c)}(p)$ and $\mathbf{agglag}_{\mathcal{F}_\alpha^{Debian}(c)}(P)$ (i.e., the instantiation of Definitions 2 and 3 in Chapter 4) in terms of the above definitions, where $\alpha \in \{\text{version}, \text{security}, \text{ stability}\}$.

Notation 2. In the remainder of this section we use the shortcut notations $\mathbf{techlag}_\alpha^{Debian}(p, c)$ for $\mathbf{techlag}_{\mathcal{F}_\alpha^{Debian}(c)}(p)$ and $\mathbf{agglag}_\alpha^{Debian}(P, c)$ for $\mathbf{agglag}_{\mathcal{F}_\alpha^{Debian}(c)}(P)$, where $\alpha \in \{\text{version}, \text{security}, \text{stability}\}$.

Based on the above definitions, we can define the technical lag of a container deployment.

Definition 16. Technical lag of a container deployment

For $\alpha \in \{\text{version, security, stability}\}$, we define:

$$\mathbf{contlag}_\alpha^{Debian} : \mathcal{C} \rightarrow \mathbb{N} : c \rightarrow \mathbf{agglag}_\alpha^{Debian}(\mathbf{installed}(c))$$

Given a container $c \in \mathcal{C}$, $\mathbf{contlag}_\alpha^{Debian}(c)$ returns the aggregated lag of all Debian package releases installed in it. For example, suppose that in March 2018 there was a Debian Stretch container c that was using only two package releases *imagemagick 8:6.9.7.4+dfsg-11* and *cups 2.2.1-8*, where *imagemagick 8:6.9.7. 4+dfsg-11* had 132 security vulnerabilities and *cups 2.2.1-8* had 4 security vulnerabilities. Assume that these packages had many other available releases in Debian Stretch at that time. The most secure of these package releases (i.e., $\mathbf{ideal}_{security}(\textit{imagemagick 8:6.9.7.4+dfsg-11})$ and $\mathbf{ideal}_{security}(\textit{cups 2.2.1-8})$) were *8:6.9.7.4+dfsg-11+deb9u4* with 58 security vulnerabilities and *2.2.1-8+deb9u1* with 3 security vulnerabilities, respectively.

Therefore, $\mathbf{techlag}_{security}^{Debian}(\textit{imagemagick 8:6.9.7.4+dfsg-11}, c) = 132 - 58 = 74$ and $\mathbf{techlag}_{security}^{Debian}(\textit{cups 2.2.1-8}, c) = 4 - 3 = 1$. Consequently, the aggregated technical lag for container c is $\mathbf{contlag}_{\mathbf{security}}^{\mathbf{Debian}}(c) = 74 + 1 = 75$ vulnerabilities, which is the sum of lags of all installed package releases.

6.2.3 Empirical Evaluation

Research Questions

In this section, we empirically analyze Debian system packages that are installed in Docker containers. We report results about the technical lag, security vulnerabilities and bugs for $2,453$ official and $4,927$ community Docker Hub images based on the Debian Linux distribution.

The research questions that we address in this study are:

RQ_0: *How often are Docker images updated?* With this preliminary research question, we aim to know first when Docker images were last updated and how often they got updated.

RQ_1: *How outdated are container packages?* We identify Debian package releases that are installed in the analyzed Docker images and we compute their technical lag in terms of versions w.r.t the highest available versions in the Debian archive.

RQ_2: *To which extent do containers suffer from security vulnerabilities in packages?* For all installed Debian packages, we quantify the number of reported vulnerabilities they have in the Debian security tracker.

RQ_3: *How vulnerable are container package releases compared to the least vulnerable versions?* For all images, we compute the technical lag that their installed package releases suffer from in terms of number of vulnerabilities w.r.t the least vulnerable versions in Debian.

RQ_4: *To which extent do containers suffer from bugs in packages?* For all installed Debian package releases, we quantify the number of reported bugs in the Debian bug tracker.

RQ_5: *How buggy are container package releases compared to the least buggy versions?* For all images, we compute the technical lag that their installed package releases suffer from in terms of number of bugs w.r.t the least buggy versions in Debian.

RQ_6: *How long do bugs and security vulnerabilities remain unfixed?* With this research question, we aim to shed some light on the time needed before vulnerabilities and bug reports are fixed and closed.

RQ_0: How often are Docker images updated?

In order to analyze the technical lag of Docker container packages, it is essential to know how often Docker maintainers update their images and when they were last updated, since images that have not been updated in a long time may have more outdated packages. This allows us to have a better understanding and carry out a fair comparison between the content of different containers.

In September 2017, *Anchore.io*, which is a company dedicated to container deployers, analyzed the official Docker images update history[14] and found that operating system images like Debian, Alpine or *Ubuntu* update images less often than non-OS images like *Redis, MySQL* or *Postgres*. They also noticed that Debian images are updated every month, which is the average compared to other OS images. Moreover, they observed that on some days many repositories push updates at the same time. Investigating this phenomenon, they found that in many cases this occurs the day after their base image *debian:latest* was updated.

[14] `anchore.com/blog/look-often-docker-images-updated/`

Figure 6.2 shows the number of images that were last updated per each year. We observe that the number of official images that were updated in 2018 is less than those that were updated in 2017, for the images that make use of the *Oldstable* version Jessie (Debian 8), while it is the opposite for the community images. Another important observation that should be taken into account for the rest of the study is that 48% of the community images and 66% of the official images were last updated before 2018. This can be explained by the number of images per repository: while official repositories have many images with different operating systems (i.e., including slim and full) and for different architectures, community repositories have predominantly only one unique image (latest) with different tags. Thus, in official repositories new images emerge while others stop being updated; and community repositories tend to keep updating their images without creating new ones.

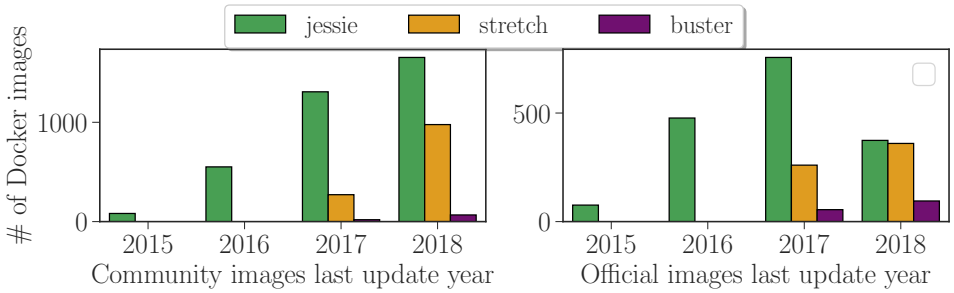

Figure 6.2: Year of last update of Docker images, grouped by Debian distribution and image type (community or official).

Findings: More than half of the Docker images have not been updated for four months.

RQ_1: How outdated are container packages?

RQ_1 investigates how outdated the packages in Docker containers are, based on a quantification of their technical lag. Therefore, we start by exploring how many packages within containers are outdated (i.e., $\mathbf{contlag}^{\mathbf{Debian}}_{\mathbf{version}}$). Figure 6.3 shows the proportion of up-to-date and outdated packages in both official and community Docker containers, grouped by their Debian version. We observe that, regardless of the Debian version, most packages are up-to-date. The median proportion of up-to-date packages per container is 82% of all installed packages. We also notice that packages inside community containers are slightly more up-to-date (median 85%) than packages inside official containers (median 78%).

We statistically confirm this observation using a non-parametric *Mann-Whitney U* test. The null hypothesis assumes that the up-to-date package distributions of the community and official containers, grouped by their Debian version, are identical. For each pair of groups *(Official-Jessie, Community-Jessie), (Official-Stretch, Community-Stretch)*,

and *(Official-Buster, Community-Buster)*, we rejected H_0 with statistical significance ($p < 0.01$) when comparing the distributions of up-to-date packages of two groups of containers. However, for each comparison, we only found a small effect size ($|d| \leq 0.28$) using *Cliff's Delta*, a non-parametric measure quantifying the difference between two groups of observations.

When restricting our analysis to recent images only (i.e., those that were last updated in 2018), we found that packages in official containers are slightly more up-to-date than packages in community containers. Since community images are based on official images, this means that from all available official images, Docker community deployers tend to use the most recently updated ones, or they manually update all outdated packages inherited from old official images. We also found that the median proportion of up-to-date packages per container, in all cases, increased to 98% of all installed packages.

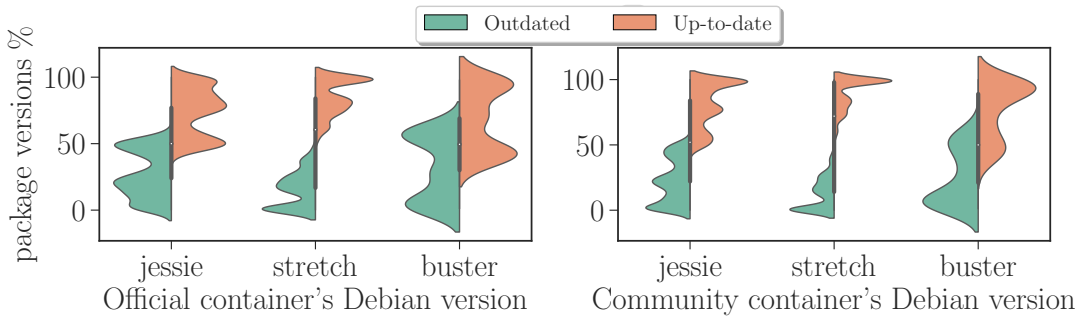

Figure 6.3: Proportion of up-to-date and outdated packages in Docker containers, grouped by their Debian version.

To quantify how outdated Debian packages are, we compute their *technical lag*, using the version-based instantiation of the technical lag framework in Section 6.2.2.

For all containers, we measured their packages' $\mathbf{techlag}_{version}^{Debian}$. Figure 6.4 shows the distribution of $\mathbf{techlag}_{version}^{Debian}$ of packages installed in Docker containers, grouped by the container's installed Debian version. At first sight, we observe that the distributions are highly skewed. However, the distribution for the containers using Stretch is more highly skewed than the others. Table 6.3 shows that the median version lag for both Jessie and Stretch containers is 1, while it is 2 versions for Buster. This small difference is related to the state of the Debian release. Because Buster is now in the *Testing* phase, many containers prefer not to depend on its packages since they are still subject to many changes, making it hard to keep up with its updating process. However, we can conclude that, in general, package releases in Docker containers are either up-to-date or lagging behind with a median of 1 to 2 versions.

Since we found that the proportion of up-to-date installed packages per container is high, we decided to investigate the used package releases and when they were created, in

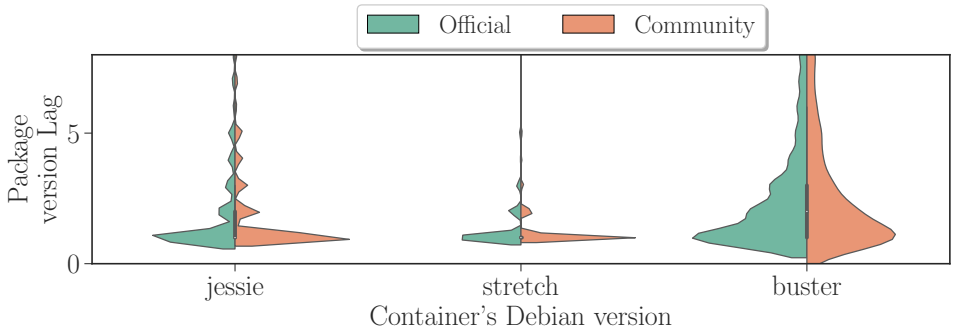

Figure 6.4: Violin plots of the distribution of $\mathbf{techlag}_{version}^{Debian}$ induced by outdated packages in Docker containers.

Table 6.3: Mean and median of packages' $\mathbf{techlag}_{version}^{Debian}$, grouped by Debian version and container type.

Containers	Official		Community	
	mean	median	mean	median
Jessie	2.13	1	1.93	1
Stretch	1.36	1	1.24	1
Buster	2.71	2	3.05	2

order to know how the updating process of Debian is working. This way we can know if images are keeping up with frequent updates of Debian or they simply make use of package releases that are no longer updated by Debian.

Considering only the up-to-date package releases this time, we traced back the date when they were first seen in Debian. Figure 6.5 shows the cumulative proportion of up-to-date package releases used in containers. We found that most of the package releases are old (i.e., they have been released many months ago). Moreover, we found that exactly 80% of the used Stretch package releases and 90% of the Jessie package releases were created before 2017-06-18, the release date of the *Stable* version of Stretch. We also found that 63% of the used Jessie package releases were created before the release date of the *Stable* version of Jessie (i.e., 2015-04-25). This means that used package releases tend to remain up-to-date because of the way in which Debian maintainers are creating and updating their packages.

Findings:

- One out of five installed package releases in containers is outdated.
- Authors of community containers tend to use recently updated official images.
- Outdated installed package releases are lagging behind one to two versions.

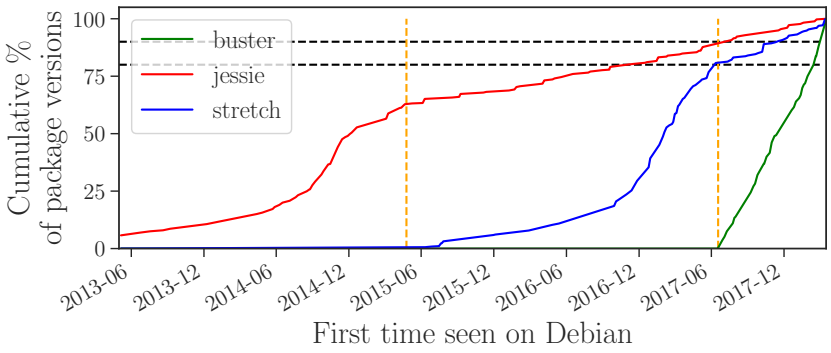

Figure 6.5: Cumulative number of used up-to-date package releases, by date of first appearance in Debian.

RQ_2: To which extent do containers suffer from security vulnerabilities in packages?

Verifying a software container for security vulnerabilities is important, since such vulnerabilities could be exploited to abuse the system. With RQ_2, we analyze if the presence of vulnerabilities in Docker containers is related to the presence of outdated package releases in the containers.

We found that only 12.2% (i.e., 488 out of 3,975) of all unique installed package releases (from both official and community containers) had security vulnerabilities. Figure 6.6 shows the distribution of vulnerabilities by their severity (not assigned, unimportant, low, medium or high) and status (open, resolved and undetermined). We found that 49.9% (i.e., 12,806) of all vulnerabilities are *resolved*, while 48.6% (i.e., 12,479) are still *open*. A small proportion of 1.6% (i.e., 401) are *undetermined*. We also observe that the majority of vulnerabilities has a *medium* (37.2%), *unimportant* (20.2%) or *high* (18.3%) severity. However, we found that all containers are affected by these severity vulnerabilities. In fact, 96% of all containers are affected by all types of vulnerabilities, except for the *not assigned* vulnerabilities. This means that this small proportion of 12.2% of package releases causes the vulnerability of nearly all Docker containers.

Next, we computed the number of vulnerabilities per container. We obtained a mean value of 1,336 vulnerabilities and a median of 601. The important difference between mean and median signals a heavily skewed distribution. Indeed, we found a container with a maximum of 7,338 vulnerabilities. Using a *Mann-Whitney U* test we found statistically significant differences ($p < 0.01$) in the number of vulnerabilities per container between official and community distributions. However, the effect size was small ($|d| < 0.3$). Table 6.4 shows more details about the distribution of the number of vulnerabilities in Docker containers.

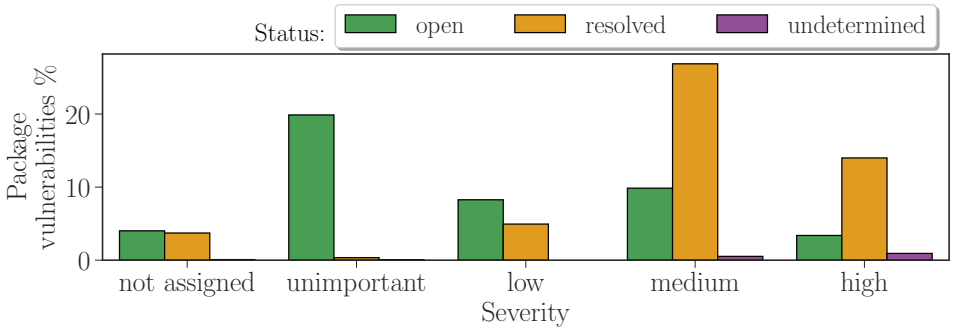

Figure 6.6: Proportion of vulnerabilities found in package releases in Docker containers, grouped by severity and status of the vulnerability.

Table 6.4: Minimum, median and maximum number of vulnerabilities per container, grouped by Debian version and container type.

Containers	Official			Community		
Debian	min	median	max	min	median	max
Jessie	155	658	7,106	155	916	7,338
Stretch	85	242	3,498	75	336	4,729
Buster	34	134	659	41	183	1,035

To study the relation between the outdated package releases in containers and the vulnerability of the container, we compared the number of outdated package releases and number of vulnerabilities per container. Considering both official and community containers, and without differentiating between vulnerabilities by severity or status, we plot the numbers in a scatter plot for different Debian versions (Figure 6.7). We visually observe a certain relationship between both metrics, especially for the Jessie containers: when the number of outdated package releases increases, the number of vulnerabilities tends to increase as well.

To verify our observations, we calculated Pearson's correlation coefficient R and Spearman's ρ over all package releases, using the following thresholds : $0 < \textit{very weak} \leq 0.2 < \textit{weak} \leq 0.4 < \textit{moderate} \leq 0.6 < \textit{moderately strong} \leq 0.8 < \textit{strong} \leq 1$. A moderately strong increasing correlation ($0.6 < R \leq 0.8$ and $0.6 < \rho \leq 0.8$) for Jessie and Buster, and only a moderate one for Stretch ($R = 0.53$ and $\rho = 0.42$) exists.

If we consider that the need for updating an outdated package release only arises when a fix for a known vulnerability is available, it is preferable to focus on the category of *resolved vulnerabilities*. We found a mean value of 347 resolved vulnerabilities per container, and a median value of 102. Repeating the above correlation analysis reveals a strong correlation between the number of resolved vulnerabilities and the number of

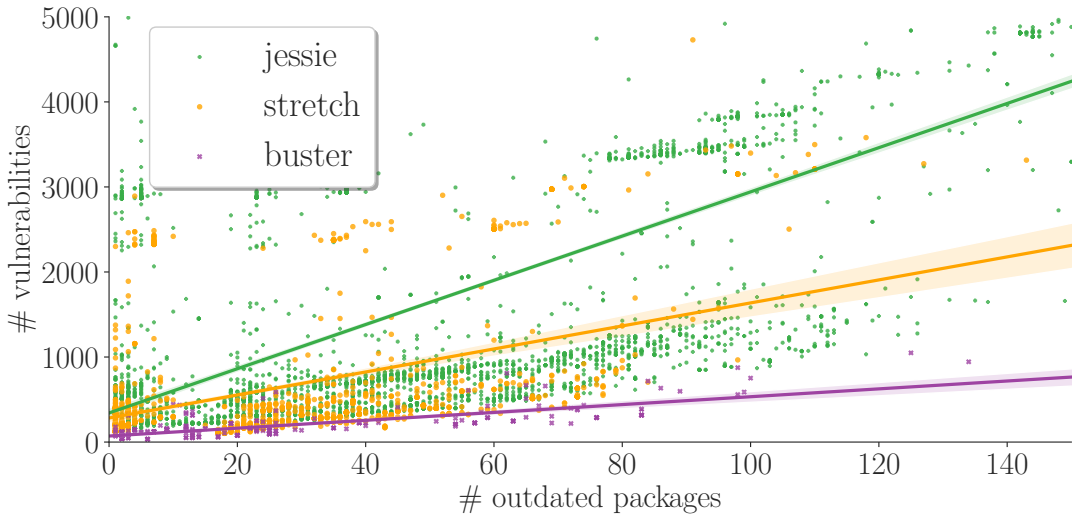

Figure 6.7: Number of outdated packages and vulnerabilities per container.

outdated package releases ($0.8 < R \leq 0.9$ and $\rho > 0.9$).

Table 6.5 shows the top 5 vulnerable official and community images with their number of vulnerabilities (i.e., #vulns), number of installed packages (i.e., #pkgs) and age. For the official images, only the top vulnerable image in a repository is presented. For example, the *perl* repository has many images with a high number of vulnerabilities; summed together, *perl* would be in the top 5. However, since these images provide the same functionality, we report only one: the most vulnerable image. A common characteristic about the top vulnerable containers is that they *have not been updated for more than two years*, and their number of installed packages is higher than the median over all containers. Thus, it is not surprising that these containers have high number of severity vulnerabilities.

Table 6.6 shows the top 5 most and least vulnerable source packages, with their number of vulnerabilities and the proportion of containers that make use of them. The three most vulnerable source packages *linux, chromium-browser* and *imagemagick* seem to have high number of binary packages: 433, 419 and 327, respectively. We did not observe any significant relation between the number of binary packages and the number of vulnerabilities. For instance, the source packages *mono* and *libreoffice* have 241 and 195 binary packages, but they have only 1 and 9 vulnerabilities, respectively. We also found some vulnerable packages (e.g., *audit* and *bzip2*) that are used by all containers, explaining why nearly all containers are affected by vulnerable packages.

Table 6.5: Top 5 vulnerable official and community Docker images. (Age in months.)

Official images	# vulns	# pkgs	age
perl:5.12.5-threaded	7,106	406	36.0
node:0.8.28	6,889	369	32.9
erlang:18.2.1	6,713	372	28.4
ruby:2.1.8	6,426	372	25.9
sentry:7.5.0	5,742	384	35.9
Community images	# vulns	# pkgs	age
weboaks/chromium-xvfb-node	7,338	448	27.8
jmoifutu/almakioski-processor-base	7,282	407	35.2
suitupalex/node-composer	7,167	384	26.2
youdowell/php-fpm-for-wordpress	7,155	339	25.5
newsdev/github-keys	7,106	405	35.5

Table 6.6: Top 5 vulnerable Debian source packages.

Most vulnerable package	#vulnerabilities	used by
linux	433	54.51%
chromium-browser	419	0.43%
imagemagick	327	28.13%
php5	186	2.3%
firefox-esr	139	0.09%
Least vulnerably package	#vulnerabilities	used by
audit	1	100.0%
bzip2	1	100.0%
db5.3	1	100.0%
pam	1	100.0%
sensible-utils	1	98.97%

Findings:

- Nearly half of the vulnerabilities have no fix.
- All containers have high severity vulnerabilities.
- The number of vulnerabilities depends on the Debian release used.
- The most vulnerable containers have not been updated for more than 2 years.
- The number of outdated packages in a container is strongly correlated to the number of resolved vulnerabilities that the same container suffer from.

RQ_3: How vulnerable are container package releases compared to their most secure versions?

In the previous research question, we found that Docker containers suffer from security vulnerabilities coming from Debian packages. To answer the current research question, we use the security-based instantiation of the technical lag framework in 6.2.2. This way, we can report the technical lag in terms of vulnerabilities that would be reduced if container deployers installed the *most secure* (i.e., least vulnerable) package releases.

Figure 6.8 shows $\mathbf{contlag}^{\mathbf{Debian}}_{\mathbf{security}}$, grouped by container Debian version. We observe that the aggregated security lag is related to the Debian version in the sense that containers with old package releases have a higher lag. Without distinguishing between image types, we found that the median security lag is 65, 1 and 2 severity vulnerabilities for Jessie, Stretch and Buster, respectively. This could be explained by the fact that Jessie package releases are older, they might have had more time to accumulate more reported vulnerabilities. We also observe a small difference between containers of official and community images. Official images tend to have a higher security lag. This is not surprising since we found that package releases are more up-do-date in community images than in official images, even if we found that community images have more vulnerabilities. The number of vulnerabilities is related to the number of installed packages, hence it is expected that community images have more vulnerabilities. This shows again that there is a relation between outdated package releases and the vulnerabilities they may induce.

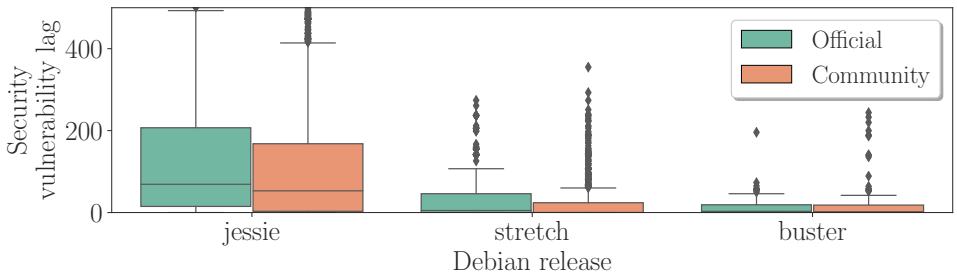

Figure 6.8: $\mathbf{contlag}^{\mathbf{Debian}}_{\mathbf{security}}$ induced by installed packages in Docker containers

However, since we did not find a very strong correlation between outdated package releases and all kinds of vulnerabilities in containers, we expect that in order to have the most secure package versions and reduce the security lag, deployers may need to downgrade their package releases. Indeed, to have the most secure package release, we found that 2.5% of the package releases installed in Docker container need to be downgraded and 22.5% of them need to be upgraded. The rest (75%) does not require any change, they are already having the most secure version (i.e., $\mathbf{techlag}^{Debian}_{security} = 0$ at the containers build date, March 2018).

As a consequence of the previous findings, we also found that in order to reduce security lag, package releases in official images require more upgrades or downgrades than in community images.

Findings:

- Debian Jessie containers have a higher security lag than other containers.
- Official images have higher security lag than community images.
- Some package releases need to be downgraded in order to reach the most secure version.

RQ_4: To which extent do containers suffer from bugs in packages?

This question concerns the presence of non-security-related bugs in Docker container package releases, and the relation between bugs and outdated package releases. Considering all package releases for both community and official images, we found that 50.1% (1,994 out of 3,975) of all unique installed source package releases have bugs. We also discovered that all containers have "buggy" package releases.

Figure 6.9 shows the distribution of bugs grouped by status (pending, forwarded, fixed) and severity (wishlist, minor, normal, important, high). The *high* category combined three different severity types: *serious, grave* and *critical*. We found that 65.5% (12,863) of all bugs are still *pending*, 7.3% (3,460) are *forwarded* and only 27.2% (30,922) are *fixed*. With respect to the severity, only 2.9% of all bugs are *high*, 27.7% are *important*, 50.2% are *normal* and the rest is *minor* or still in the *wishlist*.

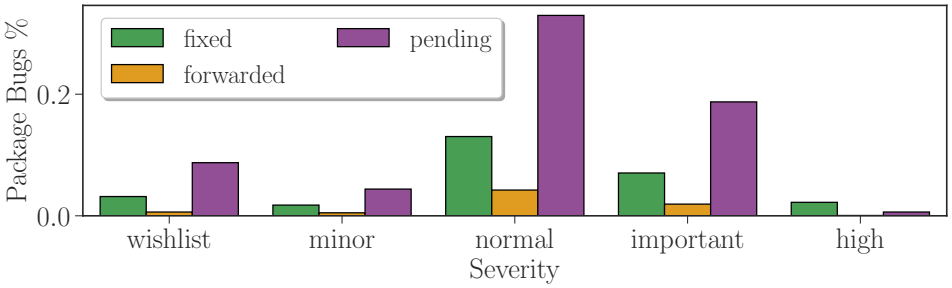

Figure 6.9: Proportion of bugs grouped by severity and status.

Since the majority of bugs are still *pending* – nearly two out of three bugs in package releases (65.5%) are without a fix, and one out of two package releases (50.1%) has a bug – we would expect the number of bugs per container to be higher than the number of vulnerabilities. Including both official and community containers, we found a mean value of 2,081 and a median value of 2,163 bugs per container. Focusing on *fixed bugs*

only, i.e., bugs that have been fixed in another release, we found a mean value of 678 and a median value of 729 fixed bugs per container. While these numbers may appear high, this is normal for Debian, that typically has thousands of open bugs at any point in time.

When comparing the number of bugs in official and community containers, we obtained a statistically significant difference ($p < 0.01$) using *Mann-Whitney U* test. The effect size was small ($|d| < 0.2$) for Buster and Stretch, and medium ($|d| = 0.28$) for Jessie. The number of reported bugs decreases with more recent versions of Debian. Table 6.7 shows details about the distribution of the number of bugs in Docker containers.

Table 6.7: Min, median and max of bugs per container grouped by Debian version and container type.

Containers	Official			Community		
Debian	min	median	max	min	median	max
Jessie	1,307	2,201	3,415	1,296	2,450	5,628
Stretch	962	1,683	2,665	828	1,759	3,285
Buster	213	560	776	278	561	1,098

We also studied the relation between the presence of outdated package releases and bugs in containers. Considering both official and community containers, and without differentiating between bug status or severity, Figure 6.10 shows a scatter plot, for different Debian versions, of the relation between the number of outdated package releases and the number of bugs found in each container. Opposite to what we observed for vulnerabilities in Figure 6.7, there only appears to be a relation between the number of bugs and number of outdated package releases for Buster.

To statistically verify our observations, we computed Pearson's R and Spearman's ρ correlation for all packages. For Jessie and Stretch, the correlation was weak to very weak ($R \leq 0.2$ and $\rho \leq 0.22$). For Buster, a moderate increasing correlation ($R = 0.58$ and $\rho = 0.55$) was observed. This is because Buster contains many new package releases, which still get new bug notifications. Jessie and Stretch, on the other hand, contain mainly stable and old package releases, that mainly contain old bugs. Indeed, 90% of Jessie bugs were created before July 2016, and 78% of Stretch bugs were created before its *Stable* release in June 2017.

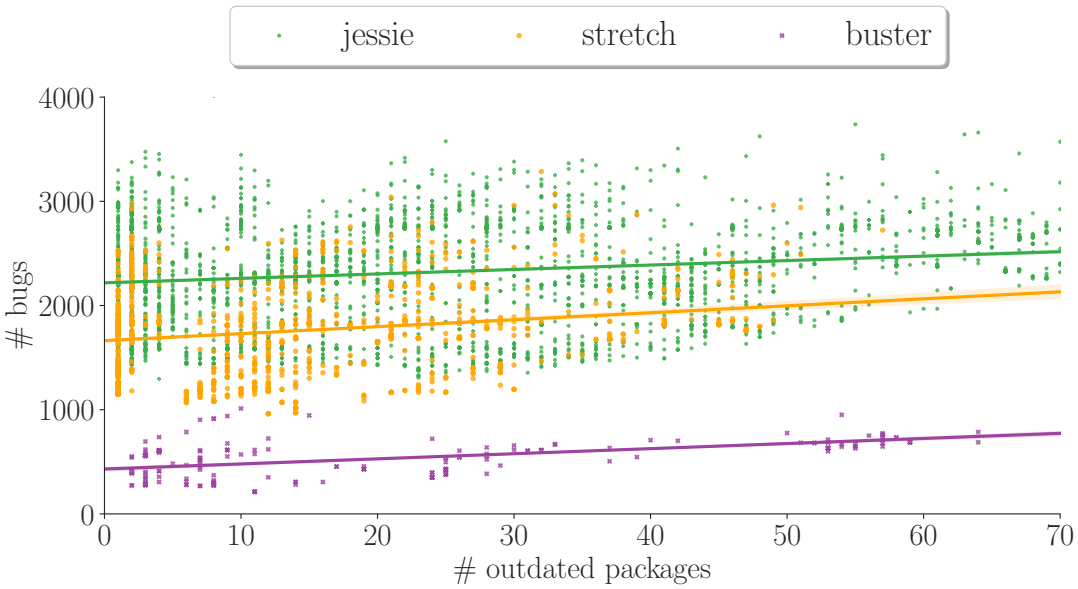

Figure 6.10: Number of outdated package releases and bugs per container.

Findings:

- All containers have buggy package releases.
- 65% of bugs in installed package releases are without a fix.
- The number of bugs is negatively correlated with the used Debian version.
- There is a weak correlation between the number of bugs and the number of outdated package releases in containers relying on the *Stable* and *Oldstable* Debian release.

RQ_5: How buggy are container package releases compared to their most stable versions?

Having found evidence that Docker containers suffer from other kind of bugs induced by installed Debian packages, we will now analyze and compute the stability lag as the difference in number of bugs between two package releases. To do so, we use the stability-based instantiation of the technical lag framework in 6.2.2. This way, we can report the technical lag in terms of the number of bugs that would be reduced if container deployers installed the *most stable* (i.e., least buggy) package releases.

Figure 6.8 shows $\mathbf{contlag}^{\mathbf{Debian}}_{\mathbf{stability}}$, grouped by container Debian version. We observe that contrary to what we found about the aggregated security lag, the aggregated stability lag is not related to the Debian version order. Debian Stretch is more recent than Jessie and older than Buster, however containers that make use of this Debian distribution have

less stability lag than other containers. Without distinguishing between image types, we found that the median stability lag is 187, 103 and 148 bugs for Jessie, Stretch and Buster, respectively. We also observe that community images have a slightly higher stability lag, even if they are more up-to-date. This is expected since we could not find a correlation between bugs and outdated package releases in the previous research question.

In order to reduce the stability lag in Docker containers, 22.4%, 12.3% and 27.5 % of the packages in Jessie, Stretch and Buster would need to be downgraded while only 6%, 4.4% and 6.3% of the package releases would need to be upgraded, respectively.

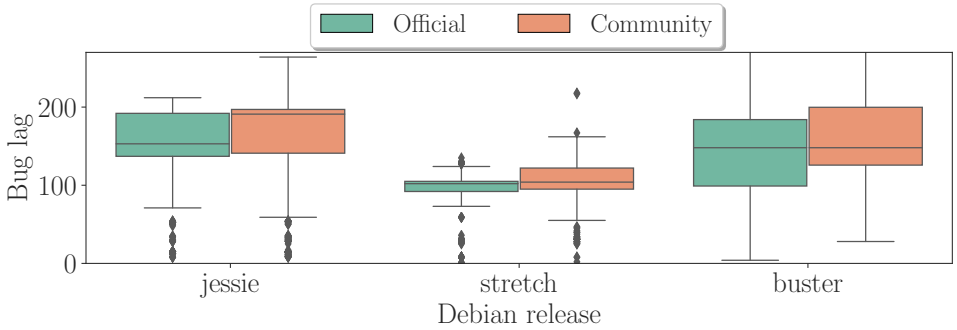

Figure 6.11: $\mathbf{contlag}^{\mathbf{Debian}}_{\mathbf{stability}}$ induced by installed package releases in Docker containers

Findings:

- Debian Stretch containers have a lower stability lag than other containers.
- To have the most stable package releases, a small proportion of package releases require release changes. Most of these changes are release downgrades.

RQ_6: How long do bugs and security vulnerabilities remain unfixed?

Since nearly half of all vulnerabilities are still *open* and 65% of all bugs are still *pending*, this question investigates how long it takes for a bug to get fixed. To do so, for all bugs, we compute the time interval between the bug report creation date and the last modification date of the bug, considering that this corresponds to the bug fix date, in case a fix was observed.

We rely on the statistical technique of survival analysis based on the non-parametric *Kaplan-Meier* statistic estimator commonly used to estimate survival functions [91]. It has been widely used in software engineering research [92, 93, 94]. Figure 6.12 shows survival curves per severity level for the event "bug is fixed" w.r.t. the bug report creation date. We observe that the time to fix a bug does not always depend on its severity level. *High* severity bugs are fixed faster than other kind of bugs. For example, it takes 53.8 and 33.5 months so that 50% of all *normal* and *minor* bugs get fixed, respectively, while it only

takes 3 months to fix *high* severity bugs. It seems like Debian maintainers prefer to start with easy bugs that are trivial to fix rather than normal ones. Nonetheless, bugs that may have an impact on releasing the package with the *Stable* release of Debian (i.e., *high severity*[15]) have the highest priority.

To find out if there are statistically significant differences between the survival curves per severity, we carried out log-rank tests for each severity pair. The differences were statistically confirmed ($p < 0.01$ after Bonferroni correction) except for the pairs *(normal, important)* and *(minor, wishlist)* where the null hypothesis could not be rejected.

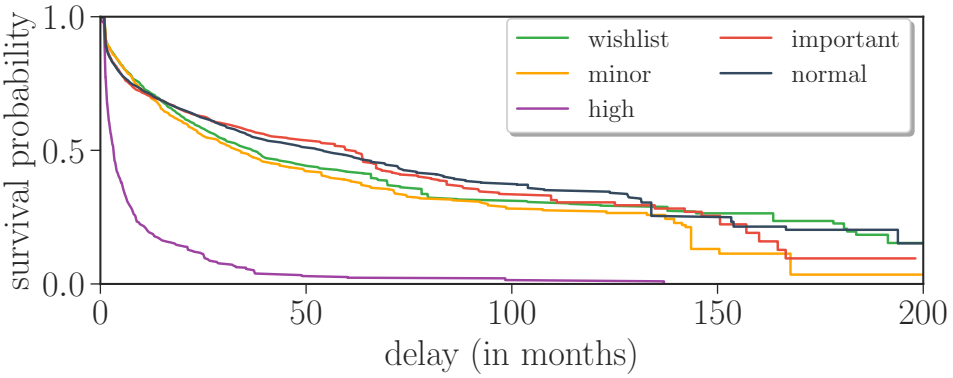

Figure 6.12: Survival probability for event "bug is fixed" w.r.t. the bug report creation date.

Findings:

- Half of the normal and minor bugs require more than 53.8 and 33.5 months respectively to be fixed.
- High severity bugs are fixed faster than other kind of bugs.

Similar to the bug survival analysis, we analyzed the survival of security vulnerabilities over time. Using the Debian security tracker we extracted the *debianbug id* for each vulnerability. With this *id*, we searched in the UDD for the creation and last modification date of the corresponding bug. We only found 62% of all vulnerabilities with a corresponding *debianbug id*. This proportion of vulnerabilities is responsible for 93% of all container vulnerabilities. For this subset we carried out a survival analysis for the event "security vulnerability is fixed" w.r.t. the bug report creation date. Figure 6.13 shows the *Kaplan-Meier* survival curves for each severity level found on the security tracker (as opposed to the severity of the bug reported in the UDD).

We observe that vulnerabilities are fixed faster than bugs. It takes 5.9 months

[15] `https://www.debian.org/Bugs/Developer.en.html#severities`

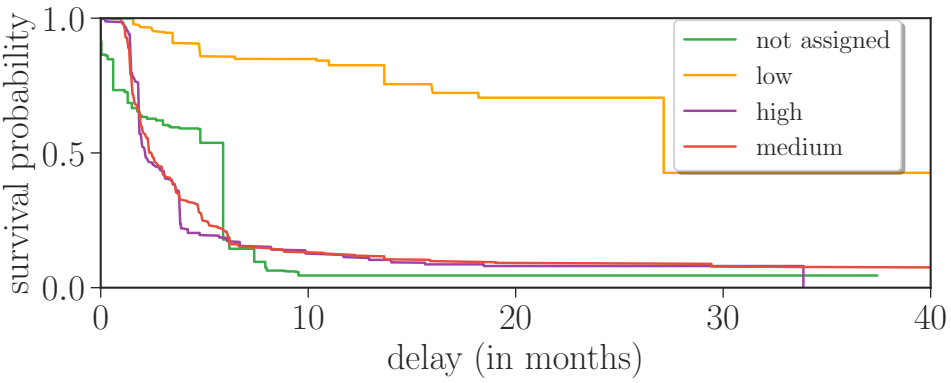

Figure 6.13: Survival probability per severity level for event "security vulnerability is fixed" w.r.t. the date of the bug arrival.

for *not assigned* severity vulnerabilities, 2.4 months for *medium* severity vulnerabilities, and 2.1 months for *high* severity vulnerabilities. *Low* severity vulnerabilities take much more time to fix: 27 months to fix 50% of them. We could not include the *unimportant* vulnerabilities since only 0.5% of them are fixed. We carried out log-rank tests to compare whether there are statistically significant differences between the survival curves depending on the severity of the bug. We could reject the null hypothesis assuming the similarity between the survival analysis curves with statistical significance ($p < 0.01$ after Bonferroni correction), except for *high* and *medium* severity vulnerabilities.

The fact that only 0.5% of the *unimportant* vulnerabilities are fixed could be expected since those vulnerabilities do not affect the binary package, but only materials and files that are not built (e.g., doc/foo/examples/[16]). To investigate this further, we identified the bug severity of the *low* severity vulnerabilities inside the Debian bug tracker (i.e., from the UDD), and found that 77.5% of these vulnerabilities have an *important* or *normal* bug severity. This correlates with the previous findings in RQ_4 and explains the results for the *low* vulnerabilities. However, we also observed that 51% of the *high* severity vulnerabilities are labeled as *important* bugs and 45% of them are considered as *high* (i.e., *serious, grave* or *critical*) bugs. This means that an upstream high vulnerable package can have a different priority downstream, depending on the downstream maintainers assessment (i.e., how a package is compiled, how it is integrated into the distribution, etc.).

Findings:

- *High* and *medium* severity vulnerabilities are fixed faster than *low* severity vulnerabilities.
- Vulnerability reports *upstream* might have different severity *downstream*.

[16]https://security-team.debian.org/security_tracker.html

6.2.4 Actionable Results

In RQ_1, we found that, in general, Debian packages used in *stable* releases are old and up-to-date (i.e., having the latest available fix). This implies that it should be easy for developers to keep up with the Debian updating process and reduce the version lag, since package maintainers are not releasing often. Moreover, new package releases in the *Stable* and *Oldstable* Debian releases are only about security patches, so from a semantic point of view, there is little to fear of breaking changes. For the *Testing* release, however, things are different: deployers should be aware of how the Debian project works.

Actionable result: Container deployers should be aware that the optimal update frequency of their base images and installed packages depends on the base Debian version.

In RQ_2, we found that the number of vulnerabilities is related to the number of outdated package releases per Debian-based container. This shows that containers could benefit from better updating procedures, allowing them to avoid security issues coming from their installed packages. Moreover, we found that the number of vulnerabilities is related to the Debian release.

Actionable result: Deployers who prefer stability to new functionalities are recommended to use the Stable and Oldstable versions that include only the most important corrections or security updates. To have a lower number of severe vulnerabilities, container deployers using the *Oldstable* Debian release should upgrade to the *Stable* release.

We found that all Docker containers are vulnerable and contain package releases with a high number of vulnerabilities and bugs. Since we did not discover a high version lag in containers, we do not think it is the responsibility of Docker deployers to avoid all vulnerabilities. Even containers with up-to-date package releases still may have a high number of vulnerabilities.

Lesson learned: No release is devoid of vulnerabilities, so deployers cannot avoid them even if they deploy the most recent packages.

We could not find a significant relation between the number of outdated package releases and the number of bugs. However, we observed that the number of bugs is related primarily to the Debian release. This means that deployers that care about bugs and new functionality and not about stability, should definitely upgrade to the Debian *Testing* release, since the updates in *Stable* and *Oldstable* releases are primarily about security bugs (i.e., severity vulnerabilities).

Actionable result: Container deployers concerned with having as few non-security bugs as possible should upgrade to the *Testing* release, at the expense of having a lower package stability.

When we analyzed the technical lag while considering different variants for the ideal software version, we found that the actions required to reduce the technical lag are related

to the chosen ideal version. We observed that in some cases, and in order to have the ideal version (i.e., most secure, most stable), the software package releases in use would need to be downgraded. However, in general, most of the package releases do not require any changes. This implies that the latest available version is perhaps not the most secure or stable version but it seems to be a good mix between stability, security and features.

Based on a survival analysis, we concluded that security vulnerabilities take less time to fix than other kind of bugs. The relation between vulnerabilities, bugs and outdated package releases shows that container deployers should give a high priority to updating when checking their container packages.

Actionable result: High security bugs are first priority for Debian maintainers; they are fixed faster than other kind of bugs. Container deployers should be aware of the newly available versions of their installed package releases and keep technical lag to the minimum to avoid this type of bugs.

Comparing our results about vulnerabilities to previous observations [55], we found Debian-based Docker containers to have an average number of vulnerabilities (i.e., 460) that is above the average for all Docker containers (i.e., 120). However, the number of vulnerabilities depends on the number of installed packages found. For example, it is not fair to compare vulnerabilities between Debian containers and Alpine[17] containers, unless we compare their size as well (in terms of number of installed packages).

6.2.5 Limitations

Our study was focused on Docker containers that make use of the Testing, Stable or Oldstable versions of Debian. The results of our analysis can therefore not be generalized to other base images in Docker. The analysis itself, however, can be easily replicated on other base images.

We chose to use the technical lag as a measurement. We only compared the used package release with the latest available version of the package within the same Debian release and without considering co-installability issues. Our results may differ when comparing with the latest available package release from the latest (e.g., *Stable*, *Testing* or *Unstable*) Debian releases, or when comparing with the latest installable package version that has no conflicts with other installed package versions.

It is not trivial to identify which package releases are affected by bugs or severity vulnerabilities. For example, the way in which we computed vulnerabilities and bugs was different. For vulnerabilities we relied only on the fixed version, since this is the way it is done in companies such as *CoreOS* or *Anchore.io*. For bugs, we relied on two sources of

[17] Alpine is a minimal image based on the security-oriented, lightweight Alpine Linux distribution with a complete package index of only 8 Mb.

information: the bug report creation date and its last modification date. Counting bugs in the same way as vulnerabilities would result in more bugs than the ones considered in this analysis.

Also, when searching for vulnerabilities in the Debian security tracker, the *debianbug id* was not found for 38% of the vulnerability reports. This may have influenced our survival analysis results. However, the missing proportion of vulnerability reports is responsible for only 7% of all analyzed container vulnerabilities.

6.3 npm Packages in Docker Containers

In addition of the operating system packages, containers might have other types of packages installed on them, for example *npm* packages. Such packages can be outdated and vulnerable as well [95, 53, 58]. In this section, we analyze official Docker Hub images that are based on Node, in order to assess the state of the npm packages installed on them.

6.3.1 Method and Data Extraction

This section describes the steps followed to obtain the dataset used in our study: (1) identifying candidate Docker images, (2) extracting npm package data, (3) collecting security vulnerabilities, and (4) computing technical lag. For step (1), we relied on the official Node image[18], which contains *Node.js*. We focused on Node images that are based on the Debian or Alpine base images, since the use of the corresponding Linux distributions is widespread in Docker[19]. In step (2), we pulled and ran the candidate images locally and identified the installed npm package versions. In steps (3) and (4), based on the npm packages found, we computed their technical lag and identify the security vulnerabilities that affect their versions.

6.3.1.1 Identifying Candidate Images

Our empirical analysis relies on the official Node image as the base to retrieve the Docker images that include npm package dependencies. As a consequence of Docker's layering mechanism (as explained in Section 6.2.1), if a Docker image is based on the Node image, then all the layers of this Node image will be included in it.

We extracted all available images (i.e., latest and tagged ones) from the 124 official repositories through the Docker API. We remotely inspected them using skopeo. We found that 961 out of 8,891 unique official images make use of Node and are based on the Debian or Alpine operating system. All these images are coming from only three repositories: *node*,

[18]https://hub.docker.com/_/node/

[19]https://www.ctl.io/developers/blog/post/docker-hub-top-10/

ghost[20] and *mongo-express*[21]. Table 6.8 shows the number of images considered for this analysis per repository and operating system.

Table 6.8: Number of analyzed images grouped by repository and operating system.

OS	**Debian**			**Alpine**	
Repository	node	ghost	mongo-express	node	ghost
Images	785	40	17	84	35

6.3.1.2 Extracting npm Package Data

When installing a *JavaScript* package from npm, a *package.json*[22] file is created locally on the machine, containing information such as the package name, version, description, etc. To inspect which packages are installed in each image, we pulled and ran the image locally, and then located all *package.json* files in it. Next, we identified package names and versions found in the files. To determine if a package release is outdated, we relied on the dataset extracted from *libraries.io* on March 13th 2018 [96]. This dataset contains metadata from the manifest of each package, based on the list of packages provided by the official registry of npm.

6.3.1.3 Collecting Security Vulnerabilities

To identify security vulnerabilities that affect installed npm package releases, we manually gathered from *Snyk.io* all 1,099 security reports that were published before May 3rd 2018[23]. Each security report contains vulnerability information about the affected package, the range of affected releases, its severity, its type (i.e., how vulnerable it is), the date of its disclosure, and the date when it was published in the database. For each security report, we identified the name and list of packages releases affected using the dataset of libraries.io [96].

6.3.2 Extending the Technical Lag Framework to npm packages used in Docker containers

In Chapter 5, we used the technical lag framework to compute the dependency version and time lag for npm package releases. Now, we extend the framework to support vulnerabilities of npm package releases installed in Docker containers.

Definition 17. Package releases

[20]Ghost is a free and open source blogging platform written in JavaScript
[21]Mongo-express is Web-based MongoDB interface, written with express
[22]https://docs.npmjs.com/files/package.json
[23]https://snyk.io/vuln?type=npm

Let $\mathcal{P} \subset \mathcal{N} \times \mathcal{V} \times \mathcal{T}$ be the set of all package releases available in npm, where $\mathcal{N}$ is the set of all possible package names, $\mathcal{V} \subset \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ the set of all possible version numbers, and $\mathcal{T}$ the set of all possible time points. Each package release $p = (p_{name}, p_{version}, p_{time}) \in \mathcal{P}$ has an associated package name $p_{name} \in \mathcal{N}$, a version number $p_{version} \in \mathcal{V}$ and a release date $p_{time} \in \mathcal{T}$. We assume a total order on $\mathcal{V}$ and $\mathcal{T}$.

Definition 18. Package release security

security : $\mathcal{P} \times \mathcal{T} \rightarrow \mathbb{N}$ such that **security**(p, t) corresponds to the number of reported vulnerabilities that p has at time t.

In some cases, a package release can have vulnerabilities with different severities. The number of vulnerabilities of a package release p can be computed using different weights for different severities. In our case, we assign the same weight to each vulnerability.

Definition 19. Package releases installed in a container

In practice, software containers might include a set of installed third-party package releases. In the case of Docker containers that are based on Node, they include npm package releases.

Let $\mathcal{C}$ be the set of all possible Node-based software containers, where each container $c \in \mathcal{C}$ has an associated build date $c_{time} \in \mathcal{T}$. We define:

installed : $\mathcal{C} \rightarrow \mathbb{P}(\mathcal{P})$ such that **installed**(c) corresponds to all npm package releases installed in the container c.

available : $\mathcal{C} \rightarrow \mathbb{P}(\mathcal{P})$ such that **available**$(c) = \{p \in \mathcal{P} \mid \exists p' \in$ **installed**(c), $p_{name} = p'_{name} \wedge p_{time} \leq c_{time}\}$ corresponds to all available releases of the installed npm package versions **installed**(c) at the container's build date c_{time}.

The goal of our empirical analysis is to study how outdated npm package releases in Docker containers are with respect to the latest available and the most secure package releases. For this reason, we consider three instantiations of the technical lag framework: time, version and security-based.

Definition 20. Time-based instantiation of the technical lag framework.

We define $\mathcal{F}^{npm}_{time}$ as the function that instantiates the technical lag framework $\mathcal{F}$ for a given container $c \in \mathcal{C}$:

$$\mathcal{F}^{npm}_{time}(c) = (\mathcal{A}, \mathbb{N}, \mathbf{ideal}_{time}, \mathbf{delta}_{time}, \mathbf{agg}_{time})$$

where:

- $\mathcal{A} =$ **available**(c), is the set of all available package releases of the container c.
- $\mathbb{N}$ is the set of possible time lag values. Intuitively, each element corresponds to a

number of *days*.

- $\mathbf{ideal}_{time}(p) = \max_{p'_{version}} \{p' \in \mathcal{P}_t \mid p'_{name} = p_{name}\}$. Intuitively, for a given package release p, the function $\mathbf{ideal}^{npm}$ returns the *highest available version* of a package release with the same name as p.
- $\mathbf{delta}_{time}(p, q) = \max(0, q_{time} - p_{time})$ computes the positive difference in number of days between the release dates of two package releases p and q.
- $\mathbf{agg}_{time}(L) = \sum_{x \in L} x, \text{ with } L \subseteq \mathbb{N}$ computes the sum over a set of time lags.

Definition 21. Version-based instantiation of the technical lag framework.

We define $\mathcal{F}^{npm}_{version}$ as the function that instantiates the technical lag framework $\mathcal{F}$ for a given container $c \in \mathcal{C}$:

$$\mathcal{F}^{npm}_{version}(c) = (\mathcal{A}, \mathcal{V}, \mathbf{ideal}_{version}, \mathbf{delta}_{version}, \mathbf{agg}_{version})$$

where:

- $\mathcal{A} = \mathbf{available}(c)$, is the set of all available package releases of the container c.
- $\mathcal{V}$ is the set of possible version lag values. Intuitively, each element corresponds to a version number.
- $\mathbf{ideal}_{version}$ is defined in the same way as for $\mathcal{F}^{npm}_{time}$
- $\mathbf{delta}_{version}(p, q)$ is defined as $(major_{lag}, minor_{lag}, patch_{lag})$ where
  $coll = \{r \in \mathcal{P}_t \mid p_{version} \leq r_{version} \leq q_{version} \wedge p_{name} = r_{name} = q_{name}\}$
  $major_{lag} = |\{major \mid r_{version} = (major, minor, patch), \forall r \in coll\}| - 1$
  $minor_{lag} = |\{(major, minor) \mid r_{version} = (major, minor, patch), \forall r \in coll\}| - major_{lag} - 1$
  $patch_{lag} = |\{(major, minor, patch) \mid r_{version} = (major, minor, patch), \forall r \in coll\}| - major_{lag} - minor_{lag} - 1$
  In this definition, *coll* is the set of all releases between p and q. We then find the number of releases that increase the major number, then the number of releases that increase the minor number while having the same major number, and finally the number of patch releases that increase the patch number while having the same major and minor numbers. From that, we derive the number of distinct major, minor and patch releases that would be needed to upgrade from r_1 to r_2.
- $\mathbf{agg}_{version}(L) = \sum_{x \in L} x, \text{ with } L \subseteq \mathcal{V}$ computes the sum over a set of versions.

Definition 22. Security-based instantiation of the technical lag framework.

We define $\mathcal{F}^{npm}_{security}$ as the function that instantiates the technical lag framework $\mathcal{F}$ for a given container $c \in \mathcal{C}$:

$$\mathcal{F}^{npm}_{security}(c) = \Big(\mathcal{A}, \mathbb{N}, \mathbf{ideal}_{security}, \mathbf{delta}_{security}, \mathbf{agg}_{security}\Big)$$

where:

- $\mathcal{A} = \mathbf{available}(c)$, is the set of all available package releases of the container c.
- $\mathbb{N}$ is the set of possible security lag values. Intuitively, each element corresponds to a number of reported vulnerabilities.
- $\mathbf{ideal}_{security}(p) = \max_{p'_{version}}\{p' \in \mathcal{A} \mid p_{name} = p'_{name} \wedge \forall r \in \mathcal{A}, r_{name} = p'_{name} \implies \mathbf{security}(p', c_{time}) \leq \mathbf{security}(r, c_{time})\}$

  Intuitively, for a given package release p, the function $\mathbf{ideal}_{security}$ returns the highest version with the *lowest number of vulnerabilities*.
- $\mathbf{delta}_{security}(p, q) = \mathbf{security}(p, c_{time}) - \mathbf{security}(q, c_{time})$ computes the difference between the number of vulnerabilities of package releases p and q.
- $\mathbf{agg}_{security}(L) = \sum_{x \in L} x,$ with $L \subseteq \mathbb{N}$, computes the sum over a set of lags.

Now, we can directly compute $\mathbf{techlag}_{\mathcal{F}^{npm}_{\alpha}(c)}(p)$ and $\mathbf{agglag}_{\mathcal{F}^{Debian}_{\alpha}(c)}(P)$ (by instantiating definitions 2 and 3 in Chapter 4) in terms of the above definitions, where $\alpha \in \{\text{time}, \text{version}, \text{security}\}$.

Notation 3. In the remainder of this section we use the shortcut notations
$\mathbf{techlag}^{npm}_{\alpha}(p, c)$ for $\mathbf{techlag}_{\mathcal{F}^{npm}_{\alpha}(c)}(p)$ and $\mathbf{agglag}^{npm}_{\alpha}(P, c)$
for $\mathbf{agglag}_{\mathcal{F}^{npm}_{\alpha}(c)}(P)$, where $\alpha \in \{\text{time}, \text{version}, \text{security}\}$.

Based on the above definitions, we can define the technical lag of a deployment of a Node-based container.

Definition 23. Technical lag of a Node-based container deployment

Let $c \in \mathcal{C}$ be a Node-based software container. For $\alpha \in \{\text{time, version, security}\}$, we define $\mathbf{contlag}^{npm}_{\alpha}(c) = \mathbf{agglag}^{npm}_{\alpha}(\mathbf{installed}(c))$.

In other words, $\mathbf{contlag}^{npm}_{\alpha}(c)$ corresponds to the aggregated lag of all npm package releases installed in a container c.

6.3.3 Empirical Evaluation

Research Questions

In this section, we analyze third-party *JavaScript* packages that are installed in Docker containers. We report the results of the analysis about the technical lag in terms of updates and security vulnerabilities for 961 official Docker Hub images based on the Node image. The research questions that we address in this study are:

RQ_1: *How outdated are npm packages used in Docker images?* We analyze the technical lag induced by npm packages within official Docker images by comparing these images at the date of their last update and at the date of data extraction.

RQ_2: *How vulnerable are npm packages in Docker images?* We use a dataset of npm vulnerability reports to identify which and how many vulnerable packages are present in images.

RQ_3: *How vulnerable are npm packages in Docker containers compared to their most secure versions?* With this research question, we compute the technical lag in terms of security between the containers now and the most secure containers that would be possible to create using the same set of used npm packages.

RQ_1: How outdated are npm packages used in official Docker images?

To answer the first research question we analyze the technical lag of npm packages in Docker images based on (1) the date of the last available image update and (2) the time when we performed the analysis (March 13th 2018). This will provide insights about the state of npm packages while their images were still being maintained, and their state at a later time, when they possibly accumulated more technical lag.

However, before answering our research questions, we investigated on the distribution of the number of installed npm packages per container. We found that *node* containers had a lower number of installed packages than the others, while *ghost* containers have the highest number of installed packages. This should be expected, since *ghost* is offering the service provided by the npm package *ghost*[24] and *mongo-express* is providing services of two packages *mongodb*[25] and *express*[26], and both of these images are built on top of *node*. Considering both operating systems, we found that the median number of installed packages is 200, 419 and 959 for *node*, *mongo-express* and *ghost*, respectively.

[24]https://www.npmjs.com/package/ghost
[25]https://www.npmjs.com/package/mongodb
[26]https://www.npmjs.com/package/express

a) Technical lag at the date of the last update

Packages that are up-to-date have no technical lag. Therefore, we first started by exploring how many packages are outdated. We found that the majority of the used npm packages are up to date. The median proportion of up-to-date packages is 64% for Debian images and 57% for Alpine images. These proportions are still fairly risky and show a high potential of problems due to outdated package releases.

Figure 6.14 shows packages $\textbf{techlag}^{npm}_{time}$ distribution of all considered Docker images, grouped by year and operating system (Alpine and Debian). We found a statistically significant difference ($p < 0.001$) when comparing the distributions of Alpine and Debian for each year using the Mann-Whitney U test. However, the difference between the time lag of outdated packages was small, according to the effect size ($|d| = 0.16$) computed with Cliff's delta, a non-parametric measure quantifying the difference between two groups of observations. We also notice that the time lag is increasing over time, which could be a consequence of old images that have been re-uploaded to Docker Hub without having updated their contained npm packages.

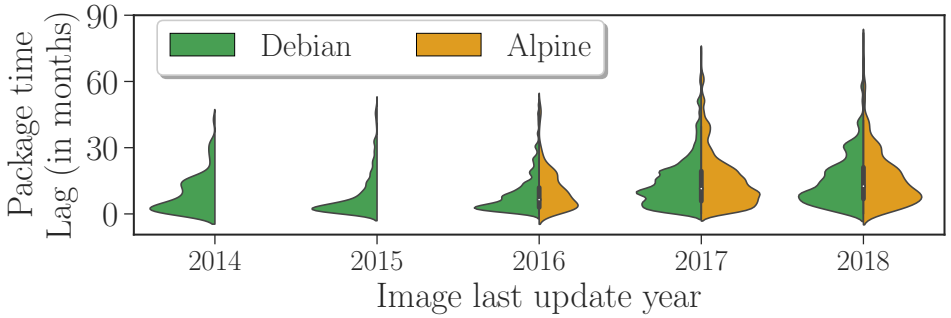

Figure 6.14: Violin plots showing yearly distribution of $\textbf{techlag}^{npm}_{time}$ (measured at the date of the images last update) for all outdated packages in Docker container, grouped by operating system.

Figure 6.15 shows packages $\textbf{techlag}^{npm}_{version}$ distribution of all considered Docker images, grouped by the image's last update year. The version lag seems to remain quite stable over time. When considering all packages in the images, we found that the median version lag is (0 major, 0 minor, 1 patch). Thus, at the time of the images last update, outdated npm packages were mainly missing patch updates, which is expected since patch updates are released frequently [69].

b) Technical lag at the date of the analysis

We computed the technical lag of used npm packages on March 13$^{\text{th}}$ 2018. Compared to the proportions at the date of the last update, we found that up-to-date packages

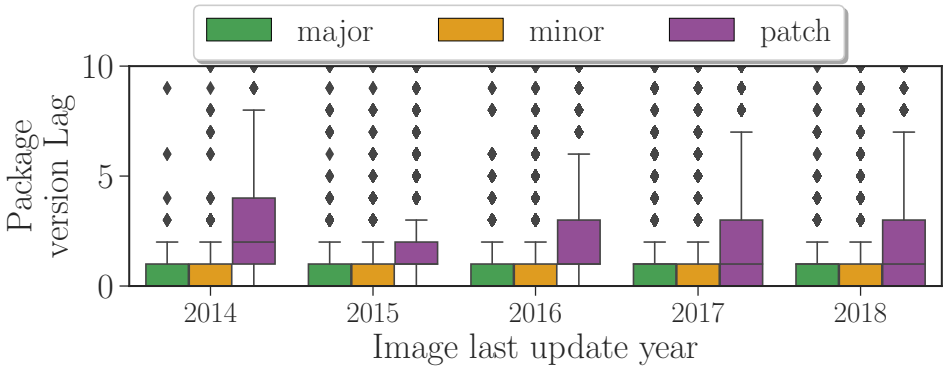

Figure 6.15: Box plots showing yearly distribution of $\textbf{techlag}^{npm}_{version}$ (measured at the date of images last update) for all outdated packages in Docker images.

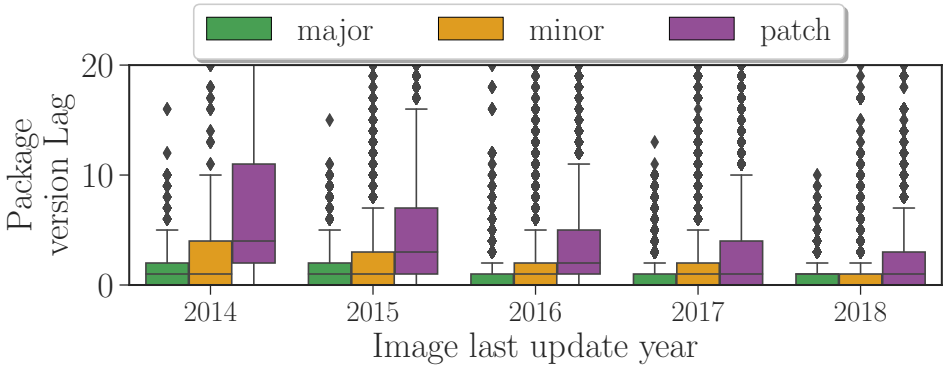

Figure 6.16: Box plots of yearly distribution of $\textbf{techlag}^{npm}_{version}$ (calculated at the date of March 13th 2018) for all outdated packages in images.

decreased to 41% and 34% for Debian and Alpine images, respectively. Figure 6.16 shows the distribution of $\textbf{techlag}^{npm}_{version}$ at the date of the analysis. The lag appears to decrease over time with different rates for patch (clearly visible), minor or major updates. Packages of images last updated in 2014 have a median version lag of (1 major, 1 minor, 4 patch). Images last updated in 2018 have a lower median version lag of (0 major, 1 minor, 1 patch). This is expected since newly updated images will have more recent package releases.

Findings: Docker deployers that use old Node images might be missing updates, including one major update.

RQ_2: How vulnerable are npm packages in official Docker images?

A security vulnerability is a fault that could be exploited to breach the system. They are the most important bugs and require more experienced developers to fix them [97]. With

Table 6.9: The top 5 vulnerability types found for npm packages in Docker containers.

Vulnerability type	**% affected containers**	**# unique affected packages**
ReDoS	100	29
Uninitialized Memory Exposure	100	7
Prototype Pollution	100	4
Access Restriction Bypass	100	1
Prototype Override Protection Bypass	79	1

this research question, we evaluate to which extent Docker images have vulnerabilities caused by npm packages.

After identifying which of the installed npm packages are affected by the vulnerabilities reported on Snyk.io, we found that from 1,099 known vulnerabilities, only 74 are affecting the npm packages in Docker images. These vulnerabilities are affecting 3.7% (i.e., 52) of all (i.e., 1,412) unique installed packages. However, all images are affected by these vulnerable packages. We found that most of the vulnerabilities (54%) were disclosed between 2017 and 2018. 40% of the vulnerability reports have medium severity, 35% have high severity and 25% have low severity.

We identified the type of affecting vulnerabilities so that maintainers can assess the severity of the vulnerabilities depending on their proper use of the containers. We found 27 types of vulnerabilities in total, affecting 52 unique packages, of which the *ReDoS* (Regular Expression Denial of Service) vulnerability type was responsible for affecting 55.7% of the packages. Table 6.9 shows the frequency for the top 5 vulnerability types affecting npm packages in Docker images.

There is a mean of 16.6 vulnerabilities per container and a median value of 10. Figure 6.17 shows a scatterplot of the last update dates of images against the number of vulnerabilities found in them. It shows that recently updated images have less vulnerabilities than older ones. However, for *low* severity vulnerabilities the evolution is different. We computed Pearson's R and Spearman's ρ correlation between the number of vulnerabilities and the date of the last update. We found that the number of *low* severity vulnerabilities has a weak positive correlation with the last update of an image ($R = 0.16$, $\rho = 0.25$). *medium* severity vulnerabilities have strong negative correlation ($R = -0.74$, $\rho = -0.8$) while *high* severity vulnerabilities have a weak to moderate negative correlation ($R = -0.28$, $\rho = -0.58$). Ignoring the unimportant (i.e., low severity) vulnerabilities, we conclude from this that the number of vulnerabilities per image is higher for older images than for more recent ones.

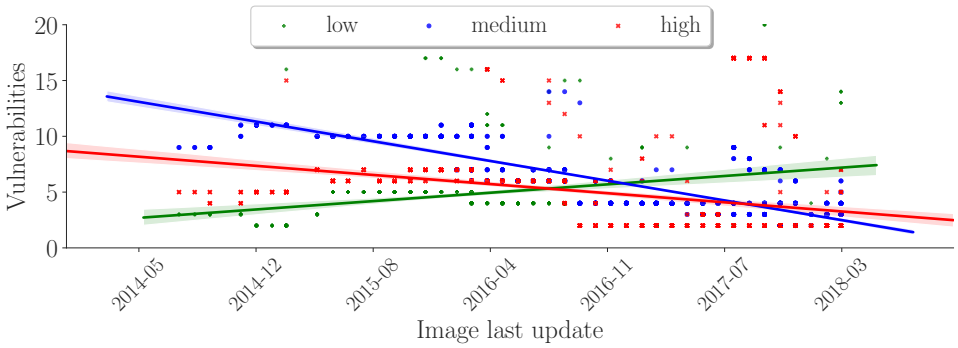

Figure 6.17: Number of vulnerabilities w.r.t to images last update date.

Findings: All official Node-based images have vulnerable npm packages, with an average of 16 security vulnerabilities per image. Older images are more likely to have more vulnerabilities.

RQ_3: How vulnerable are npm packages in Docker images compared to their most secure versions?

After identifying all vulnerable packages installed in Docker containers, we computed the difference in number of vulnerabilities between the used and the most secure package versions (i.e., $\textbf{techlag}^{npm}_{security}$). Then, we investigated on how much vulnerabilities Docker containers would avoid by changing their packages to the most secure package versions (i.e., $\textbf{contlag}^{\textbf{npm}}_{\textbf{security}}$).

To answer this research question, we started by computing the number of vulnerabilities for all npm package versions, without distinguishing between their types. Then, we chose the most secure and recent version of all npm packages found installed in Docker images. Figure 6.18 shows the distribution of the proportion of packages that require a change to have the most secure npm package versions in a Docker container. The figure shows that official Docker images only have a small proportion of packages being used with the most secure version. In fact, we found that only 9.3% of all package versions used in containers are from the most secure npm package versions (i.e., they do not require any changes). The rest needs to be updated to have the most secure versions (i.e., 50.2% need to be upgraded and 40.5% need to be downgraded). This is surprising and shows that many older package versions are not affected by the reported vulnerabilities.

Now, we suppose that Docker deployers concerned by npm packages security would follow our instructions to have the most secure package versions. We compute the security lag $\textbf{contlag}^{\textbf{npm}}_{\textbf{security}}$ that they would be reducing from their containers. We find that

the number of vulnerabilities would reduce from a mean of 16.7 vulnerabilities to 0.6 vulnerabilities per container. The median of $\mathbf{contlag}^{\mathbf{npm}}_{\mathbf{security}}$ that would be reduced is 10 vulnerabilities.

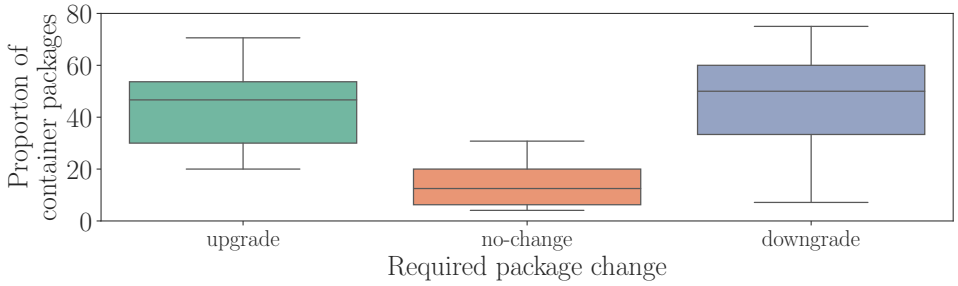

Figure 6.18: Proportion of packages that require changes in order have the most secure npm package versions in a Docker images

Findings: Recent package releases are not necessarily more secure than older package releases. To reduce security technical lag, two out of five installed npm packages need to be downgraded.

6.3.4 Limitations

In this analysis, we only focused on *official* Docker images, which are expected to be more secure and well maintained since they are the base images of other official and community images. Thus, the number of vulnerabilities found in the analyzed images is an underestimation of the real number of vulnerabilities that might be found in community images that depend on them (without updating) and where more dependencies, activity and development are expected. Hence, the results cannot be generalized to community images.

In a similar vein, we only analyzed Node-based images. Other official images that are not based on the Node image might have npm packages installed on them. Our findings cannot be generalized to them. Finally, our results cannot be generalized to packages in other languages (or package managers). However, the methodology itself can be applied easily to such types of packages.

Our findings might be biased by the limited (and low) number of security reports that were available on Snyk.io for npm packages. Our results are therefore an underestimation of the actual number of npm package vulnerabilities in Docker images, as it is very likely that many vulnerabilities are unknown or not reported. However, even with a small number of vulnerability reports we could find a relation between outdated images and their number of npm vulnerabilities.

Also, because of the lack of vulnerability reports data, we only analyzed the presence of vulnerabilities in Docker images at the date of the analysis. To show if packages were vulnerable when images were last updated, a time-related analysis similar to the one considered in 6.3.3-A could be performed.

6.4 Discussion

If containers would always depend on the most recent available version of their used packages, they would benefit from the latest functionality, security updates and bug fixes. However, maintainers might be more focused on other software characteristics such as package stability, or they just choose not to upgrade certain packages because of the considerable effort that may be involved in doing so. For this reason, we studied the presence of technical lag in Docker containers, and related it to the presence of bugs and severity vulnerabilities.

As highlighted before, it is important to verify not only vulnerabilities, but also other bugs. Indeed, bugs make the system behave in unexpected ways, resulting in faults, wrong functionality or reduced performance. Researchers already found that performance bugs are similar to security bugs, in that they require more experienced developers to fix them [97]. Hence, it is essential to include bug analysis tools into existing automated scan and security management services such as *Anchore.io* or *Quay.io*.

Moreover, most research and tools around Docker focus on system packages. However, we found that even on the basis of the small number of vulnerability reports that we analysed for npm packages, potential security vulnerabilities are frequently present in the packages used in Docker images. Thus, a detailed analysis of such packages should definitely be considered in the future to advance in the state of the art and to provide more complete services.

The *Anchore.io* survey [89] showed that container deployers care more about package vulnerabilities than having packages up-to-date. However, we found that less outdated containers have less vulnerabilities, for both cases (i.e., npm and Debian). Thus, we believe that including the technical lag as a measure of how outdated packages are, can empower automated scan and security management tools to give better insights about the security of Docker containers.

Recommendation: Docker scan and security management tools should improve their platforms by adding data about other kind of bugs and include the measurement of technical lag to offer deployers information of when to update, for both third-party and system packages used in containers.

6.5 Conclusion

This chapter presented two empirical analysis of the state of system and third-party packages in public Docker containers. We studied how outdated container packages are and how this relates to the presence of bugs and severity vulnerabilities. For both analysis, our approach aimed at proposing and using an automated method for tracking and identifying package bugs and vulnerabilities from trusted data sources.

In the first analysis, we studied 7,380 popular unique images that are based on Linux-based Debian distribution while considering both official and community images. We observed that most container packages have the latest fix available in Debian, even for old packages (e.g., *Stable*). However, we found that all containers have vulnerable and buggy packages. Studying outdated packages in more detail, we found that their number is correlated with the number of vulnerabilities found in a container.

We observed that in Debian, taking care of security vulnerabilities is more important than taking care of bugs, in the sense that vulnerabilities are fixed faster than other kinds of bugs. This results a high number of open bugs for the *Stable* and *Oldstable* releases. Therefore, even up-to-date installed package versions could be affected by these open bugs.

These findings indicate that container deployers whose major concerns are stability and security need to rely on better updating procedures. In contrast, container deployers that care more about functionality and bugs should rely on the newest Debian releases.

In the second analysis, we focused on the use of *JavaScript* packages in official Docker images based on the Node image for *Debian* and *Alpine* Linux distributions. 961 unique images were retrieved and analyzed against 1,099 security reports extracted from *Snyk.io*, a well-known registry of vulnerabilities for *npm JavaScript* packages. We also studied the impact of the presence of third-party packages on the technical lag and security of Docker images.

For both analysis, the findings reveal the presence of outdated packages in Docker images and the risk of potential security vulnerabilities. We found that the technical lag and the number of vulnerabilities are related to the last update date of images, which suggests that Docker users should keep up with the updating process of their base images and installed packages.

Chapter 7

ConPan: A Tool to Analyze Health of Software Packages in Docker Containers

---

Software containers may include buggy and vulnerable packages that put at risk the environments in which the containers have been deployed. Existing quality and security monitoring tools provide only limited support to analyze Docker containers, thus forcing practitioners to perform additional manual work or develop ad-hoc scripts when the analysis goes beyond security purposes. This limitation also affects researchers desiring to empirically study the evolution dynamics of Docker containers and their contained packages.

In this chapter, we present ConPan, an automated tool to inspect the characteristics of packages in Docker containers, such as their outdatedness (technical lag) and other possible flaws (e.g., bugs and security vulnerabilities). ConPan comes with a command-line interface and API, and the analysis results can be presented to the user in a variety of formats. The content of this chapter is mainly based on a publication in the MSR 2019 conference proceedings [98].

7.1 Introduction

In Chapter 6, we have shown that despite being largely adopted, Linux-based Docker images often contain buggy or vulnerable releases of packages which may have been fixed in more recent package releases, thus exposing the production environment where they have been deployed to potential risks [55, 99].

Docker users can rely on a limited number of tools to scan and monitor their images, primarily for security vulnerabilities. For instance, *Anchore.com* [100] inspects container security using CVE-based security vulnerability reports, while *Dagda* [101] relies on the OWASP [102], Red Hat Oval [103] and the Offensive Security exploit database [104]. Finally, *Snyk* [105], an Open Source security platform, does not only scan and monitor, but also suggest fixes for detected vulnerabilities. Docker Hub provides its own tool, *Docker Trusted Registry* which scans each layer and checks the results against a periodically updated vulnerability database. If practitioners wish to evaluate other aspects than security vulnerabilities, such as evaluating the *outdatedness* (e.g., in terms of version or time lag) or quality issues of a container (e.g., in terms of the bugs [106] of its system and third-party packages), they are forced to develop ad-hoc scripts which may be time-consuming and error-prone. This limitation also affects empirical research in software container mining and analysis [107, 108, 109], requiring scholars to re-invent the wheel because of the need to retrieve and analyze the data by themselves.

In this chapter, we present ConPan, a tool that we developed to simplify the analysis of Docker Hub images and their installed packages. ConPan gathers and processes information about security vulnerabilities, bugs and technical lag of installed packages, leveraging on different publicly available databases. ConPan is an easy-to-use open source tool that: (i) allows to track and retrieve data about packages in containers from multiple sources in an easy and consistent way; (ii) has been designed to be extensible to cover new data sources; and (iii) provides the possibility to output its results to external analysis and/or visualization tools thanks to its flexible output formats (e.g., pandas dataframes [110]).

7.2 Overview of ConPan

ConPan aims to support both practitioners and researchers desiring to analyze Docker containers. Its goal is to collect and fetch data about software packages that are installed in Docker containers, leaving the tasks of storing and analysing the data to other tools. It can be used either as a command-line tool or as a *Python* library. Release 1.0.0 of ConPan supports the analysis of Debian packages included in Docker images based on the the corresponding Linux distribution.

The overall structure of ConPan is summarized in Figure 7.1. Its core is composed of five tasks: (i) pulling and running Docker images; (ii) identifying the installed packages; (iii)

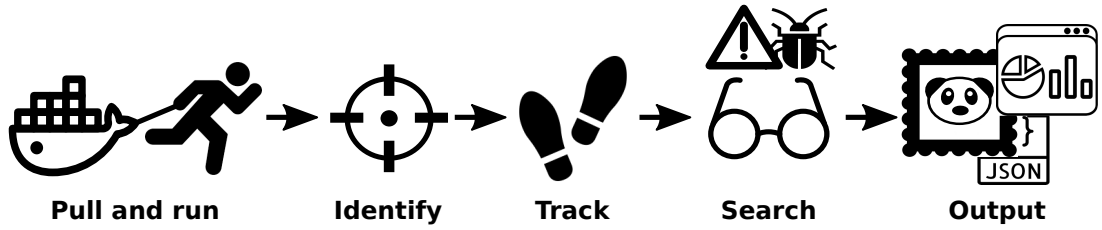

Figure 7.1: Overview of ConPan. The user interacts with the tool via either the command line or through its API. Once the tool is initialized, the target Docker image is pulled, the contained packages are extracted and traced back to the corresponding package managers, and vulnerabilities and other bugs are identified and returned as output (as pandas dataframes, JSON documents and/or charts).

tracing them back to their package managers; (iv) searching for their known vulnerability reports or other reported bugs and quality issues; (v) reporting the results in a specific output format. ConPan also provides general information about the analyzed Docker Hub image, fetched from the Docker Hub registry using its API [1].

To be able to trace back installed packages to their package managers, ConPan relies on datasets containing historical information about where and when packages were released. For example, the dataset corresponding to Debian packages is obtained via the Debian archive [111], which contains daily snapshots of all Debian packages from the official and security Debian Snapshot repositories. ConPan can easily include other datasets of popular package repositories (e.g., npm[2] and PyPI[3]) by relying on freely available services. ConPan downloads the latest Debian dataset and, using regular Debian tools (`dpkg -l`) in the container, identifies the installed packages and compares them to the latest releases to assess how outdated they are.

To search for known package vulnerabilities and bugs, ConPan relies on the method used in Chapter 6, using the Debian security tracker [112] and the Ultimate Debian Database [90].

7.3 ConPan in Action

This section describes how to install and use ConPan, highlighting its main features.

[1] https://docs.docker.com/registry/spec/api/
[2] https://www.npmjs.com/
[3] https://pypistats.org/api/

7.3.1 Installation

ConPan has been developed and tested mainly on GNU/Linux platforms. It is very likely to work out of the box on any Linux-like (or Unix-like) platform, provided that the right version of *Python* is available (i.e., *Python* 3.5). ConPan can be cloned via GitHub and then installed using *Python*'s pip package manager. Listing 7.1 shows how to install ConPan from its source code.

There are other required dependencies that are outside PyPI like Docker and the package *apt_pkg*, all of them can be found on `github.com/neglectos/ConPan`, together with further installation instructions.

Listing 7.1: How to download and install ConPan

```
# Installation from source code using pip
$ pip3 install git+https://github.com/neglectos/ConPan
# To uninstall
$ pip3 uninstall conpan
```

7.3.2 Use

Once installed, ConPan can be used through a command-line interface (CLI) or through the API of a *Python* library. We showcase these two types of executions below.

7.3.2.1 CLI

Using ConPan as a CLI does not require much effort. Listing 7.2 shows how easy it is to call ConPan on a Docker Hub image. Three parameters are required: i) the type of packages (Debian packages in this case); ii) the Docker image to be analyzed; and iii) the path to the historical package dataset extracted from Debian archive. The latter dataset is provided with the tool. In other cases where ConPan relies on online APIs such as the npm registry, the data path is not needed.

Listing 7.2: How to use the ConPan CLI

```
# Call ConPan from command line
$ conpan -p debian  -c <Docker image>:<tag> -d path/to/data
```

The output of the execution of Listing 7.2 would be the general information about the analyzed Docker image, plus the number of installed packages, vulnerabilities and bugs. The output also includes three figures showing the proportions of the outdated packages, plus the proportion of vulnerabilities and bugs grouped by their severity.
A concrete example of the output of ConPan will be shown in subsection 7.3.3.

7.3.2.2 API

The API of ConPan can be accessed from within any *Python* script with minimal effort, assuming the user knows how to program in *Python*. Listing 7.3 shows how to use the ConPan API. The ConPan module is imported at the beginning of the file, then the package kind, Docker image and path to the historical data (if needed) parameters are set and used in order to call ConPan. The generated output consists of one JSON file and four pandas dataframes, which are summarized below.

- *general info*: a JSON file containing general information about the analyzed Docker image, such as size, architecture, number of pulls, among others.
- *installed_packages*: a dataframe containing the set of all installed packages.
- *tracked_packages*: a dataframe containing the set of installed packages that are coming from the package manager and were not installed from external sources.
- *vulnerabilities*: a dataframe containing the set of all vulnerabilities with their severity, status, corresponding packages, etc.
- *bugs*: a dataframe containing the set of all bugs with their severity, status, corresponding packages, etc.

Listing 7.3: How to use ConPan as a python library

```
#! /usr/bin/env python3
from conpan.conpan import ConPan
# Parameters
kind = 'debian'
image =  <Docker image name>
dir_data = 'path/to/data/'
# Call ConPan
cp = ConPan(packages=kind, image=image, dir_data=dir_data)

# Results
(general_info, installed_packages, tracked_packages,
vulnerabilities, bugs)  = cp.analyze()
```

7.3.3 Reporting

The output generated by ConPan can be exploited directly using *pandas*, one of the most commonly used *Python* libraries for data analytics and dataframes [110]. The reported data can be visualized by means of *matplotlib* [113] or *seaborn* [114] libraries, or by converting the dataframes to JSON files and storing them to a persistent NoSQL document-based

storage, such as an *ElasticSearch* database [115], or as raw data in a CSV format. *Jupyter notebooks* [116] can be used for data analysis and early prototyping of data visualization in a transparent way, since they allow to store both the results and the code needed to reproduce them.

The ConPan library can be very useful for researchers desiring to analyze container packages. The tool can be used to extract data about a large number of Docker containers and to create datasets to be used in empirical research. The dataset contains different information related to software packages installed in containers, thus providing a powerful basis to perform empirical studies. We have relied on ConPan ourselves in Chapter 6 to empirically analyze installed system packages in a large dataset of Docker Hub images that are based on the Debian Linux distribution.

For deployers desiring to monitor their Docker containers, they can use the tool's CLI or integrate its functionalities in the automation of their Docker image builds. For instance, Listing 7.4 shows the output of ConPan executed from the CLI on the community Docker Hub image *google/mysql*[4], which is an image of MySQL server for Google Compute Engine. The listing includes general information such as the description, the number of pulls and stars, and the last time the image was updated. The output shows that the image was not updated sine November 2015.

Listing 7.4: General information about the Docker image *google/mysql*

```
Results:
General information about the Docker image:  google/mysql
- description: MySQL server for Google  Compute Engine
- star_count: 18
- pull_count: 46692
- full_size: 96687899
- last_updated: 2015-11-13T01:19:18.235940

Results about installed packages in:  google/mysql
# installed packages: 144
# tracked packages: 81
# vulnerabilities: 240
# bugs: 474
```

Figure 7.2 shows the pie charts generated by ConPan to highlight the percentage of outdated packages, vulnerabilities and bugs. We observe that the image has a high proportion of outdated packages (93.8%), and a high number of vulnerabilities (240) and bugs (474). We also observe from the breakdown in severity that most vulnerabilities are medium and high.

[4]https://hub.docker.com/r/google/mysql

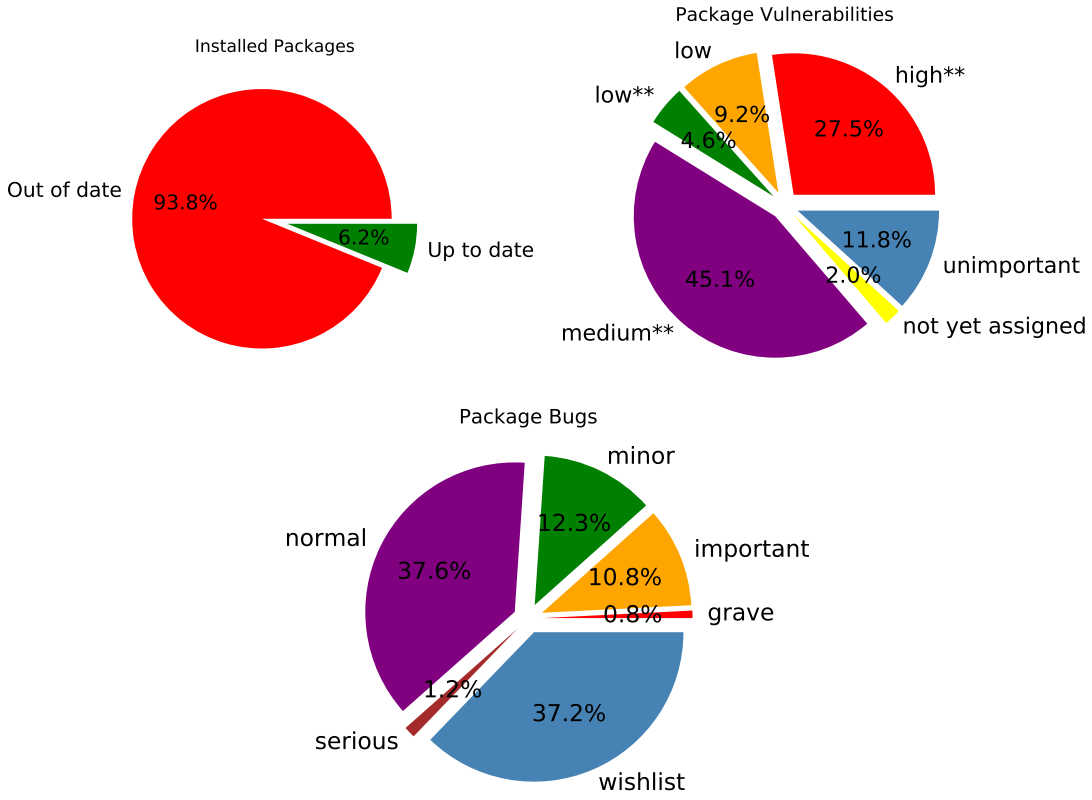

Figure 7.2: Statistics about Debian packages in the community Docker Hub image **google/mysql**

7.4 Summary

Deploying applications and services using containerization technologies is becoming a popular practice in software engineering, thanks also to the increasing popularity of Docker containers. They offer isolation, portability and reusability by providing all needed artifacts and dependencies shipped in one package. However, as shown in previous chapters, Docker containers may contain vulnerable and outdated packages that may put at risk the environments where the containers are deployed. Nevertheless, little support is given to practitioners and researchers desiring to assess the status of their containers, thus forcing them to resort to the tedious task of writing error-prone ad-hoc scripts.

For this reason, we developed ConPan, a tool that simplifies the monitoring and analysis of software packages installed in Docker containers, by reporting how outdated, vulnerable and buggy they are. ConPan can be used stand-alone as a CLI or integrated through its *Python* API with other processes (e.g., empirical studies or automation of Docker image builds). The output of ConPan has been conceived to cover a wide spectrum of technologies such as *Python* libraries for data processing and visualization (e.g., *pandas,*

matplotlib), NoSQL document-based databases, JSON and CSV formats.

Besides the software packages of the host operating system, a container may include third-party packages that are needed at runtime or for the development of other included applications. Two of the most popular third-party packages are *JavaScript* and *Python* packages that are hosted in the npm and PyPI package repositories, respectively. We aim to extend ConPan to support this kind of packages. This should be easy to achieve. In fact, we have already started to support npm packages by relying on basic npm commands[5]. After extracting the list of all npm installed package versions we can assess how outdated they are with respect to their latest available releases by relying on the npm registry API[6].

[5]https://docs.npmjs.com/cli/ls.html
[6]http://registry.npmjs.org/

Chapter 8

Conclusion and Outlook

This dissertation started by presenting the research context and goals of the thesis. The main goals were to empirically analyze how software components are reused by other components of open source software ecosystems, and to provide support to developers by creating a technical lag framework that can assess the health of an ecosystem's software components. In this chapter we recall the contributions of this thesis and discuss the limitations of our research. Finally, we present future research opportunities opened by the contributions of this thesis.

8.1 Contributions

In Chapter 3, we performed an exploratory analysis on the use of testing-related Maven libraries in Java projects. We found that testing-related libraries have been used in Java software projects since the first commit in their GitHub repository. This shows the importance of external libraries. We also observed that software projects tend to stick to the use of a specific library, since only 5% of the projects performed a migration to another competing library. As a consequence updating the library remains the only possibility for software projects to benefit from new testing-related functionalities. If software projects do not update the libraries they rely on, they might suffer from technical lag.

In Chapter 4, we gave examples where technical lag is present and we investigated the usefulness of technical lag as the difference between the versions of a collection of deployed software components and the most available ideal versions of the same collection of components. We carried out interviews and surveys with software practitioners to ask them about the technical lag concept. We found that all interviewees were favorable towards the idea. To be able to compute and analyse technical lag, we defined a generic formal framework of technical lag for component-based software repositories.

In Chapter 5, we operationalised the framework of technical lag to the npm case

study. We instantiated it based on version lag and time lag. We computed such lag for the whole registry of npm package releases and their dependencies and for GitHub applications that make use of npm packages. We found an increase of technical lag over time. One of the causes of technical lag appears to be the use of too strict package dependency constraints, disallowing packages and applications to benefit from more recent releases of their dependencies. We also observed that the technical lag induced by transitive dependencies can be very high and it is related to the dependency tree depth of package releases. Finally, we observed some changes in the technical lag over time due to policy changes made in the npm package manager.

In Chapter 6, we analyzed technical lag for system and third-party packages included in Docker images, by computing and instantiating it based on version, time, security vulnerabilities and other bugs. We computed technical lag using different variants of the ideal package version (i.e., most secure, least buggy, latest) and we found that technical lag is present in Docker containers in all cases. We observed that technical lag is weakly related to the chosen ideal version. In fact, we only found a small proportion of package versions that needed to be downgraded to obtain the ideal version. In most of the cases, the latest version is the most secure and the least buggy version. We provided many findings and actionable results to help Docker deployers in deploying the most preferred package version.

To be able to reproduce our analysis in Chapters 5 and 6, we provided replication packages in Appendix-C. Finally, in Chapter 7, we presented a tool that can support Docker deployers in assessing the health of their containers by analyzing how outdated, buggy and vulnerable their included packages are.

8.2 Threats to Validity

Each chapter presented the limitations related to the work and analysis performed in it. However, one of the major limitations of this thesis is the potential lack of generalization of the results. For example, the empirical studies presented in chapters 3, 5 and 6 need to be reproduced on other component-based software ecosystems to assess whether the findings can be generalized. A future topic of research would be to compare the results and findings across different component-based software ecosystems.

The proposed formal framework for measuring technical lag aimed to be as generic as possible, hence it should be applicable to any type of software repository as well as to other ways of measuring technical lag. The specific instantiations of this framework to the npm, Debian and Docker case studies are, however, only partially generalizable, since different repositories may use different notations and ways for expressing and interpreting version numbers and version constraints, and may provide different ways to support the semantic

versioning specification. As a result, the findings we have obtained for each empirical study cannot be generalized to other software ecosystems, since they inevitably depend on repository-specific factors such as the policies, tools, practices and values adopted by the community (cf. Bogart et al. [9]).

The concrete way in which we instantiated the technical lag framework in our case studies, and the way we have aggregated technical lag, has a direct effect on our observed findings. We do not consider this as a *real* threat to validity, since the purpose of the technical lag framework is to define and explore different useful ways of measuring technical lag, each of which may provide different results that should be interpreted differently.

8.3 Future Work

Topics of research:

Our results open the door for more research on technical lag and its measurement in reusable software libraries of other kinds, ranging from Linux-based deployments to embedded systems built with reusable components. We also expect that a better understanding of technical lag can be used to improve how dependency constraints are defined and managed over time, so that the overall technical lag for a whole repository can be reduced. Of course, this needs the collaboration and training of a large fraction of component developers, but given the high potential benefits we believe this is feasible once i) the theoretical model has been more thoroughly evaluated in other real case studies and ii) suitable tools are provided.

As mentioned in Chapter 3, technical lag can be seen as an extension of the concept of technical debt. The metaphor of "technical debt" was introduced in 1992 [73]. It tries to capture the problems caused by not writing the best possible code, requiring code improvements later on. The difference between code "as it should be" and code "as it is" is a kind of debt for the developing team. If technical debt increases, code becomes more difficult to maintain. A similar concept is "design debt", which extends the concept to the design of software components [117]. A possible direction could be to study this relation between technical debt and technical lag as the difference between "how dependencies are" specified, and how they "should be" specified in the ideal situation.

According to Conway's law, software mimics the organizational-social structure around it [118]. Understanding and taking into consideration this socio-technical congruence can be beneficial to improve software engineering practices. A possible topic of research that goes into this direction would be to translate technical lag to address social aspects, through the formal definition of some notion of "social lag". A similar analogy has already been undertaken by translating the idea of "technical debt" to "social debt" [119].

From a research point of view, our technical lag framework can be used to explore different ways of measuring technical lag, taking into account security vulnerabilities, closed issues, pull requests, etc. More precise lag measurements could rely on change logs as more detailed information about what changed in a software component. Another possibility would be to involve dynamic analysis of the actual source code to uncover the presence of breaking changes and security vulnerabilities. This can be very challenging for a dynamically typed and interpreted language such as JavaScript, but solutions are being proposed [38]. Having obtained more detailed information about the causes of technical lag, it should become possible to provide estimation models of the effort required to reduce it.

It is also useful to compare the extent of technical lag across different software component repositories and different package managers, in order to assess which policies, practices, culture and tools lead to the best compromise. Inspired by the work of Lauinger et al. [58] on the client-side use of *JavaScript* libraries, we would also like to include other types of external applications to our analysis, like deployed websites.

In a recent work, Valiev et al. [5] have conducted a mixed-methods study on the PyPI ecosystem in order to identify the factors affecting the sustainability of open-source *Python* projects. They found that the number of connections and the relative position in the dependency network of PyPI are significant factors affecting the chances of a project becoming dormant; the practice of producing backwards compatible releases does not appear to influence project dormancy, etc. Inspired by this study, we aim to investigate more on the factors behind software developers updating practices.

When studying how outdated Docker images are, we did not differentiate between specific container characteristics such as their size, service, targeted audience, or provided functionality. Moreover, we did not differentiate between package release types (e.g., patch, minor or major) when calculating technical lag. In future work we would like to consider other measures of technical lag while considering container and package characteristics and all available releases in a project. For instance, in many cases vulnerability fixes are first done in the *Testing* or *Unstable* releases before entering the *Stable* and *Oldstable* releases Debian distribution. Besides the operating system packages, containers have other types of packages installed on them, for instance, *PyPI* and *npm* packages. Such packages can be vulnerable as well [95, 53, 58]. In this thesis, we have carried out an empirical study for npm packages. In future studies, we aim to include other types of packages. We also plan to carry out a comparison with other operating systems and other base images.

Tooling:

Regardless of the preferred choice, it is important to offer proper measurement and recommendation tools to developers that indicate when, where and why technical lag

increases, and whether it introduces breaking changes in the code. Such support should be part of more extensive monitoring tools (preferably part of a continuous integration process) that also take into account the presence of bugs and security vulnerabilities, unmaintained packages, known incompatibility issues, transitive dependencies, technical debt, among others. Such a tool could additionally provide suggestions on how to reduce technical lag by making certain dependency constraints more permissive. Tools could offer as well support in the opposite direction to make maintainers of reusable libraries aware of the technical lag in the software that depends on them. For this reason, we plan to include technical lag in open source software data analytic tools such as GrimoireLab [120].

For the npm repository in particular, an example of a useful tool for finding and selecting reusable packages is npms.io [1]. It allows developers to search for packages with similar functionality, and select the most appropriate one among those, by relying on useful characteristics such as the package quality, maintenance or popularity. The concept of technical lag could be added easily.

An assessment of technical lag at the level of the entire package repository is also useful, in order to understand how the structure of the repository as well as the practices of the used package manager influence technical lag. Indeed, we observed that the policies adopted by a package manager can push toward more or less technical lag. For example, the decision taken by npm to change the default use of tilde by the use of caret resulted in a growth of technical lag because of the increased use of a more strict constraint. A technical lag measurement tool could therefore be helpful to carry out "what if" scenarios, in order to assess upfront how particular changes in the package manager policy may affect current and future technical lag in the package repository.

In chapter 7, we presented a tool ConPan that analyzes Debian system packages installed in Docker containers. However, a container may include other third-party packages that are needed at runtime or for the development of other included applications. Two of the most popular third-party packages are *JavaScript* and *Python* packages that are hosted in the npm and PyPI package repositories, respectively. For this reason, it is important to extend ConPan to support this kind of packages. We have designed the ConPan architecture to be easily extended and include this feature.

We have already started the extension of ConPan to npm *JavaScript* packages. In future work and for third-party *Python* packages, we will collect all names and versions of the installed *Python* packages in the target Docker container. Then, we will use the PyPI registry API to see how outdated they are by comparing them with their latest available releases. To search for any known vulnerabilities of the installed packages, for both npm and PyPI packages, we will rely on available open source vulnerability databases like npm Security advisories[2].

[1] See `https://npms.io` and `https://docs.npmjs.com/getting-started/searching-for-packages`
[2] https://www.npmjs.com/advisories

We also plan to improve the visualization part of the ConPan CLI by providing a variety of charts to show the distribution of the number of vulnerabilities, bugs and missed updates for each installed package. Finally, we aim to evaluate the tool by asking developers to what extent ConPan meets their expectations and to what extent it helps them to achieve their goals.

8.4 Closing Summary

Open source software component repositories are constantly evolving and increasing in size. Not updating to the ideal available release of component dependencies may negatively affect software development by not benefiting from new functionality, vulnerability and bug fixes available in other versions. On the other hand, automatically updating to other releases may introduce incompatibility issues.

The main goal of this thesis has been to develop a new formal framework that can be instantiated to support software practitioners in assessing technical lag as an aspect of the health of the software components they depend on. To validate the framework, we carried out empirical studies on different software ecosystems and we provided several actionable results in order to reduce technical lag when deploying or developing software systems. More specifically, we empirically analyzed technical lag for the whole registry of npm packages and their direct and indirect dependencies, and for system and third-party packages installed in Docker containers hosted on Docker Hub. We also developed a tool to assess how outdated, buggy and vulnerable packages included in Docker containers are.

The findings and developed tools aim to guide and help open source software developers and deployers to keep their software in a healthy shape. The thesis highlights the risk of having bugs and vulnerabilities and missing new functionalities through outdated dependencies. We expect that the proposed technical lag framework can be made even more useful by building specific tools that developers and people deploying software and maintaining those deployments can include in their continuous integration systems. Our observed findings can be turned into actionable guidelines, and with time, be made applicable to software package and container managers.

Appendix A

Interviews with Software Practitioners

In order to assess the usefulness of the technical lag concept, we carried out interviews with software practitioners during the gathering of open source software developers in *FOSDEM 2019*[1]. Each interview was grouped into 4 parts:

- **Profile:** To gather demographic information about the interviewee such as his or her background and experience.
- **Software Characteristics:** To collect information about the software projects that the interviewee is involved in.
- **Updating Process:** To obtain information about the current updating process and policy that is followed in the software project in which the interviewee is involved.
- **Technical lag:** After explaining the concept of technical lag to the interviewee, we ask him/her about its expected usefulness.

This appendix includes all answers to all questions that have been asked during the interviews to all participants.

Profile:

Question: What is your role in the software project(s) you are involved in?
P1: I am a development coach. I am not involved in coding but I am working as a mentor to developers.
P2: I am the only developer.
P3: I am the leader of the team.

[1] https://fosdem.org/2019/

P4: I am a developer.
P5: My role is a developer advocate and I come from an operations background. For my company, I am helping people to use and operate our technology.

Question: Which professional experience do you have?
P1: As a developer I have 5 years experience, and as a coach I have 9 months. I also did some mentoring and computer science and machine learning training.
P2: I have 5 years of experience in programming, however, as a professional experience I have only 9 months, that's because I have just graduated my masters.
P3: I started working in computer engineering in 2003, that's 15 years now. I also worked in a research group for some years.
P4: I have 3 to 4 years of development, plus 4 years of research.
P5: More than 20 years.

Software Characteristics:

Question: Which kind of project is your software project? e.g, library, application, open source, industrial, etc.
P1: I am working on a pretty big project. The functionalities that the tool provides are mainly services. The tool is not open source.
P2: I am working on a tool that is a developer activity tracker. It is an internal tool that is not open sourced yet.
P3: The project is open source and it is mainly a library.
P4: The tool is more like an open source platform, it can be used as an application or as a library.
P5: It is an open source application.

Question: To which domain is your deployment dedicated?
P1: We mainly do infrastructure work.
P2: The tool is for my team and it is purely for management. Our team members contribute to open software projects and my tool tracks their contributions. So it is dedicated for managers, team leaders and also developers.
P3: The tool is for IT, data extraction, data analysis, etc.
P4: The tool is for IT practitioners, for data processing, data management.
P5: It is used by software IT companies. It is for deploying containers.

Question: What is the size of your software project?
P1: We are five to twelve people working full time on the project.
P2: I do not think it is a big project. In terms of lines of codes, we have 2000 lines of code.
P3: The project is pretty big, we have separate modules that can be used individually, but they can be used as one project as well. In general, I think we have about 6000 lines

of code.
P4: It is medium size, we are 4 people working on it (only development), but one or two people can hold all the knowledge.
P5: The project is pretty big and it is a popular one in the world.

Question: Which technology do you use for the development/deployment of the software project?
P1: We use Python with many other technologies, e.g. Docker, Ansible, etc
P2: The tool is developed in Python and for the front-end I use some of JavaScript, mostly Angular.
P3: We use Python for the programming language, we use other technologies like JavaScript, Flask, Docker, ElasticSearch and many others.
P4: We use Python, MySQL, JavaScript, and frameworks like Django, Angular, etc.
P5: For the main application, we use Go, and for another software that we use to deploy our software, we use a mix of Go and Ruby.

Question: How critical is the dependency management in your software?
P1: We use many libraries. So I think that the dependency management is pretty critical in our case.
P2: Some of the dependencies are not that important, but some of them are contributing to the main functionalities of the tool.
P3: It is very important, for example Kibana and ElasticSearch are always changing and our tool is based on them, so we have to keep up with their updating. Sometime it is hard for us to update from a major version to an other one, because new major versions have breaking changes.
P4: It depends on the type of dependencies. Our tool is built on top of another tool, so we try to keep an eye on its new releases. However, we also have dependencies that are not core ones, so we do not update these dependencies.
P5: The dependency management is the hottest spot in our project.

Updating Process:

Question: When and how do you update the software you depend on?
P1: It is a trade-off, sometimes we do the updating manually, sometimes we do it automatically. Like, if you do it manually, it may take some time to do it, but if you do it automatically, you may have a new update that has breaking changes. Also, doing it automatic means that you will always be in the bleeding edge, which means if a break happens you will need to deal with it every six months maybe, the effort will be distributed and will not be noticed by the business. While if you do it manually, maybe you will need to say that you need a six months of dependency refactoring. Anyway, it is always a trade-off.

P2: Actually, I did not update them in a while. But that is because I did not face any issues yet.
P3: Our dependencies are pinned to one specific version. In the case where there is a known vulnerability, we update the dependencies manually because we have to test them. For the main technologies that we use, we try to keep up with their updating process. We also have a case where we use an outdated but stable version of a dependency. The dependency is not so important for us, it only has one task. That's why we do not care about updating it.
P4: We always do the updating manually. When we know that there is a new release of one of the core dependencies that we use, we schedule a date of when we can do the migration/updating. Still, sometimes we can't keep up with the updating pace of the tools that we depend on. We are not a big team to be able to keep an eye and also keep up with the updating process of our dependencies.
P5: The tool that manages our dependencies looks for new updates and asks us if we update, so just one click and then everything is updated automatically. So, I can say it is automatic with a bit of human interaction. For some dependencies, even if we get a notification of a new available dependency release, we do not update.

Question: Why do you update?
P1: For security reasons, performance, novelty, just not to have all tools which are not used by anyone and not maintained anymore. In fact, I think that we should use a dependency that have many people working on it, and that has people enjoying working on it.
P2: I would update if there is some important vulnerability fix or some very good new feature.
P3: We mainly update for features.
P4: We do it for features.
P5: Mostly for security purposes, just to make sure that there are no open vulnerabilities.

Question: Do you keep tracking if your software dependency is outdated?
P1: I keep tracking the software for security updates. I sometime track features but not as important as security updates.
P2: No, I don't really do it.
P3: We are in contact with the developers of the core dependencies, so we receive notifications from time to time about the new releases.
P4: We are always having an eye on the tools that we use. But we don't have some automatic tracking.
P5: Yes, we have our tool that tracks the dependencies we use.

Question: Is there any guideline that tells you, when and how you have to update?
P1: I think it is quite psychological. I think the good thing for me is to see benefits of it.

P2: No there is no such guideline.
P3: No we don't, but yes, we always have that feeling, because the developers of the tools that we depend on are always adding many things and we are always asking these questions of should we update? and what are we going to gain if we update? and how much time we will need for that? We also have many transitive dependencies that need to be updated also if we update other tools, so yes we are always looking for metrics that tell us why we should update.
P4: No, we don't actually have any measurement that tells us how outdated we are.
P5: I don't think that we have any automated guideline for that. Even for our costumers we do not ask them to update our tool, because they are just two to three versions behind, so that's fine. So I can say that our guideline is that "update when you can". Obviously, if someone is lagging behind with one year then "good luck".

Question: Do you contact or consult with other people before updating?
P1: When you do something, you should always talk about it.
P2: I am the only developer.
P3: Yes, we contact each other to ask about each others opinions.
P4: Yes, nobody does the updating alone.
P5: I just do it. Because I have my own development environment.

Question: How much time does the updating take? Person time and real time (Human effort vs Computer effort)?
P1: I remember downgrading a dependency for availability purposes and because the version that we downgraded to was a good version. It took about one to two hours.
P2: Until now it is not important, it is just a matter of running the pip manager.
P3: Sometimes it takes just some minutes, and in some cases it really takes weeks because of the refactoring work behind.
P4: For the core tools, it takes so much time from starting to finishing the updating, sometimes it takes even months. The time needed is more about machine effort, to update we have to migrate some data and that's the part where it takes so much time.
P5: I would say, usually about 10 minutes of my time and a couple of hours for the services to download all requirements.

Question: Which are the most important characteristics of more recent versions for you to decide to which version to upgrade? Example: stability, absence of bugs, fixes of security vulnerabilities, backward compatibility, new functionality, performance, etc
P1: Security, that's for sure. Features, maybe performance but that's less relevant for me. If a new version has only new features but I will not need them, then I do not think I will update to it.
P2: I don't like to spend a lot of time in the front-end, so I think if there are some new features that will make my work easier, I would use it. Security is not an issue for me

since the tool is just internal, for now.
P3: Security is always important, especially when we work with critical and personal information. We have one case when we did a major update that took us a lot of time for security reasons. However, new features are important.
P4: The most important characteristics would be new functionalities. In some cases, the most stable one (absence of bugs) is also desirable.
P5: I think security is important. And also just getting access to the new features. For me the latest is more important than security, since it will have fixes for the open vulnerabilities. However, I think sometimes new vulnerabilities are emerged within the latest version.

Technical lag:

Before starting the questions of this part, we explained the technical lag concept and we gave examples similar to the ones presented in chapter 4.

Question: There are many different cost and benefit measurements that could be used when computing the technical lag, for example: new functionalities, resolved bugs, fixed vulnerabilities, lines of code, commits, version numbers, time, conflicts, etc. Which ones do you think are the most interesting to consider?
P1: I would consider the *changelog* between versions, what is new and what changed between the versions.
P2: I think intuitively, one should look for how many versions is the dependency missing, but I think it is a mix between all units, features, breaking changes and fixed vulnerabilities. Number of commits is also important because it shows how much effort people are putting into the project.
P3: Usually vulnerabilities, but of course bugs. Also, it depends on the tools, sometimes some bugs are not critical so you can live with them.
P4: In general, I think security is more important. So the number of vulnerabilities is more important.
P5: I think there are two important metrics, features and vulnerabilities. They can sum up everything, they are almost like a reverse *changelog*. But also I think what would be interesting is to know how many people had problems in order to have the ideal update. So it is kind of effort vs benefits.

Question: In the case of the software projects you are involved in: What do you think is the ideal dependency version that you would like to use?
P1: The ideal version that we would use is the most secure, however, we only use the bleeding edge version (i.e. the latest version), because it is easy.
P2: I think the one that is working and stable is pretty fine for me. I just want the

development to be as easy as possible for me.
P3: Well, since I am the leader, what I told you is what we do. However, in general I think it is a balance between the most secure and the latest versions.
P4: In our case, we prefer the latest versions (features).
P5: I think the latest major stable version would be good, but if you have appetite for security then do patches as well.

Question: What do you think of the concept of technical lag?
P1: I like it. I would consider cost. And I think it also motivates me to update or not. For example, knowing the most secure version and the features that I would miss if I use it is interesting. But also I would like to talk about the cost and benefits part, if you just go to the minor version, maybe you will get only minor stuff, but the updating will be easy. While if you go to a major version maybe you will have more stuff but the updating will require more cost, and more work.
P2: It is interesting. I think it would be more interesting to give different metrics when reporting the lag.
P3: I think it is good to know this kind of balance.
P4: I think it is helpful. If we know the ideal version that we can target, we can consider the concept of technical lag. Depending on the projects, it can be a motivation to update.
P5: I think it is great. It is definitely something we are missing. For the larger enterprise customers, the more we can say to them "this is why you should keep being up to date versus 10 years old version", the more is better.

Appendix B

Online Surveys with Software Practitioners

In order to understand the characteristics of the most desirable software versions we created a Google form and shared it on Facebook groups of software developers (e.g., DevOps Engineers Group, etc). This appendix includes the list of questions asked in the survey.

Survey Questions:

Q: Did you contribute to the development of any software project before?
Options: Yes or No.

Q: Years of experience.
Options: From 1 to 10 years

Q: Which programming languages did/do you use?
Options: multiple choices (Java, Python, JavaScript, PHP, Android, Other)

Q: What is your role in the software project?

Q: Are/Were you in charge of updating the used software libraries in your project?

Q: In your opinion, what is the most appropriate version of a library to use?
Options: multiple choices (latest available, most stable, has many contributors, least complex, most tested, most documented, most secure, has the minimum size)

Appendix C

Replication Packages

To be able to reproduce the analyses in Chapters 5 and 6, we have created replication packages. They contain Jupyter notebooks [116], scripts and data and they require Python 3.5+ to be installed, as well as all the dependencies listed in "requirements.txt".

All code and data required to reproduce the empirical analysis in Chapter 5 about npm package dependencies are available on `https://doi.org/10.5281/zenodo.1420075`

All code and data required to reproduce the empirical analysis in Chapter 6 about Debian packages used in Docker containers are available on `https://doi.org/10.5281/zenodo.2350504`.

List of Figures

List of Tables

Bibliography

[1] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, pages 1–36, February 2018.

[2] Charles W Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[3] Laszlo A. Belady and Meir M Lehman. A model of large program development. *IBM Systems journal*, 15(3):225–252, 1976.

[4] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. Metrics and laws of software evolution-the nineties view. In *Proceedings Fourth International Software Metrics Symposium*, pages 20–32. IEEE, 1997.

[5] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the pypi ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 644–655. ACM, 2018.

[6] Christopher Bogart, Anna Filippova, Christian Kästner, and James Herbsleb. Survey of ecosystem values. http://breakingapis.org/survey/. accessed: 28/10/2017.

[7] R. G. Kula, D. M. German, T. Ishio, and K. Inoue. Trusting a library: A study of the latency to adopt the latest Maven release. In *Int'l Conf. on Software Analysis, Evolution, and Reengineering*, pages 520–524, March 2015.

[8] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. Experience paper: a study on behavioral backward incompatibilities of java software libraries. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 215–225. ACM, 2017.

[9] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an API: Cost negotiation and community values in three software ecosystems. In *Int'l Symp. Foundations of Software Engineering*, pages 109–120. ACM, 2016.

[10] Forrest Shull, Janice Singer, and Dag IK Sjøberg. *Guide to advanced empirical software engineering*. Springer, 2007.

[11] Richard Stallman. *Free software, free society: Selected essays of Richard M. Stallman*. Lulu.com, 2002.

[12] Richard Stallman. Why "free software" is better than "open source". 1998. *Available in http://www.gnu.org/philosophy/free-software-for-freedom.html*, 2002.

[13] Ron Goldman and Richard P Gabriel. *Innovation happens elsewhere: Open source as business strategy*. Morgan Kaufmann, 2005.

[14] Katherine Stewart and Tony Ammeter. An exploratory study of factors influencing the level of vitality and popularity of open source projects. *ICIS 2002 Proceedings*, page 88, 2002.

[15] Konstantinos Manikas and Klaus Marius Hansen. Software ecosystems–a systematic literature review. *Journal of Systems and Software*, 86(5):1294–1306, 2013.

[16] Alexander Serebrenik and Tom Mens. Challenges in software ecosystems research. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, page 40. ACM, 2015.

[17] Mircea F Lungu. *Reverse engineering software ecosystems*. PhD thesis, Università della Svizzera italiana, 2009.

[18] Mircea Lungu, Romain Robbes, and Michele Lanza. Recovering inter-project dependencies in software ecosystems. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 309–312. ACM, 2010.

[19] Cyrille Artho, Kuniyasu Suzaki, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Why do software packages conflict? In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 141–150. IEEE Press, 2012.

[20] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[21] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser. Measuring dependency freshness in software systems. In *International Conference on Software Engineering*, pages 109–118, 2015.

[22] Ali Mili, Rym Mili, and Roland T Mittermeir. A survey of software reuse libraries. *Annals of Software Engineering*, 5(1):349–414, 1998.

[23] Neil A Maiden and Cornelius Ncube. Acquiring cots software selection requirements. *IEEE software*, 15(2):46–56, 1998.

[24] Shawn A Bohner. Extending software change impact analysis into cots components. In *27th annual NASA Software engineering workshop*, pages 175–182. IEEE, 2002.

[25] William B Frakes and Kyo Kang. Software reuse research: Status and future. *Transactions on Software Engineering*, 31(7):529–536, 2005.

[26] S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning versus breaking changes: A study of the Maven repository. In *International Conference on Source Code Analysis and Manipulation*, pages 215–224, September 2014.

[27] Christian Macho, Shane McIntosh, and Martin Pinzger. Automatically repairing dependency-related build breakage. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 106–117. IEEE, 2018.

[28] Alexandre Decan and Tom Mens. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, 2019.

[29] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. Mining trends of library usage. In *International Workshop on Principles of Software Evolution*, pages 57–62. ACM, 2009.

[30] Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. Mining library migration graphs. In *Working Conf. Reverse Engineering (WCRE)*, pages 289–298, 2012.

[31] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. A study of library migrations in Java. *Journal of Software: Evolution and Process*, 26(11):1030–1052, 2014.

[32] Amine Benelallam, Nicolas Harrand, César Soto Valero, Benoit Baudry, and Olivier Barais. The maven dependency graph: a temporal graph-based representation of maven central. *arXiv preprint arXiv:1901.05392*, 2019.

[33] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. The emergence of software diversity in maven central. *arXiv preprint arXiv:1903.05394*, 2019.

[34] Matúš Sulír and Jaroslav Porubän. A quantitative study of java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 17–25. ACM, 2016.

[35] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the JavaScript package ecosystem. In *Int'l Conf. Mining Software Repositories (MSR)*, pages 351–361. IEEE, 2016.

[36] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? An empirical case study on npm. In *International Symposium on Foundations of Software Engineering*, pages 385–395. ACM, 2017.

[37] Raula Gaikovina Kula, Ali Ouni, Daniel M German, and Katsuro Inoue. On the impact of micro-packages: An empirical study of the npm JavaScript ecosystem. *arXiv preprint arXiv:1709.04638*, 2017.

[38] Gianluca Mezzetti, Anders Moller, and Martin Toldam Torp. Type regression testing to detect breaking changes in node. js libraries. In *European Conference on Object-Oriented Programming (ECOOP)*, 2018.

[39] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 559–563. IEEE, 2018.

[40] Jesus M Gonzalez-Barahona, Gregorio Robles, Martin Michlmayr, Juan José Amor, and Daniel M German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.

[41] Pietro Abate, Roberto Di Cosmo, Jaap Boender, and Stefano Zacchiroli. Strong dependencies between software components. In *International Symposium on Empirical Software Engineering and Measurement*, pages 89–99. IEEE Computer Society, 2009.

[42] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240, 2012.

[43] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Learning from the future of component repositories. *Science of Computer Programming*, 90:93–115, 2014.

[44] Maëlick Claes, Tom Mens, Roberto Di Cosmo, and Jérôme Vouillon. A historical analysis of Debian package incompatibilities. In *Working Conf. Mining Software Repositories*, pages 212–223, 2015.

[45] James Turnbull. *The Docker Book: Containerization is the new virtualization*. 2014.

[46] Docker Inc. Docker - build, ship, and run any app, anywhere. https://www.docker.com/. accessed: 01/11/2018.

[47] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C Gall. An empirical analysis of the Docker container ecosystem on GitHub. In *14th Intl Conf on Mining Software Repositories*, pages 323–333. IEEE Press, 2017.

[48] Zhigang Lu, Jiwei Xu, Yuewen Wu, Tao Wang, and Tao Huang. An empirical case study on the temporary file smell in dockerfiles. *IEEE Access*, 2019.

[49] Jiahong Zhou, Wei Chen, Guoquan Wu, and Jun Wei. Semitagrec: A semi-supervised learning based tag recommendation approach for docker repositories. In *International Conference on Software and Systems Reuse*, pages 132–148. Springer, 2019.

[50] Alexandre Decan, Tom Mens, and Maëlick Claes. An empirical comparison of dependency issues in OSS packaging ecosystems. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 2–12. IEEE, 2017.

[51] Raula Gaikovina Kula, Coen De Roover, Daniel M German, Takashi Ishio, and Katsuro Inoue. A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 288–299. IEEE, 2018.

[52] Ruturaj K Vaidya, Lorenzo De Carli, Drew Davidson, and Vaibhav Rastogi. Security issues in language-based sofware ecosystems. *arXiv preprint arXiv:1903.02613*, 2019.

[53] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *International Conference on Mining Software Repositories*, 2018.

[54] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. Technical Report 1902.09217v1, arxiv, February 2019.

[55] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on Docker Hub. In *International Conference on Data and Application Security and Privacy*, pages 269–280. ACM, 2017.

[56] M. Cadariu, E. Bouwers, J. Visser, and A. van Deursen. Tracking known security vulnerabilities in proprietary software systems. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 516–519, March 2015.

[57] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, May 2017.

[58] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web. In *NDSS Symposium*, 2017.

[59] Samim Mirhosseini and Chris Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 84–94. IEEE Press, 2017.

[60] Jesus M Gonzalez-Barahona, Paul Sherwood, Gregorio Robles, and Daniel Izquierdo. Technical lag in software compilations: Measuring how outdated a software deployment is. In *IFIP International Conference on Open Source Systems*, pages 182–192, 2017.

[61] S. Moser and O. Nierstrasz. The effect of object-oriented frameworks on developer productivity. *Computer*, 29(9), 1996.

[62] Ahmed Zerouali and Tom Mens. Analyzing the evolution of testing library usage in open source java projects. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 417–421. IEEE, 2017.

[63] Ahmed Zerouali. Analysis and observations of the evolution of testing library usage. In *2017 the Seminar Series on Advanced Techniques and Tools for Software Evolution (SATToSE)*, 2017.

[64] Alex Zhitnitsky. We analyzed 60,678 libraries on github - here are the top 100. http://blog.takipi.com/we-analyzed-60678-libraries-on-github-here-are-the-top-100/, April 2015.

[65] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Working Conf. Mining Software Repositories*, pages 207–216. IEEE, 2013.

[66] Mathieu Goeminne, Alexandre Decan, and Tom Mens. Co-evolving code-related and database-related changes in a data-intensive software system. In *CSMR-WCRE*, pages 353–357, 2014.

[67] M. Goeminne and T. Mens. Towards a survival analysis of database framework usage in Java projects. In *Int'l Conf. Software Maintenance and Evolution*, pages 551–555, September 2015.

[68] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela Damian. The promises and perils of mining GitHub. In *Working Conf. Mining Software Repositories*, pages 92–101, 2014.

[69] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse*, pages 95–110. Springer, 2018.

[70] Ahmed Zerouali, Tom Mens, Jesus Gonzalez-Barahona, Alexandre Decan, Eleni Constantinou, and Gregorio Robles. A formal framework for measuring technical lag in component repositories — and its application to npm. *Journal of Software: Evolution and Process*, page e2157, 2019.

[71] J. Dietrich, K. Jezek, and P. Brada. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *CSMR-WCRE*, pages 64–73, 2014.

[72] Georgios Digkas, Mircea Lungu, Alexander Chatzigeorgiou, and Paris Avgeriou. The evolution of technical debt in the apache ecosystem. In *ECSA*, pages 51–66. Springer, 2017.

[73] Ward Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1993.

[74] Robert M Groves, Floyd J Fowler Jr, Mick P Couper, James M Lepkowski, Eleanor Singer, and Roger Tourangeau. *Survey methodology*, volume 561. John Wiley & Sons, 2011.

[75] Eleni Constantinou and Tom Mens. An empirical comparison of developer retention in the RubyGems and npm software ecosystems. *Innovations in Systems and Software Engineering*, 13(2-3):101–115, 2017.

[76] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. Adding sparkle to social coding: An empirical study of repository badges in the npm ecosystem. In *Proceedings of the 40th International Conference on Software Engineering*, pages 511–522. ACM, 2018.

[77] Andrew Nesbitt and Benjamin Nickolls. Libraries.io open source repository and dependency metadata (version 1.2.0) [data set]. Zenodo, March 2018.

[78] Christopher Bogart, Christian Kästner, and James Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *Automated Software Engineering Workshop*, pages 86–89. IEEE, 2015.

[79] Valerio Cosentino, Javier L Cánovas Izquierdo, and Jordi Cabot. A systematic mapping study of software development with github. *IEEE Access*, 5:7173–7192, 2017.

[80] Jérôme Vouillon and Roberto Di Cosmo. On software component co-installability. *ACM Trans. Softw. Eng. Methodol.*, 22(4):34:1–34:35, October 2013.

[81] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the evolution of technical lag in the npm package dependency network. In *International Conference on Software Maintenance and Evolution*. IEEE, 2018.

[82] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. On the relation between outdated docker containers, severity vulnerabilities, and bugs. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 491–501. IEEE, 2019.

[83] Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. On the impact of outdated and vulnerable javascript packages in docker images. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 619–623. IEEE, 2019.

[84] David Bernstein. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[85] Adrian Mouat. *Using Docker: Developing and Deploying Software with Containers*. O'Reilly Media, Inc., 2015.

[86] Carl Boettiger. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[87] Anthony Bettini. Vulnerability exploitation in docker container environments. https://www.blackhat.com/docs/eu-15/materials/eu-15-Bettini-Vulnerability-Exploitation-In-Docker-Container-Environments-wp.pdf, 2015. accessed: 01/01/2018.

[88] Fintan Ryan. Containers in production - is security a barrier? a dataset from Anchore. http://redmonk.com/fryan/2016/12/01/containers-in-production-is-security-a-barrier-a-dataset-from-anchore/, December 2016. accessed: 01/01/2018.

[89] Anchore.io. Snapshot of the container ecosystem. https://anchore.com/wp-content/uploads/2017/04/Anchore-Container-Survey-5.pdf, April 2017. accessed: 01/01/2018.

[90] Lucas Nussbaum and Stefano Zacchiroli. The ultimate Debian database: Consolidating bazaar metadata for quality assurance and data mining. In *Working Conf. Mining Software Repositories (MSR)*, pages 52–61, 2010.

[91] Manish Kumar Goel, Pardeep Khanna, and Jugal Kishore. Understanding survival analysis: Kaplan-meier estimate. *International journal of Ayurveda research*, 1(4):274, 2010.

[92] Ioannis Samoladas, Lefteris Angelis, and Ioannis Stamelos. Survival analysis on the duration of open source projects. *Information and Software Technology*, 52(9):902–922, 2010.

[93] Giuseppe Scanniello. Source code survival with the Kaplan Meier estimator. In *International Conference on Software Maintenance and Evolution*, pages 524–527. IEEE, 2011.

[94] Bin Lin, Gregorio Robles, and Alexander Serebrenik. Developer turnover in global, industrial open source projects: Insights from applying survival analysis. In *International Conference on Global Software Engineering*, pages 66–75. IEEE, 2017.

[95] Jeremy Valance. Improving open source security with anchore and snyk. https://anchore.com/blog/improving-open-source-security-with-anchore-snyk. accessed: 21/12/2018.

[96] Andrew Nesbitt and Benjamin Nickolls. Libraries.io open source repository and dependency metadata. March 2018.

[97] Shahed Zaman, Bram Adams, and Ahmed E Hassan. Security versus performance bugs: a case study on Firefox. In *8th Working Conference on Mining Software Repositories*, pages 93–102. ACM, 2011.

[98] Ahmed Zerouali, Valerio Cosentino, Gregorio Robles, Jesus M Gonzalez-Barahona, and Tom Mens. A tool to analyze packages in software containers. In *Mining Software Repositories 2019 (MSR)*, 2019.

[99] Jayanth Gummaraju, Tarun Desikan, and Yoshio Turner. Over 30% of official images in docker hub contain high priority security vulnerabilities. https://banyanops.com/blog/analyzing-docker-hub/, 2015.

[100] Anchore. Anchore.io, jan 2019.

[101] Elias Grande. Dagda, jan 2019.

[102] OWASP. Owasp, jan 2019.

[103] Matthew Wojcik, D Proulx, J Baker, and R Roberge. Introduction to oval. *The MITRE Corporation*, 2005.

[104] ExploitDB. Offensive security exploit database, jan 2019.

[105] Snyk.io. Synk.io. https://snyk.io/features/container-vulnerability-management/, March 2019. accessed: 10/03/2019.

[106] Kevin Crowston, Hala Annabi, and James Howison. Defining open source software project success. *Int'l Conf. Information Systems (ICIS)*, page 28, 2003.

[107] Thanh Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.

[108] Antony Martin, Simone Raponi, Théo Combe, and Roberto Di Pietro. Docker ecosystem–vulnerability analysis. *Computer Communications*, 122:30–43, 2018.

[109] Tianyin Xu and Darko Marinov. Mining container image repositories for software configuration and beyond. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 49–52, 2018.

[110] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[111] Debian. snapshot.debian.org. https://snapshot.debian.org/. accessed: 25/01/2019.

[112] Debian. Security bug tracker. https://security-tracker.debian.org/tracker/. accessed: 25/01/2019.

[113] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90–95, 2007.

[114] Michael Waskom. seaborn: statistical data visualization. https://seaborn.pydata.org. accessed: 25/01/2019.

[115] Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine*. " O'Reilly Media, Inc.", 2015.

[116] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. *Jupyter Notebooks-a publishing format for reproducible computational workflows.* 2016.

[117] Joshua Kerievsky. *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.

[118] Melvin E Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[119] Damian A Tamburri, Philippe Kruchten, Patricia Lago, and Hans Van Vliet. Social debt in software engineering: insights from industry. *Journal of Internet Services and Applications*, 6(1):10, 2015.

[120] Bitergia. Grimoirelab: Free, libre, open source tools for software development analytics. https://chaoss.github.io/grimoirelab/. accessed: 28/01/2019.