Noname manuscript No.
(will be inserted by the editor)

A multi-dimensional analysis of technical lag in Debian-based Docker images

Ahmed Zerouali · Tom Mens · Alexandre Decan · Jesus Gonzalez-Barahona · Gregorio Robles

Received: date / Accepted: date

Abstract Container-based solutions, such as *Docker*, have become increasingly relevant in the software industry to facilitate deploying and maintaining software systems. Little is known, however, about how outdated such containers are at the moment of their release or when used in production. This article addresses this question, by measuring and comparing five different dimensions of *technical lag* that *Docker* container images can face: package lag, time lag, version lag, vulnerability lag, and bug lag. We instantiate the formal technical lag framework from previous work to operationalise these different dimensions of lag on *Docker Hub* images based on the *Debian Linux* distribution. We carry out a large-scale empirical study of such technical lag, over a three-year period, in 140,498 *Debian* images. We compare the differences between *official* and *community* images, as well as between images with different *Debian* distributions: *OldStable*, *Stable* or *Testing*. The analysis shows that the different dimensions of technical lag are complementary, providing multiple insights. *Official Debian* images consistently have a lower lag than *community* images for

A. Zerouali
Vrije Universiteit Brussel, Brussels, Belgium
Université de Mons, Mons, Belgium
E-mail: ahmed.zerouali@vub.be

T. Mens
Université de Mons, Mons, Belgium
E-mail: tom.mens@umons.ac.be

A. Decan
Université de Mons, Mons, Belgium
E-mail: alexandre.decan@umons.ac.be

J. Gonzalez-Barahona
Universidad Rey Juan Carlos, Madrid, Spain
E-mail: jgb@gsyc.es

G. Robles
Universidad Rey Juan Carlos, Madrid, Spain
E-mail: grex@gsyc.urjc.es

all considered lag dimensions. The amount of lag incurred depends on the type of *Debian* distribution and the considered lag dimension. Our research offers empirical evidence that developers and deployers of *Docker* images can benefit from identifying to which extent their containers are outdated according to the considered dimensions, and mitigate the risks related to such outdatedness.

Keywords: technical lag, container images, Docker, outdated packages, security vulnerabilities, bugs, Debian, empirical analysis.

1 Introduction

During the last years, new ways of deploying software have become mainstream to support new software architectures, such as those based on micro-services. One of the most successful among them is based on containers, and in particular on *Docker* (Bernstein, 2014; Turnbull, 2014). *Docker* containers emerged as a lightweight solution (Mouat, 2015) capable of provisioning multiple, partially isolated applications on a single host. Each application runs in a separate container, including all external libraries and applications needed to perform its job, all of them sharing system libraries, configuration files, and an operating system in the same host (Merkel, 2014). Containers are run from *images* that include file systems and all data needed for their execution. To ease container deployment, collections of ready to be deployed container images are made available on registries. One of the largest of such registries is *Docker Hub*, with more than 5.8M images (as of June 2019).

Docker images in registries are usually deployed in production, either directly or indirectly, via *derived images*. Derived images are based on other images, by adding and/or removing files from them. Therefore, images need to be carefully tested, both before and after deployment. Switching to a new image version is risky, since new versions may come with backward incompatible changes and may give rise to unforeseen co-instability issues (Artho et al., 2012; Claes et al., 2015; Vouillon and Di Cosmo, 2011). Therefore, there is a strong tendency to avoid updating to new versions of container images in a given deployment. On the other hand, those images need to be updated at some point, because new versions fix bugs, avoid known security vulnerabilities, and include new or improved functionality. This was confirmed by different studies and surveys (Anchore.io, 2017; Bettini, 2015) showing that among *Docker* practitioners, the absence of security vulnerabilities is more important than benefiting from new features. This causes a dilemma between *updating* deployed images to their latest versions and benefiting from the fixed vulnerabilities and bugs, or *staying* with the currently deployed and working version.

In previous work, Zerouali et al. (2019b) introduced the *technical lag framework* as a formal model to capture this dilemma of whether or not to update. This model allows to quantify the differences between two versions of a package, and to aggregate these differences between collections of packages. In this article, we apply the formal model to *Docker* container images, to compute

the technical lag between package releases contained in image versions, and then between the image versions themselves.

We focus on *Docker Hub* images that are based on the *Debian Linux* distribution. Such images were already studied in Zerouali et al. (2019c), and this article extends the previous research in multiple ways. While Zerouali et al. (2019c) considered only one snapshot in time of *Docker Hub* images, this article carries out an evolutionary analysis of *Docker Hub* images at their release dates, providing insights in how the lag of *Docker* images evolved across *Debian* releases over a three-year time period. The size of the used dataset of *Docker* images is also an order of magnitude larger in this article. We consider all images available in the most relevant *Docker Hub* repositories, yielding 19 times more images: 140,498 compared to only 7,380 in (Zerouali et al., 2019c). Moreover, this article applies the technical lag framework of (Zerouali et al., 2019b) to *Docker* images for the first time. While (Zerouali et al., 2019b) considered only three different variants of technical lag, this article evaluates and compares five different dimensions (i..e, framework instantiations) of technical lag: package lag, time lag, version lag, vulnerability lag and bug lag.

This naturally leads us to five research questions (RQ_1 to RQ_5; one per considered lag dimension), aiming to understand how the technical lag evolves over time in *Debian*-based *Docker* images. To answer these questions, we empirically analyzed the five considered dimensions of technical lag for these images. The results of these analyses led us to derive some insights about *Debian*-based *Docker* images. Among them, we can highlight the following ones. *Official Debian* images consistently have a lower lag than *community* images for all considered lag dimensions. The amount of lag incurred depends on the type of distribution and the considered lag dimension. While *Debian* images tend to have a very low proportion of outdated packages, they all suffer from bugs and security vulnerabilities to a certain extent. *Debian Testing* images have higher package and version lag than other images, while images relying on older and more stable *Debian* distributions have higher vulnerability lag, which means those "more stable" images are more vulnerable from a security point of view. Moreover, we find all dimensions of lag to be increasing over time in *OldStable* images, while they are more stable in the case of *Stable* images. For *Testing* images, only time and version lag tend to increase over time.

We expect that the findings of the analysis will be of benefit to *Docker* developers, helping them to create better images, while at the same time allowing deployers to assess the outdatedness of the images they rely on.

The remainder of this article is structured as follows. Section 2 provides background on *Docker* images and the technical lag framework. Section 3 discusses related work. Section 4 explains the research method and data extraction process, and presents a preliminary analysis of the selected dataset. Section 5 instantiates the formal technical lag framework to the case study of the *Docker* images. Section 6 carries out a multi-dimensional empirical analysis of the technical lag incurred by *Debian*-based images on *Docker Hub*. Section 7 highlights the novel contributions, discusses our findings, and outlines possible

directions for future work. Finally, Section 8 discusses the limitations of this work and Section 9 concludes.

2 Background

This section provides the necessary background on technical lag (Section 2.1) and *Docker* images (Section 2.2), which is required for understanding the remainder of this article.

2.1 Technical Lag Framework

The concept of technical lag was initially introduced by Gonzalez-Barahona et al. (2017) to quantify how outdated a deployed software component is, reflecting *"the increasing lag between upstream development and the deployed system when no corrective actions are taken"*. However, the notion of technical lag is not only useful for *deployers* of software components, but also for the *developers* of such components. Developers can use technical lag to decide whether or not to seize the opportunity of updating the external dependencies of the components they maintain. That way they can assess, on an informed basis, the risks of relying on outdated dependencies (Cox et al., 2015).

To empirically evaluate the practical impact of technical lag from the perspective of open source software *developers*, we have previously conducted several quantitative case studies (Decan et al., 2018a; Zerouali et al., 2018). In particular, we assessed the technical lag induced by outdated package dependencies in the *npm* registry of reusable JavaScript libraries. To evaluate the importance of technical lag from the perspective of software *deployers*, we carried out a case study on *Docker* container images based on the *Debian Linux* distribution (Zerouali et al., 2019c). In that study, we considered three ways of measuring technical lag, namely, in terms of version updates, security vulnerabilities and bugs.

To formally capture the different perspectives on technical lag, as well as the different ways of measuring it, we introduced a technical lag measurement framework, and validated it on the above case studies (Zerouali et al., 2019b):

> We define a *technical lag framework* $\mathcal{F}$ as a tuple $(\mathcal{C}, \mathcal{L}, \mathsf{ideal}, \Delta, \mathsf{agg})$ where $\mathcal{C}$ is a set of component releases; $\mathcal{L}$ is a set of possible lag values; $\mathsf{ideal} : \mathcal{C} \to \mathcal{C}$ is a function returning the most preferred component release; $\Delta : \mathcal{C} \times \mathcal{C} \to \mathcal{L}$ is a function computing the difference (in terms of lag induced) between two component releases; $\mathsf{agg} : \mathbb{P}(\mathcal{L}) \to \mathcal{L}$ is a function aggregating the results of a set of lag values.

Given such a framework $\mathcal{F}$, we can formally define the *technical lag* induced by using a component release instead of the ideal release for that component. The technical lag induced by using a less than ideal release can be computed by

the difference function Δ between that release and the ideal one. This lag can be aggregated over a set of component releases D by applying an aggregation function agg over all its members.

$$\mathsf{techlag}_{\mathcal{F}} : \mathcal{C} \to \mathcal{L} : c \to \Delta(c, \mathsf{ideal}(c))$$
$$\mathsf{agglag}_{\mathcal{F}} : \mathbb{P}(\mathcal{C}) \to \mathcal{L} : D \to \mathsf{agg}(\{\mathsf{techlag}_{\mathcal{F}}(c) \mid \forall c \in D\})$$

In practice, how to define the ideal (e.g., the most stable, most recent, or most secure component), how to measure the difference Δ between two releases, and how to aggregate the technical lag over a set of component releases (e.g., using the maximum or the sum) will depend on the considered scenario of use.

2.2 On Docker images

Docker Hub is the world's largest registry and community for container images. *Docker Hub* is organized in repositories, each containing versioned *Docker* images. Repositories may be public (with unrestricted access) or private. Public repositories are categorized into *official* and *community* repositories. The *official* status of a repository can be seen as a quality label, signaling that the repository contains secured and well-maintained images, produced by well-known organizations (e.g., *ElasticSearch*, *MySQL*, or *Debian*). Images in *official* repositories are frequently used as the basis for other *Docker* images, because of their perceived good quality. *Community* repositories can be created by any user or organization (Boettiger, 2015).

Since *Docker* images are used to create runnable containers, they include complete operating systems. Images that include *Linux*-based operating systems follow the packaging model of the chosen *Linux* distribution (e.g., *Alpine* or *Debian*). *Docker* images may also include a collection of third-party packages coming from specific package repositories, such as *npm* or *PyPI* (the most used package repositories for *JavaScript* and *Python*, respectively). Once a certain version of an image is built, packages remain *frozen* in it.

To facilitate search and use in *Docker Hub*, images are labeled with the name of the repository and a tag (e.g., debian:stretch). An image can be tagged more than once, and therefore may have more than one label (e.g., debian:stretch and debian:stable). The name of *community* image repositories (e.g., grimoirelab/full:0.2.26, bitnami/mysql:latest) usually starts with the name of the organisation producing the images.

The build configuration of an image is declared using a *Dockerfile* (Docker Inc., 2020b) consisting of a list of commands used to produce the image. Each command results in a new *layer* with a unique hash signature. Hence, a Dockerfile produces a *stack* of layers, one layer for each command. In addition, a Dockerfile can be based on another Dockerfile, including all its layers. This

way, an image can be built on top of another one, leading to a hierarchy of images. When a new image container is created, a new writable layer will be added on top of the underlying layers. This layer is called the *container layer*. All changes made to the running image, such as writing new files, modifying existing files, and deleting files, are performed on this writable container layer.

Listing 1 Dockerfile of the image debian:buster-backports

```
1 FROM debian:buster
2 RUN
    ↪ echo "deb http://deb.debian.org/debian buster-backports
    ↪ main" > /etc/apt/sources.list.d/backports.list
```

For example, Listing 1 shows the content of the Dockerfile of the image debian:buster-backports. Since the Dockerfile contains two commands (`FROM` and `RUN`), building the image with this Dockerfile on *17-Nov-2019* produced a list of two layers: `[5ae19949497e, 52c713817fee]`.

This image can be used by other Dockerfiles as their base image using the command `FROM debian:buster-backports`. Each image produced from those Dockerfiles will inherit the layers of the base image. For example, Listing 2 shows a fragment of the Dockerfile[1] used to build community image shogun/shogun-dev:latest. The `FROM` command on line 1 pulls the image. Line 2 just provides information about the maintainer of the Dockerfile. The `RUN` command starting in line 3 provides a series of instructions to build the image. For example, the instruction `apt-get upgrade -y` upgrades already installed packages, and the instruction `apt-get install -qq` installs a new set of packages listed after the options (e.g., *make*, *gcc*, etc.).

Listing 2 Excerpt of the Dockerfile of shogun/shogun-dev:latest

```
1 FROM debian:buster-backports
2 MAINTAINER shogun@shogun-toolbox.org
3 RUN apt-get update -qq &&
    ↪ apt-get upgrade -y &&
    ↪ apt-get install -qq --force-yes --no-install-recommends
    ↪ make gcc g++ libc6-dev libbz2-dev ccache libarpack2-dev
    ↪ ...
```

The resulting shogun/shogun-dev:latest image on *17-Nov-2019* includes the layers found in its base image debian:buster-backports, as evidenced by the two

[1] `https://github.com/shogun-toolbox/shogun/blob/develop/configs/shogun-sdk/Dockerfile`

first items in its list of layers:

```
[ 5ae19949497e, 52c713817fee,
  ed222bc780e3, a0c30a484e89, 4c6e5392d047, ...]
```

3 Related Work

This section presents a summary of related research, organized into four subsections. Section 3.1 reviews studies about software package outdatedness, Section 3.2 reviews studies about software security vulnerabilities, and Section 3.3 presents related work about the *Docker* technology. Section 3.4 highlights the novel contributions and the main differences between this study and prior related work.

3.1 On outdated software components

Today's software systems are heavily dependent on components retrieved from software package registries (e.g., *Maven* for *Java*, *npm* for *JavaScript*, the *Debian* distribution of *Linux* packages, etc.). Such package registries are growing quickly and facing very frequent package updates (Decan et al., 2019). Many researchers have studied the impact of relying on outdated software components. Kula et al. (2015) empirically analysed thousands of *Java* libraries distributed on *Maven* to study their latency in adopting the latest version of their dependencies. They found that maintainers are reluctant to adopt the latest version of a library at the beginning of a project. They also found that maintainers are more inclined to use the latest available version when they actually introduce a new dependency in their project. In a follow-up work, they studied library migration for over 4,600 *GitHub* software projects and 2,700 library dependencies (Kula et al., 2017). They observed that four out of five of the projects have their dependencies outdated. A survey with project maintainers revealed that a large majority of them were unaware of such outdated dependencies. Cogo et al. (2019) studied the effect of package downgrades in *npm* on technical lag. They observed that one fifth of all downgraded packages increase the technical lag of client packages. More specifically, downgrades of major versions introduce more technical lag than downgrades of minor and patch versions (i.e., they tend to downgrade further than the latest previous working version). This caused an unnecessary (i.e., avoidable) increase of technical lag for 13% of the downgrades.

Salza et al. (2020) investigated the ecosystem of mobile apps for Android, by studying whether, when, how, and why such apps update the third-party libraries they depend on. They used the technical lag concept to quantify the difference between the library versions used in the mobile apps and the latest available version of these libraries. Their main finding was that mobile developers rarely update library dependencies, and when they do, they mainly tend to update dependencies related to the graphical user interface. Mezzetti et al.

(2018) note that the dynamic nature of *JavaScript* often causes unintended breaking changes to be detected too late, because its semantic versioning system relies on the ability to distinguish between breaking and non-breaking changes when libraries are updated. Developers tend then not to update their dependencies, at the risk of not including security critical updates. To mitigate this, they present a technique, which they call type regression testing, to automatically determine when an update affects other *JavaScript* packages. Møller and Torp (2019) presented a model-based variant of this solution that solves some of its scalability limitations.

Gonzalez-Barahona et al. (2009) investigated the problem of outdated dependencies in Linux-based software component distributions. Legay et al. (2020) surveyed 170 Linux users and observed that keeping packages up to date is an important concern from them, and that some distributions are perceived to be quicker in deploying package updates than others. Abate et al. (2009, 2012) identified different types of dependencies that may arise between software components, and proposed solutions for managing such dependencies. Abate et al. (2014) proposed a framework to detect future problems related to challenging upgrades and outdated packages, and validated it on *Debian*.

Many tools are available to monitor, for a given software system, the *freshness* of its dependencies (i.e., the libraries the software system requires). For example, *npm* provides the npm outdated command to check the registry to see if any (or, specific) installed packages are currently outdated[2]. Similar tools are available for other ecosystems such as *Maven* (e.g., Versions Maven Plugin[3]), *Node.js* (e.g., David[4]) and *RubyGems* (e.g., gem outdated[5]). Most of the existing solutions, however, only mention the latest available version compared to the installed one. In other words, such tools aim to detect outdatedness, but do not quantify nor measure its extent.

All of the aforementioned tools are not specific to *Docker*, but can be used to check the freshness of packages installed within *Docker* containers. There are also tools that specifically focus on *Docker* containers. Some of them are open source, such as Watchtower[6] and Ouroboros[7] that continuously monitor if there are new versions of the (base) images that running images are using, and update those images automatically from *Docker Hub* if this is the case. Other tools are commercialised by companies like Anchore.io, Quay.io and Snyk.io, and check more low-level details of the installed system and third-party libraries, and offer information about the freshness and vulnerabilities in *Docker* images.

[2] https://docs.npmjs.com/cli/outdated.html
[3] http://www.mojohaus.org/versions-maven-plugin/
[4] https://david-dm.org/
[5] https://guides.rubygems.org/command-reference/#gem-outdated
[6] https://github.com/containrrr/watchtower
[7] https://github.com/pyouroboros/ouroboros

3.2 On security vulnerabilities

Several researchers observed that outdated dependencies are a potential source of security vulnerabilities. Cox et al. (2015) analyzed 75 *Java* projects that manage their dependencies through *Maven*. They observed that projects using outdated dependencies were four times more likely to have security issues and backward incompatibilities than systems that were up-to-date.

Decan et al. (2018b) carried out an empirical analysis of security vulnerabilities in the *npm* ecosystem of *JavaScript* packages. They analyzed how and when these vulnerabilities are discovered and fixed, and to which extent this affects direct or indirect dependent packages. 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. 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.

Zapata et al. (2018) offered a different perspective by analyzing vulnerable dependency migrations at the function level for 60 *JavaScript* packages. They provided evidence that many outdated projects are free of vulnerabilities as they do not really rely on the functionality affected by the vulnerability. Because of this, the authors claim that security vulnerability analysis at package dependency level is likely to be an overestimation.

Zimmermann et al. (2019) provided evidence that the *npm* ecosystem suffers from single points of failure, i.e., a very small number of maintainer accounts could be used to inject malicious code into the majority of all packages. This problem is increasing over time, and unmaintained packages threaten large code bases, as the lack of maintenance causes many packages to depend on vulnerable code, even years after a vulnerability has become public.

A survey with *Docker* deployers revealed that the absence of security vulnerabilities is a top concern when deciding whether to deploy *Docker* images (Bettini, 2015). Another survey showed that in addition to security, *Docker* developers and deployers are concerned about other software package checks such as making sure there are no bugs in major third-party software or verifying whether third-party software versions are up-to-date (Anchore.io, 2017). Yet another survey showed that only 19% of developers claim to test their *Docker* images for vulnerabilities during development (Vermeer and Henry, 2019). This signals a tendency to deliver images without inspecting them in detail for security weaknesses.

Combe et al. (2016) provided a comprehensive overview of security vulnerabilities in the *Docker* container ecosystem. They defined an adversary model that pointed out several vulnerabilities affecting the usages of *Docker*. Shu et al. (2017) performed a large-scale study on the state of security vulnerabilities in *official* and *community Docker Hub* repositories. They proposed the *Docker Image Vulnerability Analysis (DIVA)* framework to automatically discover, download, and analyze *Docker* images for security vulnerabilities. By

studying a set of 356,218 images they observed that both *official* and *community* repositories contain an average of 180 vulnerabilities. Many images had not been updated for hundreds of days, calling for more systematic methods for analysing the content of *Docker* containers.

With respect to tool support, Zerouali et al. (2019a) developed ConPan[8], an open source tool that analyses the technical lag, vulnerabilities and bugs in packages installed in *Docker* images. Kwon and Lee (2020) proposed DIVDS, a *Docker* image vulnerability diagnostic system. The system diagnoses security vulnerabilities in *Docker* images when they are uploaded to or downloaded from the *Docker Hub* image repository.

3.3 Other studies on *Docker*

Beyond research on security vulnerabilities, other empirical studies have been conducted on *Docker* containers. Cito et al. (2017) characterized the *Docker* ecosystem by discovering prevalent quality issues and studying the evolution of *Docker* images. Using a dataset of over 70,000 *Dockerfiles* they contrasted the general population with samplings containing the top 100 and top 1,000 most popular projects using *Docker*. They observed that the most popular projects change more often than the rest of the *Docker* population. Furthermore, based on a representative sample of projects, they observed that one out of three Docker images could not be built from their *Dockerfiles*.

Lu et al. (2019) offered another perspective on the quality of Docker images, by focusing on what they refer to as *temporary file smells*. In the building process of *Docker* images, temporary files are often used. If such temporary files are imported and subsequently removed in different layers by a careless developer, it leads to the presence of unneeded files, resulting in larger images. This restricts the efficiency and quality of image distribution and thus affects the scalability of services. Through an empirical case study on 3,242 real-world *Dockerfiles* on *Docker Hub* the presence of this temporary file smell was observed in a wide range of *Dockerfiles*.

Online services such as *Docker Hub* and *Docker Store* host open source software repositories for a large number of reusable *Docker* images. Effectively reusing these images requires a good understanding of them, and semantic tags facilitate this understanding. To address this problem, Zhou et al. (2019) proposed *SemiTagRec*, a semi-supervised learning based tag recommendation approach for *Docker* repositories.

Henkel et al. (2020) studied the need for more effective semantics-aware tooling in the realm of Dockerfiles, in order to reduce the quality gap between Dockerfiles written by experts and those found in open-source repositories. They identified and addressed three challenges in learning from, understanding, and supporting developers writing Dockerfiles: (i) nested languages in Dockerfiles, (ii) rule mining, and (iii) the lack of semantic rule-based analysis.

[8] `https://github.com/neglectos/ConPan`

They observed that best practices and rules for *Docker* have arisen, but developers are often unaware of these practices and therefore do not tend to follow them. On average, Dockerfiles on *GitHub* were found to have nearly five times more rule violations[9] than those written by *Docker* experts.

Socchi and Luu (2019) carried out a deep analysis of *Docker Hub*'s security landscape. They collected and analyzed a large amount of metadata and vulnerability information about *certified* [10] official and community images on *Docker Hub*. They observed that certified and verified [11] repositories do not lead to a significant improvement of the overall security of *Docker* images on *Docker Hub*. They predicted that the average number of unique vulnerabilities found across all types of repositories is expected to grow with a rate of approximately 105 vulnerabilities per year between 2019 and 2025 if *Docker Hub* continues evolving the same way.

3.4 Novel Contributions

Based on this previous and related work, we decided to focus our study on applying a theoretical framework of technical lag to quantify outdatedness of container images, and with it, of container deployments (since those containers are in many cases deployed directly in production environments). We instantiate technical lag in five different ways, and use it to measure *Docker* images with different aims. As a result of this study, we also provide actionable insights to developers, deployers and tool providers of *Docker* images, that they can use in their decision processes to produce updated images, or to deploy them in production.

Our study is not the first one to analyse *Docker Hub* images, but none of the previous studies we are aware of focused on studying images for a specific operating system, exploring in detail their characteristics. We improved this situation by producing a large, curated dataset of *Debian*-based *Docker Hub* images, and analysing it, finding interesting insights on how the different *Debian* distributions and images based on them are updated over time, and how suitable they are for fulfilling different optimisation criteria for assessing decisions at deployment time.

We also included data about bug reports and vulnerabilities for *Debian* packages, retrieved from the *Debian* Ultimate Database and the *Debian* Security Tracker. Although relying on these databases is not novel by itself, the combination of data from both of them with the history of *Debian* packages used in *Docker* images is, to our knowledge, new in the literature. It allowed us to perform detailed analysis on vulnerabilities and bugs in *Debian*-based *Docker* images, adding useful dimensions to our study.

[9] An example of rule violation is forgetting the `-y` flag when using `apt-get install`

[10] Certified images are built with best practices, tested and validated against the Docker Enterprise Edition and pass security requirements.

[11] Verified images are high-quality images from verified publishers. These products are published and maintained directly by a commercial entity.

Another relevant contribution is the combination of the main data sources we used: a newly created dataset of *Debian*-based *Docker Hub* images, and another dataset produced from the whole collection of *Debian* packages over time. To our knowledge, this is the first time that data related to deployments (of container images) and packaged modules (of *Debian* packages) used by them are studied together. It allows us to determine with precision how outdated images are, given the availability of packages that could have been used in building them.

We also analyse technical lag for *Docker* images from two different points of view: as it was at their last modification date and as it was at the date of the analysis (more details in Section 4). The first viewpoint is a good measure of how outdated images were when they were produced, with respect to packages available at that time. Therefore, that measure informs on how much better the producers of those images could have performed. The second viewpoint reflects how far away those images at production time were from an ideal "most possible up-to-date" image. This is a good measure of the outdatedness of contained packages if these images were to be used for production at the moment we performed our study. Therefore, it informs on how appropriate they are for deployment in production at that time, given a certain set of criteria (such as minimising the number of vulnerabilities).

4 Method and data extraction

This section introduces the case study of *Debian*-based *Docker* images and the methodology to extract these images, their package releases and their related bugs and security vulnerabilities. We decided to focus on *Docker* images based on a *Linux* distribution, because applications in them are usually installed using well-defined packages. We selected the *Debian* distribution because of its maturity and widespread use in *Docker Hub* (DeHamer, 2020). On 6 October 2019, the *Debian* repository on *Docker Hub* had more than 284M pulls[12].

The *Debian* project maintains packages for several simultaneous release lines, referred to as *distributions* (Gonzalez-Barahona et al., 2009). The most important distributions are *Testing*, *Stable* and *OldStable*. *Testing* packages are frequently updated since they are continuously inspected for defects. Whenever the *Testing* distribution as a whole reaches a certain level of quality and stability (e.g., lack of critical bugs, successful compilation, etc), it is "frozen", and its packages are used to produce a new *Stable* distribution. Upon release of a *Stable* version, the former one becomes *OldStable*, which in turn becomes *Oldoldstable*. While package 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* images on *Docker Hub* only for *Testing*, *Stable* and *OldStable*. Table 1 provides general information about these considered *Debian* distributions.

[12] `https://registry.hub.docker.com/v2/repositories/library/debian/`

Table 1 General information about the considered *Debian* distributions.

Distribution type	Version number	Distribution name	Release date as stable
OldStable	*Debian* 8	*Jessie*	2013-04-25
Stable	*Debian* 9	*Stretch*	2017-06-17
Testing	*Debian* 10	*Buster*	2019-07-06

The process to compute the technical lag for *Debian*-based images on *Docker Hub* is composed of the following steps:
(1) identify *Docker* images that are based on *Debian*;
(2) retrieve the list of installed package releases from these images, and match them with the list of all known *Debian* packages;
(3) map (a) vulnerability reports and (b) bug reports to these package releases.
As shown in Figure 1, combining all these steps results in a dataset that we use in Section 6 to measure and analyse the technical lag of *Debian* images.

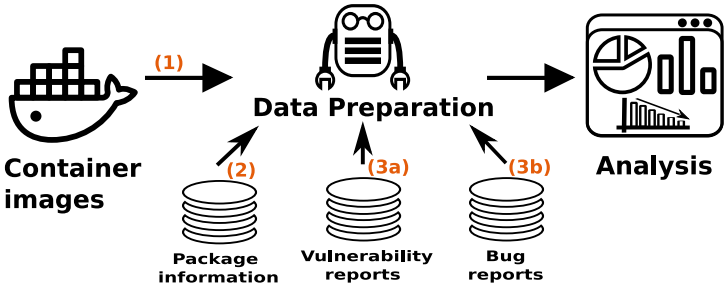

Fig. 1 Process of the *Docker* container package analysis.

Step 1 – *Debian*-based *Docker* images

We extracted the list of all repositories that are available on *Docker Hub* in May 2019 using the Docker Registry HTTP API (Docker Inc., 2020a). We found 150 public *official* repositories and 1,929,428 *community* repositories. We extracted the name and the list of layers of all images provided by each repository in June 2019. We found 27,760 images in *official* repositories and 5,842,567 images in *community* repositories.

Based on the image layering mechanism explained in Section 2.2, we identified whether an image is built on top of one of the *official Debian* images provided by *Docker Hub*[13]. Since several image names (or tags) can correspond to the same image, we removed duplicate images by comparing the hash values of their layers.

[13] `https://hub.docker.com/_/debian`

Among all *Docker Hub* images, we found 9,581 *official Debian* images, as well as 924,139 *community Debian*-based images coming from 400,193 repositories. Out of the 9,581 *official Debian* images, 9,330 images used one of the three *Debian* distributions considered in Table 1, i.e., either *Testing* (*Debian* 10), *Stable* (*Debian* 9) or *OldStable* (*Debian* 8). The remaining 251 images used older *Debian* distributions and were therefore excluded from our dataset. The observed high number of *community Debian*-based images is unsurprising since anyone can upload images on *Docker Hub*. Many of these are experimental or personal images, or images that are never expected to be reused by deployers or by developers of other images. Therefore, we decided to focus on those images that are most worthwhile for deployers and developers based on the number of pulls that *Docker Hub* reports for each *community* repository. We started downloading *Debian* images from *community* repositories sorted by decreasing number of pulls. After one full month of continuous downloads, the dataset was composed of 131,168 images. This corresponds to 14.2% of all *Debian community* images, with a number of pulls ranging from 1,954 to tens of millions. We stopped downloading more images at this point,[14] since these images already represented 81% of the total number of pulls of all *Debian* images, and were therefore representative of the usage of *Debian*-based images from *community* repositories. Moreover, by doing so, we ensure to exclude the long tail of experimental or personal images.

Table 2 Number of *Docker* images per *Debian* distribution.

***Debian* images**	***Testing***	***Stable***	***OldStable***	**total**
official	741	5,504	3,085	9,330
community	3,114	75,534	52,520	131,168
Total	3,855	81,038	55,605	140,498

Table 2 shows the breakdown of all downloaded *official* and *community* images per considered *Debian* distribution. While these images were available on *Docker Hub* in June 2019, they were not all created and published at the same time. Figure 2 shows the evolution of the number of images proportionally to the number of extracted images from *Docker Hub* between 2016 and 2019, grouped by their origin (*official* or *community*) and *Debian* distribution. We observe that 62.7% of the *official* images and 59.3% of the *community* images were last updated before 2019. Regardless of the origin, we observe an exponential increase over time for images based on the *Testing* distribution ($R^2 = 0.95$) and *Stable* distribution ($R^2 = 0.85$), while the proportion of *OldStable* images is linearly increasing over time ($R^2 = 0.93$).

[14] Downloading all available images would have taken at least 6 extra months, and would have required considerably more storage capacity.

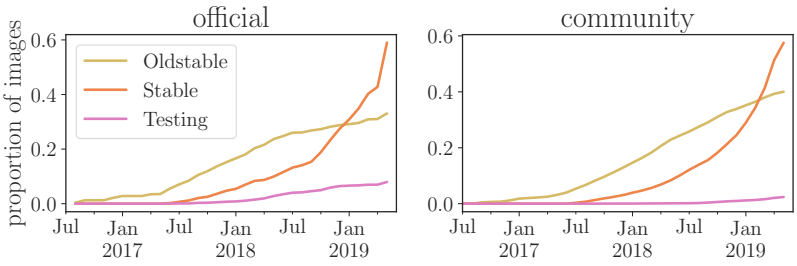

Fig. 2 Evolution of the number of images proportionally to the number of extracted images from *Docker Hub*, grouped by *Debian* distribution (*Testing*, *Stable*, *OldStable*) and image origin (*official* or *community*).

Step 2 – *Debian* package releases

We pulled each of the considered *Docker* images and ran `dpkg`, the official package management tool of *Debian*, to get the list of installed package releases (i.e., `dpkg -l`). We compared each package release against the list of all package releases available for the considered *Debian* distributions. This list was obtained by downloading the daily snapshots of all binary packages for *OldStable*, *Stable* and *Testing* from the official and security *Debian Snapshot* repositories[15]. Based on this list, we identified 46,272,487 package releases in the set of considered images, covering more than 99% of all the package releases we extracted using `dpkg`. We found a median of 185 installed packages in *official* images, and 377 in *community* images. Figure 3 shows the evolution of the statistical distribution of the number of packages installed in considered *official* and *community* images. We observe that the number of installed packages per image does not change over time, and that *community* images have a higher number of packages installed compared to *official* images. This is expected, since *community* images are usually built on top of *official* ones. We investigated whether the number of installed packages relates to the *Debian* distribution in which the image was found, but could not find any significant difference between these distributions.

Step 3 – Vulnerability and bug reports

The third step focuses on identifying vulnerability and bug reports, and mapping these reports to installed package releases identified in step 2. We relied on the *Debian Security Tracker*[16] to identify security vulnerabilities in *Debian* packages. This security tracker is maintained by the *Debian Security*

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

[16] `security-tracker.debian.org/tracker/data/json`

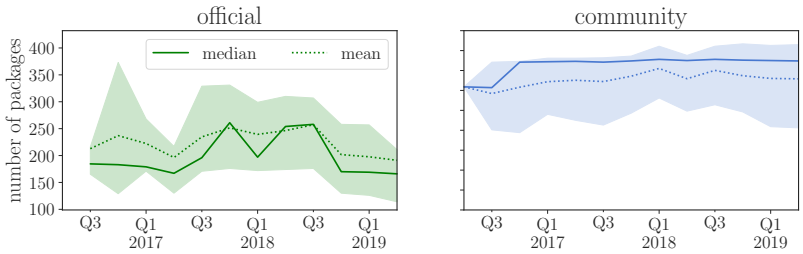

Fig. 3 Quarterly evolution of the population of packages installed in considered *Docker* images. (The shaded area corresponds to the interval between the 25th and 75th percentile.)

Team using data from various data sources (e.g., CVE database[17] or National Vulnerability Database "NVD"[18]). We extracted a list with all 29,111 known vulnerabilities as of August 2019.

Each vulnerability report contains information about the affected packages, its status (e.g., *open* or *resolved*), affected distributions, fixed versions, etc. A report also indicates the severity of a vulnerability, ranging from *unimportant* to *high*. We decided to ignore reports whose severity is either *unimportant*, *not yet assigned* or *end-of-life* since our focus is on relevant security vulnerabilities. We mapped these reports to the package releases identified in step 2. We considered a package release to be vulnerable if its package is involved in a vulnerability report, and the corresponding vulnerability is either not (yet) resolved, or resolved in a more recent release than the considered one.

The final list of vulnerabilities contains 5,119 reports involving 557 distinct packages (around 10% of all packages identified in step 2). Figure 4 shows the proportion of vulnerabilities with respect to their status (*open* or *resolved*) and their severity (*low*, *medium* or *high*). We observe that most vulnerabilities (73.3%, i.e., 4,435) are *resolved*, regardless of their severity. Most vulnerabilities have a *medium* severity (59.1%), followed by *low* (21.1%) and *high* severity (19.8%).

For bug reports, we relied on the *Ultimate Debian Database*[19], a continuously updated system that collects a variety of data (e.g., packages, bugs, upload history) and stores them in a publicly available database (Nussbaum and Zacchiroli, 2010). We queried this database for all known bug reports for the packages identified in step 2. Bug reports indicate the severity of a bug, ranging from *wishlist* to *critical*[20]. We ignored those that have a *wishlist* severity since they do not really correspond to bugs. We considered a package release to be affected by a bug if its package is involved in a bug report, and

[17] `cve.mitre.org/cve/`
[18] `nvd.nist.gov`
[19] `udd.debian.org/bugs/`
[20] `https://www.debian.org/Bugs/Developer`

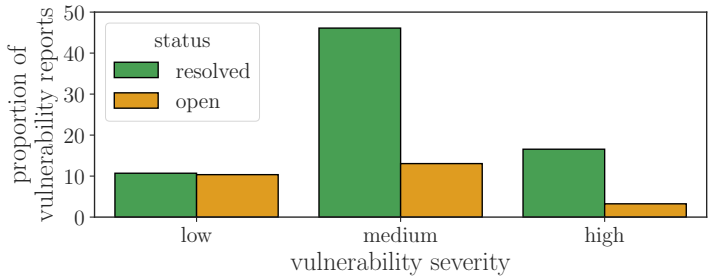

Fig. 4 Proportion of reported vulnerabilities in *Debian* packages contained in considered *Docker* images, grouped by resolution status and severity.

the corresponding bug is either not (yet) fixed, or fixed in a more recent release than the considered one.

The final list of bugs contains 45,953 reports involving 3,400 distinct packages (around 61% of all packages identified in step 2). Figure 5 shows the proportion of bug reports with respect to their status (*open* or *resolved*) and their urgency (*minor, normal, important, serious, grave* or *critical*). We observe that a large proportion of bugs, regardless of their urgency, is still *open* (55.5%). Highest urgency labels *serious, grave* or *critical* are only found in 7.5% of all considered bug reports. Most bugs have either a *normal* (56.8%) or *important* urgency (24.6%). The remaining ones have a *minor* urgency (11.1%).

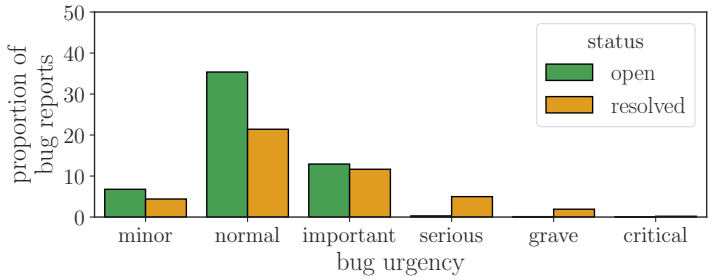

Fig. 5 Proportion of reported bugs in *Debian* packages contained in considered *Docker* images, grouped by resolution status and urgency.

5 Technical Lag Instantiations for *Debian*-based *Docker* Images

The goal of our empirical analysis is to investigate and compare the technical lag of *Debian*-based *Docker* images, using different ways of measuring such lag. More specifically, we consider five different dimensions:

- **package lag** indicates whether a given package release is outdated;
- **time lag** quantifies the time difference between package releases;
- **version lag** quantifies the number of missed versions between package releases;
- **vulnerability lag** measures the difference in number of vulnerabilities;
- **bug lag** measures the difference in number of bugs.

Each of these dimensions correspond to a specific instantiation of the technical lag framework $\mathcal{F} = (\mathcal{C}, \mathcal{L}, \mathsf{ideal}, \mathsf{delta}, \mathsf{agg})$ presented in Section 2 and defined in detail in (Zerouali et al., 2019b). To formalise these instantiations we need to introduce a number of auxiliary sets and functions.

5.1 Package releases

To start with, we formalise the notion of package releases contained in *Debian* distributions. We assume the existence of a set $\mathcal{N}$ of all strings, an ordered set $\mathcal{T}$ of all possible points in time, and a set $\mathcal{V}$ of all possible version numbers for *Debian* packages. We assume a total order on $\mathcal{V}$, following *Debian* specifications[21].

We define $\mathcal{P} \subset \mathcal{N} \times \mathcal{V} \times \mathcal{T}$ as the set of all package releases available in *Debian*. A package release $p \in \mathcal{P}$ is a triple $(p_{\text{name}}, p_{\text{vers}}, p_{\text{time}})$ where $p_{\text{name}} \in \mathcal{N}$ is the package name, $p_{\text{vers}} \in \mathcal{V}$ is the version number of the package release, and $p_{\text{time}} \in \mathcal{T}$ denotes its release date.
For all $p, q \in \mathcal{P}$, we have $(p_{\text{name}} = q_{\text{name}} \wedge p_{\text{vers}} = q_{\text{vers}}) \Longrightarrow p_{\text{time}} = q_{\text{time}}$.

As shown in Section 4, a package release could be affected by vulnerabilities and bugs. We therefore define two functions $\mathsf{vuln}(p) : \mathcal{P} \to \mathbb{N}$ and $\mathsf{bugs}(p) : \mathcal{P} \to \mathbb{N}$, returning respectively the number of reported vulnerabilities and bugs for a given package release p.

5.2 *Debian* distributions

Each *Debian* distribution has its own set of available package releases. Let $\mathcal{D} = \{\mathit{OldStable}, \mathit{Stable}, \mathit{Testing}\}$ be the set of considered *Debian* distributions (see Table 1). For each $d \in \mathcal{D}$ we define $\mathcal{P}_d \subset \mathcal{P}$ as the restriction of all package releases to those available in distribution d. By definition, we have $\mathcal{P} = \bigcup_{d \in \mathcal{D}} \mathcal{P}_d$.

The availability of a package release $p \in \mathcal{P}$ depends on its release date p_{time} and the currently considered date. We define $\mathsf{packages}(d, t) : \mathcal{D} \times \mathcal{T} \to \mathbb{P}(\mathcal{P})$ as a function returning the set of package releases available in distribution d at time t, as follows: $\mathsf{packages}(d, t) = \{p \in \mathcal{P}_d \mid p_{\text{time}} \leq t\}$.

A package release can be installed using the command-line tool `apt` (or one of its variants such as `apt-get` and `aptitude`) that, given the name of a package, installs the highest available version of that package. We found

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

that in 99% of the considered cases in our dataset, the *highest* package release coincides with the *latest* package release.

Let $\mathsf{apt}(n, \mathcal{C}) : \mathcal{N} \times \mathbb{P}(\mathcal{P}) \to \mathcal{P}$ be a function mimicking the behaviour of `apt`. Given a package name n and a set $\mathcal{C}$ of available package releases, $\mathsf{apt}(n, \mathcal{C})$ returns package release $p \in \mathcal{C}$ whose version number is the highest available one for that package. Formally, $\mathsf{apt}(n, \mathcal{C}) = \mathsf{max}_{p_{\text{vers}}}\{p \in \mathcal{C} \mid p_{\text{name}} = n\}$.

5.3 *Debian*-based *Docker* images

Let $\mathcal{I} \subset \mathcal{N} \times \mathcal{D} \times \mathcal{T} \times \mathbb{P}(\mathcal{P})$ be the set of all considered *Debian*-based *Docker* images. $\mathcal{I}$ can be partitioned in two nonempty subsets $\mathcal{I}_{\text{off}}$ and $\mathcal{I}_{\text{com}}$ corresponding to the *official* and *community* images, respectively.

An image $i \in \mathcal{I}$ is a quadruple $(i_{\text{name}}, i_{\text{dist}}, i_{\text{time}}, i_{\text{pkg}})$ where $i_{\text{name}} \in \mathcal{N}$ is the name of the image, $i_{\text{dist}} \in \mathcal{D}$ is the *Debian* distribution used by the image, $i_{\text{time}} \in \mathcal{T}$ is the image's release date, and $i_{\text{pkg}} \subseteq \mathcal{P}$ is the set of package releases installed in the image. We only consider package releases available in the currently considered distribution i_{dist}, i.e., i_{pkg} is restricted to the elements of $\mathsf{packages}(i_{\text{dist}}, i_{\text{time}})$.

5.4 Dimensions of technical lag

For each considered image $i \in \mathcal{I}$, we will compare the set i_{pkg} of installed package releases with the set of package releases with the highest version number at i_{time}. Formally, this corresponds to comparing each installed package release $p \in i_{\text{pkg}}$ with the highest available release $q = \mathsf{apt}(p_{\text{name}}, \mathsf{packages}(i_{\text{dist}}, i_{\text{time}}))$.

For each such pair (p, q) of package releases, we will measure and quantify the difference between them according to five dimensions: package lag, time lag, version lag, vulnerability lag and bug lag. To achieve this, we define a series of functions Δ_α with $\alpha \in \{pkg, time, vers, vuln, bugs\}$ that compute the difference between two package releases. For each of these difference functions, the values should be interpreted as "lower is better".

- $\Delta_{\text{pkg}} : \mathcal{P} \times \mathcal{P} \to \{0, 1\} : (p, q) \to |\{p' \in \{p\} \mid p'_{\text{name}} = q_{\text{name}} \wedge p'_{\text{vers}} < q_{\text{vers}}\}|$

  This Boolean function can be used to compare a package p with its highest available release q. It returns 0 if p is up-to-date, and 1 if p is outdated (i.e., if q has a higher version number than p).

- $\Delta_{\text{time}} : \mathcal{P} \times \mathcal{P} \to \mathbb{Z} : (p, q) \to \mathsf{days}(q_{\text{time}} - p_{\text{time}})$

  This function computes the time difference, in days, between the release dates of two package releases p and q.

- $\Delta_{\text{vers}} : \mathcal{P} \times \mathcal{P} \times \mathbb{P}(\mathcal{P}) \to \mathbb{N}$:
  $(p, q, \mathcal{C}) \to |\{r \in \mathcal{C} \mid (p_{\text{name}} = r_{\text{name}} = q_{\text{name}}) \wedge (p_{\text{vers}} < r_{\text{vers}} \leq q_{\text{vers}})\}|$

  If p and q are two distinct releases of the same package in a given collection $\mathcal{C}$, this function returns the number of package releases that can be found between them (by comparing their version number). If p and q are releases of different packages, the function returns 0.
  If $\mathcal{C}$ is clear from the context, we write $\Delta_{\text{vers}}(p, q)$ as a shortcut for $\Delta_{\text{vers}}(p, q, \mathcal{C})$.

- $\Delta_{\text{vuln}} : \mathcal{P} \times \mathcal{P} \to \mathbb{Z} : (p, q) \to \mathsf{vuln}(p) - \mathsf{vuln}(q)$

  This function computes the difference in number of security vulnerabilities between two package releases.

- $\Delta_{\text{bug}} : \mathcal{P} \times \mathcal{P} \to \mathbb{Z} : (p, q) \to \mathsf{bugs}(p) - \mathsf{bugs}(q)$

  This function computes the difference in number of bugs between two package releases.

5.5 Technical lag framework instances

Using the above auxiliary functions, we instantiate the technical lag framework $\mathcal{F} = (\mathcal{C}, \mathcal{L}, \mathsf{ideal}, \Delta, \mathsf{agg})$ of Section 2.1 according to the five considered dimensions. Each *Docker* image requires its own framework instantiation since different images do not share a same set of available components $\mathcal{C}$.

Therefore, for each $\alpha \in \{pkg, time, vers, vuln, bugs\}$, we define a function $\mathcal{F}_\alpha(i) : \mathcal{I} \to \mathcal{F}$ that, given an image i, returns an instance of the technical lag framework for that image. Concretely, $\mathcal{F}_\alpha(i) = (\mathcal{C}, \mathcal{L}, \mathsf{ideal}, \Delta, \mathsf{agg})$ where:

- $\mathcal{C} = \mathsf{packages}(i_{\text{dist}}, i_{\text{time}})$ is the set of all package releases available at time i_{time} in the corresponding distribution i_{dist}.
- $\mathcal{L} = \mathbb{Z}$, i.e., we only use integers as possible lag values. While the exact interpretation of $\mathcal{L}$ varies depending on α, its values are always understood as "lower is better".
- $\mathsf{ideal}(p) = \mathsf{apt}(p_{\text{name}}, \mathcal{C})$, i.e., the ideal release of a package is selected by `apt` among the ones that are available for the current distribution i_{dist} at i_{time}.
- $\Delta(p, q) = \Delta_\alpha(p, q)$ as defined in Section 5.4.
- $\mathsf{agg}(X) = \mathsf{max}(X)$ if $\alpha = time$;
  $\mathsf{agg}(X) = \Sigma X$ otherwise.

Finally, for each $\alpha \in \{pkg, time, vers, vuln, bug\}$, we define $\alpha\text{-}\mathsf{lag} : \mathcal{I} \to \mathbb{Z}$, a function that, given a *Docker* image i, instantiates a technical lag framework instance $\mathcal{F}_\alpha(i)$ and returns the aggregated lag of all package releases currently installed in the given image with respect to the highest available package

releases, as follows:

$$\alpha\text{-}\mathsf{lag}(i) = \mathsf{agg}(\{\Delta(p, \mathsf{ideal}(p)) \mid p \in i_{\text{pkg}}\})$$

These five α-lag functions, whose interpretation is summarized in Table 3, will be used in Section 6 to empirically analyse the technical lag of *Debian*-based *Docker* images.

Table 3 Summary of the five technical lag instantiations for a given *Docker* image i.

framework instantiation	Δ *lag at package release level*	agg *lag at image level*
pkg-lag(i)	1 if outdated, 0 otherwise	# outdated packages
time-lag(i)	release time difference	maximum difference
vers-lag(i)	# missed versions	sum of versions
vuln-lag(i)	difference in # vulnerabilities	sum of vulnerabilities
bug-lag(i)	difference in # bugs	sum of bugs

5.6 Example

Let us illustrate the definitions of the previous subsections using a fictitious example. We assume a *Debian*-based *Docker* image i = (example, *Stable*, 2019-05-08, i_{pkg}). Table 4 shows all package releases of i_{pkg}. The two last columns of this table correspond to the number of reported vulnerabilities and bugs, respectively.

Table 4 Package releases in i_{pkg}.

p_{name}	p_{vers}	p_{time}	vuln(p)	bugs(p)
curl	7.52.1-5	2017-04-30	19	38
base-files	9.9+deb9u3	2017-12-10	0	6
acl	2.2.52-3	2017-02-25	0	0

The technical lag framework instances quantify, along five dimensions, the outdatedness of package releases in i_{pkg} and, by extension, in image i. Let $\mathcal{F}_\alpha(i)$ be the technical lag framework instances for $\alpha \in \{pkg, time, vers, vuln, bugs\}$, as defined in Section 5.5. All these instances share a same single ideal function that, for each package release $p \in i_{\text{pkg}}$, selects a package release from the set packages(*Stable*, 2019-05-08) of available ones. Such *ideal* releases are selected following the behaviour of `apt`, the default package manager of *Debian*. They correspond to the highest available version in the currently considered distribution (*Stable* in this example) to quantify the lag of installed packages.

Table 5 shows the highest available releases for the three considered packages (i.e., it shows the subset of packages(*Stable*, 2019-05-08) selected by ideal). We observe that the versions of two of the three considered package releases in

Table 5 Subset of packages(*Stable*, 2019-05-08), restricted to the highest available releases for the three considered packages of i_{pkg}.

p_{name}	p_{vers}	p_{time}	vuln(p)	bugs(p)
curl	7.52.1-5+deb9u9	2019-02-07	1	41
base-files	9.9+deb9u9	2019-04-28	0	7
acl	2.2.52-3	2017-02-25	0	0

i_{pkg} are not the highest ones among package releases available at i_{time}, namely curl@7.52.1-5+deb9u9 and base-files@9.9+deb9u9. On the other hand, package release acl@2.2.52-3 is already installed in image i and is therefore up-to-date and will not induce any lag.

For each package release, we can compute its package, time, version, vulnerability and bug lag, by means of difference functions Δ_{pkg}, Δ_{time}, Δ_{vers}, Δ_{vuln} and Δ_{bug} respectively, as explained in Section 5.4. The technical lag values are obtained by comparing each $p \in i_{\text{pkg}}$ to its *ideal* release ideal(p).

Table 6 Package, time, version, vulnerability and bug lag for package releases in i_{pkg}.

package name	Δ_{pkg}	Δ_{time}	Δ_{vers}	Δ_{vuln}	Δ_{bug}
curl	1	648	9	18	-3
base-files	1	504	6	0	-1
acl	0	0	0	0	0

Table 6 reports on these lags for the three considered packages. For example, we observe that the installed package release of base-files in i is outdated since $\Delta_{\text{pkg}}(p) = 1$. It is 6 versions behind its *ideal*. It has a time lag of 504 days. It has the same number of vulnerabilities as the ideal ($\Delta_{\text{vuln}}(p) = 0$), but suffers from 1 bug less ($\Delta_{\text{bug}}(p) = -1$). As expected, the lag values for acl are zero, as this package release is up-to-date in i.

Since these values represent the technical lag at the level of individual package releases, they need to be aggregated to compute the lag for the entire image i. Using the aggregations summarised in Table 3 we find that this image has pkg-lag(i) = 2 outdated packages, time-lag(i) = 648 days, its packages missed vers-lag(i) = $9 + 6 = 15$ versions, and suffers from vuln-lag(i) = 18 more vulnerabilities. On the other hand, they are affected by 4 bugs less (bug-lag(i) = $(-3) + (-1) = -4$).

6 Multi-Dimensional Analysis of Technical Lag for *Debian* images

In this section, we empirically analyze and compare the technical lag incurred by 9,330 official and 131,168 community images on *Docker Hub* that are based on the *Debian* distribution. We do this for five dimensions of technical lag: package lag, time lag, version lag, vulnerability lag and bug lag. These dimensions correspond to the instantiations of the technical lag framework that

were defined in Section 5.5. The five research questions that we address are therefore:

- RQ_1: How does *package lag* evolve in *Debian*-based *Docker* images? The answer to this question will show how outdated *Docker* images are, considering how many of their included packages could have been updated to newer versions.
- RQ_2: How does *time lag* evolve in *Debian*-based *Docker* images? The answer to this question adds detail to the previous one, quantifying how old (compared to available versions) are packages installed in *Debian* images.
- RQ_3: How does *version lag* evolve in *Debian*-based *Docker* images? This question complements the previous questions with another dimension, focusing on the number of releases in the interim and giving insights on how different the installed and available package versions are.
- RQ_4: How does the *number of vulnerabilities* and the *vulnerability lag* evolve in *Debian*-based *Docker* images? Since security vulnerabilities are of special interest to people deploying containers in production, this question focuses on evaluating how up-to-date containers are from this security point of view.
- RQ_5: How does the *number of bugs* and the *bug lag* evolve in *Debian*-based *Docker* images? With this question, we provide a metric that is useful for deployers interested in having as many bugs fixed as possible in their production systems.

The remainder of this section provides empirical evidence for each research question. All code and data required to reproduce the analysis in this article are available in a replication package (Zerouali, 2020).

As part of the empirical analysis, we carried out comparisons of statistical distributions using the *Mann-Whitney U* test, a non-parametric test where the null hypothesis H_0 checks if two distributions are identical without assuming them to follow a normal distribution, the alternative hypothesis H_1 being that one distribution is stochastically greater than the other. For all statistical tests in the paper considered together, we wish to achieve a global confidence level of 99%, corresponding to a value of $\alpha = 0.01$. To achieve this overall confidence, the p-value of each individual test is compared against a lower α value, following a Bonferroni correction[22]. To report the *effect size* of the statistical tests, we use *Cliff's Delta d*, a non-parametric measure that quantifies the difference between two populations beyond the interpretation of p-values. Using the thresholds provided in Romano et al. (2006), we interpret the effect size to be *negligible* if $|d| \in [0, 0.147[$, *small* if $|d| \in [0.147, 0.33[$, *medium* if $|d| \in [0.33, 0.474[$ and *large* if $|d| \in [0.474, 1]$.

[22] If n different tests are carried out over the same dataset, for each individual test one can only reject H_0 if $p < \frac{0.01}{n}$. In our case $n = 28$, i.e., $p < 0.00036$.

RQ_1: How does *package lag* evolve in *Debian*-based *Docker* images?

This research question investigates how many package releases were outdated in *Debian*-based *Docker* images at the time of building the images, and how this evolved over time. Figure 6 shows the quarterly evolution of the distribution of pkg-lag(i) for images i available in *official* and *community* repositories. We observe that the number of outdated package releases does not tend to change much over time: when new images are released they do not seem to include more or less outdated package releases than older images.

Considering the whole observation period, we found that at least half of all *official* images have no package lag, while *community* images have a median lag of 7 outdated packages. Using the Mann-Whitney U test, we observe a statistically significant difference with large effect size ($|d| = 0.50$), i.e., *community* images have more package lag than *official* images. This is reasonable since *official* images serve as the starting point for the *community* images, and therefore are expected to be well maintained. For deployers, this means that using the latest *official Debian* images when they are first released and using packages directly from the *Debian* repository is quite similar in terms of outdated packages.

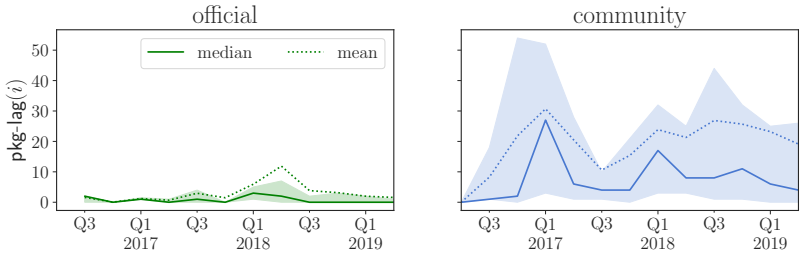

Fig. 6 Evolution of the distribution of pkg-lag(i) for images $i \in \mathcal{I}_{\text{off}} \cup \mathcal{I}_{\text{com}}$ (*official* and *community*).

We also investigated whether the package lag depends on the image's distribution. Table 7 summarises the statistical results of the comparison between images with different *Debian* distributions. Using Mann-Whitney U tests we reject H_0, and conclude that *OldStable* images have more outdated packages than *Stable* images, while *Testing* images have more outdated packages than *Stable* images. In both cases, the effect size is medium. Even though for *OldStable* versus *Testing* images H_0 is rejected as well, the effect size is negligible so we cannot draw any conclusions.

Figure 7 shows the quarterly evolution of pkg-lag grouped by *Debian* distribution. We observe that the package lag of *Debian OldStable* images is increasing over time. Given that the lag of *official* images is generally very low (see Figure 6), this increasing package lag for *OldStable* is mainly due to

Table 7 Statistically significant Mann-Whitney U tests and their corresponding effect sizes when comparing pkg-lag for images in different *Debian* distributions.

population A	direction	population B	effect size	|d|
OldStable	$>$	***Stable***	**medium**	**0.37**
OldStable	$<$	*Testing*	negligible	0.09
Stable	$<$	***Testing***	**medium**	**0.44**

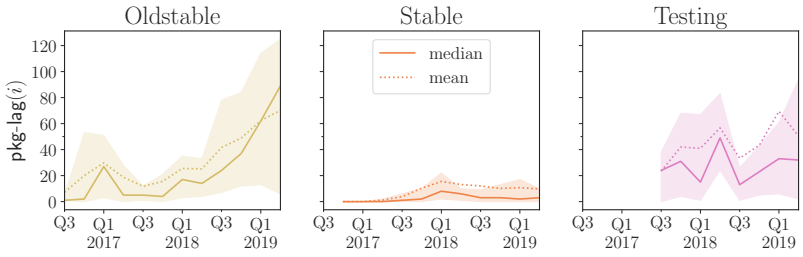

Fig. 7 Evolution of the distribution of pkg-lag(i) for images $i \in \mathcal{I}$, grouped by i_{dist} (*OldStable*, *Stable* or *Testing*).

community images[23]. We expected such result since deployers of *OldStable* images are less concerned about up-to-date packages in their images, and hence do not take measures to obtain the latest available packages when images are built.

Figure 7 also reveals a higher number of outdated packages for *Testing* images than for *Stable* images. This can be expected since packages in *Testing* images are, by their very nature, updated regularly with changes, new features and bug fixes. This updating process brings frequent package updates into the *Testing* repository making it difficult for *Docker* images to keep using the latest available package releases. In contrast, packages in *Stable* and *OldStable* distributions tend to be updated only with security patches. In addition, it is well known that *Debian Testing* should not be used for production deployments, which means that its images are in many cases used for experiments, or for environments where usual production requirements are not applicable. This further corroborates that producers of *community* images based on *Testing* are less likely to care about updating them regularly.

In Figure 6, a spike can be observed for *community* images during 2017-Q1. Figure 7 reveals that this spike is caused by *OldStable* images. Since this spike is not observed for *official* images, we think that it is caused by packages that are not provided by the *official Debian* images but added by *community* developers. We investigated the reason for this spike, and found that is was caused by two local events. The first occurred in November 2016, two months after

[23] Extra analysis and results, distinguishing the evolution trends both for *official* and *community* images, can be found in our reproduction package.

the release of *Debian OldStable* 8.6[24] and its corresponding *Docker* image[25], and one month before the last update of the latter. The second event occurred in March 2017, two months after the release of *Debian OldStable* 8.7[26] and its corresponding *Docker* image[27], and one month before the last update of the latter. The three-months period between the *Debian* release and the last update of its *Docker* image led to the outdatedness of many used packages since many of them were being updated in the *Debian* repositories during this period, explaining the spike in package lag observed in Figure 6. These events illustrate how our method captures cases when packages in *Docker* images are not updated quickly after having new available releases, leading to outdated images.

We also observed a smaller peak in Figure 6 during 2018-Q1, for both *official* and *community* images. This peak is visible as well for *OldStable* and *Stable* images in Figure 7. The peak is again caused by the newly updated packages in the latest downstream *Debian* releases (i.e., *Debian OldStable* 8.10 and *Debian Stable* 9.3 in December 2017[28] and *Debian Stable* 9.4 in March 2018[29]). The peak is smaller than what we observe in 2017-Q1 because the *Debian* official *Docker Hub* repository increased its release frequency of images since June 2017, the release date of the new *Stable* release Stretch. This change was reported in a blog by de Visser (2017), explaining that *Debian* images are updated every month, and that in a few days many *official* repositories pushed new updated images at the same time. In many cases this occurred the day after the base image debian:latest was updated, corroborating the low package lag that we found for *official* images.

We also computed the *proportion* of outdated packages in the considered images and we found very similar evolution patterns as for pkg-lag. Over the whole observation period, the median proportion of outdated packages in *official* and *community* images was very low: 0% and 2.4%, respectively. This means that *Debian* images come with the highest and latest available package releases when they are first created. The proportion of outdated packages can become more important over time when images are not updated anymore or when they do not contain any automatic update command (e.g., `apt upgrade -y`) in their Dockerfiles.

[24] https://www.debian.org/News/2016/20160917

[25] https://github.com/docker-library/official-images/commit/a0884f0cd8758a0a30cf187f25ef217e3915979f

[26] https://www.debian.org/News/2017/20170114

[27] https://github.com/docker-library/official-images/commit/fbbcd34e82dcea6e75f5a5ea465d49912d996261

[28] https://www.debian.org/News/2017/index.en.html

[29] https://www.debian.org/News/2018/20180310

Summary: While the majority of *official Debian*-based images do not include outdated packages because they are updated frequently, *community* images have a median package lag of 7 outdated packages per image. This represents a very low proportion ($< 3\%$) of outdated packages per image. *Testing* images have higher package lag compared to other images, because *Testing* packages are frequently updated in the *Debian* repository. *OldStable* images have an increasing package lag over time because their deployers are not concerned about having up-to-date packages. In general, using *official Debian* images when they are first released as base images is quite similar to directly using packages from the *Debian* repository, in terms of package outdatedness.

RQ_2: How does *time lag* evolve in *Debian*-based *Docker* images?

This research question investigates the time lag of *Debian*-based *Docker* images and how this evolves over time. Figure 8 shows the quarterly evolution of the distribution of time-lag(i) for images i in *official* and *community* repositories. We observe that time lag tends to fluctuate over time for both origins, so there is no clear relation between an image's creation date and its time lag.

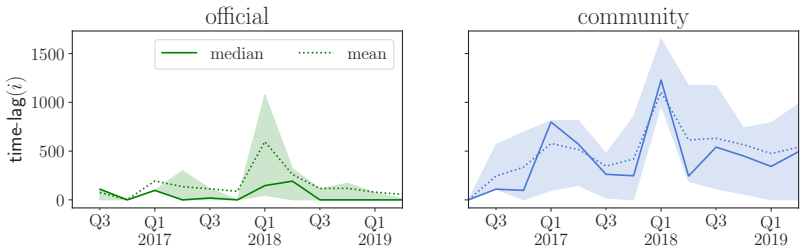

Fig. 8 Evolution of the distribution of time-lag(i) for images $i \in \mathcal{I}_{\text{off}} \cup \mathcal{I}_{\text{com}}$ (*official* or *community*).

We also observe that *community* images can have a high median value (even exceeding 1,000 days in Q1 of 2018), while *official* images tend to have a low median time lag that never exceeds 192 days and is even much lower most of the time. When we compute the median time lag over the whole observation period, we find that at least 50% of all *official* images have no time lag, while the median time lag for *community* images is well over a year (466 days, to be precise). Using the Mann-Whitney U test, we found a statistically significant difference between images from different origins with large effect size ($|d| = 0.50$). This implies that *community* images have a larger time lag than *official* images. This corroborates our observations in RQ_1 about the difference between outdated packages in *official* and *community* images.

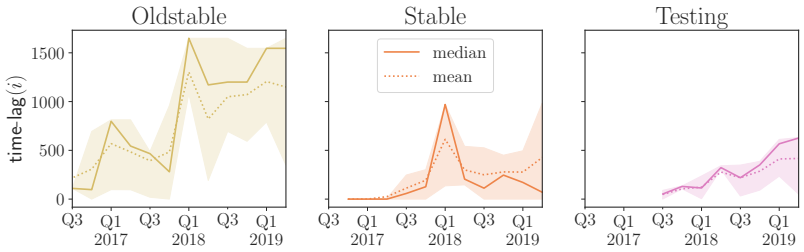

Fig. 9 Evolution of the distribution of time-lag(i) for images $i \in \mathcal{I}_{\text{off}} \cup \mathcal{I}_{\text{com}}$, grouped by i_{dist} (*OldStable*, *Stable* or *Testing*).

Figure 9 shows the quarterly evolution of time-lag grouped by *Debian* distribution. We observe that time lag is increasing over time for *OldStable* and *Testing* distributions. We investigated more about this phenomenon in our data and we found that this increase is caused by *community* images only; *official* images do not have an increasing time lag, regardless of their *Debian* distribution. While this observation is expected in the case of *OldStable*, because package versions are old and have had more time to accumulate time lag, the case of *Testing*, where packages are recent, is different. It means that people building *community* images based on *Testing* are not updating their image packages as they are having frequent updates in the *Debian* repository, and as new *official* images are available. This is consistent with the explanation that *Testing* deployers are less concerned about up-to-date packages because *Testing* is seldom used in production, and should be more popular in cases where production requirements (e.g., less build problems, packages being up-to-date, well tested updates) are not applicable.

The previous finding that deployers can rely on *official* images when they are first released almost as much as in updating directly from the *Debian* repositories also holds, since most of these images have a very small time lag.

We performed Mann-Whitney U tests to compare the time lag of images in function of their *Debian* distribution. Table 8 reports on the effect sizes of the statistically significant tests. We conclude that *OldStable* images have a higher time lag than *Stable* images, with large effect size. Similarly, *OldStable* images have a higher time lag than *Testing* images, with medium effect size. We also found evidence of a lower time lag for *Stable* images than for *Testing* images, but the effect size was negligible, so we cannot draw any conclusions about this.

Table 8 Statistically significant Mann-Whitney U tests and their corresponding effect sizes when comparing time-lag for images in different *Debian* distributions.

population A	direction	population B	effect size	|d|
OldStable	$>$	***Stable***	**large**	**0.50**
OldStable	$>$	***Testing***	**medium**	**0.47**
Stable	$<$	*Testing*	negligible	0.10

In Figure 8 and Figure 9 we can observe spikes of increased time lag during 2018-Q1. This is a consequence of the availability of new, not yet adopted releases for packages whose previous version was released many years before. For example, sensible-utils, a package that was present in 44.6% of all images at that time, released a new version on 2017-12-22. Since the latest previous version of that package was released on 2013-06-17, it induced a time lag of at least four years and a half (1,649 days).

Summary: While *official Debian*-based images are mostly up-to-date in terms of time lag, the median time lag of *community* images is well over a year, and highest for *OldStable* images. Moreover, for *OldStable* and *Testing* images this time lag tends to increase over time. Since *official* images are quite close, in terms of time lag, to updating from the *Debian* repository, there is little reason for builders of *Debian*-based images to maintain their own directly from the *Debian* repositories, instead of relying on the *official Docker Hub Debian* images.

RQ_3: How does *version lag* evolve in *Debian*-based *Docker* images?

The time lag, presented in RQ_2, provided a good first estimation of how outdated a *Docker* image is. However, this measurement is not sufficiently precise to evaluate the underlying source of the lag. For example, different *Debian* package releases contained in the same image could have exactly the same time lag, while one of them has received a large number of frequent updates, whereas the other one has received only very few updates. As another example, at the image level, some images may contain many installed packages, while others have only a few ones. As explained in Section 5, version lag allows to capture this, by counting the number of missed versions at package release level, and by computing an aggregated sum at image level (see Table 3).

Figure 10 shows the quarterly evolution of the distribution of $\mathsf{vers\text{-}lag}(i)$ for images i in *official* and *community* repositories. We observe that *official* images have a lower version lag than *community* images. Considering the whole observation period, we found that at least 50% of all *official* images do not have any missed versions, while *community* images have a median version lag of 7 missed versions. Using the *Mann-Whitney U* test, we found a statistically significant difference between *official* and *community* images with large effect size ($|d| = 0.51$), i.e., *community* images have a higher version lag than *official* images.

We observe in Figure 10 for *community* images a large peak in 2017-Q1 and a smaller one in 2018-Q1 for the same reasons as previously identified for package lag in RQ_1. Indeed, as a corollary of the technical lag definitions in Section 5, it holds that $\mathsf{pkg\text{-}lag}(i) = x \Rightarrow \mathsf{vers\text{-}lag}(i) \geq x$. In fact, we found that 66.7% of the images have their package lag equal to their version lag. This means that roughly two thirds of the images that have outdated packages are outdated by a single version only, which means they are really close to being up-to-date.

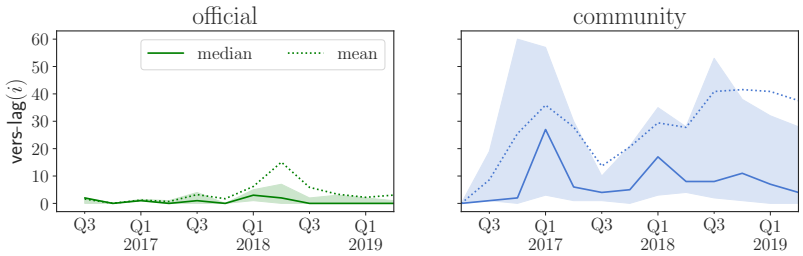

Fig. 10 Evolution of the distribution of vers-lag(i) of considered images $i \in \mathcal{I}_{\text{off}} \cup \mathcal{I}_{\text{com}}$ (*official* or *community*).

We also studied the evolution of version lag of *Docker* images, grouped by *Debian* distribution. Figure 11 reveals an increasing version lag over time for *OldStable* and *Testing* images, similar to what we observed for package lag, and for the same reasons, we also observe that version lag is higher for *Testing* images, followed by *OldStable* images, and finally *Stable* images. Table 9 summarises the statistical results of the Mann-Whitney U tests. We could reject H_0 for all comparisons, and conclude that *Testing* images have higher version lag than *Stable* images with large effect size, whereas *Testing* images have a higher version lag than *OldStable* images with small effect size. *OldStable* images have a higher version lag than *Stable* images with medium effect size.

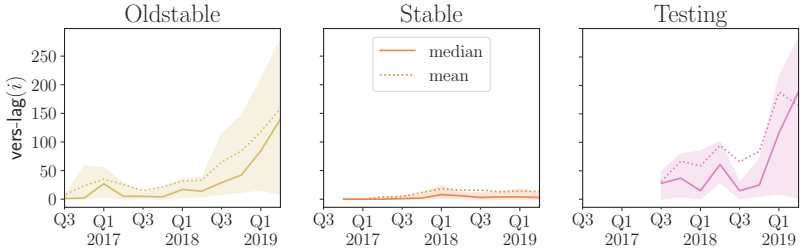

Fig. 11 Evolution of the distribution of vers-lag(i) for images $i \in \mathcal{I}$, grouped by i_{dist} (*OldStable*, *Stable* or *Testing*).

Table 9 Statistically significant Mann-Whitney U tests and their corresponding effect sizes when comparing vers-lag for images in different *Debian* distributions.

population A	direction	population B	effect size	|d|
OldStable	$>$	***Stable***	**medium**	**0.37**
OldStable	$<$	*Testing*	small	0.20
Stable	$<$	***Testing***	**large**	**0.48**

The above observations for version lag differ from the observed findings for time lag in RQ_2. More specifically, *OldStable* images have a lower version lag than *Testing* images, while they had a higher time lag; and *Testing* images have a higher version lag than *Stable* images, while such an effect was negligible for time lag. This observed difference between time lag and version lag can be explained with two arguments. First, package releases for *OldStable* tend to be older than for *Stable*, thus they will have higher version lag. Second, package releases for *Testing* are being updated on a daily basis, and many new packages versions are created in a short period of time. As a result, the version lag of outdated package releases (multiple successive versions) tends to increase while the time lag increases at a lower rate (short time span between successive versions). This explains why *Testing* images have a lower time lag but a higher version lag than *OldStable* images, illustrating the value of having different measures of technical lag.

Summary: While the majority of *official Debian*-based images are up-to-date in terms of version lag, the median version lag of *community* images is 7 missed versions. *Testing* images have a higher version lag than *Stable* and *OldStable* images because they contain packages that are being updated on a daily basis. *OldStable* and *Testing* images have an increasing version lag over time, for the same reasons as identified for package and time lag.

RQ_4: How does the *number of vulnerabilities* and the *vulnerability lag* evolve in *Debian*-based *Docker* images?

While time lag and version lag reflect to which extent an image or a package release is outdated, they do not reveal anything about the security risks that such outdatedness may incur. Related work (see Section 3.2) has revealed that outdated software packages or containers often have an increased number of security vulnerabilities. Reducing such security vulnerabilities in software containers is important, since those vulnerabilities could be exploited to abuse the system.

RQ_4 aims to study the relation between outdatedness and vulnerabilities in *Debian*-based *Docker* images, by computing their vulnerability lag using the definitions in Section 5 (i.e., the difference in number of vulnerabilities between the current and the ideal package release, and its aggregated sum at image level). To identify the vulnerabilities that are affecting the source packages installed in *Docker* images, we rely on a dataset of security vulnerabilities available in the *Debian Security Tracker* on 30 August 2019.

Number of vulnerabilities. Before actually analysing the vulnerability lag, we report on the *number of vulnerabilities* over time. Table 10 reports the characteristics for the distributions of the number of vulnerabilities in *Debian*-based images. We observe that *all* images suffer from vulnerabilities. While *official* images have a median of 315 vulnerabilities, *community* images have

a higher median value of 567 vulnerabilities. This is expected, since the preliminary analysis revealed that *community* images tend to have more installed packages than *official* ones (see Figure 3). As such, they are more subject to security vulnerabilities. A Mann-Whitney U test confirmed, with medium effect size ($|d| = 0.41$), that the population of the number of vulnerabilities for *official* images was lower than for *community* images.

Table 10 Characteristics of the distributions of number of vulnerabilities, grouped by *Debian* distribution and image origin.

	official			***community***		
	mean	**median**	**max**	**mean**	**median**	**max**
OldStable	482.4	519	1,424	743.3	796	1,858
Stable	345.4	308	1,480	485	501	2,157
Testing	133.1	144	658	211.6	174	1,055
all	373.8	315	1,480	581.9	567	2,157

Figure 12 shows the quarterly evolution of the distribution of the *number of vulnerabilities* for images in *official* and *community* repositories. We observe that older images have more known vulnerabilities than recent images, since the median and mean values are decreasing over time. Since older packages have been used for longer, and thus had more time to have their vulnerabilities found and reported, it is not surprising that older images composed of such older packages have more vulnerabilities. We also observe an inverse peak for *official* images during 2017-Q2. Investigating this phenomenon, we found that this decrease in the number of vulnerabilities coincides with the release of the new *Stable* release *Stretch* in June 2017. Since new images were created with packages from this new *Stable* version, and since these packages have fewer known vulnerabilities, we have a lower median number of vulnerabilities for all images during that period.

For the inverse peak for *community* images during 2016-Q3, we believe that it is caused by the lower number of installed packages that *Docker community* images had at this period (see Figure 3). Indeed, lower number of installed packages will lead to lower number of vulnerabilities and bugs present in *Docker* images.

Using a Mann-Whitney U test we compared images belonging to different *Debian* distributions. Table 11 summarises the results. We observe that *OldStable* images tend to have a higher number of vulnerabilities than other images, and *Testing* images tend to have less vulnerabilities.

We also found that all images suffer from vulnerabilities, even more popular *official* images like *gcc, node and pypi*, etc. Table 12 shows the repositories of the top 5 vulnerable *official* and *community* images with their number of vulnerabilities, number of installed packages and age (in months). For both image origins, only the top vulnerable image in a repository is presented. For example, the *node* repository has many images with a high number of vulnerabilities. However, since these images provide the same functionality, we report only one: the most vulnerable image. A common characteristic among

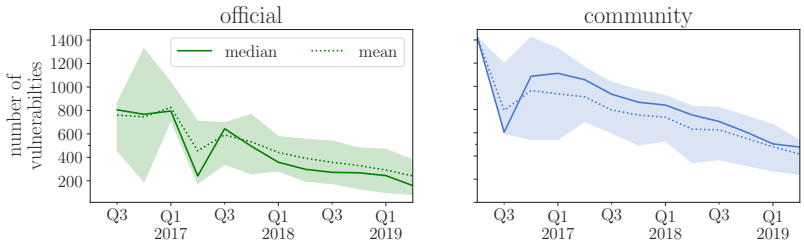

Fig. 12 Evolution of the distribution of *number of vulnerabilities* of considered images $i \in \mathcal{I}_{\text{off}} \cup \mathcal{I}_{\text{com}}$ (*official* or *community*).

Table 11 Statistically significant Mann-Whitney U tests and their corresponding effect sizes when comparing the *number of vulnerabilities* in images for different *Debian* distributions.

population A	direction	population B	effect size	|d|
OldStable	$>$	***Stable***	**large**	**0.50**
OldStable	$>$	***Testing***	**large**	**0.84**
Stable	$>$	***Testing***	**large**	**0.68**

these top vulnerable images 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 images. Thus, it is not surprising to see a high number of vulnerabilities for these images.

Table 12 Source repositories of the top 5 vulnerable *official* and *community* images.

***official* repositories**	**# vuln.**	**# pkg.**	**age** (in months)
gcc	1,480	404	27
erlang	1,424	377	37
node	1,423	375	37
pypy	1,399	354	38
rails	1,333	386	33
***community* repositories**			
surround/ws-master	2,157	1,257	28
rocker/ropensci	2,060	790	25
smartpension/sp-base	1,981	710	26
aldryn/base-project	1,858	563	31
micheee/maven-jdk-nodejs-bower-grunt	1,791	530	37

Vulnerability lag. Having shown that *Docker Hub* images suffer from vulnerabilities, we now focus on their *vulnerability lag* to quantify how vulnerable image package releases are compared to their latest available releases. This is important, because when package versions with less vulnerabilities are available, the decision to upgrade is a priori easy to take. This means that having images with a positive vulnerability lag is due to either to the unawareness of

these vulnerabilities, or to some other practical reason (no interest in minimizing vulnerabilities, incompabilities, balance with other requirements, time pressure, etc.). Considering the whole observation period, we found that at least 50% of all *official* images have no vulnerability lag, while *community* images have a median lag of 10 vulnerabilities. The observation that *community* images tend to have higher vulnerability lag than *official* ones is confirmed by a Mann-Whitney U test with large effect size ($|d| = 0.5$). Figure 13 shows the quarterly evolution of the distribution of vuln-lag(i) of images $i \in \mathcal{I}_{\text{off}} \cup \mathcal{I}_{\text{com}}$ grouped by image origin (*official* or *community*). We do not observe any increasing or decreasing trend.

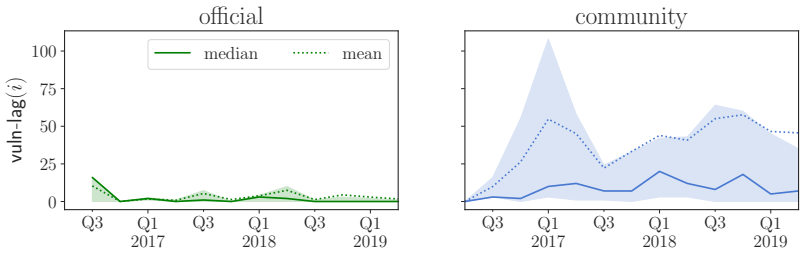

Fig. 13 Evolution of the distribution of vuln-lag(i) of considered images $i \in \mathcal{I}_{\text{off}} \cup \mathcal{I}_{\text{com}}$ (*official* or *community*).

Similarly, Figure 14 shows the quarterly evolution of the distribution of the vuln-lag(i) for all images $i \in \mathcal{I}$ grouped by *Debian* distribution i_{dist}. Over the entire observation period, the median vuln-lag was 2, 4 and 21 vulnerabilities for *Testing*, *Stable* and *OldStable*, respectively. Table 13 reports the results of the statistical comparisons. *OldStable* images tend to have a higher vulnerability lag than *Stable* and *Testing* images (medium effect size). The difference between *Stable* and *Testing* images was negligible.

Comparing vulnerability lag with the previous lag dimensions, we observe that its evolution over time in the case of *OldStable* and *Stable* images is similar to the evolution of version lag in the same subset of images. This is related to the nature of *OldStable* and *Stable* distributions; in most of the cases new package updates in these distributions are about important bug and vulnerability fixes (as explained in Section 4). Therefore having a higher version lag in images with these distributions implies having a higher vulnerability lag. It is also noticeable how images based on *Stable* and *Testing* have a similar, very low, vulnerability lag. This suggests that vulnerabilities are taken seriously both in *Stable* and *Testing* images, causing new images to be produced with enough frequency to keep this lag low. The fact that *Stable* is managed this way is no surprise, but the case of *Testing* is different: other lag dimensions discussed up till now were remarkably higher for *Testing*. This says a lot about the perceived importance of vulnerabilities, even for non-stable releases such as *Testing*.

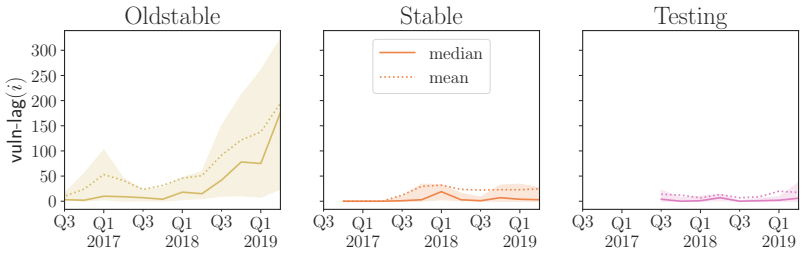

Fig. 14 Evolution of the distribution of vuln-lag(i) for images $i \in \mathcal{I}$, grouped by i_{dist} (*OldStable*, *Stable* or *Testing*).

Table 13 Statistically significant Mann-Whitney U tests and their corresponding effect sizes when comparing vuln-lag for images in different *Debian* distributions.

population A	direction	population B	effect size	|d|
OldStable	>	***Stable***	**medium**	**0.32**
OldStable	>	***Testing***	**medium**	**0.42**
Stable	>	*Testing*	negligible	0.09

Summary: Both *official* and *community Debian*-based images suffer from a high *number of vulnerabilities* (median of 556). However, the majority of *official* images do not suffer from *vulnerability lag*, whereas *community* images have a median vulnerability lag of 10 vulnerabilities. *OldStable* images have more vulnerabilities and a higher vulnerability lag than other images, and this lag is increasing over time as a consequence of their increasing version lag.

RQ_5: How does the *number of bugs* and the *bug lag* evolve in *Debian*-based *Docker* images?

Just like security vulnerabilities, software bugs can be very problematic in software containers. Bugs make a software system behave in unexpected ways, resulting in faults, wrong functionality or reduced performance. Research question RQ_5 therefore aims to study the presence of known bugs, still open in the installed versions of packages, in *Docker* images, and to which extent such bugs could be reduced if images would make use of the highest available releases of the installed packages. Similar to RQ_4, to identify the bugs that are affecting the source packages installed in *Docker* images, we relied on a dataset of bug reports available in the *Ultimate Debian Database* on 30 August 2019.

Number of bugs. Before actually analysing the bug lag, we study the *number of bugs* over time. Table 14 reports the characteristics for the distribution of the number of bugs in *Debian*-based images. We observe that *all* images suffer from a high number of bugs. While *official* images have a median of 5,127 bugs, *community* images tend to have more with a median value of 6,700, the

large majority of them being inherited from the *official* images on which they are based. A Mann-Whitney U test reveals with medium effect size ($|d| = 0.47$) that *community* images have more bugs than *official* images. This difference can be explained by the higher number of packages contained in *community* images.

Table 14 Characteristics of the distributions of number of bugs, grouped by *Debian* distribution and image origin.

	official			***community***		
	mean	**median**	**max**	**mean**	**median**	**max**
OldStable	5,140.4	5,126	7,857	6,490.7	6,916	11,523
Stable	5,216.9	5,173	8,819	6,300.3	6,111	14,269
Testing	4,372.1	4,894	7,323	6,269.2	6,342	10,653
all	5,124.5	5,127	8,819	6,375.8	6,700	14,269

To verify if the number of bugs depends on the *Debian* distribution, we grouped and compared images by distribution. Table 15 summarises the statistical results. While H_0 was rejected in all cases, the effect size was small to negligible. This small difference is in favour of images with older *Debian* distributions.

Table 15 Statistically significant Mann-Whitney U tests and their corresponding effect sizes when comparing the *number of bugs* in images for different *Debian* distributions.

population A	**direction**	**population B**	**effect size**	**|d|**
OldStable	>	*Stable*	negligible	0.10
OldStable	>	*Testing*	small	0.15
Stable	>	*Testing*	negligible	0.10

Because the number of bugs per image is high, we hypothesise that most of these bugs are of lesser importance to the *Debian* community, and tend to stay open for a long time. This is in line with the fact that 55.5% of the bugs affecting *Docker* images are still open without fixes (see Section 4). Since *Debian* provides six different categories of bug urgency, and few bugs are classified in four of them (see Figure 5), we grouped all bugs into two combined categories of bug urgency: *Low* (67% of all bugs) including all *minor and normal* bugs, and *High* (33% of all bugs) including all *important, serious, grave* and *critical* bugs. Comparing both categories in *Docker* images, we observed that the median number of reported bugs of *Low* urgency (4,815) is about three times higher than the median number of reported bugs of *High* urgency (1,672), thus validating our hypothesis.

Figure 15 shows the quarterly evolution of the distribution of the number of bugs for images in *official* and *community* repositories. Unlike the increasing trend we witnessed for the number of vulnerabilities, the number of bugs affecting the images appears to be quite stable over time. This is mainly because of two reasons: (1) the high number of open bugs, and (2) new package

versions come with some bug fixes but later new bugs are found in them. As was the case for the number of vulnerabilities and for the same reasons, we also observe an inverse peak in number of bugs for *official* images during 2017-Q2.

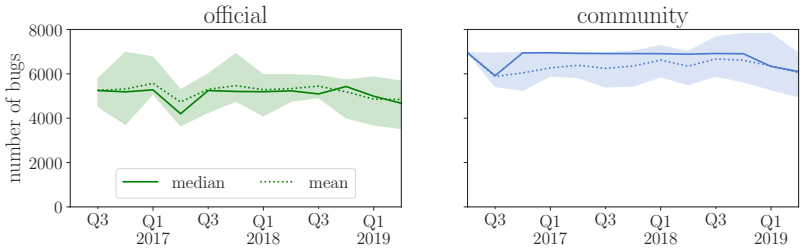

Fig. 15 Evolution of the distribution of *number of bugs* of considered images $i \in \mathcal{I}_{\text{off}} \cup \mathcal{I}_{\text{com}}$ (*official* or *community*).

Comparing Figure 15 with Figure 12, we observe a much higher number of bugs than number of vulnerabilities in *Docker* images. Table 14 confirms these observations by showing higher values for the distributions of the number of bugs in *Docker* images. Moreover, we found that there are two package versions (from *mawk* and *attr* packages) that are installed in nearly all of the images despite having a number of both high and low urgency bugs. This was not observed in the case of vulnerabilities. This difference shows how vulnerabilities are dealt with compared to "regular" bugs, which seem to be tolerated to some extent. Table 16 provides more information about the most prevalent buggy package versions.

Table 16 Statistics about the top 10 buggy source package versions that are affecting most of the images, with their proportion of affected images and the number of high and low urgency bugs they have.

package	version	# images	# high bugs	# low bugs
mawk	1.3.3-17	99.9%	6	21
attr	1:2.4.47-2	99.3%	3	13
libxau	1:1.0.8-1	80.2%	0	1
jbigkit	2.1-3.1	71.5%	1	0
acl	2.2.52-3	59.7%	1	6
bzip2	1.0.6-8.1	58.3%	1	16
gzip	1.6-5	58.2%	4	19
zlib	1:1.2.8.dfsg-5	58.0%	1	5
lz4	0.0∼r131-2	57.9%	2	2
pam	1.1.8-3.6	57.8%	14	62

Bug lag. Having shown that *Debian* images suffer from many bugs, we focus on their *bug lag* to quantify how buggy image package releases are compared to their highest available releases. Considering the whole observation period

and without distinguishing *Debian* images by their origin or distribution, we found that at least 50% of all images do not have a bug lag. This corroborates our previous finding showing that the number of bugs in images is stable over time. Based on a Mann-Whitney U test, we observe that *community* images have higher bug lag than *official* images with small effect size ($|d| = 0.26$).

Figure 16 shows the quarterly evolution of the distribution of the bug-lag(i) for images $i \in \mathcal{I}_{\text{off}} \cup \mathcal{I}_{\text{com}}$, grouped by image origin (*official* or *community*). We observe that for old images, the bug lag is very low. This corresponds to what we observed in the previous analysis (i.e., the number of reported bugs is increasing over time, and most of the bugs remain open).

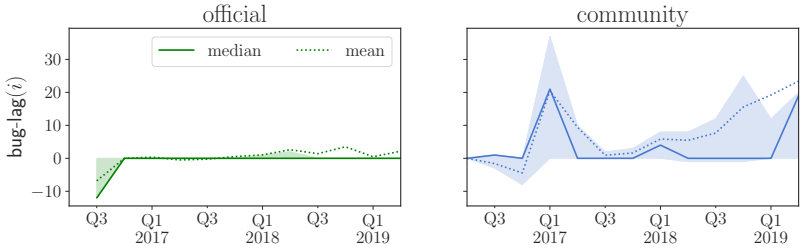

Fig. 16 Evolution of the distribution of bug-lag(i) for images $i \in \mathcal{I}_{\text{off}} \cup \mathcal{I}_{\text{com}}$ (*official* or *community*).

Figure 17 shows the quarterly evolution of the distribution of bug-lag(i) for images $i \in \mathcal{I}_{\text{off}} \cup \mathcal{I}_{\text{com}}$, grouped by their *Debian* distribution. *OldStable* images and *Testing* images appear to have a higher bug lag than *Stable* images. The statistical results reported in Table 17 confirm these observations, with a medium effect size for the comparisons that involve *Stable* images, but a negligible effect size between *OldStable* and *Testing*. The median bug-lag is 13 bugs for *Testing*, 0 for *Stable* and 5 for *OldStable*. The difference between *Stable* and *Testing* images can be explained by the fact that new updates in *Testing* come with fixed bugs. Package releases in this distribution are being checked and inspected daily. When bugs are found, they are fixed in new updates and this creates the bug lag between package releases.

Table 17 Statistically significant Mann-Whitney U tests and their corresponding effect sizes when comparing bug-lag for images in different *Debian* distributions.

population A	comparison	**population B**	**effect size**	$\lvert d \rvert$
OldStable	$>$	***Stable***	**medium**	**0.38**
OldStable	$<$	*Testing*	negligible	0.07
Stable	$<$	***Testing***	**medium**	**0.37**

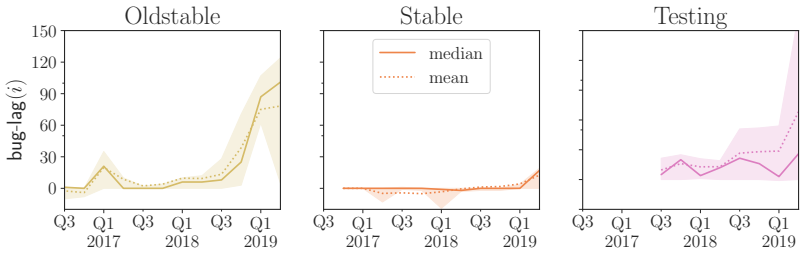

Fig. 17 Evolution of the distribution of bug-lag(i) for $i \in \mathcal{I}$, grouped by i_{dist} (*OldStable*, *Stable* or *Testing*).

Summary: All *Debian*-based images suffer from a high number of bugs (median of 6,483), with the number of bugs by image staying relatively stable over time. This means that the number of bugs being closed in packages in those images is similar to the number of new bugs reported in those packages. On the other hand, more than half of the bugs in deployed packages do not have a fix, even though *community* images tend to have a higher bug lag than *official* images. *Testing* images have a higher bug lag than *Stable* images because they tend to come with bug fixes. *OldStable* and *Testing* images have an increasing bug lag over time as a consequence of their increasing version lag.

7 Discussion

One of the core promises of *Docker* is that images are stable environments that can be downloaded and run even years after their creation. This is very good for ensuring the behavior of the deployed image, by offering a stable environment that will work as intended. While deployment decisions in the real world often require a delicate balance between sticking to a working version and upgrading to a more up-to-date version, container images ensure that specific versions can be easily deployed. From that perspective, old and outdated package versions that are frozen in container images are not a problem, but rather an advantage of the model of containerization. On the other hand, if *Docker* images do not include up-to-date package versions, they will miss the latest functionality, security updates and bug fixes, which is a problem in most production environments. Therefore, real systems are always balancing between staying with "the version that works" or updating to newer versions. People deciding when to deploy a new version of a container image in production always have to face this delicate balance, by deciding whether they should stick to the old image or switch to a more recent one. Our technical lag framework helps deployers in this difficult compromise.

This section discusses some implications of the results of our analysis of the technical lag of *Debian*-based *Docker* images: implications for the different

views of developers and deployers; insights specific to *Debian* and *Debian*-based *Docker* images; evidence of how the different lag dimensions help to characterise for different usage scenarios to which extent images are up-to-date; and some avenues for future research.

7.1 Deployer versus developer perspective

When measuring time lag and version lag for a *Docker* image, we considered the package versions *at image creation date*. In other words, we measured the lag of each included package as if it was deployed at build time. This corresponds to the difference between the time and version of the packages in the image with respect to the latest available packages at that time. This kind of information is mainly useful for developers building that image.

When measuring the bug lag and vulnerability lag for a *Docker* image, we counted the bugs and vulnerabilities in each included package according to the information *at the time of running the analysis*. It is very well possible that a package, which had no known bugs at the time of building an image, presented some bugs when we ran our analysis. Indeed, as time passes, new bugs may have been found for that image. Following this line of reasoning, the information we provide for bug lag and vulnerability lag is useful when considering the image "as if being deployed now": we use all the information we have now about bugs and vulnerabilities in the packages included in an image compared to more recent versions of those packages.

Actionable results: At the moment of building a *Docker* image, developers need to decide which package versions to incorporate. To do so, they can use and balance the proposed measures of time lag, version lag, bug lag and vulnerability lag to decide on a more informed basis whether to use an outdated package version or to replace it by a more recent version in their images. Existing tools that check for package freshness in *Docker* containers should strive to include the quantitative information provided by the technical lag measurement framework.
In addition to this, *Docker* container deployers can use bug lag and vulnerability lag to evaluate how good a *Docker* image "ages", offering them a perspective on how well developers built the image. Such information should be integrated into tools that help in deciding whether to deploy a given image, or to find alternatives containing less known bugs and vulnerabilities.

7.2 Lessons learned for *Debian*-based *Docker* images

In the response to RQ_1 we found a small proportion of outdated packages in *Debian* images. This means that developers tend to keep their images up-to-date when they build and release them. As a consequence, deployers can benefit from up-to-date collections of packages just by regularly updating the images they deploy to the latest available one. They get a clear benefit by switching to

new images, with little effort from their side: they do not need to take special actions on the image, except for the usual testing before deployment.

In the specific case of the working distribution *Debian Testing*, there is more effort involved in updating images, since *Testing* images have a higher number of outdated packages with very frequent updates. As a consequence, keeping images up to date would require updating them very frequently.

Lessons learned: The effort needed to update *Debian* packages installed in images concerns only a small proportion of packages.
Official *Debian* images are very similar, in terms of outdatedness, to installing directly from *Debian* repositories, at the time images were built. This is especially true in the case of *Stable* and *OldStable* images.

Actionable results: Deployers should use recent official *Debian* images for deployment, or for basing their own images, instead of building them directly from the *Debian* repositories, as long as they are following *Stable* or *OldStable*. This will cause little difference in terms of package outdatedness. Deployers who want to use *Testing* images should expect newer versions of these images to come with more frequent package updates than *Stable* and *OldStable* images.

The reason why packages tend to be regularly updated in images can be traced to a common practice in *official Debian* repositories: releasing new images with permissive tags referring to the latest images (e.g., latest, stretch) whenever new package versions are available and pinning the old ones with strict tags (e.g., stretch-20200224). Then, *community* images can be based on top of *official* images using the latest image tags (e.g., FROM debian:latest) and to update the installed packages, *community* images just need to be rebuilt whenever a new version of their base image is released, without modifying the *Dockerfile*.

However, images do not necessarily need to be derived from a base image containing up-to-date packages: image developers can include in the *Dockerfile* a command for updating packages (apt upgrade -y) as part of the building process in order to benefit from more recent releases.[30] To verify if this is a common practice, we inspected the Dockerfiles of all *Debian community* images in our dataset and found that only a small proportion (6.2%) makes use of the (apt upgrade) command. This is another reason explaining why *official* images are more up-to-date than the *community* ones.

Actionable result: To benefit from recent package releases, container deployers should rely on permissive tags of *official* images, and rebuild the images each time a new version of the base image is released. As an alternative, if they want to upgrade only some specific packages, they should use (apt upgrade) commands in the Dockerfiles.

The observations we made for $RQ_{1,2,3}$ suggest that developers of *Debian* images are careful about the state of the packages they contain. To confirm

[30] An example of this was provided with the Dockerfile for the community image shogun-dev:latest presented in Section 2.2.

this finding, we considered a snapshot of all *Debian* packages as of August 2019, for all considered *Debian* distributions. For each of them, we tracked when their latest available release was created. Figure 18 shows the evolution of the number of latest package releases at a given time t proportionally to the number of all latest package releases available on 30 August 2019. Most latest package releases were produced a long time ago. Nearly 92% of the *Stable* and 93% of the *OldStable* latest package releases were created before 2017-06-18, the release date of *Stable* distribution *Stretch*. Therefore, in any image built after that date, most of the packages are likely up to date, just because there was no new release even if the release contained in the image is very old. Moreover, since new package releases in *Stable* and *OldStable* only serve to add security patches, there is little risk of breaking changes.

For *Testing* images, things are different: new releases enter the distribution more frequently, and they can introduce breaking changes, since these changes may include new functionality or modifications of existing functionality.

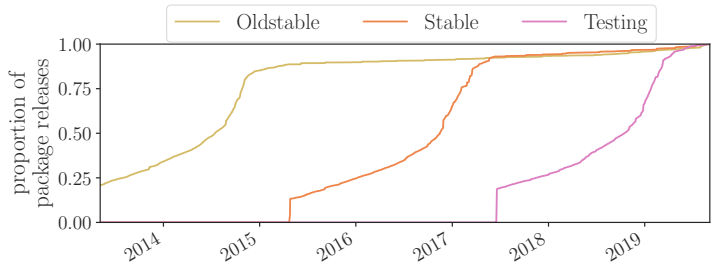

Fig. 18 Evolution of the number of latest package releases at time t proportionally to the number of all latest package releases available on 30 August 2019, grouped by *Debian* distribution. The value 0.5 for *Stable* in 2017, for example, means that 50% of the latest releases of all packages in *Stable* were created before 2017.

Actionable results: Developers of *Debian*-based *Docker* images can take advantage of Debian's slow release cycle (several months to release new package versions) to reduce their technical lag with small, punctual effort.
Image deployers wanting to benefit from up-to-date packages with respect to their corresponding distribution should rely on recent images based on *Stable* and *OldStable*.
Deployers of *Testing* images should be aware that new releases enter the distribution more frequently, with the risk of including breaking changes.

In RQ_4 we studied the number of vulnerabilities, finding that all *Debian* images contain at least some packages with a high number of vulnerabilities. This phenomenon is not caused by the developers building those images, since the observed version lag in images is low (cf. RQ_3): even images with only up-to-date packages at their build time may still suffer from many vulnerabilities.

Deployers concerned about security can mitigate the situation by deploying recent images, which have less *known* vulnerabilities.

Debian-based *Docker* images have an average number of 568 vulnerabilities, well above the average number of vulnerabilities for all *Docker Hub* images (180 vulnerabilities reported by Shu et al. (2017)). The number of vulnerabilities depends on the number of packages contained in the image. For example, it would be unfair to compare the absolute number of vulnerabilities between images with 100 and 1,000 packages. Nevertheless, the vulnerability of an image is determined mainly by the total number of vulnerabilities present in it, since any of them can potentially be exploited, regardless of the number of packages included in the image.

We observe that *Debian* images have a positive vulnerability lag in all situations. This vulnerability lag is usually low at the time of image creation, and tends to increase as the packages included in the image become more outdated. Hence, better updating procedures could be beneficial to avoid extra security vulnerabilities.

We observed a lower vulnerability lag for *Testing* images, even if such images come with a higher version lag. This implies that most of the package updates in *Debian Testing* are about fixing non-security bugs. RQ_5 confirmed this finding, since we observed a higher bug lag for *Testing* images.

Lessons learned: All images suffer from vulnerabilities, even the lightest official images (e.g., debian:slim). Deployers cannot avoid vulnerabilities even if they deploy the most recent images.
Testing images have lower vulnerability lag than other images. This is because *Testing* package releases are still *under development* and are not exposed yet to *Stable* users; hence they have less reported vulnerabilities.

Actionable results: Container deployers should reduce their number of vulnerabilities by relying on the package-level quantitative information provided by the technical lag measurement framework to decide which included packages need to be upgraded to more recent releases.

For *Stable* and *OldStable* images we occasionally observed a negative bug lag. This suggest that for such images, the *ideal* choice is not necessarily the highest available release if the goal is to minimize the number of known bugs. We also observed that all *Debian* images suffer from many bugs of low urgency. This is due to the fact that the number of reported bugs increases over time, and many of the bugs of lesser importance are never addressed, leading to their accumulation over time.

Lesson learned: All images suffer from bugs. Deployers cannot avoid all bugs even if they rely on the most recent package releases.

Actionable results: Even if bugs cannot be avoided entirely, deployers could benefit from the quantitative information about bug lag, by narrowing down to only the specific kind of bugs that are relevant for them (e.g., "critical bugs for i386"). Based on such information they could for example choose an image that strikes the best balance between a low bug lag and an acceptable version lag.

7.3 Interest of having different dimensions of technical lag

In order to assess the practical value of the technical lag notion, we performed 20-minute semi-structured interviews with five open source software practitioners at *FOSDEM 2019* (Zerouali, 2019). Each interview was structured into four parts: 1) the participant's profile; 2) software projects in which the participant has been involved; 3) questions related to the software dependency updating process followed by the participant; and 4) the potential value of the technical lag concept. A transcription of all interviews can be found in the replication package (Zerouali, 2020).

All interviewees were highly educated (having a Master's degree or higher) and experienced software engineers (with an average of 10 years of experience). 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, and another one worked as a development coach. We learned that the interviewees deal with updating software dependencies in different ways. All participants agreed that project dependency management is an important and critical task. As one interviewee noted: *"The dependency management is the hottest spot in our project"*. With respect to the potential value of the technical lag concept, one of the interviewees noted that the technical lag *"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"*. When asked about the preferred way of measuring technical lag, all interviewees suggested that multiple complementary measurements would probably be needed. Suggested ways of measuring lag included counting the number of versions, the number of commits, the amount of breaking changes, but also by analysing changelogs, and measuring the number of newly added functionalities. In addition to this, most of the interviewees agreed about the importance to include security vulnerabilities as one of the ways to measure technical lag. One of the participants specifically noted *"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"*.

Moreover, in previous surveys in the *Docker* landscape (Anchore.io, 2017; Bettini, 2015), *Docker* practitioners clearly pointed out that they care about multiple aspects (including the presence of vulnerabilities, bugs and outdated packages) before deploying application containers. This corroborates our find-

ings that multiple ways of measuring lag are needed to see the full picture. For example, while an outdated package with just one missed version update could be seen as a minor issue, it would be considered differently if the most recent version incorporates several fixes to known vulnerabilities. We observed this situation for many *Debian* images. For instance, the surround/ws-master repository on *Docker Hub* contained many images with a version lag of less than 120 versions, but at the same time their vulnerability lag was more than 525 vulnerabilities.

Actionable results: Technical lag can be measured in different ways, offering complementary information. Depending on their context and goals, container developers and deployers should use different lag measurements. Tool creators should include the various ways of measuring technical lag to empower their container scan and security management tools; this will help container developers and deployers in choosing what images to use and when to update them.

7.4 Future work

This study analysed the technical lag of *Debian*-based *Docker* images based on the difference between installed package releases and an *ideal* release for each of them. This ideal release was selected following the behaviour of apt, the default package manager of *Debian*, and corresponds to the highest available package release in the corresponding *Debian* distribution. Using this notion of ideal is quite common, following the rationale that "if *Debian* decided to substitute the old release with this new one, it is likely that this is more desirable". However, such releases may not always be the right choice since they might not be ideal for some deployments. Recent releases could be considered as less ideal than older ones because they are not yet heavily tested, because they may introduce backward incompatibilities or because they are affected by more vulnerabilities or bugs. It would therefore be of interest to explore other ideal functions for measuring image lag. The technical lag framework can be easily tailored to use other criteria for selecting an ideal release.

Several automated scan and security management services have emerged to enable *Docker* image deployers to assess the vulnerability of their deployed images. Examples of such services are *Snyk.io*, *Anchore.io* and *Quay.io*. Based on our multi-dimensional lag analysis we posit that such services should include other notions of lag as well, since this would empower container developers and deployers to gain better insights into the health of their *Docker* images.

While measuring technical lag is useful to get an overview of the freshness of an image and its installed packages, it is not sufficient to know what is missing. It is equally important to assess the *effort* required to reduce the incurred technical lag. Updating an image could be very time consuming and disruptive (e.g., because of the need to test the changes, to update scripts and configuration files, and to deal with backward incompatibilities). Tools that estimate the effort required to update an image or its packages would be very

valuable for practitioners, especially in combination with tools that measure technical lag, such as ConPan, a tool to inspect *Docker* images and report about the technical lag of their installed packages (Zerouali et al., 2019a).

8 Threats to Validity

Given the empirical nature of our work, the presented results are exposed to many potential threats. We discuss these threats and classify them following the recommendations in Wohlin et al. (2000).

Construct validity concerns the relation between the theory behind the experiment and the observed findings. The main threat of this kind comes from imprecisions in the data sources we used to identify vulnerabilities and bugs. Our analyses assumed that the data collected from the *Ultimate Debian Database* and from the *Debian Security Tracker* reliably represent a sound and complete list of bugs and vulnerabilities for *Debian* packages. However, they only contain bugs and vulnerabilities that (1) are known, (2) have been reported and (3) disclosed. As a result, the reported analysis underestimates the actual situation of packages in terms of bugs and vulnerabilities.

A second threat to construct validity concerns the way we identified packages being affected by vulnerabilities and bugs. The *Debian Security Tracker* and, to a lesser extent, the *Ultimate Debian Database* report on the *source packages* being affected, and not on the *binary packages*. This required us to map the installed binary *Debian* packages to their source packages. This mapping is not bijective, since several binary packages can be mapped to the same source package (e.g., binary packages libreoffice-writer and libreoffice-calc are both mapped to source package libreoffice). As a result, there is no automatic way to identify what binary package is actually affected by a bug or a vulnerability reported for its source package. We considered all of them to be affected in such cases, leading to a conservative overestimation of the number of bugs or vulnerabilities. To mitigate this threat, we systematically removed duplicate vulnerabilities and bugs after having collected them for all binary packages in an image.

Another threat to construct validity stems from how we identified installed packages in *Debian*-based images. We relied on the list of packages available in *official* and *security* Debian *snapshot repositories* to identify available package releases, and we considered a package release in this list as "installed" in an image if it was marked as such by dpkg, the official *Debian* package installer. Consequently, we did not identify package releases that: (1) are compiled from source code; (2) come from other repositories specified in /etc/apt/sources.d; or (3) are handled by package managers that do not interface with dpkg, such as flatpak or npm. Unfortunately, it is not possible to exhaustively take into account such packages given the large number of possible ways to install them in *Debian*. This implies that we underestimated the number of installed packages or failed to identify some package updates. We are convinced that this threat does not greatly affect our findings since we expect most users to

use dpkg to install packages, either directly or through one of the package managers that relies on dpkg, such as apt, the default package manager in *Debian* (The Debian GNU/Linux FAQ, 2019).

Internal validity concerns choices and factors internal to the study that could influence the observed results. The main threat comes from the way we instantiated the technical lag framework and its underlying functions. The definition of the ideal function was based on the behaviour of apt, the default package manager in *Debian*. It corresponds to the behaviour a user would obtain by updating the system without making any particular choice. The different delta functions were carefully chosen to precisely capture the aspects we aim to measure. The chosen aggregation functions agg are a bit more subjective since, regardless of the particular choice, the underlying idea is always to identify or compute an appropriate witness value given a collection of values. We aggregated lag using sum and maximum because we expect these to be the most relevant and representative for each of the considered lag dimensions at image level. To mitigate this threat, we repeated some of the analyses with other aggregation functions and, although this led to variations in the reported values, the observations and conclusions we made in this paper still hold.

Another threat to internal validity is our decision to analyze *all* images available in a repository, rather than selecting a *single* representative image. We did so, because it allowed us to gain a better understanding of the evolution of technical lag. While many repositories only contain a single image (the *latest*) that is regularly updated, other (especially in *official* repositories) may contain a larger number of images. We found that *official* repositories have a median number of 39 images, while *community* images have a median number of 2 images. This might have led to an "underexposure" of the specificities of single-image repositories in our comparisons, especially between *official* and *community* images.

Conclusion validity concerns the degree to which the conclusions we derived from our data analysis are reasonable. Since our conclusions are mostly based on empirical observations, our work is unlikely to be affected by such threats.

External validity concerns whether the results and conclusions can be generalized outside the scope of this study. The observations we make and the conclusions we draw for *community* images are based on a subset of all *community* images, selected from the most popular repositories, in terms of number of pulls. By doing so, we ensure not to have missed any popular *Debian*-based images from *community* repositories, and to have obtained a subset of *community* images that is representative for most users.

Although the proposed approach can be applied to non-*Debian*-based images, we cannot claim that our findings generalise to these variants. First, *Debian* images might have more installed packages than other images (e.g., those based on the lightweight Linux distribution *Alpine*). Second, not all distributions share *Debian*'s careful and conservative policy of deploying package updates. Different policies can lead to significantly different observations depending on the considered technical lag dimension. For instance, because

ArchLinux or *Fedora* have a more generous package update policy, we expect to find more outdated packages (i.e., a higher package lag) in their images. Conversely, we expect to find a lower vulnerability and bug lag, since updates that fix security vulnerabilities and bugs are more likely to be quickly available in such images.

Similarly, our findings cannot be generalised to containerisation systems beyond *Docker*, because its specificities, such as its layering mechanism, might have played a role in the observed findings.

9 Conclusion

This paper focused on the phenomenon of outdated *Docker* images, reflecting the dilemma of developers and deployers between updating the packages of their *Docker* images and sticking to the currently deployed and working versions. We presented an empirical analysis of the evolution of public *Docker Hub* images based on *Debian Linux* distributions.

To study how outdated such *Docker* images are when they are first released, we instantiated the formal technical lag framework along five different dimensions: package lag, time lag, version lag, bug lag and vulnerability lag. We studied 140,498 popular *Docker Hub* images, corresponding to public *Docker* images based on a *Debian* distribution, coming from both *official* and *community* repositories. We identified installed system packages and tracked their bugs and security vulnerabilities from official trusted data sources.

We observed that most of the packages in *Debian*-based *Docker* images are up-to-date. However, we also found that all images contain package releases having security vulnerabilities and bugs. By aggregating at image level the five dimensions of technical lag of package releases, we carried out a longitudinal multi-dimensional analysis of the technical lag of *Debian* images, enabling us to extract multiple actionable insights for image developers and deployers.

For each dimension we found that *community* images have higher lag than *official* images. Depending on the lag dimension, images belonging to specific *Debian* distributions were found to have higher lag than for other distributions. For example, version lag was highest for images relying on *Debian Testing*, while vulnerability lag was highest for *OldStable* images. We also found that in some cases, the lag is increasing over time, for example for package and version lag in *Debian Testing* images. Since we found a positive vulnerability lag in *Docker* images, image developers and deployers whose major concern is security need to rely on better updating procedures. In contrast, image deployers that care more about their packages freshness need to rely on the *Stable* distribution of *Debian*, since images relying on other distributions have a higher version lag.

Docker image management and monitoring tools could benefit from incorporating the different aspects of image outdatedness reflected by the five dimensions of technical lag to better support image developers and deployers in their decisions to create, use or update their images.

Acknowledgements

This research is carried out in the context of the Excellence of Science project 30446992 SECO-Assist financed by FWO-Vlaanderen and F.R.S.-FNRS. We acknowledge the support of the Government of Spain through project "BugBirth" (RTI2018-101963-B-100).

References

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. doi: 10.1109/ESEM.2009.5316017.

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. doi: 10.1016/j.jss.2012.02.018.

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. doi: 10.1016/j.scico.2013.06.007.

Anchore.io. Snapshot of the container ecosystem. https://anchore.com/wp-content/uploads/2017/04/Anchore-Container-Survey-5.pdf, April 2017. accessed: 01/12/2019.

Cyrille Artho, Kuniyasu Suzaki, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Why do software packages conflict? In *Working Conf. Mining Software Repositories*, pages 141–150, 2012. doi: 10.1109/MSR.2012.6224274.

David Bernstein. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014. doi: 10.1109/MCC.2014.51.

Anthony Bettini. Vulnerability exploitation in docker container environments. *FlawCheck, Black Hat Europe*, 2015.

Carl Boettiger. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015. doi: 10.1145/2723872.2723882.

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 *International Conference on Mining Software Repositories*, pages 323–333. IEEE Press, 2017. doi: 10.1109/MSR.2017.67.

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. doi: 10.1109/MSR.2015.27.

F. R. Cogo, G. A. Oliva, and A. E. Hassan. An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering*, 2019. doi: 10.1109/TSE.2019.2952130.

Theo Combe, Antony Martin, and Roberto Di Pietro. To Docker or not to Docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62, 2016. doi: 10.1109/MCC.2016.100.

Joël Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. Measuring dependency freshness in software systems. In *International Conference on Software Engineering*, pages 109–118. IEEE Press, 2015. doi: 10.1109/ICSE.2015.140.

Max de Visser. A look at how often docker images are updated. https://anchore.com/look-often-docker-images-updated/, September 2017. Accessed: 20 August 2020.

Alexandre Decan, Tom Mens, and Eleni Constantinou. On the evolution of technical lag in the npm package dependency network. In *Int'l Conf. Software Maintenance and Evolution*, pages 404–414. IEEE, September 2018a. doi: 10.1109/ICSME.2018.00050.

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*, 2018b. doi: 10.1145/3196398.3196401.

Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1):381–416, Feb 2019. ISSN 1573-7616. doi: 10.1007/s10664-017-9589-y.

Brian DeHamer. Docker hub top 10. https://www.ctl.io/developers/blog/post/docker-hub-top-10/, 2020. Accessed: 20 August 2020.

Docker Inc. Docker Registry HTTP API V2. https://docs.docker.com/registry/spec/api/, 2020a. Accessed: 20 August 2020.

Docker Inc. Dockerfile reference. https://docs.docker.com/engine/reference/builder/, 2020b. Accessed: 20 August 2020.

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. doi: 10.1007/s10664-008-9100-x.

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. Springer, 2017. doi: 10.1007/978-3-319-57735-7_17.

Jordan Henkel, Christian Bird, Shuvendu K Lahiri, and Thomas Reps. Learning from, understanding, and supporting DevOps artifacts for Docker. *International Conference on Software Engineering*, 2020.

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. doi: 10.1109/SANER.2015.7081869.

Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, 2017. ISSN 1573-7616. doi: 10.1007/s10664-017-9521-5.

Soonhong Kwon and Jong-Hyouk Lee. Divds: Docker image vulnerability diagnostic system. *IEEE Access*, 2020. doi: 10.1109/ACCESS.2020.2976874.

Damien Legay, Alexandre Decan, and Tom Mens. On package freshness in Linux distributions. In *Int'l Conf. Software Maintenance and Evolution – NIER Track*, 2020.

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. doi: 10.1109/ACCESS.2019.2905424.

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

Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. Type regression testing to detect breaking changes in Node. js libraries. In *European Conference on Object-Oriented Programming*, 2018. doi: 10.4230/LIPIcs.ECOOP.2018.7.

Anders Møller and Martin Toldam Torp. Model-based testing of breaking changes in Node.js libraries. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 409–419. ACM, 2019. doi: 10.1145/3338906.3338940.

Adrian Mouat. *Using Docker: Developing and Deploying Software with Containers*. O'Reilly Media, Inc., 2015.

Lucas Nussbaum and Stefano Zacchiroli. The ultimate Debian database: Consolidating bazaar metadata for quality assurance and data mining. In *Working Conference on Mining Software Repositories*, pages 52–61, 2010. doi: 10.1109/MSR.2010.5463277.

Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices? In *Annual Meeting of the Southern Association for Institutional Research*, 2006.

Pasquale Salza, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Filomena Ferrucci. Third-party libraries in mobile apps: When, how, and why developers update them. *Empirical Software Engineering*, 25:2341–2377, 2020. doi: 10.1007/s10664-019-09754-1.

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. doi: 10.1145/3029806.3029832.

Emilien Socchi and Jonathan Luu. A deep dive into Docker Hub's security landscape – a story of inheritance? Master's thesis, University of Oslo, 2019.

The Debian GNU/Linux FAQ. The Debian package management tools. https://www.debian.org/doc/manuals/debian-faq/pkgtools.en.html, August 2019. Accessed: 20 August 2020.

James Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

Brian Vermeer and William Henry. Shifting Docker security left. https://snyk.io/blog/shifting-docker-security-left/, 2019. accessed: 02/11/2019.

Jérôme Vouillon and Roberto Di Cosmo. On software component co-installability. In *Joint European Soft. Eng. Conf. and ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, 2011. doi: 10.1145/2025113.2025149.

C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering - An Introduction*. Kluwer, 2000. doi: 10.1007/978-1-4615-4625-2.

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 *International Conference on Software Maintenance and Evolution*, pages 559–563. IEEE, 2018. doi: 10.1109/ICSME.2018.00067.

Ahmed Zerouali. *A Measurement Framework for Analyzing Technical Lag in Open-Source Software Ecosystems*. PhD thesis, University of Mons, September 2019.

Ahmed Zerouali. Replication package for Debian-based Docker images. https://doi.org/10.5281/zenodo.3765315, 2020.

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. doi: 10.1007/978-3-319-90421-4_6.

Ahmed Zerouali, Valerio Cosentino, Gregorio Robles, Jesus M Gonzalez-Barahona, and Tom Mens. Conpan: a tool to analyze packages in software containers. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 592–596. IEEE Press, 2019a. doi: 10.1109/MSR.2019.00089.

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*, 2019b. doi: 10.1002/smr.2157.

Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M. Gonzalez-Barahona. On the relation between outdated Docker containers, severity vulnerabilities, and bugs. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 491–501. IEEE, Feb 2019c. doi: 10.1109/SANER.2019.8668013.

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. doi: 10.1007/978-3-030-22888-0_10.

Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX Security Symposium*, pages 1–16, 2019.