Analyzing Technical Lag in Docker Images

Ahmed Zerouali
ahmed.zerouali@umons.ac.be
University of Mons

***Abstract*—Packaging software into containers is becoming a common practice when deploying services in cloud and other environments. *Docker* is currently one of the most popular container technologies for building and deploying containers. A key part of this technology is the concept of registry, where container images are stored and shared. The largest one of these registries is *Docker Hub*, with hundreds of thousands of public *Docker* images ready for deployment. A container image includes usually a collection of software packages, in many cases corresponding to a Linux-based distribution. By analyzing packages in images, we can learn to which extent those images offer packages that have newer versions available. To measure how "outdated" a container image is, we introduce the concept of "technical lag" for the container, as the difference between a given image, and the most updated image that is possible with the same collection of packages. Using this concept as the framework for the analysis, we empirically measure and study technical lag for 2,245 *Docker Hub official* images based on the *Alpine Linux* distribution, with a total of 63,581 packages. Our results show a strong presence of technical lag inside of those images, indicating that even well-maintained containers could benefit of better procedures for updating.**

***Index Terms*—Empirical analysis, *Docker* containers, technical lag**

I. Introduction

In order to reduce conflicts between computing environments and increase development productivity, packaging software into isolated and executable containers has become a common practice. *Docker* [1] containers emerged as a lightweight approach to provision multiple applications on a single host, sharing and wrapping system libraries, configuration files, and code of the same operating system.

As an open source and *Linux*-based platform, *Docker* provides support for *Linux* and non-*Linux* operating systems. This led to the creation of registries that provide a common place for users willing to build, update and share their *Docker* images with others. One of the largest registries is *Docker Hub* [2], which distributes a large number of public images.

Docker Hub images are distributed using *repositories* allowing users to develop and maintain versioned images. A public repository can be an *official* or a *community* repository. Community repositories can be created by any other user or organization. An official repository contains public and certified images from official companies (e.g., ElasticSearch, Debian Alpine, etc.). Official repositories can be an essential base operating system that serve as the starting point for other users, thus they are more secure and well maintained.

Docker Hub images, and *Linux*-based images in particular, usually include a collection of software packages corresponding to the used *Linux* distribution. Once an image is built, its packages remain *frozen* until the image is updated, rebuilt and re-uploaded to the registry. Such images provide interesting datasets from which one can gather information about how maintained these images are, or how outdated and vulnerable their contained packages are. Gummaraju et al. already observed that about one third of the *official* images in *Docker Hub* contain high priority security vulnerabilities [3], demonstrating a strong need for more automated and systematic methods of applying security updates.

In this paper, we focus on *Docker* images by studying how *outdated* their contained packages are, compared to their latest available releases. To this extent, we rely on the notion of *technical lag*, a concept first introduced by Gonzalez-Barahona et al. [4]. In order to estimate the effort needed to deploy the most recent version of a deployed software, technical lag can be used and measured as the difference (e.g., time, functionality, etc.) between the version of the software deployed in production and the most recent version available of this software.

We analyze this technical lag for containers, as the increasing lag between a given image and the most up-to-date image available. More specifically, we empirically measure and study technical lag for $2,245$ *Docker Hub* images based on the *Linux* distribution *Alpine*, retained mainly from 42 *official* repositories, with a total of $63,581$ package versions. We focus on a single research question: ***How can we quantify technical lag induced by packages in Docker images?***

II. Related work

González-Barahona et al. [4] have proposed a theoretical model of "technical lag" to measure how outdated software components are. They explored many ways in which technical lag can be measured. They also presented specific cases for which it is useful to analyze the evolution of technical lag.

Ayed et al. [5] tackled the issue of enhancing the description and search of *Docker* images, proposing *Docker2RDF*, a system that extracts information from the *Docker Hub* description pages and builds a semantic description of *Docker* images. In order to enhance the support for discovering *Docker* images, Brogi et al. [6] introduced *DockerFinder*, a microservice-based prototype that permits searching for images based on multiple attributes, e.g., image name, image size, or supported software distributions.

Cito et al. [7] conducted an empirical study on a dataset of 70,000 Dockerfiles, and contrasted this general population

with samplings containing the Top-100 and Top-1000 most popular projects using *Docker*. Their goal was to characterize the *Docker* ecosystem, discover prevalent quality issues, and study the evolution of Docker images. Among other results, they found that the most popular projects change more often than the rest of the *Docker* population, with on average 5.81 revisions per year and 5 lines of code changed. Furthermore, they found that 34% of all Docker images, from a representative sample of 560 projects, were not able to be built.

Shu et al. [8] performed a large scale study on the state of security vulnerabilities in both community and official *Docker Hub* repositories. They proposed the *Docker Image Vulnerability Analysis (DIVA)* framework to automatically discover, download, and analyze *Docker* images for security vulnerabilities. They studied a set of 356,218 images and found that both official and community repositories contain more than 180 vulnerabilities on average. Another finding is that many images had not been updated for hundreds of days.

Not specifically focused on *Docker* images, Kula et al. [9] studied the impact of dependency updates in the *GitHub* ecosystem. They empirically studied library migration of a set of 4,600 *GitHub* repositories and 2,700 library dependencies, and found that 81.5% of the studied projects keep their outdated dependencies. Surveying developers about this, they found that 69% of the interviewees were unaware of their vulnerable dependencies.

III. Method

A. Choice of the base image

While it is possible to create *Docker* images *from scratch*, most *Docker* images are based on so-called *parent images*. These parent images are mostly *base images* that do not rely on any other image, except for the *Docker*-reserved minimal image "scratch". In order to find publicly available information about system packages installed in *Docker* images, we chose a base image that is based on a *Linux*-distribution.

One of the most popular base images is *Alpine*[1]. It is a minimal image based on the security-oriented, lightweight *Alpine Linux* distribution[2] with a complete package index that is no more than 8MB in size. Because of *Alpine*'s widespread use in *Docker Hub* we chose to locally mirror and analyze *Docker* images that rely on *Alpine*'s lightweight base image[3]. *Alpine* has a quite simple and good package manager that is attracting people to migrate their images to it [10].

B. Data extraction

Our empirical analysis will only focus on *official* repositories, as they are supposed to be more secure and well maintained. A *Docker* repository can have many images, or so called *tags*. A maintainer can create an image repository *"imageRepo"* and then upload tagged images to it, e.g., *"imageRepo:debian", "imageRepo:v1"*, etc.

[1]https://hub.docker.com/_/alpine/
[2]https://alpinelinux.org/about/
[3]http://pkgs.alpinelinux.org/

To know which *Docker* images make use of the *Alpine* base image, we extracted information about all available images from the 122 *official* repositories. We found that by the end of the year 2017 (i.e., 27 December 2017), *Docker Hub* contained more than 12,840 *official* images. From this list of images, we searched for those tagged with the word "alpine", as it is a common practice in *Docker Hub* to use this word in the tag of images based on *Alpine*, e.g., "imageRepo:alpine", "imageRepo:v1-alpine". Out of 12,840 *official* images, we found 2,253 images (i.e., 17.5%) from 42 *official* repositories to be considered for our data extraction.

We locally downloaded each image, and extracted the version of *Alpine Linux* used, as well as the name and version of all packages installed in the image. From 261 distinct used packages, we found 82,949 package versions installed in 2,253 images, with an average of 36 package versions per image.

From the *Alpine* package manager we extracted the build date for each version of each installed package in the downloaded *Docker* images. Unfortunately, we found that the information of 23% of the package versions is no longer available in the package manager. After discarding the images and package versions with incomplete information, we retained 2,245 *Docker* images and 63,581 package versions for our empirical analysis.

IV. How can we quantify technical lag induced by packages in Docker images?

A. Measuring technical lag

The concept of technical lag seems similar to the metaphor of "technical debt" [11], [12]. Technical debt refers to the qualitative difference between code "as it should be" and code "as it is". Technical lag refers to the increasing delay between the most appropriate upstream versions of packages used by a software system and those actually used in the deployed system [4].

Technical lag of a *Docker* image can refer to anything that makes an image container out-of-date with respect to the most recent available image. For example, more recent images may have fixed security vulnerabilities, added packages with additional functionality, updated packages to fix bugs, made changes to the configuration files, etc.

In our analysis, we consider the technical lag of an image as the technical lag of the collection of package versions installed by the image. Therefore, we need to analyze the technical lag of these installed package versions. Technical lag can be calculated in different ways. In this paper, we will use two different definitions:

1) *package time lag* is the time difference (in days) between the release date of installed package version and the date of the latest available package release.
2) *package version lag* computes the total number of releases that the installed package version is behind the latest available package release. In this case, we consider all types of releases (major, minor and patch releases).

For example, version *2.4.10-r0* of package *icinga2* was created on *2016-05-25*. If we find it installed in a certain

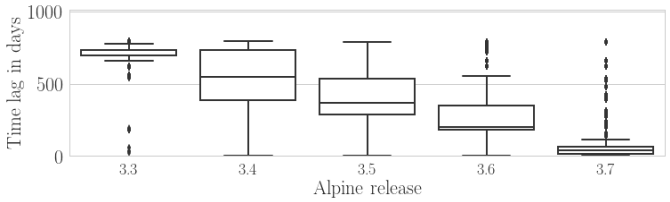

Figure 1. Distribution of package time lag for outdated packages, grouped by Alpine release

image, we compare it to the latest available version before the last update. Considering that the last update is *2017-12-01*, the latest available version would be *2.8.0-r0* that was created on *2017-11-23*. The *package time lag* in this case would be 546 days. The *package version lag* would be 3, because there are three releases between these two dates in the *Alpine* package manager (i.e., *2.5.4-r2, 2.6.3-r1* and *2.8.0-r0*).

To calculate the impact that technical lag has on images and repositories, we can rely on the outdated packages' time or version lag. However, in our case, we consider technical lag impact as:

1) *image lag impact* computes the number of packages in a *Docker* image that are out-of-date because they have a non-zero package lag.
2) *repository lag impact* as the median number of outdated packages per image contained in the repository.

B. Package time lag

For each used package version in our dataset (63,581 in total), we searched for its build date using the name and version number found in the image container, and compared this to the build date of the latest available package version. Based on this, we computed the *package time lag*. We found 59,693 (94% of all installed versions) of all package versions to be out of date (i.e., *package time lag*> 0), while the number of images did not change. The latter implies that all 2,245 analyzed *Docker* images have at least one outdated package version installed.

Figure 1 shows box plots of the distribution of *package version time lag*, grouped by the installed *Alpine Linux* version. We notice that the time lag is related to the *Alpine* version: the lag decreases as the *Alpine* version increases. This is expected, since base images with newer versions of *Alpine* often come with newer versions of packages. However, we observe a very high technical lag for all distributions, as medians and means are higher than the time difference between today and the date of the creation of the *Alpine Linux* releases. Table I shows a decreasing time lag w.r.t to the *Alpine Linux* releases. We also observe an increasing difference between the mean and median values indicating that the data distribution is becoming more skewed over time.

We checked whether we could find images based on *Alpine Linux* 3.3 that are still being updated, since that version of *Alpine Linux* is no supported since November 2017. We found

Table I
NUMBER OF IMAGES PER *Alpine Linux* RELEASE, AND MEDIAN AND MEAN PACKAGE TIME LAG (IN DAYS).

Alpine release	release date	number of images	median time lag	mean time lag
3.3	2016-01-06	84	733	728.8
3.4	2016-05-31	962	550	538.8
3.5	2016-12-22	337	371	435
3.6	2017-05-24	578	203	287.5
3.7	2017-11-30	284	42	133.9

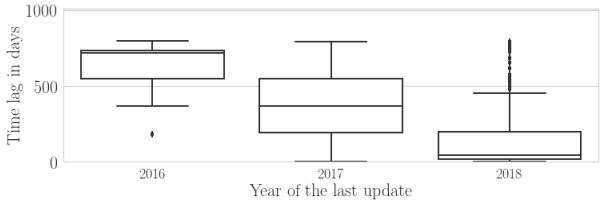

Figure 2. Distribution of the package time lag for outdated packages, grouped by the year of the last update of the image container.

only one image that was updated after the end of the support, namely the *alpine:3.3* image itself.

Figure 2 shows box plots of the distribution of the *package time lag* for outdated packages, grouped by the year of the last update of the image containers. As expected, the time lag decreases, because recently updated containers have packages with a lower time lag. The mean time lag in the distribution of 2016 is close to two years (i.e., 663 days), while in 2017 it is close to one year and two months (i.e., 403 days). Note, however, that the number of newly created images is increasing, and this affects the distribution of the time lag, since old images could have been stopped being updated. In fact, we found that 18% (i.e., 394) of the images stopped being updated in 2016.

To investigate how images tend to update their packages, we extracted a second snapshot of the images, 18 days after our first data extraction. We found that 25% (i.e., 552) of the images had been updated, and the median time difference between the last two updates for these images is 27 days.

For these images, we verified whether they updated their packages or not. We found that from 20,785 outdated package versions, only 595 were updated. These updates happened to important packages such as *openssl, libcrypto1.0, libssl1.0* and *apk-tools*. However, these updates happened in only 50% (i.e., 278) of the re-updated images, most of them in the images of *php, tomcat* and *wordpress*. This reveals that *Docker* images based on *Alpine* do not tend to update most of their outdated packages, they only update important packages such as *openssl*.

These results provide insight to maintainers and users willing to create or use images on top of *official* repositories, about how up-to-date are the packages installed inside these images. Our analysis reveals that they should expect many outdated packages, making the image vulnerable to security issues in many cases [8].

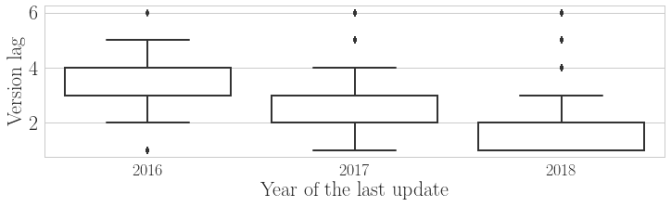

Figure 3. Distribution of package version lag for outdated packages, grouped by the year of the last update.

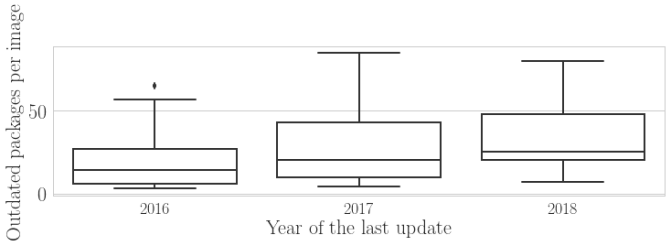

Figure 4. Distribution of image lag impact induced by outdated packages in *Docker* images, grouped by year of last update

C. Package version lag

To analyze the *package version lag*, we identified all available versions of outdated packages used in at least one of the analyzed *Docker* images of the *Alpine* package manager. We calculated how many package versions were released between the build date of the used version and the latest available release of that package.

Figure 3 shows the distribution of *package version lag* for outdated packages, grouped by the year of the last update. We notice that version lag decreased from 2016 to 2018, corresponding to the reduction in time lag observed in Section IV-B. This implies again that the number of newly created images influenced the distribution of the version lag, since we can still find a maximum version lag of 6 releases in both years 2017 and 2018.

D. Image and repository lag impact

To analyze the *image impact lag* of *Docker* containers we computed the total number of outdated packages installed within each container. Unlike our previous observations, Figure 4 reveals that the number of outdated packages is increasing, which means that the images are adding more packages without upgrading them later.

To calculate the *repository impact lag*, we relied on the same approach as the one used for *image lag impact*. We computed repository lag impact as the median number of outdated packages per image. Figure 5(a) shows the first 5 official *Docker* repositories having the *highest* median number of outdated packages per image, while Figure 5(b) shows the 5 official *Docker* repositories with the *lowest* median number of outdated packages per image. A manual inspection in *Docker Hub* between the repositories in the figure (a) and those in figure (b) showed a large difference of vulnerabilities found

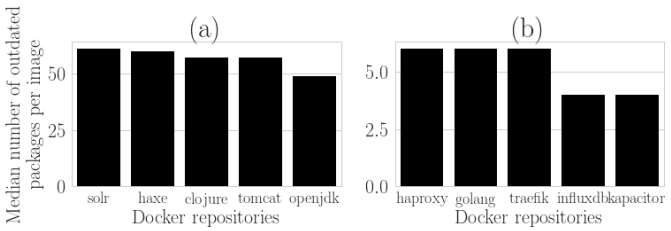

Figure 5. Median number of outdated packages per image for *official Docker* repositories.

in their images. Nearly all scanned images of the repositories *solr*, *haxe* and *clojure* have vulnerabilities.

V. Conclusions, Limitations and Further Research

We analyzed technical lag induced by outdated packages installed in *Docker* images based on the *Alpine Linux* distribution. We computed this technical lag based on the difference in time and difference in number of releases between the deployed package version and the latest available release of each package. We found 94% of all package versions to be outdated across our dataset of 2,245 analyzed *Docker* images, indicating a strong presence of technical lag inside these images. This demonstrates that even well-maintained container images could benefit from better procedures of updating.

We studied technical lag because in an ideal world, images should depend on the most recent available version of their used packages, in order to benefit from the latest functionality, security updates and bug fixes. However, in many cases, maintainers are more focused on other software characteristics such as package stability, or they just choose not to upgrade certain packages because of the considerable effort that may be involved in doing so ("if it ain't broke, don't fix it"). It therefore may be useful to explore more sophisticated notions of technical lag that are able to quantify the actual effort needed to deploy the most available stable package version.

While searching the available information in the *Alpine* package manager, we did not find relevant information for some package versions. This may have led to an underestimation of the technical lag reported in this article. However, we are confident about our dataset, since we used 77% (i.e., 63,581 package versions) of the original dataset.

When studying the technical lag, we did not differentiate between the specific characteristics of packages or *Docker* images, such as their age, size, service, targeted audience, or functionality provided. This remains a topic of future work. We also did not perform an analysis of *Docker* images that rely on other base images such as *Debian* base images or other community and official base images. Comparing the effect of the *Linux* distribution (and more generally, the quality of the base image) on technical lag also remains a topic of future work. Finally, we aim to study snapshots of *Docker* images over time to be able to carry out a more fine-grained study on the evolution of technical lag.

REFERENCES

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

[2] Docker Inc. Docker hub. https://hub.docker.com/. accessed: 01/01/2018.

[3] Jayanth Gummaraju, Tarun Desikan, and Yoshio Turner. Over 30% of official images in docker hub contain high priority security vulnerabilities. Technical report, Technical report, BanyanOps, 2015.

[4] 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.

[5] Ahmed Ben Ayed, Julien Subercaze, Frederique Laforest, Tarak Chaari, Wajdi Louati, and Ahmed Hadj Kacem. Docker2rdf: Lifting the docker registry hub into rdf. In *Services (SERVICES), 2017 IEEE World Congress on*, pages 36–39. IEEE, 2017.

[6] Antonio Brogi, Davide Neri, and Jacopo Soldani. Dockerfinder: Multi-attribute search of docker images. In *Cloud Engineering (IC2E), 2017 IEEE International Conference on*, pages 273–278. IEEE, 2017.

[7] 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 *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 323–333. IEEE Press, 2017.

[8] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 269–280. ACM, 2017.

[9] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, May 2017.

[10] Brian DeHamer. Docker hub top 10. https://www.ctl.io/developers/blog/post/docker-hub-top-10/. accessed: 01/01/2018.

[11] Ward Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1993.

[12] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, Nov 2012.