On the Usage of *JavaScript*, *Python* and *Ruby* Packages in Docker Hub images

Ahmed Zerouali[a], Tom Mens[b], Coen De Roover[a]

[a] *Software Languages Lab - Vrije Universiteit Brussel*
ahmed.zerouali@vub.be, coen.de.roover@vub.be
[b] *Software Engineering Lab - University of Mons*
tom.mens@umons.ac.be

Abstract

Docker is one of the most popular containerization technologies. A *Docker* container can be saved into an image including all environmental packages required to run it, such as system and third-party packages from language-specific package repositories. Relying on its modularity, an image can be shared and included in other images to simplify the way of building and packaging new software. However, some package managers allow to include duplicated packages in an image, increasing its footprint; and outdated packages may miss new features and bug fixes or contain reported security vulnerabilities, putting the image in which they are contained at risk. Previous research has focused on studying operating system packages within *Docker* images, but little attention has been given to third-party packages. This article empirically studies installation practices, outdatedness and vulnerabilities of JavaScript, Python and Ruby packages installed in 3,000 popular community *Docker Hub* images. In many cases, these installed packages missed important releases leading to potential vulnerabilities of the images. Our findings suggest that maintainers of *Docker Hub* community images should invest more effort in updating outdated packages contained in those images in order to significantly reduce the number of vulnerabilities. In addition to this, Python community images are generally much less outdated and much less subject to vulnerabilities than NodeJS and Ruby community images. Specifically for NodeJS community images, elimination of duplicate package releases could lead to a significant reduction in their image footprint.

Keywords: software containerization, package management, software vulnerability, outdated software, *Docker Hub*, third-party library

1. Introduction

In order to cope with continuous changes in the software development process and the need to accelerate the release time, software containerization has emerged as an easy and efficient approach to wrap and ship software in a lightweight way. Containerization enables provisioning several applications on a single host, sharing and packaging system packages, binaries and configuration files [1]. Today, *Docker* [2] is the most widespread containerization technology [3]. Practitioners around the world rely on it to package and distribute their software and its dependencies as images via services like *GitHub* and *Docker Hub*. With more than 4.1M image repositories in October 2020 [1], the latter is the world's leading service for finding and sharing container images.

Docker Hub categorizes images into two types. *Official* images are created by official organizations like *Debian* [2] and they are inspected and designed to be used as base images for other images. *Community* images are created by regular users and can be published without any inspection. In most cases, *Docker Hub* images do not only include packages for operating systems (e.g., *Debian* and *Alpine*) but also contain third-party packages from language-specific package repositories (e.g., *Python* packages from *PyPI*, *JavaScript* packages from *npm*, and *Ruby* packages from *RubyGems*). Some of these packages are required by other packages and applications as development or runtime dependencies (e.g., the *PyPI* package *six* helps to develop *Python* code that is compatible with *Python* 2 and *Python* 3), while other packages are stand-alone programs that can be executed from the command line to provide specific services (e.g., the *npm* package *elasticdump* is used to dump data from *Elasticsearch* to json files, etc). Such packages should be treated with care since non-useful ones will lead to a larger image footprint which could reduce the performance of images and affect the scalability of their services [4, 5]. Moreover, these packages can be outdated, implying that they may miss the newest features and bug fixes, and can contain known security vulnerabilities that can be exploited by hackers to abuse the image when it is running or to abuse the host system in which the image is deployed [6]. A survey with *Docker* users showed that security is a top concern when deciding whether to deploy *Docker* images [7]. Another survey showed that in addition to security, *Docker* users are concerned about other software package checks like verifying whether third-party software versions are up-to-date [8]. Studying the usage of packages within *Docker* containers will help practitioners to provide and deploy better containers.

Previous studies [9, 10, 11] have focused on the security of system packages within *Docker* images, overlooking third-party packages. Given the increasing use of the latter, it is important to study the usage and impact of outdated and vulnerable releases of such third-party packages.

In prior work we provided preliminary evidence of outdated and vulnerable *JavaScript* packages in only three official *Docker Hub* image repositories [12]. This article significantly extends upon that work by studying the usage of third-party *node JavaScript*, *Python* and *Ruby* packages provided by their respective package repositories *npm*, *PyPI* and *RubyGems* in a large dataset of community *Docker Hub* repositories. In particular, we compare third-party package instal-

[1] `https://hub.docker.com/search?q=&type=image`
[2] `https://hub.docker.com/_/debian`

lation practices across different community *Docker Hub* images. In addition, we assess the outdatedness and vulnerabilities of packages contained within *Docker* images at two time points:

- Analysing *Docker* images at their *last observed modification date* (within an observation period from June 2016 until December 2019) allows us to assess how much *Docker* maintainers care about the outdatedness and vulnerabilities of their images when they release them. This helps to provide recommendations to container maintainers on how to improve their images.

- Inspecting *Docker* images *at their extraction date* (12 December 2019) allows us to reflect upon the state of images as if they were deployed at the moment when we performed our study. This helps to provide recommendations to image users, allowing them to choose the most appropriate and most secure images.

To do so, we rely on the inheritance mechanism of *Docker* and analyse the top downloaded community images that are based and built on top of the official *Docker* images for *node*, *Python* and *Ruby*. More specifically, we focus on six main research questions:
RQ_0 **How prevalent are installed third-party packages in Docker Hub images?** This research question compares package installation practices between different community images. It helps users of these images to know what to expect when they install a *node*, *Python* or *Ruby* image.
RQ_1 **Do image maintainers add additional packages to their base *Docker Hub* images?** We study this question for two types of third-party packages: (1) core packages that are essential and inherited from the base images, and (2) non-core packages that maintainers add on top of those included in the base image. It helps the community to know which packages are most frequently installed within *Docker* images, as their maintainers can focus their attention on them.
RQ_2 **Do image maintainers include outdated third-party packages in their last image update?** We assess the outdatedness of *Docker Hub* images at their last observed modification date. The answer to this question helps community maintainers to shape their best practices by knowing the outdatedness of the images they produce, which will allow them to know how to improve these images and eventually create more up-to-date ones.
RQ_3 **How outdated are third-party packages in *Docker Hub* images?** We show the state of *Docker Hub* community images at their extraction date, which helps users of such images to know how outdated the most downloaded images are and how they could be improved to become more up-to-date.
RQ_4 **Do image maintainers include third-party packages with known vulnerabilities in their images?** We identify the vulnerabilities that were known in each image at its last observed modification date and were affecting its installed third-party packages. This allows us to understand how concerned image maintainers are about vulnerable third-party packages.

RQ_5 **How vulnerable are third-party packages in *Docker Hub* images?** We identify which and how many vulnerable third-party packages are present in community images at their extraction date. This will help users of such images to know which actions to take when they rely on them.

All along these research questions, we compare the results of *Docker Hub* images derived from the official base images *node*, *Python* and *Ruby*. In general, we found that the number of installed core and non-core packages is related to the used base image. For example, *node* images tend to have a higher number of packages installed compared to *Ruby* images, because *npm* packages have higher numbers of transitive dependencies compared to *RubyGems* [13]. *Python* images have the lowest numbers of installed packages because they do not have any extra packages (i.e., core packages) already inherited from the base images. We found that core packages are more outdated than non-core ones. However, both types of packages suffer from security vulnerabilities, endangering the environments where the image can be deployed.

The remainder of this article is structured as follows: Section 2 discusses related work. Section 3 explains the research method and data extraction process, and presents a preliminary analysis of the considered dataset. Section 4 empirically studies the research questions and presents the results of this paper. Section 5 highlights the novel contributions, discusses our findings, and outlines possible directions for future work, while Section 6 discusses the limitations of this work. Section 7 concludes.

2. Related work

2.1. Docker related studies

Due to its lightweight, self-sufficient and portable containers [14, 15], *Docker* has become one of the most popular and largely used containerization technologies. According to the 2020 Stack Overflow Developer Survey [3], *Docker* is the most wanted platform and the second most loved platform. For this reason, it has been subject to many research studies covering multiple aspects such as security, quality and evolution of *Docker* containers.

Cito et al. [16] 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. Moreover, based on a representative sample of 560 projects, they observed that 34% of all *Docker* images could not be built from their *Dockerfiles*. Lu et al. [4] 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*. Henkel et al. [17] advocate for more effective semantics-aware tooling for *Dockerfiles* in order to reduce the quality gap between *Dockerfiles* written by experts and those found in *GitHub* repositories. They found that on average, *Dockerfiles* on *GitHub* have nearly five times more rule violations than those written by *Docker* experts. They observed that best practices and rules for *Docker* have arisen, but that *Docker* developers are often unaware of them.

Beyond quality and evolution aspects, several studies focused on the security of *Docker* image containers. Gummaraju et al. [10] analysed *Docker Hub* images in order to understand how vulnerable they are to security threats. One of their main findings is that over 30% of the official images contain *high priority* security vulnerabilities. Shu et al. [9] carried out a larger-scale analysis from both community and official repositories. They found that both types of images contain more than 180 vulnerabilities on average; many images have not been updated for hundreds of days; and vulnerabilities commonly propagate from parent to child images. Socchi et al. [18] carried out an empirical analysis of *Docker Hub*'s security landscape. They extracted and analyzed a large amount of metadata and vulnerability reports about certified[3] and verified[4] image repositories on *Docker Hub*. They observed that the introduction of these two kinds of images do not lead to a significant improvement of the overall security of images on *Docker Hub*. They predicted that the average number of unique vulnerabilities caused by system packages contained in images 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.

In earlier work, we conducted an empirical analysis in which we studied the relation between outdated system packages in *Debian*-based image containers, their severity vulnerabilities, and their bugs [11]. Using the concept of technical lag [19], we computed the difference between the outdated system packages and their latest available releases in terms of versions, vulnerabilities and bugs. We found that no *Debian*-based image is free of vulnerabilities and bugs, so deployers cannot avoid them even if they deploy the most recent packages in these images. We also concluded that *Docker* image scanning tools should include metrics like the number of bugs and outdated packages to evaluate the health of the *Docker* images. Later in [12], we evaluated how outdated and vulnerable third-party packages are in 961 official *node*-based images coming from three *Docker Hub* repositories *node, ghost* and *mongo-express*. We found that the presence of outdated *npm* packages in official *node* images increases the risk of security vulnerabilities, suggesting that maintainers of official *Docker*

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

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

images should keep their installed *JavaScript* packages up to date, since official images are used as the basis for creating community images. However, we found that most of the *npm* packages in official images are up-to-date.

2.2. Novel Contributions

The current study differs from previous work in several ways. While most of the previous studies focused on assessing system packages in *Docker Hub*, we focus on the usage of third-party packages that are not related to a particular operating system. Our study significantly extends [12] where we only studied *node* images coming from three official *Docker* repositories, considering a smaller number of vulnerability reports, i.e., 1,099 compared to 3,967 vulnerability reports considered in this study. Moreover, in this article we focus mainly on *community* images since official images are supposed to be maintained by official organizations and are thus expected to be more secure and well maintained. Therefore, the number of vulnerabilities found in the official images is an underestimation of the real number of vulnerabilities that might be found in community images (that depend on them), where more dependencies, activity and development are expected.

Furthermore, we study community images that are based on three base images *node*, *Python* and *Ruby*, and use different datasets to analyze them. For example, to gather package release histories, we rely on the official registries of the package manager and to study the link between different packages within the same image, we rely on *libraries.io* dataset. We also report on installation practices (RQ_0 and RQ_1) and study the images at two time points: (1) at their last observed modification date (RQ_2 and RQ_4); and (2) at the date of their extraction (RQ_3 and RQ_5). The former time point is a good indicator of how maintainers care about the outdatedness and vulnerabilities of their images when they produce them. Therefore, the analysis at that time point informs on how much better the producers of those images could have performed. The latter time point reflects the state of those images at production time. This is a good measure of the quality of contained third-party packages if the images were to be used for production at the moment we extracted them.

3. Considered Dataset

This section presents the method used to obtain a representative dataset of community *Docker Hub* images and the third-party packages used within them. The followed process is depicted in Figure 1. More specifically, the steps are:

1. **Identifying candidate images:** as it is our goal to analyze community images containing third-party packages coming from the repositories *npm*, *PyPI* and *RubyGems*, we have to assure that the considered images run and make use of such packages.

2. **Extracting the installed packages:** we pull and run the candidate images locally and extract the installed third-party package releases.

3. **Collecting package releases history:** to be able to assess the outdatedness of images, we use the registries of *npm*, *PyPI* and *RubyGems* and collect the list of all package releases found through the previous step.

4. **Collecting security vulnerabilities:** using a snapshot of vulnerability reports obtained with permission from Snyk [5], we identify all known vulnerabilities that affect the package releases installed in the analyzed images.

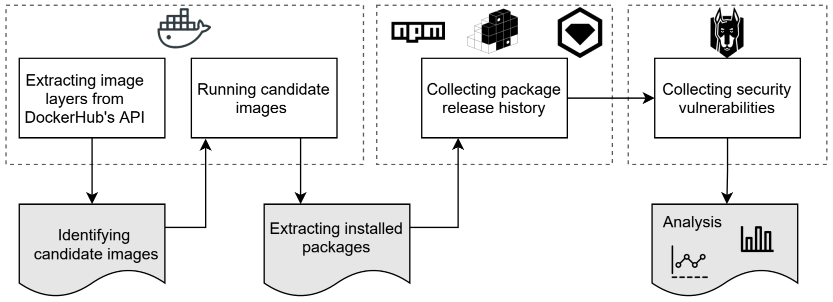
Extracting image layers from DockerHub's API
Running candidate images
Collecting package release history
Collecting security vulnerabilities
Identifying candidate images
Extracting installed packages
Analysis

Figure 1: Process and setup followed to obtain a representative dataset of community *Docker Hub* images and their installed third-party packages.

3.1. Identifying Candidate Images

The candidate *Docker Hub* images for this study are the community images that make use of *npm*, *PyPI* or *RubyGems* packages. We focus on these packages for three reasons: (1) their widespread use (e.g., *npm* is by far the largest package manager in terms of number of hosted packages and *Python* is the most wanted programming language in 2020 [3]) and prevalence among *Docker Hub* images; (2) their usage of semantic versioning [20] which allows us to compare between installed and released package version numbers; and (3) their releases have sufficient vulnerability reports in the *Snyk* database. To ensure that we only download images with such packages, we rely on *Docker*'s inheritance mechanism used by previous studies [11, 12]. Every *Docker* image is built from a Dockerfile which contains a set of instructions (e.g., `FROM`, `RUN`), each creating a layer that represents an intermediate image that is defined by a unique ID (i.e., layer signature). When an image is derived from another image (using `FROM`), it inherits all its layers with their IDs.

To retrieve candidate images, we identify the layers of all images available in the official repositories of *node* [21], *Python* [22] and *Ruby* [23] using *Docker Hub*'s API. We extract the layers of all available community images tagged with `latest` on *Docker Hub*'s registry and check whether these images contain

[5]`https://snyk.io`

the layers of the predefined base images. We found 86,248 images with such characteristics, of which 61,007 *node* images, 21,774 *Python* images and 3,467 *Ruby* images. To do a fair comparison, we only consider the top-1,000 most frequently pulled (i.e., most downloaded using "*docker pull*") images from each group (3,000 in total). These images are representative since they were last updated between June 2016 and December 2019 and they cover 86% of the total number of pulls for *node* images, 96% for *Python* images, and 92% for *Ruby* images. In total, this corresponds to 878M pulls out of a total of 972M pulls for all candidate images. All selected images make use of either the *Debian* or the *Alpine* operating system, with the exception of 8 images that make use of *Ubuntu*. Because of their small number, we therefore replaced these 8 *Ubuntu* images by other popular images that make use of *Debian* or *Alpine*. Table 1 shows the number of images considered for this analysis per base image and operating system.

Table 1: Considered *Docker Hub* images grouped by base image and operating system.

OS	**Base image**		
	node	***Python***	***Ruby***
Debian	597	554	748
Alpine	403	446	252

We also found that the number of considered *Docker* images increases over the years because of the increasing popularity of *npm*, *PyPI* and *RubyGems* packages. Nearly half of the considered images (457 for *node*, 534 for *Python* and 457 for *Ruby*) were last updated in 2019.

3.2. Extracting the Installed Packages

To determine the installed packages in considered images, we pulled and ran them locally on 12 December 2019. Afterwards, we executed the appropriate commands to list the installed packages from each considered package manager. More specifically, to identify which *npm* packages are installed in *node*-based images, we used the command "*npm ls -g*" [24]. To identify *PyPI* packages installed in *Python*-based images we used the command "*pip freeze*" [25] with versions *pip2* and *pip3* of the *pip* package manager. To extract installed *RubyGems* packages from *Ruby*-based images, we used the command "*gem list*" [26].

3.3. Collecting Package Release History

Each package manager usually has a reference package repository such as *npm* [6] for *npm*, *PyPI* [7] for *pip* and *RubyGems* [8] for *RubyGems*, where all package releases are stored. To determine if an installed package release is

[6]`https://registry.npmjs.org/{PACKAGE}`
[7]`https://pypi.org/pypi/{PACKAGE}/json`
[8]`https://rubygems.org/api/v1/versions/{PACKAGE}.json`

outdated, we compared it against the list of all package releases available in the corresponding package repository. To do so, we used the API of each package repository and extracted the available package releases.

We found that the considered *Docker Hub* images make use of 28,363 distinct package releases coming from 12,208 distinct packages. Table 2 shows the breakdown of these packages and their installed releases by package repository.

Table 2: Number of distinct packages and package releases found installed in *Docker Hub* images, grouped by package repository.

	npm	***PyPI***	***RubyGems***
# distinct packages	5,123	3,867	3,218
# distinct package releases	10,435	8,656	9,272

3.4. Collecting Security Vulnerabilities

To identify whether *Docker Hub* images suffer from vulnerabilities affecting third-party packages installed in them, we use the database of vulnerability reports collected by *Snyk* [9], a continuous security monitoring service which has the biggest database of third-party package vulnerabilities. More specifically, we rely on vulnerability reports that are discovered and published before April 12th 2020, the day when we received a snapshot of the vulnerability database from *Snyk*. Note that this dataset also includes vulnerabilities that were published after the date when we extracted the installed packages (December 12th 2019). We include these vulnerabilities since they are not different from the other vulnerabilities, except that they were published or discovered after the package extraction date. Each vulnerability report contains information about the affected package, the range of affected releases, the severity of the vulnerability, its origin (i.e., the package manager of the vulnerable package), the date of its disclosure, and the date when it was published in the database.

To determine if an installed package release is vulnerable, we compare its version number to the range of version numbers affected by a certain vulnerability and specified in the vulnerability reports. From an original dataset of 3,967 vulnerabilities (i.e., 55.6% affecting *npm* packages, 27.1% affecting *PyPI* packages and 17.3% affecting *RubyGems* packages), we only found 632 vulnerabilities (i.e., 16% of the entire dataset) that affect the packages installed in *Docker Hub* images. Of these vulnerabilities, 272 affect *npm* packages, 152 affect *PyPI* packages and 208 affect *RubyGems* packages. Figure 2 shows the number of vulnerabilities with respect to their severity (low, medium, high, critical) and package repository.

[9]`https://snyk.io/vuln`

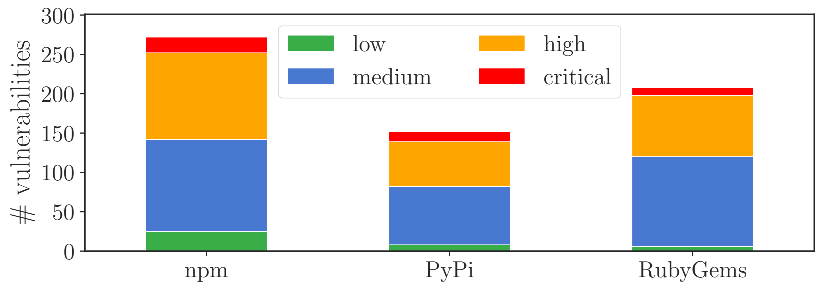

Figure 2: Number of vulnerabilities affecting 5.4%, 3.7% and 6% of distinct *npm*, *PyPI* and *RubyGems* package releases installed in *Docker Hub* images, grouped by severity and package repository.

4. Empirical Analysis Results

This section answers the research questions defined in Section 1 by means of empirical analyses and comparisons between third-party packages installed in *node*-based, *Python*-based and *Ruby*-based community *Docker Hub* images. As part of the analyses, we carry out statistical comparisons using the *Mann-Whitney U* test, a non-parametric test where the null hypothesis H_0 states that there is no difference between two distributions. For all statistical tests, we set a global confidence level of 99%, corresponding to a significance level 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[10].

If the null hypothesis can be rejected, we report the *effect size* with *Cliff's delta d*, a non-parametric measure that quantifies the difference between two populations beyond the interpretation of p-values. Following the guidelines of [27], 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 $|d| \in [0.474, 1]$.

All code and data required to reproduce the analysis in this article is available in a replication package [11].

RQ_0 : How prevalent are installed third-party packages in Docker Hub images?

This research question analyses and compares how many package releases are installed in *Docker Hub* images. The violin plots in Figure 3 show the distribution of the number of installed package releases in community images. *node*-based images have many more installed package releases than the other images, with a median number of 717 installed package releases, compared to 17 for *Python* and 42 for *Ruby* images.

[10]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 $n = 18$, i.e., $p < 0.00055$.
[11]`https://doi.org/10.5281/zenodo.4075073`

The violin plots do not reveal a clear difference between *Alpine* and *Debian* images in terms of installed package releases. To confirm our observation, we carry out a *Mann-Whitney U* test. The null hypothesis assumes that the installed package releases distributions in *Alpine* and *Debian* images are identical. For the pairs of (*node-Alpine*, *node-Debian*) and (*Python-Alpine*, *Python-Debian*) images, H_0 was rejected with small effect size ($|d| \leq 0.22$) when comparing the distributions of their installed package releases. However, H_0 could not be rejected for the pair of (*Ruby-Alpine*, *Ruby-Debian*) images. Given that, at best, we only found a small effect size, the number of installed third-party package releases does not seem to be related to the used operating system.

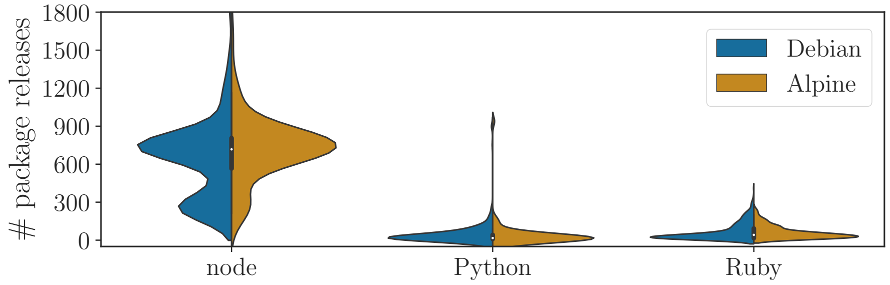

Figure 3: Distribution of the number of installed package releases in *Docker Hub* community images, grouped by base image and image operating system.

We also investigated whether some package releases are installed multiple times in community images. We did not find duplicated package releases in *Ruby* and *Python* images. However, nearly half (47%) of the installed package releases in *node* images are duplicated ones. After removing duplicates, we obtained a median number of 377 distinct installed *npm* package releases in *node*-based images. This is much lower than the median number of 717 when considering all releases. The top four *npm* packages that have duplicate package releases in nearly all images (i.e., in more than 92.8% of the images) are glob, core-util-is, minimatch and readable-stream.

This difference in number of duplicate installed package releases is explained by the installation approach followed by the package managers. *npm* allows installing the dependencies of a package release under a nested sub-directory independently of the other installed package releases, even if there are some package releases that share the same set of dependencies [12]. This is not allowed by *pip* and *RubyGems*. However, there is a way to flatten the *npm* dependency tree of installed packages by sharing the common dependencies and removing the duplicated ones using the command "*npm dedupe*" [13]. For the rest of the

[12]For more details: https://medium.com/learnwithrahul/understanding-npm-dependency-resolution-84a24180901b

[13]`https://docs.npmjs.com/cli/dedupe`

article, installed package releases will be computed and considered only once, ignoring duplicates. Table 3 reports the number of installed package releases.

Table 3: Characteristics of the number of installed package releases in *Docker Hub* images, grouped by base image.

base image	mean	min	median	max
node	705.5	127	717	5,129
node (distinct package releases)	373.0	101	377	1,401
Python	45.2	1	17	939
Ruby	65.3	7	42	410

Note that *Docker* images containing duplicate identical package releases are different from images containing multiple distinct releases of its installed packages. The latter might exist and be necessary in order to avoid conflicts between required dependencies. We found that 862 of all installed *npm* packages (16.8%), 190 of *Ruby* packages (5.9%) and only 15 of *PyPI* packages (0.4%) have been installed with distinct releases within the same images.

To better understand how the same packages are installed with different releases, we computed the number of distinct packages installed in each image, without considering whether they are installed with different releases or not. Table 4 reports the distribution of distinct third-party packages in *Docker Hub* images. The median number of installed packages in *node* images is 7.2% less compared to installed package releases (350 versus 377). We found that all *node* images, 52.1% of *Ruby* images and only 1.3% of *Python* images had the same packages installed with distinct releases. This explains a lower *median* number of installed *node* and *Ruby* packages (Table 4) compared to the number of installed package releases (Table 3). It is expected to find the same packages installed with different releases in *node* and *Ruby* images, as it is allowed by the package managers *npm* and *RubyGems*. It is more surprising to find such cases for *PyPI* images, since the package manager *pip* does not allow installing the same package with multiple releases. A deeper investigation showed that this small proportion (i.e., 0.4%) of *PyPI* packages were installed using different versions of the *pip* package manager (i.e., *pip2* and *pip3*). This is possible since it is permitted to have the two major versions of *pip* installed within the same image.

Table 4: Characteristics of the number of installed distinct packages in *Docker Hub* images, grouped by base image.

base image	mean	min	median	max
node	340.8	100	350	1,108
Python	45.17	1	17	939
Ruby	63.8	7	41	406

Findings

node images have over eight times more installed package releases than *Python* or *Ruby* images. The number of installed third-party packages is not related to the used operating system. Every *node* image includes several distinct releases of a same package.

RQ_1 Do image maintainers add additional packages to their base Docker Hub images?

To create a properly working community image, maintainers tend to add additional packages to the base images from which they derive their community images, allowing them to satisfy all dependencies that are required by the software applications that their images are packaging. RQ_1 therefore aims to study this subset of added packages in community *Docker Hub* images, distinguishing between two kinds of third-party packages: *core* and *non-core*. Core packages refer to those packages that are inherited by the community image from its official base image, while non-core packages refer to the subset of packages that are added on top of the base image.

We identified the core packages installed in the base images *node:latest, python:latest* and *ruby:latest*. We found 370 core packages for *node:latest*, while *ruby:latest* contains 55 core packages, which are the standard ones that come with *Ruby* [14]. The only core packages that were installed in *python:latest* are the default ones: *pip, wheel* and *setuptools*.

Figure 4 shows the distribution of the number of *non-core* package releases that remain in the community images after removing all releases corresponding to the aforementioned core packages. The violin plots reveal that the distributions for *node*, *Python* and *Ruby* are much more similar, different from what we observed in Figure 3. The median number of non-core package releases is 26 for *node*, 17 for *Python* and 15 for *Ruby* images. We also found 69 *node* images and one *Ruby* image that did not add any non-core package.

We performed pairwise comparisons of these three distributions with a *Mann-Whitney U* test and could reject the null hypothesis in all cases (after Bonferroni correction) implying that the distributions are statistically different. As summarised in Table 5, the effect size was, however, negligible to small for all conducted comparisons. In the comparison between *Python* and *Ruby* images, the effect size is in favour of *Python* images. This is different from what we noticed in RQ_0 where *Ruby* images appeared to have more package releases than *Python* images.

Next, we investigated whether the number of non-core package releases in community images relates to the year when the image was last updated. Figure 5 shows the distribution of the number of non-core package releases in community images, grouped by base image and last update year. In the case of *Python*

[14]This set of packages is maintained by *Ruby* core: `https://stdgems.org/`

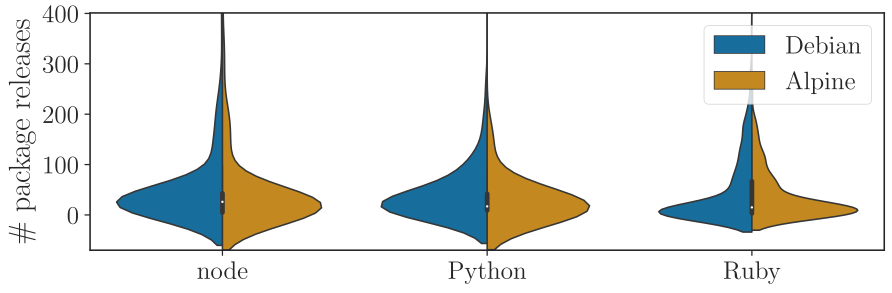

Figure 4: Distribution of the number of non-core package releases in *Docker Hub* community images, grouped by base image and operating system.

Table 5: Mann-Whitney U test and effect size results for pairwise comparisons between distributions of number of non-core package releases used in *node*, *Ruby* and *Python* images.

population A	**direction**	**population B**	**effect size**	**|d|**
node	>	***Python***	negligible	0.06
node	>	***Ruby***	small	0.19
Python	>	***Ruby***	negligible	0.12

and *Ruby* images, the number of added package releases is slightly increasing over time. For *node* images, however, we observe that the number of non-core *npm* package releases is *decreasing* over time. This is surprising since the *npm* repository is getting larger each day; by June 2019 *npm* contained over one million packages. An in-depth investigation [15] reveals that, over time, more and more *npm* packages are being included as core packages in the *node* base image. It is likely that maintainers of older images needed to add more non-core packages themselves, while maintainers of recent images found most of these packages already installed in the base image, reducing their need to add them manually to their images.

Table 6 shows the top three non-core third-party packages that we found installed by most of the community *Docker* images, grouped by their package repository. The *npm* package fstream is one of the *Streams* packages in *npm* that handle end-to-end information exchange (e.g., writing files), and it is maintained by *npm*. The *PyPI* package six is a package that helps to create *Python* code that is compatible on both *Python* 2 and *Python* 3 versions. The most used *RubyGems* package did_you_mean is a package that helps developers to avoid typos when creating code.

When installing a package release in a *Docker* image, all its required dependencies will be installed within the image as well. For this reason, we computed how many of the added non-core packages might have been installed without

[15]This extra analysis can be found in the replication package of this paper.

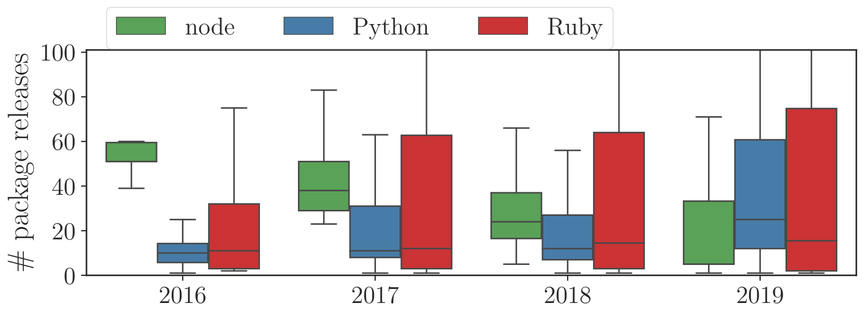

Figure 5: Distribution of the number of non-core package releases in *Docker Hub* community images, grouped by base image and the year of last update to the image.

Table 6: Top three of third-party packages and the proportion of *Docker* images that make use of them, grouped by package repository.

package repository	package	% images
npm	fstream	90.3
	block-stream	90.2
	builtin-modules	85.2
PyPI	six	73.3
	requests	63.1
	urllib3	62.6
RubyGems	did_you_mean	90.4
	rubygems-update	84.8
	rack	40.5

being dependencies of other packages. This allows us to know which packages are installed individually by the maintainers as top-level packages and not automatically by the package manager as dependencies of other packages [16]. To do so, we relied on (the latest) version 1.6.0 of the Libraries.io Open Source Repository and Dependency Dataset [28] containing metadata of all package dependencies from 37 different package repositories, including *npm*, *PyPI* and *RubyGems*. We found that from 11,607 non-core packages, 5,495 (47%) might have been installed as top-level packages. More specifically, the median number of top-level packages is 9 for *node* images, 8 for *Python* images and 4 for *Ruby* images.

[16]Note that a dependency that is automatically installed is also mandatory in the image since it is required by the package that has been manually installed.

Findings

The number of non-core packages in community images is similar for *node*, *Python* and *Ruby* images. Over time, *Python* and *Ruby* images have included more non-core packages, while *node* images have included fewer of them, due to an increasing number of core packages offered by the *node* base image.

RQ_2 Do image maintainers include outdated third-party packages in their last image update?

This research question investigates whether image maintainers release community images with outdated packages. For each image, we collect the package releases that were installed in it when it was last updated and compare them to the list of available package releases at that time.

Figure 6 shows the proportion of outdated third-party packages in the community images, grouped by their type (core or non-core). We observe similar distributions of outdated packages, with a median proportion of 50% outdated non-core packages and a total of 323 images having *all* their non-core packages outdated; whereas a median of 58% of core packages were outdated, and 201 images had all of their core packages outdated. For all images (with the exclusion of *Python* images that do not have core packages) at least 20% of their included core packages were outdated when they were last updated. This suggests that maintainers ship their community images with outdated (core and non-core) third-party packages. It also suggests that image maintainers do not update their inherited core packages when they rely on other (base) images. We used a *Mann-Whitney U* test to verify if the proportions of outdated core and non-core packages were statistically different. H_0 was rejected with small effect size ($|d| = 0.26$) in favour of outdated core packages. We therefore conclude that *Docker* images tend to have more outdated core packages than non-core ones.

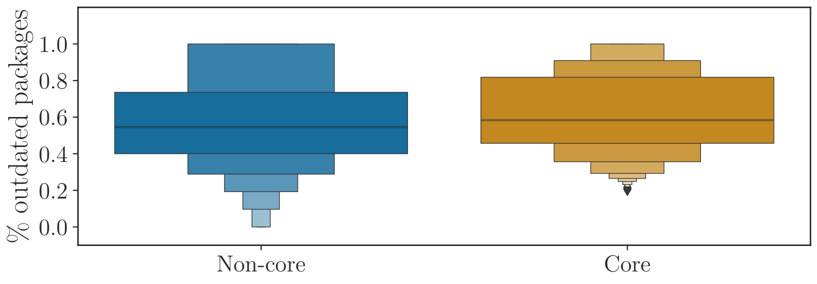

Figure 6: Boxen plots of the distribution of the proportion of *outdated* third-party packages in community images (computed at the date of the last image update), grouped by their type (core or non-core).

Since more than half of the (combined core and non-core) packages in community images are outdated, we aimed to quantify how much outdated they are. To gain insights about the type of changes missed by such outdated packages, we define four degrees of outdatedness based on the semantic versioning policy [20]:

1. *up-to-date*: the package is up-to-date;
2. *patch*: the package is only missing patch updates;
3. *minor*: the package is missing at least one minor but no major updates;
4. *major*: the package is missing at least one major update.

Figure 7 shows the distribution of the proportion of **non-core** packages according to their degree of outdatedness, grouped by base image. We observe a higher proportion of up-to-date packages for *Python* images (median of 66.5%) compared to *node* and *Ruby* images (median of 48.6% and 41.2%, respectively). We also observe that the majority of outdated non-core packages in *node* images are missing a *major* update (median of 40% of installed packages), against 4.1% that are missing a *minor* update and 4.4% that are missing a *patch* update. For *Python* and *Ruby* images, the highest proportions are for *minor* outdated packages with a respective median of 20% and 22% of the packages installed in a community image.

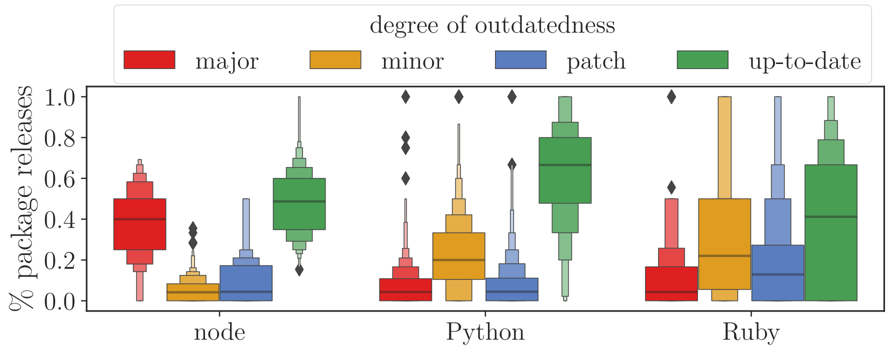

Figure 7: Boxen plots of the distribution of the proportion of third-party **non-core** package releases in community images (computed at the date of the last observed image's update) per degree of outdatedness, grouped by base image.

Similarly, Figure 8 shows the distribution of the proportion of **core** packages according to their degree of outdatedness, grouped by base image (*node* or *Ruby*). We do not show the distribution of *Python* core packages since, as previously mentioned, *Python* images do not have any core packages. We observe that *node* core packages have similar degrees of outdatedness as non-core packages (Figure 7), while more differences can be observed between core and non-core packages for *Ruby*. *node* images have a much higher proportion of up-to-date core packages (median of 50.8%) compared to *Ruby* images (median of 18.1%). We also observe that the majority of outdated core packages in *node* images are missing a *major* update (median of 26.1% of installed packages), against 10.1% that are missing a *minor* update and 9.9% that are missing a *patch* update. For *Ruby* images, the highest proportions are for *minor* outdated packages with a median of 37.5% of the packages installed in a community image. This reveals that the difference in outdatedness that we can see in Figure 6 is caused by *Ruby* packages.

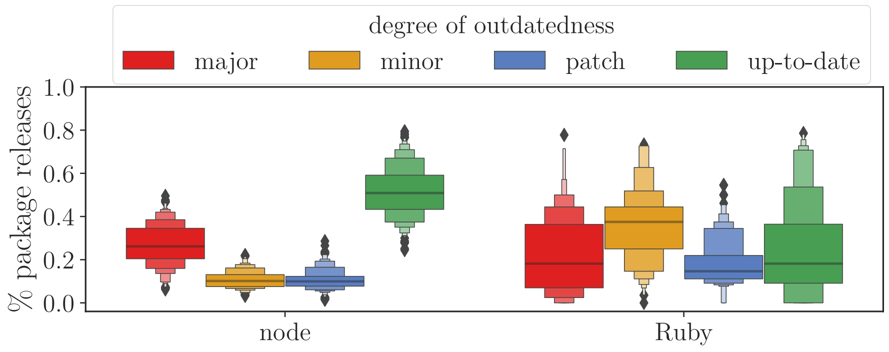

Figure 8: Boxen plots of the distribution of the proportion of third-party **core** package releases in community images (computed at the date of the image's last update) per degree of outdatedness, grouped by base image.

In a similar way we analyzed package outdatedness based on the time of the last update to the image [17]. Older *node* images tend to have more up-to-date packages than recent ones, while *Ruby* images follow an opposite trend. We also observed that the proportion of outdated *major* packages is increasing over time in the case of *node* images. We could not observe any clear trend in the case of *Python* images.

Findings

The majority of third-party packages in community images were already outdated at the time of the last update to the image. Core packages in images are more outdated than non-core ones. At their last modification, older *node* images contained fewer outdated packages than more recent ones, while the opposite was observed for *Ruby* images.

[17]Figures and detailed results of this analysis of the evolution of package outdatedness over time, can be found in our replication package.

RQ_3 How outdated are third-party packages in Docker Hub images?

While the analysis of RQ_2 helps to create awareness about outdatedness among maintainers of community images at the time they last updated their images, RQ_3 helps the users of such images to assess the outdatedness of the available (and popular) images they can rely on now. More specifically, RQ_3 studies how outdated third-party packages are in images as if they were deployed at the date the analysis was conducted (i.e., 12 December 2019).

Figure 9 shows the median proportion of packages in community images per degree of outdatedness, grouped by the year in which their image was last updated and by the base image. We observe a difference in the proportions of up-to-date and outdated packages between images derived from different base images. *node* images seem to have higher proportions of up-to-date packages than the other images, except in 2019 when all images seem to have similar proportions of up-to-date packages. Older images have higher proportions of outdated packages, which is expected since they are missing new package releases that were created after their last update. Considering the whole period (and without differentiating between outdated packages), we found a median proportion of up-to-date packages of 33.6% for *node*, 21% for *Python* and 15.4% for *Ruby* images. This shows that *Ruby* images have higher proportions of outdated packages than the other images. However, it is important to mention that, while *node* images have more up-to-date packages, they also have the highest proportion of packages missing *major* updates. In contrast, the majority of outdated packages in *Python* and *Ruby* images only miss *minor* updates.

Findings

Older community images have more outdated third-party packages than recent images. *node* images have the highest proportions of packages missing *major* updates.

RQ_4 Do image maintainers include third-party packages with known vulnerabilities in their images?

Security vulnerabilities can be exploited to abuse *Docker* images or the environment in which these images are deployed. Fixing them typically requires experienced developers [29]. Verifying *Docker* images for vulnerabilities is therefore essential. RQ_4 relies on our dataset of vulnerability reports to study if the images that *Docker* maintainers create contain vulnerable third-party packages with known vulnerabilities at their last modification date.

To identify which vulnerabilities were known before the last update to an image, we rely on the disclosure date of the vulnerability. We consider all vulnerabilities that are affecting the package releases installed in *Docker* images and that have been disclosed before the date of the image last update.

In total, the analyzed packages in *Docker* images suffered from 441 known vulnerabilities when images were last updated, 165 for *npm*, 95 for *PyPI* and 181 for *RubyGems* packages. 30 of these vulnerabilities were of *low* severity,

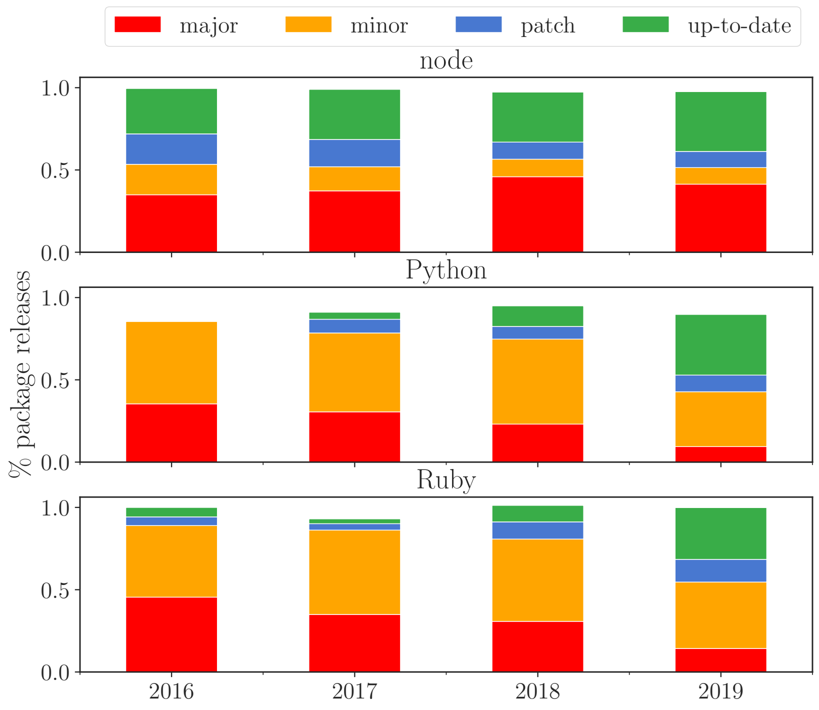

Figure 9: Median proportion of third-party packages per degree of outdatedness in community images on 12 December 2019, grouped by image last update year and base image.

216 *medium*, 166 *high* and 29 were *critical*. These vulnerabilities were affecting 245 of the installed packages (and a total of 789 distinct package *releases*), 111 from *npm*, 49 from *PyPI* and 85 from *RubyGems*. We also found that 74% (i.e., 2,219) of the images were affected by these vulnerabilities, 42.8% of *node*, 12.2% of *Python* and 45% of *Ruby* images.

Figure 10 shows the distribution of the number of known vulnerabilities found in community images and affecting third-party packages, grouped by base image and vulnerability severity. We observe that *node* images have the highest numbers of vulnerabilities while *Python* images have the lowest numbers. We also observe that most of the vulnerabilities affecting images are of a *medium* or *high* severity. Without differentiating between vulnerabilities, *node*, *Python* and *Ruby* images have a median number of 4, 1 and 6 vulnerabilities, respectively.

We compared the vulnerability distributions in *node*, *Python* and *Ruby* images using a *Mann-Whitney U* test. We found a statistically significant difference (after Bonferroni correction) between the pairs of distributions (*Python*, *node*) and (*Python*, *Ruby*) in terms of number of vulnerabilities. As summarised in Table 7 we found a large effect size for these pairs. This indicates that *Python* images have significantly less vulnerabilities than *node* or *Ruby* images. For the

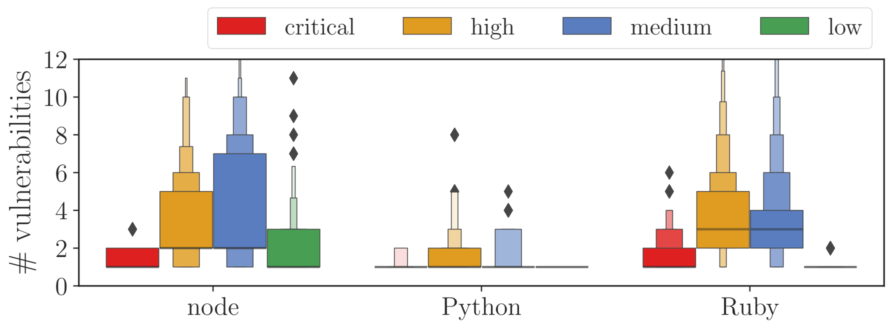

Figure 10: Distribution of the number of known vulnerabilities affecting community images when they were last updated, grouped by base image and vulnerability severity.

pair (*node*, *Ruby*), H_0 was rejected with a negligible effect size, implying that we could not observe a difference in number of vulnerabilities between *node* and *Ruby* images when they are last updated.

Table 7: Mann-Whitney U test and effect size results for pairwise comparisons between vulnerability distributions of *node*, *Ruby* and *Python* images.

population A	direction	population B	effect size	|d|
node	>	***Python***	**large**	0.74
Ruby	>	***Python***	**large**	0.69
Ruby	-	***node***	negligible	0.06

Findings

At their last modification date, *Python* images had considerably fewer vulnerabilities than *node* and *Ruby* images.

We found that *Docker* community images suffer from 103 types of vulnerabilities. Table 8 shows the most prevalent vulnerability types that were affecting *Docker* images when they were last updated. We observe that the vulnerability *Arbitrary Code Execution (ACE)* affects nearly all of *Ruby* images, while the most prevalent vulnerability in the case of *node* images is *Denial of Service (DoS)* affecting nearly two thirds of the images. For *Python*, we could not find any vulnerability types affecting more than 8% of the images, the most prevalent vulnerability is *Information Exposure* affecting only 7.7% of *Python* images. We have also found that *Denial of Service (DoS)* is the only common prevalent vulnerability between *node*, *Python* and *Ruby* images. As can be noticed in the table, there are some cases where only a small number of packages is responsible for high proportion of images, e.g., 40.3% of *node* images are suffering from the *Time of Check Time of Use (TOCTOU)* vulnerability because of only one unique core package chownr.

Table 8: Top three vulnerability types affecting most images with the number of unique packages inducing them and the proportion of *Docker* images that were suffering from them at their last update, grouped by base image.

base image	vulnerability type	% images	# packages
node	Denial of Service	64.7	11
	Regular Expression DoS	54.2	38
	Time of Check Time of Use	40.3	1
Python	Information Exposure	7.7	8
	Arbitrary Code Execution	5.2	7
	Denial of Service	4.8	8
Ruby	Arbitrary Code Execution	99.3	5
	Cross-site Scripting	39.2	23
	Arbitrary Command Execution	35.2	1

Comparing *core* and *non-core* third-party packages, we found that only 26 of the *core* packages are suffering from the vulnerabilities, while they are responsible for 49.3% of the propagated vulnerabilities in the community images. This is different from *non-core* packages where 221 of them cause the rest (i.e., 50.7%) of the propagated vulnerabilities in the images. This difference is explained by the fact that *core* packages are used by almost all images, therefore a vulnerability affecting only one *core* package release will affect all images making use of it.

Findings

Because of their widespread use across all images, core packages are responsible for nearly half of the vulnerabilities in community images.

We found that outdated packages are responsible for the majority of vulnerabilities present in the studied *Docker* images. We inspected the latest package releases for vulnerabilities and compared them to the package releases used in *Docker* images. We found that 69.9% of the vulnerabilities in images could have been avoided if the concerned images would have used the latest available releases of their outdated packages. Indeed, more recent package releases come with bug and vulnerability fixes and thus tend to have less vulnerabilities than older ones [30, 11, 31].

Recommendation

Maintainers of *Docker Hub* images could reduce more than two thirds of the vulnerabilities by updating their outdated packages to the latest available releases before publishing their images.

RQ_5 : How vulnerable are third-party packages in Docker Hub images?

While RQ_4 only studied vulnerabilities that were known before images last update, RQ_5 evaluates how vulnerable images are if they were deployed at their extraction date. Using the same vulnerability dataset, we analyzed the vulnerable package releases (both core and non-core) installed in community images and identified the most prevalent vulnerabilities.

In total, the analyzed packages in community images suffer from 632 distinct reported vulnerabilities, of which 272 for *npm* packages, 152 for *PyPI* and 208 for *RubyGems*. 39 of these reported vulnerabilities are of a *low* severity, 305 are *medium*, 245 are *high* and only 43 are *critical*. The vulnerabilities are affecting a total of 317 installed packages (and a total of 1,442 distinct package *releases*), of which 158 from *npm*, 65 from *PyPI* and 94 from *RubyGems*. All *node* and *Ruby* images are affected by these vulnerabilities, while only 814 (i.e., 81.4%) *Python* images are affected. Table 9 summarises these results.

Table 9: Breakdown of distinct number of reported vulnerabilities, their severity, the number of installed packages affected, the number of package releases affected and the number of community images affected by the vulnerabilities.

# metric	*node*	*Python*	*Ruby*	All
low vulnerabilities	25	8	6	39
medium vulnerabilities	117	74	114	305
high vulnerabilities	110	57	78	245
critical vulnerabilities	20	13	10	43
all vulnerabilities	**272**	**152**	**208**	**632**
affected packages	158	65	94	317
affected package releases	562	325	555	1,442
affected community images	1,000	1,000	814	2,814

For each image, we computed the number of vulnerabilities that its installed packages are suffering from. Figure 11 shows the distribution of the number of vulnerabilities found in community images and affecting third-party packages, grouped by base image and vulnerability severity. Similar to Figure 10 we observe that *node* images have the highest numbers of vulnerabilities while *Python* images have the lowest numbers. We also observe that most of the vulnerabilities affecting images are of *medium* or *high* severity. Without differentiating between vulnerability severities, we found that *node*, *Python* and *Ruby* images have a median number of 19, 2 and 14 vulnerabilities, respectively.

We performed a pairwise comparison between the distributions of number of vulnerabilities in installed *node*, *Python* and *Ruby* images. The *Mann-Whitney U* test was rejected in all cases (after Bonferroni correction), revealing a statistically significant difference between all pairs of distributions. As summarised in Table 10, there was a large effect size for the pairs (*Python*, *node*) and (*Python*, *Ruby*), and a medium effect size for the pair (*node*, *Ruby*). This shows how *node* images accumulate more vulnerabilities compared to *Ruby* images, since we could not find a statistical difference in number of vulnerabilities between these two populations of images when they were last updated (see RQ_4).

Several months after their last update, community images have been shown to accumulate more outdated packages and thus suffer from more vulnerabilities [11, 30]. For this reason we observed a higher number of vulnerabilities in

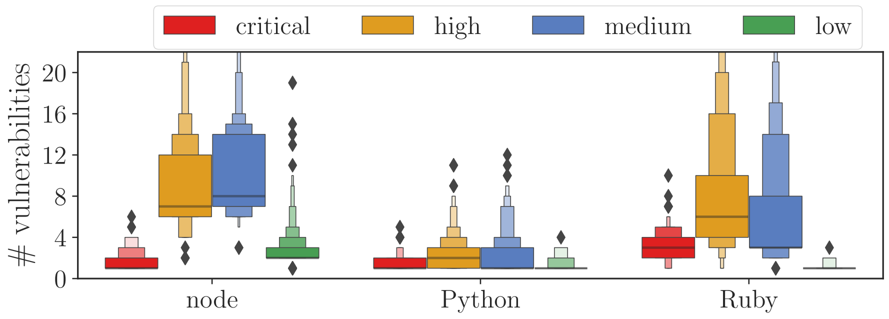

Figure 11: Distribution of the number of vulnerabilities affecting community images, grouped by base image and vulnerability severity.

Table 10: Mann-Whitney U test and effect size results for pairwise comparisons between distributions of number of vulnerabilities in installed *node*, *Ruby* and *Python* images.

population A	**direction**	**population B**	**effect size**	**|d|**
node	>	***Python***	**large**	0.96
Ruby	>	***Python***	**large**	0.88
node	>	***Ruby***	medium	0.37

RQ_5 compared to what was found in RQ_4. Moreover, we found that community images suffer from 125 distinct types of vulnerabilities (22 more than RQ_4). Table 11 shows the most prevalent vulnerability types affecting images [18]. All *node* images suffer from the *Prototype Pollution* vulnerability and nearly all of *Ruby* images suffer from the *Arbitrary Code Injection* and *Arbitrary Code Execution* vulnerabilities, while the most prevalent vulnerability type in *Python* images is *Denial of Service* affecting 39.3% of the images.

Similarly to RQ_4, only a small number of 31 *core* packages is affected by the vulnerabilities. These packages are responsible for 50.4% of the vulnerabilities in community images, while the remaining 49.6% of vulnerabilities are found in the 287 *non-core* packages.

Findings

At their extraction date, *node* images exhibit the most vulnerabilities while *Python* images exhibit the fewest. Community images without new updates may accumulate more outdated packages over time, and thus suffer from more vulnerabilities than at their last update. Core packages are responsible for 50.4% of the vulnerabilities in community images. *node*, *Python* and *Ruby* images have different types of vulnerabilities.

[18]Note that the vulnerabilities in Table 11 are different from the ones in Table 8

Table 11: Top three vulnerability types affecting most images, and the proportion of community images that are suffering from them, grouped by base image.

base image	vulnerability type	% images	# packages
node	Prototype Pollution	100.0	17
	Arbitrary File Overwrite	99.7	6
	Arbitrary File Write	99.5	3
Python	Denial of Service	39.3	14
	Information Exposure	27.0	8
	Insufficient Randomness	21.7	1
Ruby	Arbitrary Code Injection	99.6	5
	Arbitrary Code Execution	99.6	5
	Denial of Service	94.6	13

The proportion of vulnerabilities in images could be reduced by 79.1% of all vulnerabilities if these images would update their outdated packages to the latest available releases. Broken down by base image, this would be 86.8% for *node* images, 64% for *Python* and 71.4% for *Ruby* images.

Recommendation

Maintainers of community images could reduce nearly 80% of their vulnerabilities by updating their outdated packages to the latest available releases before publishing their images.

5. Discussion

In addition to operating system packages, *Docker* images include other third-party packages that are needed by the software they ship as a container. Usually, these third-party packages come from language-specific package repositories such as *npm*, *PyPI* and *RubyGems*. Including these packages in *Docker* images should be treated with care since non-useful and harmful ones could lead to larger-sized and more vulnerable images. For this reason, we decided to study the usage of third-party packages of three popular programming languages in community *Docker Hub* images.

In response to RQ_0 we found that *node* images have more installed packages than *Python* and *Ruby* images, and contain many duplicate package releases. This might be explained by the observation [13] that *JavaScript* applications tend to require more dependencies than applications written in *Python* or *Ruby*. The same observation might also explain why *node* images tend to have a larger size [32].

Recommendation

Maintainers of *Docker Hub node* images could reduce the relatively large number of installed npm packages by eliminating duplicate package releases.

Our study also showed that while *Alpine* is a lightweight Linux operating system, maintainers that make use of it include a similar number of packages as maintainers that use *Debian*. This implies that empirical studies on *Docker* images should consider packages from third-party package managers in addition to the system packages inherited from the relatively small base images.

Lesson learned

Choosing a community image with a different operating system will not affect the number of third-party packages.

After ignoring core packages installed in community images in RQ_1, we found a similar number of installed packages in *node*, *Ruby* and *Python* images. This suggests that the effort spent on these packages by community maintainers could be similar as well, whether they are maintaining a *node*, *Ruby* or *Python* image. We also noticed that the number of core packages is the highest in *node* images. This relates to the lack of a standard library for *JavaScript*, requiring many packages to be included as core packages. According to its creator Brendan Eich, *JavaScript* is not maintained by "*a single person or pair of people who design well like the founders of Unix were or the creators of Perl, Python, and Ruby are.*" Instead, there are "*a bunch of people from different companies. They would do a terrible job if they were in charge of building a big standard library.*"[19]

We also noticed that core packages are more used over time in the case of *node* images, since older images required more dependencies to be added by community maintainers than recent ones. Even if *JavaScript* does not have an extensive standard library, the *node* community seems to keep track of the most useful packages and offers them as core packages in its official images.

Recommendation

Maintainers of official repositories should offer images with the most prevalent dependencies for maintainers of community images to derive from.

In response to RQ_2, nearly half of all (core and non-core) third-party packages in community images were found to be outdated already at the time of the last update to the image. Core packages were more outdated than non-core ones, suggesting that maintainers of community images often forget to update the packages they inherit from their official base images. The proportion of out-

[19]https://www.infoworld.com/article/3048833/brendan-eich-javascript-standard-package-will-stay-small.html

dated packages increases over time as new package releases become available. Indeed, RQ_3 showed that if *Docker Hub* images were deployed at the analysis date, they would suffer from higher proportions of outdated packages. In many cases, and especially in *node* images, several packages are outdated by at least one major release. Our findings show that community images are less concerned about having up-to-date third-party packages than official images, since previous studies showed that outdated *npm* packages in official *node* images have a median of only one missing patch update [12].

Reasons behind this large number of outdated packages in *Docker Hub* images could be related to the nature of *Docker* containers, since by definition the use of container images provides isolation from evolving dependencies and changes in packages that may break working systems. It could be also possible that image maintainers stick to package releases that "just work", even if they are outdated, since upgrading to new versions may by risky and requires effort [33]. In fact, previous studies [34, 35] on other ecosystems showed that developers are not likely to prioritize a library update, as it is perceived as additional work and in many cases developers ignore updates because of a poor awareness of the benefits.

Comparing outdated packages across *Docker* images in RQ_3, we found that most of the outdated packages in *node* images miss at least one major release. For this reason we looked at the history of releases of all packages installed in *Docker* images, and found that *npm* packages have more major updates than *PyPI* or *RubyGems* packages. 4.8% of *npm* package releases are major releases, against 2.9% and 2.8% for *PyPI* and *RubyGems*. We also noticed that *PyPI* and *RubyGems* have higher proportions of pre-1.0.0 version releases (i.e., 0.y.z). 25.6% of *npm* package releases have a pre-1.0.0 version number against 45.8% and 37.6% of *PyPI* and *RubyGems* package releases. This suggests that the difference we found between *Docker* images in terms of major outdated packages could be related to the updating and development procedures of third-party package developers in each ecosystem [20].

Recommendation

Docker Hub users should inspect the outdatedness of packages they inherit from base images before they include packages themselves. Image maintainers should reflect on their updating practices so fewer outdated packages are provided through Docker Hub.

Having outdated packages in community images is not necessarily a problem for image users, since by default *Docker* provides isolation from evolving dependencies and changes in packages that may break working systems. On the other hand, outdated packages may miss vulnerability fixes and thus put *Docker* images and the systems they provide at risk. RQ_4 studied if image maintainers ship their software in images with known vulnerable package releases. We

[20]E.g., packages still in the 0.x.y phase cannot be outdated missing major releases.

found 74% of the images to suffer from vulnerabilities. 69% of these vulnerabilities could have been avoided if maintainers would have updated their packages to the latest available releases before publishing their images on *Docker Hub*. Image maintainers should therefore invest more effort in updating packages to provide more secure images. We also noticed that *Python* images had less vulnerabilities than the other images. This could be explained by the absence of *core* packages for *Python* images (see RQ_1) and the low proportions of outdated *PyPI* packages (see RQ_2), or this could simply mean that *Python* maintainers pay more attention to the security of their packages.

Recommendation

Image maintainers should invest more effort in updating packages to provide more secure community images. Python images are less outdated and hence less vulnerable than other images.

We found that only a small number of non-core packages might have been manually installed as top-level packages in community images (see RQ_1). We therefore think that most of the vulnerabilities come from transitive dependencies. In fact, the security team of *Snyk* has already shown in the *2020 State of Open Source Security Report* that the majority of open source vulnerabilities continue to be discovered in indirect dependencies, i.e., 86% in *npm* and 81% in *RubyGems*. While only 11% of the vulnerabilities have been found in the indirect dependencies of *PyPI*. This is in line with our findings since we found fewer vulnerabilities in *Python* images.

Furthermore, reflecting on the results reported in Table 4 and RQ_1, we only found a median of 2.4% and 9.5% of the *npm* and *RubyGems* (non-core) packages in images that were manually installed by the image maintainers, while *Python* images have a median of 47% of manually installed packages.

Recommendation

Transitive dependencies come with a high number of vulnerabilities. Image maintainers should reduce the number and depth of their transitive dependencies or monitor them alongside the top-level packages.

RQ_5 showed that vulnerable packages were present in nearly all of the studied images at the date of the analysis. This is expected since, as time passes by, images accumulate more outdated packages and more disclosed vulnerabilities. Because of their widespread use and despite their small number, core packages inherited from official base images are responsible for 50% of the vulnerabilities in derived community images. Developers of such packages should therefore be aware that abandoning or not maintaining these packages may have a big impact on community images.

RQ_4 and RQ_5 only revealed a minority of outdated core packages as being vulnerable. This is not surprising since there are many other reasons why packages receive updates: to fix bugs, improve performance, add new functionality, change package metadata and licensing, improve documentation, and so on. Moreover, the number of reported vulnerable outdated packages is an underestimation, as packages may also have vulnerabilities that have not yet been discovered or been included in the vulnerability database.

Recommendation

Docker Hub users should inspect third-party packages for vulnerabilities before relying on community images. Updating third-party packages will reduce the number of vulnerabilities in Docker images.

6. Threats to validity

The empirical nature of our research exposes to potential threats to validity. We present them following the classification and recommendations of [36]. The main threat to *construct validity* comes from imprecision in, or incompleteness of, the data source we used to identify vulnerabilities. We assumed that the *Snyk.io* vulnerability database represents a sound and complete list of vulnerability reports for third-party packages. This may have lead to an underestimation of our results since some vulnerabilities may not have been disclosed yet and therefore are missing from the database.

Another threat to construct validity stems from how we identified installed packages in *Docker* images. We relied on the list of package releases available in *npm*, *PyPI* and *RubyGems*, and we considered a package release in this list as "installed" in an image if it was marked as such by the corresponding package manager i.e., *npm*, *pip* and *RubyGems*. Consequently, we did not identify package releases that are compiled from source code, installed in virtual environments or installed via other package managers like *Yarn* [21]. As such, we may have underestimated the number of installed packages. We do not expect this threat to affect our findings much since we expect most users to install packages via the default package managers of the base images they use.

A third threat to construct validity concerns our choice to remove duplicated package releases installed in images. Not removing such package releases could have led to higher number of (non distinct) vulnerabilities, especially in *node* images. Another similar threat to validity concerns our method to identify which packages could have been installed automatically or manually by the image maintainers. We relied on an analysis of the dependency network of installed packages, however, there is a possibility of developers requiring a package that was already installed as a dependency for another package, or was already installed as a top package in a parent image.

As a threat to *internal validity*, we only relied on a sample of 3,000 *Docker Hub* images for our observations on the usage of third-party packages. These images were the most popular ones in terms of number of pulls. While one may argue that this sample is not representative of all community images making

[21]`https://yarnpkg.com/`

use of the considered third-party packages, the selected images represent 90% of the total number of pulls of all possible image candidates that make use of the considered third-party packages. As a result, we consider the chosen sample of community images to be representative for most users of *Docker Hub* images.

As a threat to *external validity* we cannot claim that our findings generalise to other third-party packages (e.g., *Maven* or *CRAN*). Similarly, our findings cannot be generalised to containerization systems beyond *Docker* and *Docker Hub*, such as *rkt* [22] and *Linux containers* [23], because technology-specific mechanisms (such as layering and inheritance) may have played a role in the observed findings. Nevertheless, because *Docker* is more popular than its competitors[24], our study is relevant and potentially useful to a large community of container users. Still, it would be interesting to compare *Docker* to other containerization technologies in future studies. It would also be interesting to study other packaging technologies that offer the possibility to create an isolated environment in which users can run application software in isolation from the rest of the system like *flatpak* [25].

7. Conclusion

This paper empirically analysed the usage of third-party *JavaScript*, *Python* and *Ruby* packages in *Docker Hub* images. We studied how prevalent, outdated and vulnerable these packages are in community images that are based on *node*, *Python* and *Ruby* base images. We studied 3,000 popular images, i.e., 1,000 from each group of images. We observed that the number of installed third-party packages is not related to the used operating system. However, it is related to the considered base image, i.e., *node*, *Python* or *Ruby*. The installed packages were grouped into two categories, core packages offered by the base image, and non-core packages added by community maintainers.

We found that when community images were last updated, they had more outdated and vulnerable core packages than non-core ones. After some time, these core and non-core packages missed more updates leading to more vulnerabilities present in *Docker Hub* community images. The presence of such vulnerable packages is considerably more pronounced for *node* and *Ruby* images, which tend to be more outdated and more vulnerable than *Python* images. In addition to this, *node* images tend to have the highest proportion of packages missing major updates, as well as a high number of duplicate package releases.

Maintainers of community images in general, of *node* images in particular, and to a lesser extent of *Ruby* images, should invest more effort in updating their outdated packages in order to reduce their number of vulnerabilities. Services

[22]`https://coreos.com/rkt/`

[23]`https://linuxcontainers.org/`

[24]Although the containerisation landscape is likely to evolve, at the time of writing the biggest competitor of *Docker*, namely *rkt*, has been archived (`https://github.com/rkt/rkt/issues/4024`)

[25]`https://flatpak.org/`

for monitoring the freshness and security of container images should be used, and should be improved further to include dedicated monitoring of third-party packages to support this effort.

Acknowledgements

We express our gratitude to the security team of *Snyk.io* for providing us with the full dataset of vulnerability reports. This research was partially funded by the Excellence of Science project 30446992 SECO-Assist financed by FWO-Vlaanderen and F.R.S.-FNRS.

[1] D. Bernstein, Containers and cloud: From LXC to Docker to Kubernetes, IEEE Cloud Computing 1 (3) (2014) 81–84.

[2] J. Turnbull, The Docker Book: Containerization is the new virtualization, 2014.

[3] Stack Overflow, 2020 stack overflow developer survey, https://insights.stackoverflow.com/survey/2020, accessed: 12/02/2021 (2020).

[4] Z. Lu, J. Xu, Y. Wu, T. Wang, T. Huang, An empirical case study on the temporary file smell in Dockerfiles, IEEE Access.

[5] M. A. Oumaziz, J.-R. Falleri, X. Blanc, T. F. Bissyandé, J. Klein, Handling duplicates in dockerfiles families: Learning from experts, in: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp. 524–535.

[6] A. Martin, S. Raponi, T. Combe, R. Di Pietro, Docker ecosystem–vulnerability analysis, Computer Communications 122 (2018) 30–43.

[7] A. Bettini, Vulnerability exploitation in Docker container environments, https://www.blackhat.com/docs/eu-15/materials/eu-15-Bettini-Vulnerability-Exploitation-In-Docker-Container-Environments-wp.pdf, accessed: 12/02/2021 (2015).

[8] Anchore.io, Snapshot of the container ecosystem, https://anchore.com/wp-content/uploads/2017/04/Anchore-Container-Survey-5.pdf, accessed: 12/02/2021 (2017).

[9] R. Shu, X. Gu, W. Enck, A study of security vulnerabilities on Docker Hub, in: International Conference on Data and Application Security and Privacy, ACM, 2017, pp. 269–280.

[10] J. Gummaraju, T. Desikan, Y. Turner, Over 30% of official images in Docker Hub contain high priority security vulnerabilities (2015).

[11] A. Zerouali, T. Mens, G. Robles, J. M. Gonzalez-Barahona, On the relation between outdated Docker containers, severity vulnerabilities, and bugs, in: International Conference on Software Analysis, Evolution and Reengineering, IEEE, 2019, pp. 491–501. `doi:10.1109/SANER.2019.8668013`.

[12] A. Zerouali, V. Cosentino, T. Mens, G. Robles, J. M. Gonzalez-Barahona, On the impact of outdated and vulnerable JavaScript packages in Docker images, in: International Conference on Software Analysis, Evolution and Reengineering, IEEE, 2019, pp. 619–623.

[13] A. Decan, T. Mens, P. Grosjean, An empirical comparison of dependency network evolution in seven software packaging ecosystems, Empirical Software Engineering 24 (1) (2019) 381–416. `doi:10.1007/s10664-017-9589-y`.

[14] Z. Li, M. Kihl, Q. Lu, J. A. Andersson, Performance overhead comparison between hypervisor and container based virtualization, in: International Conference on Advanced Information Networking and Applications (AINA), IEEE, 2017, pp. 955–962.

[15] A. Acharya, J. Fanguède, M. Paolino, D. Raho, A performance benchmarking analysis of hypervisors containers and unikernels on ARMv8 and x86 CPUs, in: 2018 European Conference on Networks and Communications (EuCNC), IEEE, 2018, pp. 282–289.

[16] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, H. C. Gall, An empirical analysis of the Docker container ecosystem on GitHub, in: International Conference on Mining Software Repositories, IEEE Press, 2017, pp. 323–333.

[17] J. Henkel, C. Bird, S. K. Lahiri, T. Reps, Learning from, understanding, and supporting DevOps artifacts for Docker, in: International Conference on Software Engineering, ACM, 2020.

[18] E. Socchi, J. Luu, A deep dive into Docker Hub's security landscape – A story of inheritance?, Master's thesis, Department of Informatics, University of Oslo (2019).

[19] A. Zerouali, T. Mens, J. Gonzalez-Barahona, A. Decan, E. Constantinou, G. Robles, A formal framework for measuring technical lag in component repositories—and its application to npm, Journal of Software: Evolution and Process.

[20] T. Preston-Werner, Semantic versioning 2.0.0, https://semver.org/, accessed: 12/02/2021 (2013).

[21] Node.js Docker Team, node, https://hub.docker.com/_/node, accessed: 12/02/2021.

[22] Docker Community, python, https://hub.docker.com/_/python, accessed: 12/02/2021.

[23] Docker Community, ruby, https://hub.docker.com/_/ruby, accessed: 12/02/2021.

[24] npm, npm-ls: List installed packages, https://docs.npmjs.com/cli/ls, accessed: 12/02/2021.

[25] Python Packaging Authority, pip freeze, https://pip.pypa.io/en/stable/reference/pip_freeze/, accessed: 12/02/2021.

[26] Ruby Community, gem list, https://guides.rubygems.org/command-reference/, accessed: 12/02/2021.

[27] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, L. Devine, Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices?, in: Annual Meeting of the Southern Association for Institutional Research, 2006.

[28] J. Katz, Libraries.io Open Source Repository and Dependency Metadata (Jan. 2020). `doi:10.5281/zenodo.3626071`.

[29] S. Zaman, B. Adams, A. E. Hassan, Security versus performance bugs: a case study on Firefox, in: Working Conference on Mining Software Repositories, ACM, 2011, pp. 93–102.

[30] J. Cox, E. Bouwers, M. van Eekelen, J. Visser, Measuring dependency freshness in software systems, in: International Conference on Software Engineering, IEEE Press, 2015, pp. 109–118.

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

[32] M. H. Ibrahim, M. Sayagh, A. E. Hassan, Too many images on DockerHub! How different are images for the same system?, Empirical Software Engineering (2020) 1–32.

[33] A. Zerouali, A measurement framework for analyzing technical lag in open-source software ecosystems, Ph.D. thesis, University of Mons (September 2019).

[34] R. G. Kula, D. M. German, A. Ouni, T. Ishio, K. Inoue, Do developers update their library dependencies?, Empirical Software Engineering 23 (1) (2017) 384–417.

[35] P. Salza, F. Palomba, D. Di Nucci, C. D'Uva, A. De Lucia, F. Ferrucci, Do developers update third-party libraries in mobile apps?, in: Proceedings of the 26th Conference on Program Comprehension, 2018, pp. 255–265.

[36] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, A. Wesslen, Experimentation in Software Engineering - An Introduction, Kluwer, 2000.