The Docker Hub Image Inheritance Network: Construction and Empirical Insights

Ruben Opdebeeck
Ruben.Denzel.Opdebeeck@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Jonas Lesy
Jonas.Sebastiaan.Lesy@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Ahmed Zerouali
Ahmed.Zerouali@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Coen De Roover
Coen.De.Roover@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

***Abstract*—Docker is a popular technology to containerise applications together with their dependencies into reproducible environments. In Docker, container images can depend on others through inheritance. Such inheritance can propagate bad practices and security vulnerabilities from a parent image to its children. Unfortunately, Docker Hub, the most popular online registry of images, lacks transparency about such inheritance. This obscures the software supply chain, possibly leaving image users unaware of quality or security issues caused by parent images. Nonetheless, we found inheritance on Docker Hub to be an understudied topic in academia to date. Therefore, the goal of this paper is to empirically investigate the practice of image inheritance on Docker Hub. To this end, we collect a dataset of 636,625 unique images belonging to popular Docker repositories and identify inheritance by comparing the images' layers. We leverage the constructed inheritance network to empirically investigate three aspects, namely the structure of the inheritance network, how child images differ from their parents, and outdatedness of parent images. Our results show that most popular community Docker Hub images directly inherit from official images rather than other community ones. We also observe that community child images are often much larger than their parent, in comparison to official child images. This may indicate the existence of gaps between the features provided by official images and those required by consumers, suggesting the need for more ready-made parent images. Finally, we find that around half of the child images use an outdated parent image at the time the child is built, although time lag is usually less than a month. However, time lag becomes much larger when we compare against the latest version of the parent image available at the analysis date, with up to 70% of child images using an outdated parent image and a median of over 5 months of time lag. This indicates that users should pay attention to the lineage of the images they consume, and motivates future work on alleviating technical lag in Docker images.**

***Index Terms*—Docker; Docker Hub; software supply chain; software ecosystems; inheritance network; technical lag**

I. Introduction

Containerisation is a key technology in the DevOps practice. It enables practitioners to package their software together with its dependencies into a *container image*. These can be used to produce *containers*, i.e., isolated environments which can be run consistently on any system [1]. As the most popular containerisation technology, Docker [2] is widely used today. Its success may in part be attributed to the ease in which containers can be created. Indeed, the concept of *image inheritance* enables practitioners to reuse and build upon existing images, often contributed by third parties.

To facilitate such reuse, Docker provides the Docker Hub container image registry. Docker Hub has grown steadily over the past years [3], currently hosting millions of different images. Consequently, it has been the subject of many research studies, including quality [4], [5], security [6], [7] and evolution [3], [8]. Nonetheless, the subject of image inheritance has seen considerably less research attention. The few studies on image inheritance largely disregard the vast amounts of information present in Docker Hub, and focus more on Dockerfiles, i.e., the specifications used to build images.

However, it is vital to understand the image inheritance network, as it contains important insights on the quality and security of Docker images. Bloated parent images, i.e., images containing large amounts of unnecessary files, reduce the efficiency and quality of child images [9]. Moreover, security vulnerabilities may be propagated from a parent image to its children [6], [10]. Finally, with the increasing importance of supply chain security, knowledge of an image's parents is paramount.

While the propagation of security vulnerabilities in Docker Hub has been investigated previously [6], [11], [12], to the best of our knowledge, a comprehensive study of the *general* properties of inheritance in Docker Hub has not yet been undertaken. Therefore, the aim of this paper is to investigate three aspects. First, we investigate the structure of the network, including the prevalence of inheritance and the most influential images. This aspect replicates prior work [13] but with a more recent and accurate inheritance network. Second, we investigate how inheriting images differ from their parents in terms of size and instructions added. In this aspect, we aim to understand how developers extend images and to uncover insights that could be used to suggest or create improved base images. Finally, we investigate the outdatedness of parent images in terms of time lag. Outdatedness of dependencies forms a threat to supply chain security. As such, insights into outdatedness of parent images can drive future work on improving the security landscape of the Docker Hub ecosystem.

To carry out our study, we create a novel dataset of inheritance on Docker Hub. Due to the size of the Docker Hub ecosystem, we focus solely on the most popular image repositories to keep the analyses feasible. Specifically, we

collect 636,625 unique images from all official and the top 1000 community Docker Hub repositories. We devise an algorithm to detect inheritance between these images and to create an inheritance network. We then visualise and quantitatively analyse this network to investigate the three aspects outlined above. The data and scripts used in this study are available for download at https://doi.org/10.5281/zenodo.8119368.

II. Background

A. The Docker Containerisation Framework

Containerisation is a widely-used technique to isolate software into its own environment. Developers can create *container images* that package an application together with an operating system and all runtime dependencies. A *container* created from the image runs the application in an environment which is independent from the host's. However, containers share memory and CPU with the host system and are therefore more lightweight than virtualisation, making them a popular alternative to achieve platform independence. Various technologies implementing containerisation exist, of which Docker is the most popular [14].

Docker images are created by building *Dockerfiles*. A Dockerfile consists of a sequence of instructions, such as `RUN` to run shell commands, or `COPY` to copy files from the host environment into the image. To build an image, for each instruction sequentially, the Docker engine creates a container, runs the instruction in it, and records the changes that are made. Each subsequent instruction builds upon the changes of those before it. Effectively, this creates a stack of changes caused by each instruction, which makes up the final image.

Since each instruction depends on previous changes, and since the first instruction does not have a predecessor, it follows that the first instruction must be a special case. Indeed, each Dockerfile must start with a `FROM` instruction, which specifies the base for the next instruction. This can either be a pre-existing image, or the special-case empty `scratch` image. The former causes *inheritance* between images, in which the layers of the existing image become a prefix of the layers of the new one. The latter, as the name implies, creates the new image from scratch.

Listing 1 exemplifies a Dockerfile to build a NodeJS application. The `FROM` instruction (line 1) specifies that the image needs to build upon an image named `node:18-alpine`, which we shall return to later. Subsequently, the `COPY` instruction copies source code files into the image, `RUN` installs the application's runtime dependencies with the `npm` package manager, and the `CMD` instruction specifies which command to invoke when a container is spun up from the resulting image.

```
1 FROM node:18-alpine
2 COPY . .
3 RUN npm install
4 CMD [ "node", "app.js" ]
```

Listing 1: Example Dockerfile to build a NodeJS image.

As mentioned, images are stacks of separate layers, each corresponding to an instruction. Layers are identified through SHA-256 digests, based on the changes made in the layer itself, as well as its dependencies (i.e., predecessor layers). Subsequently, images are also given a SHA-256 digest, based on its layers and certain other properties. For instance, images built from the same instructions may be distinguished based on the architecture on which they run, such as *amd64* or *armv7*.

Image repositories store images after they are built. Each repository has a unique name and can store multiple images which are individually addressable by *tags*, often used to distinguish different versions. Within a repository, an image can be given multiple tags. For instance, an image providing a Python 3.11.4 runtime could be tagged with `3.11.4`, `3.11`, and `3`. When a Python 3.11.5 image is made later, and assigned the tags `3.11.5`, `3.11`, and `3`, the latter two will be disassociated from the earlier image, leaving only `3.11.4` pointing to it. Finally, the last built version of an image is automatically tagged `latest` by the Docker engine.

Given a name and a tag, an image can be addressed as `<name>:<tag>`. In the example of Listing 1, line 1 specifies `node:18-alpine` as a base image, referring to an image in the `node` repository with the tag `18-alpine`. If no tag is specified in a `FROM` instruction, `latest` is implied.

B. The Docker Hub Ecosystem

Images can be pushed to and pulled from centralised image repositories, enabling reuse of images. *Container registries*, such as Docker Hub[1], Google Container Registry[2], and Quay[3], aggregate many public image repositories, forming ecosystems of Docker content. Of these, Docker Hub is the largest and most popular public registry to date with more than 9.4M images (as of August 2022). It has facilitated billions of image downloads, with images like `ubuntu`, `alpine`, and `redis` having over a billion downloads each.

Repositories in Docker Hub are divided into official and community repositories. Images in official repositories must undergo a validation process, must maintain security and quality standards, and must be maintained regularly. Importantly for inheritance, official images may only inherit from other official images. Because of their high quality standards, official images often serve as good base images. Examples of official repositories include `debian`, `python`, and `nginx`.

Community repositories do not need to meet the same strict requirements as official ones. Nonetheless, this does not mean they are necessarily of lower quality. Many popular open-source projects provide their own image repositories, of which `datadog/agent` and `grafana/grafana`, are popular examples.

III. Constructing the Inheritance Network

To carry out the empirical study, we first need to construct an inheritance network of Docker Hub images. In the follow-

[1]https://hub.docker.com/
[2]https://gcr.io/
[3]https://quay.io/

ing sections, we first describe the collection of the relevant data (Section III-A). Afterwards, we describe how we leverage image layer information to identify inheritance and how we construct the inheritance network (Section III-B).

A. Data Collection

To build the Docker Hub inheritance network and perform the empirical study, we require a dataset of Docker Hub images. This dataset needs to adhere to several key requirements so that the resulting inheritance network is accurate. First, our network's nodes should represent individual images rather than entire repositories of multiple image versions. Second, the dataset should contain *all* image versions of a specific image repository, rather than just the `latest` tag. Prior work has shown many images' parents to be unidentifiable when old image versions are not considered [12], and that up to 20% of images do not have a `latest` tag [3]. Third, we want our dataset to cover both official and community repositories, and want to avoid restrictive filters, such as operating systems or programming languages, where possible.

We cannot rely on Dockerfiles to build our dataset [8], [13], as they may specify base images with the `latest` tag, which could have been modified after the image was built and pushed. Therefore, although Dockerfiles specify the inherited repository exactly, the image version may not be accurate. Moreover, Dockerfiles may not always be available.

We will instead leverage image contents to identify the exact parent. Rather than downloading image contents, which comes with a large overhead [6], we will use layer digests as a proxy for layer contents. Using layer digests will enable our approach to scale efficiently and to identify exact parent images accurately, regardless of the tag used in a Dockerfile. However, our approach will only be able to identify inheritance if both the parent and child images are present in the dataset.

To collect this dataset, we list all image repositories (Section III-A1), and then retrieve the image metadata for all tags in a popular subset of these repositories (Section III-A2).

1) Gathering the Image Repository List: As a first step in gathering a dataset of image layer information, we require a list of the image repositories on Docker Hub. Unfortunately, Docker Hub does not publicly expose such a list for community repositories. Moreover, simple crawling approaches used by prior work [3] can no longer be applied[4]. Instead, we resort to an exhaustive search via Docker Hub's search API.

Rather than relying on a large, pre-generated list of search terms as applied in previous work [6], we use an incremental approach. Specifically, we generate a 1521-word dictionary composed of all minimal search terms that the API accepts, i.e., all 2-character combinations of allowed characters in repository names ("a–z", "0–9", ".", "-", or "/"). We initially execute a search query for each search term. However, since the search API is limited to returning 10,000 results, these short search terms may not uncover all repositories. Instead, when the API returns the maximum number of results, we extend the search term by prefixing and suffixing each allowed character (e.g., "aa" becomes "aaa", "aab", "baa", ...) and running the search process anew for each expanded term. We deduplicate, extract and store all of the relevant information of each search result, most importantly, the repository name, its total pull count, and whether the repository is official.

[4]Replicating the approach used by Zhao et al. [3] results in a mere 689 community repositories, while prior work has uncovered over a million repositories [10].

	Official	Community
# repos	175	10,422,531
# repos with 0 pulls	2	3,616,703
# repos with >1B pulls	23	49
Total pull count	106,756,687,443	330,183,083,133
# repos in our dataset	171	995
# tags in our dataset	96,094	1,018,429
# unique images in our dataset	44,917	591,708

TABLE I: Metrics of Docker Hub and our final dataset.

After executing this collection process, we found a total of 10,422,706 image repositories, of which 175 official ones. Table I depicts metrics of the resulting dataset. Interestingly, we note that a large fraction (over 3.5 million) of repositories have never been pulled and are therefore likely not being used in practice.

2) Gathering the Image Metadata: The second phase of our data collection consists of first collecting all tags in a repository. Afterward, we collect image metadata, most importantly its list of layers, for each discovered tag. Since multiple tags can correspond to the same image, we will already remove duplicate images by comparing their layer digests. In case duplicate images occur across repositories, we keep official images in favour of community ones.

Given the size of the Docker Hub ecosystem and the large number of repositories discovered earlier, it is necessary to apply filters to keep the analysis feasible. First, we decide to focus only on images of the *amd64* architecture, since it is the most used [12] and inheritance between different architectures is impossible. Second, for community repositories, we only include the 1000 most pulled repositories, which will prevent low quality and unused images from polluting the dataset while still keeping a large number of images to analyse. Therefore, our dataset will consist of all *amd64* images for each tag of all official and the top-1000 community repositories.

We executed this collection process on March 19th, 2023. Collecting the image metadata of all official repositories took nearly 11 hours, whereas it took close to 4.5 days for the top-1000 community ones. In total, we failed to gather image metadata for 4 official and 5 community repositories. These failures were either due to the repository being intentionally empty (e.g., temporary or deprecated repositories), or due to the repository being moved, renamed, or removed between the time of repository collection and image collection. An overview of the number of repositories, tags, and images used in the rest of study is provided in the bottom rows of Table I. Our final dataset consists of 636,625 images belonging to 1166 image repositories.

Algorithm 1: Inheritance detection algorithm

input : A list of n images $I = [i_1, \ldots, i_n]$ with their layers

output: A set of direct inheritances $< P_i, C_i >$ between parent P_i and child C_i

```
1 I ← SortByLayers(I);
2 previous-root ← 1;
  // Outer loop over all images
3 for i ← 2 to n do
    // Loop over candidate parents
4   for j ← i − 1 to previous-root do
5     if I_j's layers are prefix of I_i's layers then
6       Output < I_j, I_i > as direct inheritance;
7       break;
8   if no parent was found for I_i then
9     previous-root ← I_i;
```

B. Building the Inheritance Network

Having collected the dataset of Docker Hub images, we proceed to construct its inheritance graph. To this end, we need to identify inheritance relationships between images. Recall from Section II-A that when an image inherits from a parent, the parent's layers will form a prefix of the child's. Consequently, an inheritance relationship between a child and parent image can be identified by investigating their layers.

To identify all inheritance relationships in our dataset, we use the algorithm depicted in Algorithm 1. First, we lexicographically sort the list of images based on their layer digests (line 1). This ensures that an image's parent, if any, is ordered before the image itself. Then, the algorithm sequentially attempts to identify a parent for each image. It does this by backtracking through the image list in reverse order in search of the first image whose layers match a prefix, and is thus the closest parent to the image. Recall that we remove duplicate images based on the layers, meaning there can be at most one such image. This backtracking halts whenever it reaches a root image, i.e., one without a known parent. Any further exploration will yield no matches, since the earlier images are certainly unrelated because of the lexicographical sorting.

Once we obtain the set of all inheritance relationships, we can construct the inheritance network, which is a directed graph. Its nodes represent each image that is involved in at least one inheritance relationship, whereas edges represent an inheritance relationship, flowing from the child to its parent. Since we only add nodes that are involved in an inheritance relationship, our network does not contain isolated nodes.

IV. Empirical Study

We now leverage the inheritance network to perform an empirical study into inheritance in Docker Hub. In the empirical study, we aim to answer 5 research questions divided into 3 topics, as outlined below.

	Isolated nodes		Connected nodes	
	# nodes	% nodes	# nodes	% nodes
Total	326,619	51.30	310,006	48.70
Official	9,878	1.55	35,039	5.50
Community	316,741	49.75	274,967	43.19

TABLE II: Network node metrics

Structure of the network

RQ_1: What does the inheritance network look like?
RQ_2: Which images are most often inherited from?

Characteristics of inheritance

RQ_3: How much do inheriting images grow in size?
RQ_4: Which instructions are added to parent images?

Outdatedness of parent images

RQ_5: How outdated are parent images?

A. Structure of the Network

RQ_1: What does the inheritance network look like?: In the first research question, we perform an initial exploratory analysis by visualising the network. Through this, we aim to understand the overall shape of the ecosystem.

Research Method: First, we visualise and explore the network using Gephi[5]. We use Gephi's built-in graph drawing algorithms to render the inheritance network. We also use its implementation of the PageRank [15] algorithm to assign weights to each node in the graph. These weights are used to determine the size of the node to draw. Intuitively, nodes which more often appear as ancestors, are ranked higher and are consequently drawn larger.

Afterwards, we will divide the inheritance network into clusters of connected components. Each component will contain a root image and all of its descendants. Since Docker does not support multiple inheritance, this will effectively turn our network into a forest of disconnected subtrees. We will then investigate the distribution of the component sizes to better understand the sizes of subcultures in the network.

Results: As can be seen in Table II, our entire inheritance network consists of 310,006 nodes. Roughly half of the images in our dataset are represented in this network. Conversely, for the other half, we could not find any inheritance involving these images in our dataset.

We observe that in absolute numbers, community images make up the majority of our connected network. Nonetheless, a larger proportion of all official images (78%) are represented in the connected network, as opposed to only 53.5% of community images. Moreover, 15,549 images are *intermediate parents*, i.e., they inherit a parent image and are themselves a parent image.

Figure 1 depicts the Gephi visualisation of the inheritance network. Nodes are coloured according to the image type (blue for official images, brown for community) and sized according to their PageRank score. Edge colours signify the type of the inheriting image, i.e., a brown edge represents an inheriting

[5]https://gephi.org/

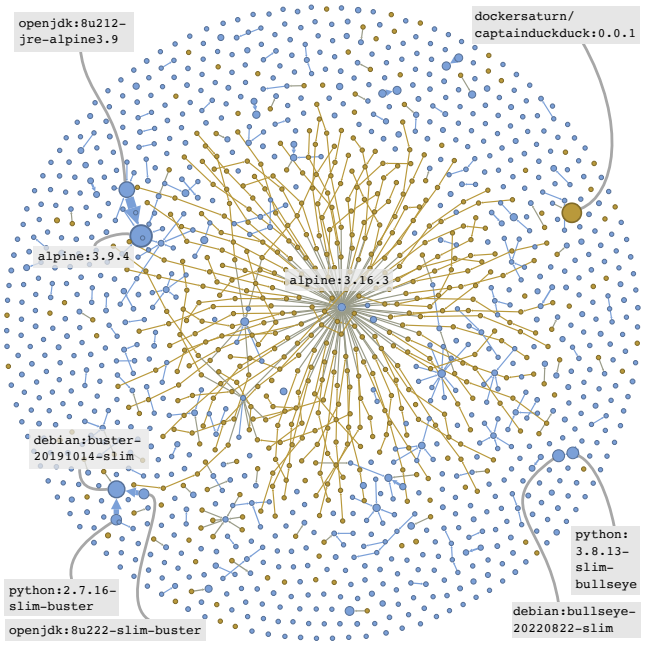

Fig. 1: Gephi visualisation of the most influential nodes in the network.

community image. Edge thickness indicates the child image's contribution towards the influence of the parent image, i.e., the total number of descendant images that inherit the parent image transitively via the child. To maintain comprehensibility, we only depict the visualisation for the most-inherited subset of the network.

Most importantly, visible on the right of the figure, we observe a community image with a high PageRank score, namely `dockersaturn/captainduckduck:0.0.1`. Upon closer inspection, this image is an anomaly, and inherits `alpine:3.6` without adding new layers. However, the inherited `alpine:3.6` image is no longer associated to a tag on Docker Hub, and is therefore not included in our dataset. If it had still been tagged, it would have been chosen as the canonical image following our deduplication strategy (cf. Section III-A). We will return to this phenomenon in Section V-A. For the remainder of our empirical analysis, we will instead relabel this image as "`alpine:3.6-detached`".

Centrally, we observe an official image, `alpine:3.16.3`, that forms the ancestor for a large subnetwork of community images. Towards the corners, we observe several influential official images with influential official children. Further exploration reveals that the parent images are operating system images, like `alpine` and `debian`, while the child images are language runtimes for Java and Python.

Finally, the visualisation shows isolated nodes corresponding to separate root images, each forming their own component in the network. In total, our network consists of 5391 components. The majority of these (3256) are minimal components of size 2, consisting of just a parent and a child. Nonetheless, we observed that larger components can consist of thousands of images, the largest having over 18,000 images. This component is rooted at `alpine:3.9.4`, indicating that it is a highly-used base image. We further found that the average component consists of 57 images.

Child type	Parent type	# edges	%
official	official	32,931	10.80
community	official	238,411	78.22
community	community	33,476	10.98

TABLE III: Inheritance relationship types

Summary: 78% of official images are involved in inheritance vs. only 53.5% of popular community ones. Most root images are inherited just once, although large outliers with thousands of descendants exist. Parent images could have been misidentified due to image aliasing.

RQ_2: Which images are most often inherited from?: The previous research question confirmed the existence of several influential base images. Moreover, Figure 1 suggests official images are more popular base images. In RQ_2, we inspect these influential base images more closely, and investigate the different types of inheritance in our dataset.

Research Method: First, we quantify the types of inheritance in our network. These types are determined based on the official status of the child and parent image, leading to three possibilities since official images cannot inherit from community ones. Then, we identify the most popular parent images in the network, both in terms of their in-degree (i.e., number of direct children) and their PageRank [15] score, calculated earlier.

Results: Table III summarises the types of inheritance between images. We notice that the majority (78%) of inheritance relationships involve a community image inheriting from an official one, whereas inheritance between two official images is the least common. Although official images make up a smaller proportion of our dataset, we observe that in nearly 89% of inheritance cases, the parent is an official image. Nonetheless, community images inheriting from other community images form a non-negligible portion.

Table IV depicts the top 5 images in the network according to their direct in-degree and their PageRank scores. We notice that for both metrics, the entire top 5 consists of images that offer either an operating system (`alpine` and `debian`) or a language runtime (`openjdk` and `python`) based on those operating systems. Moreover, the top 5 consists entirely of official images. The most inherited community image in our dataset is `fluent/fluent-bit:1.8`, ranked 63rd with 780 direct children.

Interestingly, we find that `alpine` and language runtimes are more popular as direct parents, whereas `debian` base images only enter the top 5 when the PageRank metric is considered. This may suggest that Debian images are more often inherited indirectly, i.e., through intermediate parents, while Alpine is more popular as a direct base image.

Highest in-degree		Highest PageRank	
Image	**In-degree**	**Image**	**PageRank**
`alpine:3.6-detached`	15,067	`alpine:3.9.4`	0.0248
`openjdk:8u212-jre-alpine3.9`	11,254	`alpine:3.6-detached`	0.0208
`python:3.8.13-slim-bullseye`	7,642	`debian:buster-20191014-slim`	0.018
`python:2.7.16-slim-buster`	6,470	`openjdk:8u212-jre-alpine3.9`	0.016
`openjdk:8u222-slim-buster`	5,931	`debian:bullseye-20220822-slim`	0.011

TABLE IV: Top-5 images with highest in-degree and PageRank scores

Summary: Official images, specifically operating systems and language runtimes, are popular parents. The top community images are rarely inherited by other popular images.

B. Characteristics of Inheritance

RQ_3: How much do inheriting images grow in size?: To characterise inheritance, we first look at the size increase of child images, both in terms of layers and image size. Image size is an important factor in the quality of images, since smaller images reduce storage and distribution costs and may lead to more efficient runtime performance [9], [16]. Therefore, answering this research question may provide insights that can help image providers optimise their images.

Research Method: First, for each inheritance relationship in our dataset, we calculate the number of layers added to the base image. We then study the distribution of the number of additional layers, comparing official against community child images. To this end, we apply the appropriate statistical tests to determine whether the difference is statistically significant.

Second, for each inheritance relationship, we calculate the total size increase caused by the additional layers. We analyse the evolution of size increase over time, based on the date the child image was pushed. Both the layer size and the image push date are available in the image metadata collected in Section III-A. However, we notice that the push date is not always available for individual images. The earliest timestamp in our dataset is in September 2020, so our evolution analysis omits child images pushed before that date.

Results: Figure 2 depicts boxenplots of the distribution of the number of layers added after inheritance, distinguishing between official and community child images. We observe that both distributions are right-skewed, indicating that the majority of inheriting images add relatively few new layers. The median number of added layers is 13 and 14 for official and community images, respectively. Moreover, the maximum number of observed added layers is 56 for official images, while for community images, higher numbers can often be found. These observations may suggest that official images are more optimised, since adding more layers may increase I/O overhead [16].

To statistically confirm these observations, we carried out the non-parametric Mann-Whitney U test that does not assume normality of the data. The null hypothesis H_0 assumes that both distributions are similar. We rejected H_0 with statistical significance $p < 0.01$. However, we only found a negligible effect size ($|d| = 0.04$) in favour of community images using Cliff's Delta, a non-parametric measure quantifying the difference between two populations beyond the interpretation of p-values.

Importantly, we observe that for some community images, the additional layer count is an over-approximation due to intermediate parents that have not been identified. For instance, inspecting the Dockerfile of the image which adds the highest number of new layers (108), we find that it actually inherits from an intermediate parent which is not hosted on Docker Hub. Instead, it is hosted on Microsoft's Artifact Registry, which is not considered in our dataset.

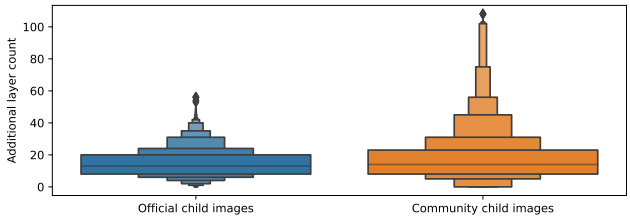

Fig. 2: Distribution of additional layer counts for official and community inheriting images.

Finally, Figure 3 depicts the evolution of size differences over time, based on the last pushed date of the image, split between official and community images. We observe a slight upward trend, showing that both official and community images are adding more and more data to their base images. Most strikingly, we find that community images add significantly more data to their parent images than official images. This may again indicate that official images are better optimised, or that community images suffer more from bloating [9].

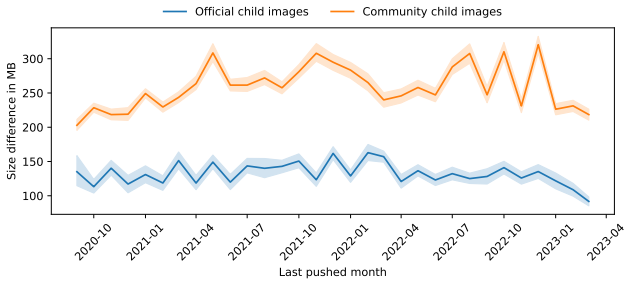

Fig. 3: Temporal evolution of average size difference after inheritance.

Summary: Although popular community child images add slightly more layers than official ones, they add a lot more data, suggesting optimisations may be possible.

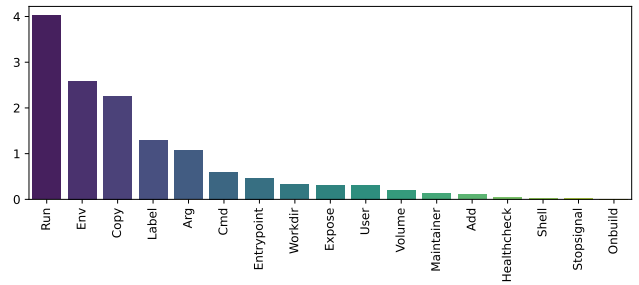

Fig. 4: Average instruction occurrence per inheritance case.

RQ_4: Which instructions are added to parent images?: In addition to size and layer increases during inheritance, we investigate which instructions practitioners use to create inherited images on Docker Hub. This may identify trends in instruction usage and may also provide information about the reasons why developers extend images.

Research Method: We identify and count the types of Docker instructions for each layer added during inheritance. These instructions are provided by Docker Hub in the image metadata already collected. We then calculate how often these instructions occur during inheritance.

Results: Figure 4 depicts the average number of times that each Docker instruction is used to extend a base image, identified through the layers that were added during inheritance. We find that `RUN`, `ENV`, `COPY`, `LABEL`, and `ARG` are the most frequent instructions, each occurring on average at least once. This may indicate that developers often run commands to extend images (e.g., to install new packages or to compile their own source code) and copy new files into them. Moreover, they may use `LABEL` to set their own image metadata, and use `ENV` and `ARG` to introduce or override (environment) variables.

Instructions such as `CMD` and `ENTRYPOINT`, which configure which command is run when a container is started, occur on average only in roughly half of extended images. The least frequent instruction is `ONBUILD`, which registers another instruction to be triggered when the image is used as a base image. The lack of `CMD` and `ENTRYPOINT` is surprising, as one would expect inheritors to override which command is executed by the container, rather than inherit the base image's command. This may indicate that many popular images provide a "framework" as a base image for others to extend further. On the other hand, since the `ONBUILD` instruction is only useful when the image is meant to be a base image, a lack thereof may suggest that the child images in our dataset are meant to be used as-is, without further inheritance. Nonetheless, we must take into account that use of the `ONBUILD` instruction may be rare, regardless of inheritance. Alternatively, it is possible that the base image already provides a suitable command, causing the lack of `CMD` and `ENTRYPOINT` instructions.

Finally, we find that usage of the `MAINTAINER` instruction is decreasing over time. In fact, it only appears in 1 of 100 images in 2023, over 5 years after its deprecation in 2017.

Summary: `RUN`, `ENV`, and `COPY` are the most common instructions. A lack of `CMD` instructions may suggest that many of the most popular child images are supposed to be inherited further, whereas a lack of `ONBUILD` instructions may suggest the opposite.

C. Outdatedness of Parent Images

RQ_5: How outdated are parent images?: Finally, we investigate the outdatedness of parent images. Since images can be updated, a child may have inherited an outdated version of its parent. Outdated images may carry security vulnerabilities and other problems, and should therefore be avoided. Although outdatedness has been studied extensively for language ecosystems [17], [18] and packages installed in Docker images [10], we find that the outdatedness of parent images has not been studied so far. Therefore, answering this research question may unearth important implications regarding the security of Docker Hub images.

Research Method: To calculate outdatedness, we again require the push date of both the parent and child image. As described earlier, this date is not available for images pushed before September 2020, and we exclude all such images. Moreover, since the push date is in fact the *last* pushed date, it is possible for a child image's push date to be earlier than that of its parent image, making the child image appear to be older than its parent. This may occur when the parent image is repushed to the same tag with the same content after the child was already published. Therefore, in the upcoming analyses, we exclude all such nonsensical inheritance relationships.

First, we investigate the difference between push dates of the parent and child image. A high push difference between a child and its parent means that the child image used an old parent image which may no longer be maintained.

However, push difference does not take into account the availability of new versions of the parent. Therefore, we also investigate the time lag [10], which quantifies the outdatedness between the parent image and its latest version. To this end, we examine each inheritance relationship and determine the most recent available version of the parent image at a specified date. Subsequently, we calculate the time difference (i.e., time lag [10]) between the push date of the latest version of the parent and the version actually inherited from.

We carry out this analysis twice, assuming two different time points to find the most recent available parent image. First, we calculate time lag at the time the child image was released, based on the child's push date. Second, we assume the child image would be deployed at the analysis time, which we set at the time of data collection (March 19th, 2023). The former produces best-case results, whereas the latter assumes the more realistic scenario that images keep being used after their initial build date. We apply the latter only to the latest (last pushed) versions of child images. Calculating time lag at the analysis time for child images which are themselves outdated would lead to unsound results, since older images are likely to become outdated with a significant time lag.

Results: *a) **Push date differences**:* Figure 5 depicts the distribution of push differences for official and community child images. We observe that the distribution is right-skewed, indicating that more often than not, inheriting images use recent variants of those images. We can also see that push date differences are lower for official children than for community ones. This can likely be attributed to the higher quality standards that official images need to follow.

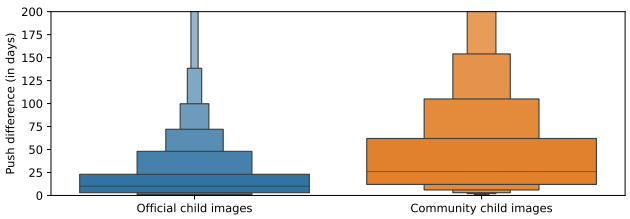

Fig. 5: Push difference between inheriting and base image.

Nonetheless, we notice that there are 677 images in our data set with a push difference of over a year, with one as high as 763 days. Looking at official images only, which are supposed to be of higher quality, Table V lists the most extreme cases of push time difference. We immediately see that most images inherit from `centos:centos7.9.2009`, which was pushed to Docker Hub on 2021–09–15. In fact, this image is involved in 92 of the 100 highest observed push differences. This version of CentOS is unmaintained as of August 2020. Therefore, any recent image depending on (official) CentOS 7 will necessarily have a high push time difference.

*b) **Time lag at the push date of the inheriting image**:* Looking at time lag on release date confirms our initial observation, as we find that over 53.8% of the images in our dataset were using the latest available version of the image they inherit from. We further found that 35.7% of outdated images (time lag greater than zero) were less than one month behind the latest available version. When considering all inherited images (outdated and up-to-date), 29.7% exhibited a time lag equal or exceeding one month. When looking at official and community images, we observed that community images tend to inherit from slightly more outdated image versions compared to official images.

These results validate our initial observation, demonstrating that image producers more often prefer recent and up-to-date images when creating their own images. Nonetheless, over a quarter of images use parent images that are already outdated by more than a month at their release date. These findings are consistent with previous studies as well [10].

*c) **Time lag at the analysis date**:* Figure 6 shows the time lag incurred by outdated inherited images, grouped by the child image's kind (i.e., official or community). We found that only 29.6% of the used images are up-to-date with zero time lag. Overall, the median time lag is 5.63 months, the average is 18.27 months while the maximum is 81.9 months. When comparing between official and community images, we found

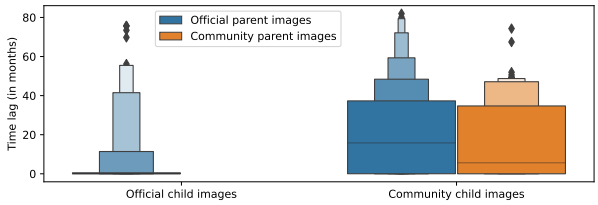

Fig. 6: Time lag of outdated inherited images, grouped by their kind (official or community), at the analysis date.

that the latter have more outdated images than the former, i.e., $p < 0.01$ using the Mann-Whitney U test, while the effect size is medium ($|d| = 0.469$) in favour of community images.

These findings demonstrate that, as expected, images grow more outdated over time. Nonetheless, since we only focus on the latest versions of images, our results are still alarming. One possible explanation may be that child images are maintained and updated less often than their parents. Therefore, the child image would get outdated as soon as a new parent image version becomes available, and stay outdated for longer. Given that community images are affected more by outdatedness, and most community images inherit from official ones, which must be maintained regularly, such an explanation appears likely.

Summary: The extent to which parent images are outdated may be a cause for concern. Nearly 30% of popular child images inherit a parent image that is outdated by over a month at the time the child is built. At analysis time, 70% of the latest version of popular child images inherit an outdated parent with a median of 5.63 months.

V. Discussion

A. Unidentifiable Parents

As observed in RQ_1 and RQ_3, our approach may fail to identify parent images in some cases. We identify three reasons this may happen.

1) The parent is not on Docker Hub, but on another registry.
2) The parent is on Docker Hub, but is omitted by a filter.
3) The parent is on Docker Hub, but is not linked to a tag.

While the first two cases can be avoided by applying our approach on a larger dataset, the final case cannot currently be addressed. Since Docker Hub allows its contributors to overwrite tags, it is possible for images to become "detached" from all tags, after which they can only be retrieved via their SHA-256 digest. Consequently, because the API only exposes tags, we cannot retrieve information about the detached image. Therefore, if the image was used as a parent in the past, we can no longer identify it as such.

In rare cases, we may be able to find an alternative image with the same layers, i.e., an image that inherits the original image but does not add any layers to it, which we encountered in RQ_1. Such images lead to "spurious parents", where we wrongly identify the parent image to be the alternative, while the child image in fact inherited from the detached image. Such spurious parents may have led to an over-approximation

Child image	Parent image	Push difference (days)
ibm-semeru-runtimes:open-11.0.18_10-jdk-centos7	centos:centos7.9.2009	531
crate:5.1.4	centos:centos7.9.2009	509
eclipse-temurin:19.0.2_7-jre-centos7	centos:centos7.9.2009	497
swift:centos7-slim	centos:centos7.9.2009	491
mongo:4.0.28	ubuntu:xenial-20210804	359

TABLE V: High push differences: official images

of community parent images. Nonetheless, we found and rectified one spurious parent (see RQ_1), and found that the other community parent images are much less influential in the network.

Unfortunately, since we have no way of enumerating detached images, we have no way of knowing which and how many potential parents we missed. Therefore, there is a potential for many of the isolated nodes in our network to in fact be related to some unknown detached parent image without an alternative, spurious parent to connect them. The impossibility of tracking these images' lineage is worrisome, especially in the context of supply chain security, where it hampers the ability for practitioners to inspect the health of their dependencies, such as indirect parents of their Docker images (cf. RQ_5).

B. Implications

1) For image consumers: Our results reveal that many images, although themselves up-to-date, use outdated parent images (RQ_5). Outdated base images ship outdated software, which is more likely to contain bugs, inefficiencies, and most importantly, security vulnerabilities. Such problems often propagate into child images [6]. Although prior work has already recommended practitioners to use up-to-date versions of images [6], [19], we believe this to be insufficient. Instead, practitioners should not only ensure the image itself is up-to-date, but also **ensure that all of the image's ancestors are both up-to-date and recent**.

2) For image contributors: The results for RQ_2 confirm that, unsurprisingly, official images are by far the most inherited from. However, RQ_3 shows that community child images may grow in size by twice the amount that official children do. This suggests that there is room for improvement in space **optimisations of community images**. It may also be indicative of a **lack of ready-made, feature-complete base images**, causing consumers to need to add common packages or dependencies. It may be worthwhile for base image providers to investigate which features are often needed by consumers, and to offer image variants that provide those out-of-the-box. This may provide further opportunities for deduplication, thereby saving storage and distribution costs.

Secondly, as revealed in RQ_5, we observe high outdatedness and time lag, especially for community images. In well-known software ecosystems like *npm* and *PyPI*, packages can specify dependencies using ranges. This enables clients to update transitive dependencies with security patches. However, inheritance in Docker images is pinned and immutable. Hence, clients cannot resolve vulnerabilities in a parent image unless the inheriting image itself is updated. Therefore, we urge official and community image maintainers alike to **remain in sync with security updates of parent images**.

3) For registry operators: As discussed in Section V-A, a lack of historical information on image tags and the inability to list image digests pose significant challenges for supply chain security research. Moreover, not exposing such information hampers practitioners who want to understand the source of the images they use. We therefore implore registry operators to **provide tag histories and image digest lists**, and to ratify this in the distribution specification [20].

Many before us have repeatedly called for built-in security tooling on Docker Hub [6], [7], [12], [19]. Docker's recently-unveiled Scout[6], an image insights tool which shows vulnerabilities and image inheritance lineage, is a promising first step. Nonetheless, we find several shortcomings, such as its apparent inability to identify community parent images, which we believe should be addressed[7]. Moreover, we urge Docker to **roll out this security analysis to *all* Docker Hub images** instead of merely official ones. We also advise other registry providers to follow suit with similar technologies.

4) For researchers: Most importantly, we advise researchers to **beware of detached images and their threats** to empirical Docker Hub research, as discussed in Section V-A.

Our research open various gates for future work. Our findings in RQ_3 motivate further empirical studies into the content developers add during inheritance, specifically to investigate why community images add twice as much content as official images, and to the differences between images and their parents. The results of RQ_5 motivate further studies on the impact of outdated base images in practice, and technical research can investigate means to alleviate outdatedness of Docker base images. Finally, tool builders can benefit from accurate inheritance networks, such as those built by our approach, to recommend base images [21], improve discoverability of images [22], or to alert Docker image maintainers of security vulnerabilities caused by base images.

VI. Threats to validity

We present the threats to validity of our research according to the guidelines of Wohlin et al. [23].

As a threat to **construct validity**, we use the last push date of an image as a proxy for the date when an image was built. However, since developers could repush the same image at a later date, this date may be inaccurate. This may have caused inaccuracies in the findings of RQ_5. Unfortunately,

[6]https://www.docker.com/products/docker-scout/

[7]Docker Scout is still in Early Access and these limitations may be fixed in a final release.

such inaccuracies could not be avoided, since Docker Hub provides no historical information.

As a threat to **external validity**, since we only analysed the official and top 1000 community repositories, our findings cannot be generalised to the larger Docker Hub ecosystem and only apply to the popular repositories considered in our dataset. Although a random sample may improve generalisability, it would be unlikely to contain community parent images, and our approach would be unable to identify such parents. Therefore, to improve generalisability, our approach could be replicated on a larger subset of the ecosystem. However, such a dataset would require much longer to collect. A second threat to validity stems from the omission of other image registries in our dataset. Therefore, we cannot claim generalisability to other registries.

As a threat to **internal validity**, we only considered the *amd64* architecture. Nonetheless, we do not expect observations to differ significantly for other architectures. Secondly, our dataset possibly lacks some parent images (cf. Section V-A), meaning we may have under-approximated how often inheritance occurs.

VII. Related Work

a) Empirical Studies on Docker: Docker and the Docker Hub ecosystem have been widely studied in prior work. Zhao et al. [3] investigate image characteristics (e.g., layer size, file types) of 47TB worth of images from Docker Hub. Cito et al. [8] study Dockerfiles for quality issues, revealing that one third of the 70,000 considered files could not be built and were thus unusable. Lin et al. [19] combine Dockerfiles and their Docker Hub images to investigate the evolution of Docker images, uncovering that Dockerfiles evolve more slowly than general-purpose source code. Ibrahim et al. [24] investigate the differences between official images and community counterparts and find that community counterparts are often more storage-optimised. On the other hand, Zerouali et al. [10] investigate technical lag in Debian-based images and find that official Debian images suffer less from technical lag than their community counterparts.

Some prior work has investigated inheritance in Docker images. Zhang et al. [13] determine inheritance based on `FROM` instructions in Dockerfiles. Our work replicates part of their study, namely the exploration of the structure of the resulting network and the most influential images. Nonetheless, their inheritance calculation relies on Dockerfiles, which, as mentioned in Section III-A, may not always be reliable or available. Other than that, they investigated differences between different categories of images, such as *Operating System* or *Language Runtime*.

Shu et al. [6] investigate the propagation of security vulnerabilities in the Docker Hub ecosystem. They find that vulnerabilities are pervasive in the ecosystem and suggest that many may be propagated from parent to child images. Similar to our work, they collect an exhaustive dataset of Docker images and calculated inheritance between them, albeit with a different algorithm resulting in a graph of layers rather than images. Moreover, they do not explore general properties of Docker Hub image inheritance. Ul Haque et al. [11] investigate a similar topic, but determine inheritance from Dockerfiles.

b) Empirical Studies on Infrastructure as Code: Since Docker is an Infrastructure-as-Code (IaC) technology, we also look at empirical studies in the IaC domain. We find that the majority of existing studies in the IaC domain focus on co-evolution of source and infrastructure code [25], code quality and smells [26]–[29], and practitioner insights [30], [31]. Nonetheless, some IaC ecosystems have been subjected to empirical studies, such as the practice of content versioning [32], and the characteristics of reusable Kubernetes code [33].

c) Empirical Studies on Dependency Networks: A final area of related work is that of dependency networks, since our inheritance network is also a form of dependency network. Decan et al. [34] study evolutionary patterns of the *npm*, *CRAN*, and *RubyGems* ecosystems and the importance of semantic versioning and dependency constraints. Later, they further examine the growth of the *npm* ecosystem, uncovering that transitive dependencies often cause fragility. Kula et al. [35] investigate thousands of *Maven* and *CRAN* artefacts, revealing popularity, adoption, and updating patterns. Blincoe et al. [36] identify dependency networks through GitHub projects and find that most ecosystems are centred around one project and interconnect with other ecosystems.

VIII. Conclusion

Docker Hub forms the largest ecosystem of container images. Its contents constitute vital parts of software supply chains. However, image inheritance, although vital to understand for security purposes, is obscured by Docker Hub. Therefore, in this paper, we construct an inheritance network of the popular subset of the Docker Hub ecosystem. Specifically, we collect a dataset of 636,625 Docker images spread across the 175 official and 1000 most popular community repositories, and compare their layers to identify inheritance. We leverage the network to empirically study the most influential images, the characteristics of image inheritance, and the outdatedness of parent images. We find that many image ancestors may remain unidentified since Docker Hub does not provide tag histories. Such "detached images" may form a substantial threat to empirical research based on Docker Hub. Moreover, we find that community images add twice the amount of data to a parent image than official ones, suggesting the former can benefit from optimisations. Finally, we uncover that at release time, most images use relatively up-to-date parent images. However, we find that if one were to deploy the images today, over 70% would use an outdated parent, with a median of over 5 months of time lag. This indicates that Docker Hub users should pay additional attention to the lineage of the images they consume, and calls for better security insights provided by image registries.

Acknowledgements

This research was partially funded by the "Cybersecurity Initiative Flanders" project and the Research Foundation Flanders (FWO) under Grant No. 1SD4321N and V431423N.

References

[1] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

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

[3] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt, "Large-scale analysis of the Docker Hub dataset," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 2019, pp. 1–10.

[4] Z. Lu, J. Xu, Y. Wu, T. Wang, and T. Huang, "An empirical case study on the temporary file smell in Dockerfiles," *IEEE Access*, vol. 7, pp. 63 650–63 659, 2019.

[5] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps, "Learning from, understanding, and supporting DevOps artifacts for Docker," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, 2020, pp. 38–49.

[6] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on Docker Hub," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. ACM, 2017, p. 269–280.

[7] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, "Docker ecosystem–vulnerability analysis," *Computer Communications*, vol. 122, pp. 30–43, 2018.

[8] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall, "An empirical analysis of the Docker container ecosystem on GitHub," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 323–333.

[9] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Cimplifier: Automatically debloating containers," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, p. 476–486.

[10] A. Zerouali, T. Mens, A. Decan, J. Gonzalez-Barahona, and G. Robles, "A multi-dimensional analysis of technical lag in Debian-based Docker images," *Empirical Software Engineering*, vol. 26, no. 2, p. 19, Feb. 2021.

[11] M. U. Haque and M. A. Babar, "Well begun is half done: An empirical study of exploitability & impact of base-image vulnerabilities," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 1066–1077.

[12] E. Socchi and J. Luu, "A deep dive into Docker Hub's security landscape – a story of inheritance?" Master's thesis, University of Oslo, 2019.

[13] Y. Zhang, Y. Zhang, Y. Wu, Y. Lu, T. Wang, and X. Mao, "Exploring the dependency network of Docker containers: Structure, diversity, and relationship," in *Proceedings of the 12th Asia-Pacific Symposium on Internetware*. ACM, 2020, p. 199–208.

[14] Stack Overflow, "2022 Stack Overflow developer survey," https://survey.stackoverflow.co/2022/.

[15] A. N. Langville and C. D. Meyer, *Google's PageRank and beyond: The science of search engine rankings*. Princeton University Press, 2006.

[16] Q. Jiang, "Improving performance of Docker instance via image reconstruction," in *Proceedings of the 2022 International Conference on Big Data Intelligence and Computing (DataCom)*. Springer, 2023, pp. 511–522.

[17] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration," *Empirical Software Engineering*, vol. 23, pp. 384–417, 2018.

[18] J. Stringer, A. Tahir, K. Blincoe, and J. Dietrich, "Technical lag of dependencies in major package managers," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2020, pp. 228–237.

[19] C. Lin, S. Nadi, and H. Khazaei, "A large-scale data set and an empirical study of Docker images hosted on Docker Hub," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 371–381.

[20] Open Container Initiative, "OCI distribution specification." [Online]. Available: https://github.com/opencontainers/distribution-spec/blob/main/spec.md

[21] Y. Zhang, Y. Zhang, X. Mao, Y. Wu, B. Lin, and S. Wang, "Recommending base image for Docker containers based on deep configuration comprehension," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 449–453.

[22] A. Brogi, D. Neri, and J. Soldani, "DockerFinder: Multi-attribute search of Docker images," in *2017 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2017, pp. 273–278.

[23] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[24] H. Ibrahim, M. Sayagh, and A. E. Hassan, "Too many images on DockerHub! How different are images for the same system?" *Empirical Software Engineering*, vol. 25, 09 2020.

[25] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code — an empirical study," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 45–55.

[26] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 189–200.

[27] R. Opdebeeck, A. Zerouali, and C. De Roover, "Smelly variables in Ansible infrastructure code: Detection, prevalence, and lifetime," in *Proceedings of the 19th International Conference on Mining Software Repositories*. ACM, 2022, p. 61–72.

[28] E. van der Bent, J. Hage, J. Visser, and G. Gousios, "How good is your Puppet? an empirically defined and validated quality model for Puppet," in *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2018, pp. 164–174.

[29] R. Opdebeeck, A. Zerouali, and C. De Roover, "Control and data flow in security smell detection for infrastructure as code: Is it worth the effort?" in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023, pp. 534–545.

[30] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry," in *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution, Industrial Track*, 2019, pp. 580–589.

[31] I. Kumara, M. Garriga, A. U. Romeu, D. Di Nucci, F. Palomba, D. A. Tamburri, and W.-J. van den Heuvel, "The do's and don'ts of infrastructure code: A systematic gray literature review," *Inf. Softw. Technol.*, vol. 137, Sep. 2021.

[32] R. Opdebeeck, A. Zerouali, C. Velázquez-Rodríguez, and C. De Roover, "On the practice of semantic versioning for Ansible Galaxy roles: An empirical study and a change classification model," *Journal of Systems and Software*, vol. 182, Dec. 2021.

[33] A. Zerouali, R. Opdebeeck, and C. De Roover, "Helm charts for Kubernetes applications: Evolution, outdatedness and security risks," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023, pp. 523–533.

[34] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in OSS packaging ecosystems," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 2–12.

[35] R. G. Kula, C. De Roover, D. M. German, T. Ishio, and K. Inoue, "A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 288–299.

[36] K. Blincoe, F. Harrison, and D. Damian, "Ecosystems in GitHub and a method for ecosystem identification using reference coupling," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 202–211.