CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Investigating and Mitigating the Impact of Technical Lag and Different architectures on Container Image Security

Ensuring Secure and Reliable Deployment of Containerized Applications

Master's thesis in Computer science and engineering

Sina Darbouy and Mosope Williamson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Investigating and Mitigating the Impact of Technical Lag and Different architectures on Container Image Security

Ensuring Secure and Reliable Deployment of Containerized Applications

Sina Darbouy and Mosope Williamson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Investigating and Mitigating the Impact of Technical Lag and Different architectures on Container Image Security
Ensuring Secure and Reliable Deployment of Containerized Applications
Sina Darbouy and Mosope Williamson

Supervisor: Philipp Leitner, Chalmers CSE
Advisor: Anders Danielsen, Plejd
Examiner: Jennifer Horkoff, Chalmers CSE

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Investigating and Mitigating the Impact of Technical Lag and Different architectures on Container Image Security
Ensuring Secure and Reliable Deployment of Containerized Applications
Sina Darbouy and Mosope Williamson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Containerization technology has revolutionized software development and deployment by providing a more lightweight and scalable alternative to traditional virtualization. However, containers running on different architectures, such as ARM or AMD, use distinct images, which could lead to more security issues when selecting different architectures. Moreover, there needs to be more knowledge of how containers become outdated when released in production. This article addresses these questions by performing a deep comparison and analysis of two major container image architectures, namely ARM and AMD, for a dataset of 2500 container images using vulnerability scanners such as Clair, Trivy, Anchore, and Snyk and by measuring and comparing three dimensions of technical lag that Docker container images can face: time lag, version lag, and vulnerability lag. Finally, we applied what we learned from this research to our partner company to enhance the security of the company's container images in production.
Our results indicate no significant difference in the context of vulnerabilities among different architectures. Furthermore, our analysis revealed that the various dimensions of technical lag are complementary, providing multiple insights. Specifically, we found that official images consistently have a lower vulnerability lag than community images. Based on our findings, we recommend to our partner company the regular monitoring and updating of container images, with a focus on official images, to minimize vulnerability lag. In addition, our research highlights the importance of taking proactive measures to manage container image security in a production environment.

Keywords: Containerization, Security, Technical lag, Vulnerability scanners, Deployment, Software development, thesis.

Acknowledgements

We would like to express our deep gratitude to the people who made this project possible. First and foremost, we want to thank our supervisor, Philipp Leitner, for his valuable guidance, support, and feedback throughout our journey. His expertise and mentorship have been significant in shaping the direction and quality of our work.
We also want to acknowledge our sponsor at the partner company, Anders Danielsen, for initiating this project and allowing us to use the company's Kubernetes infrastructure. Their support and trust in us have been greatly appreciated, and we are grateful for the resources they provided.
Lastly, we thank our examiner, Jennifer Horkoff, for her insightful feedback and suggestions that helped us improve our work. Her expertise and attention to detail have been really helpful in improving my thesis. we appreciate their thorough examination and helpful input.

Sina Darbouy and Mosope Williamson, June 2023

Contents

List of Figures

List of Tables

1

Introduction

The use of containerization technologies has experienced a significant surge in recent years due to the emergence of modern software demands such as scalability, portability, microservices architecture, and resource efficiency. Bentaleb et al. [1] observed that "container technologies have emerged as a new paradigm to address intensive scientific application problems.". Moreover, the easy deployment of container images within a reasonable time frame and the few required computational resources they need make container technologies more viable solutions than virtual machines, hence their increasing popularity.

The rapid and broad adoption of containerization has also brought to light several security challenges that need to be addressed, particularly in container image vulnerabilities. One key issue is the technical lag between discovering vulnerabilities and releasing patches and updates to address them. This lag can leave containers vulnerable for extended periods, which poses a significant security risk to organizations. In addition, Zerouali et al. [2] confirmed that there needs to be more knowledge about whether containers used to deploy and maintain software systems are up-to-date when released or during production.

Another issue that needs to be paid more attention to is the choice of architectures for Docker images; choosing a specific architecture may cause more vulnerabilities. For example, Haq et al. [3] in an article presented a security analysis of Docker containers running on ARM architecture, which is commonly used in embedded devices such as Internet of Things (IoT) devices. By addressing the security issues, they highlighted potential security threats and vulnerabilities from using ARM containers at the ARM-based edge nodes. The result of this article inspired us to investigate the vulnerabilities associated with using Docker images based on ARM and AMD architecture, which is one of our research focuses.

1.1 Purpose of the study

The primary purpose of this study is to find potential vulnerabilities and problems associated with technical lag and different architectures in the context of container image security and to propose recommendations for mitigating their impact.

The outcome of this research can affect the decision in choosing between architec-

tures (ARM and AMD) in terms of which choice best mitigates container image vulnerability. In addition, the study can affect the balance between updating the Docker images and the risk of breaking changes to keep the security. This balance is essential because if we update the Docker images too frequently, there is a risk of encountering problems that could disrupt the stability of the production.

This would involve conducting a thorough analysis of vulnerabilities arising from using different architectures and the lag between their discovery and the release of patches and updates for Docker images. In addition, the study would aim to provide organizations with a better understanding of the potential risks and challenges associated with choosing the architecture or not updating the version of the images.

1.2 Research question

- **RQ1: What potential vulnerabilities can arise from choosing different architectures(ARM and AMD) for Docker images?** To ensure the security of their applications, developers must be aware of the potential risks associated with choosing different architectures (ARM and AMD) for Docker images. By understanding the factors that contribute to container security, developers can make informed decisions on which architecture to use.
- **RQ2: What is the extent of technical lag in Docker images in terms of vulnerability, version, and time lag?** New versions of software libraries are released over time. In this research question, we calculate and analyze the duration required (delay) for a software library to be updated to a recently released version. By understanding how to reduce the impact of technical lag, developers can better protect their applications and users.
- **RQ3: How much technical lag exists in Docker images used in industrial settings in terms of vulnerability, version, and time lag?** We apply what we have learned from this research to our partner company and give them recommendations based on the findings from the analysis of the company's container images in production.

1.3 Thesis outline

In Chapter 2, readers will find a thorough examination of the key concepts and relevant literature crucial for comprehending the study's topic. This comprehensive overview aims to equip readers with the necessary background knowledge to facilitate their understanding of the research findings discussed in the following chapters.

Chapter 3 of the research report provides a detailed explanation of the research methodology adopted in the study. It outlines the research design steps, collection and analyzing of data, and arrival at the findings. The chapter also highlights the different tools and techniques utilized to ensure the reliability and validity of the research results.

Chapter 4 presents the research study's findings based on the methodology described in Chapter 3. In addition, the chapter provides a detailed account of the analysis conducted on vulnerability across architectures and technical lag, which were the primary research objectives. The results presented in this chapter are supported by the data collected and analyzed using the research methods and tools described in the previous chapter, demonstrating the effectiveness of the research methodology.

Chapter 5 discusses the results from Chapter 4. It also addresses the research questions posed in Chapter 1. Finally, the chapter provides a comprehensive analysis of the findings, examining how they contribute to answering the research questions and discussing the study's limitations while suggesting areas for further research to build upon the findings.

Chapter 6, the conclusion, summarizes the main findings from the preceding chapters, highlighting the impact of technical lag and different architectures on container image security and providing recommendations for improving container image security practices.

2

Background

This section offers an in-depth analysis of the concepts and related literature relevant to our research topic, focusing on the key themes and ideas in the field. The review aims to identify the gaps and limitations that our research aims to address. Additionally, this chapter will explore the various research methodologies and methods employed in the literature and the relevance of these approaches to our research.

2.1 Docker

Docker is a powerful containerization architecture that has revolutionized how applications are developed, deployed, and run. At its core, Docker provides a way to package applications and their dependencies into portable containers that can be easily distributed and run on any compatible system. Using Docker, developers can build, ship, and run applications quickly and easily without worrying about the underlying infrastructure. In addition, Docker allows applications to be isolated from the host system, providing a consistent and reliable environment that can be easily replicated across different environments. [4]

2.2 Docker images

Docker images are the building blocks of Docker containers. They are snapshots of a file system that include an application's necessary components, including the application code, runtime, libraries, and other dependencies. Docker images are built into layers, each representing a change to the file system. For example, the first layer may include the operating system and runtime, while subsequent layers add the application code and any necessary libraries. Finally, these layers are combined to create a final image that includes all the application's necessary components. Docker optimizes storage and transfer by building images in layers, allowing images to be created and distributed quickly and efficiently. If a new version of the application is released, only the changed layers need to be updated, making updating and distributing images fast and efficient. [5]

2.3 Docker image repository

A Docker image repository is a collection of related Docker images, usually organized around a specific application or project. These repositories can be public or private and are typically hosted by a registry, such as Docker Hub or a private registry. [6]

2.3.1 Community and official docker hub images

Docker hub hosts a set of curated docker images reviewed to a high standard by the docker hub team with clear documentation that covers every detail on each particular image repository. These container images curated by the docker hub team are called official images. On the other hand, community images are created by docker users, hosted on the docker hub, and made available for anyone within the docker's community of users [7]. Although the community of users maintains community images for that specific docker image, it may not have as rigorous standards as an official image.

2.4 Layers

Docker images comprise a base layer and one or more child layers built on top of it, using a union file system. Each layer represents a different image part and is identified by a unique layer ID. The layered approach allows for more efficient use of storage space and faster image build times. However, each layer may contain vulnerabilities or security issues that could be exploited, so it is crucial to identify and mitigate these issues before deploying the image in production. Security scanners can analyze each Docker image layer to identify any security issues or vulnerabilities. [5]

2.4.1 Versions or tags in image

To identify and manage different versions of an application, Docker allows developers to assign version tags to images. Version tags are simply labels assigned to a specific image and can be used to differentiate between different versions of the same application. For example, an image for a web server might have tags such as "latest," "v1.0," and "v2.0" to differentiate between the most recent version, the first release, and the second release, respectively. [4]

It is worth noting that updating container images in a registry such as Docker Hub is not always a linear process. While the Docker image versioning scheme allows for images to be identified by a specific tag, such as "v1.0" or "latest," these tags are often mutable and can be updated over time. In addition, the content of an image can change without altering its tag, such as when a security vulnerability is patched.

2.5 Digest

In Docker images, a digest is a content addressable identifier that uniquely identifies the image's content by taking a collision-resistant hash of its bytes. The digest consists of a serialized hash result that includes the algorithm used to calculate the digest and the hex-encoded result of the hash. The algorithm is crucial in ensuring that the correct content is obtained, and while a variety of algorithms can be implemented, compliant implementations should use sha256 to maintain the uniqueness of the digest. [8]

In Docker, it is possible to set multiple tags for one image, meaning an image can have several different tags that all refer to the same underlying image data. For example, an image with the name "nginx:latest" and an image with the name "nginx:1.21.0" may both have the same underlying image data, identified by a unique digest value. In the Docker Hub repository, images with different tags but the same digest value will be displayed as separate entries.

2.6 Architecture

In Docker, architecture refers to the underlying operating system and hardware architecture that an application is built to run on. Docker enables the execution of containers on diverse architectures, allowing a single image to incorporate different variants tailored for various architectures and occasionally different operating systems like Linux, Windows, and macOS. [9]

The Docker Official Images available on Docker Hub offer a wide range of architectures. As an illustration, let us consider the Busybox image, which includes support for various architectures such as amd64, arm32v5, arm32v6, arm32v7, arm64v8, i386, ppc64le, and s390x. When running this image on an x86_64 (amd64) machine, it automatically pulls and runs the corresponding amd64 variant. [9]. This thesis will compare two popular architectures (amd64 and arm64v8).

2.7 Container scanning

Container scanning is a technique for examining the contents of container images to identify security vulnerabilities and other issues that may pose a risk to the system. Container scanning tools use layers of images to scan the file system and compare package versions against a known vulnerabilities database. The scan results are then used to generate a report that identifies potential vulnerabilities, outdated packages, and other issues that may need to be addressed. Developers can use this report to prioritize their efforts and focus on the most critical vulnerabilities first. Container scanning is an essential practice for ensuring the security of containerized applications, and it can help to identify and mitigate vulnerabilities before attackers exploit them. [10]

2.8 Vulnerability assessment: CVE and CVSS

CVE stands for Common Vulnerabilities and Exposures, a glossary for categorizing vulnerabilities. To assess the severity of a vulnerability, the glossary employs the Common Vulnerability Scoring System (CVSS) [11]. In addition, the CVE score is frequently employed to prioritize vulnerability security. The primary goal of the CVE glossary is to monitor and categorize vulnerabilities in consumer software and hardware. The project is overseen by the MITRE Corporation, funded by the US Division of Homeland Security [12]. Vulnerabilities are gathered and cataloged through the Security Content Automation Protocol (SCAP), which assesses the vulnerability details and assigns a unique identifier to each vulnerability.

After vulnerabilities undergo evaluation and identification, they are included in the MITRE glossary, which is accessible to the public. The National Institute of Standards and Technology (NIST) subsequently analyzes the vulnerabilities listed. In addition, the NIST's National Vulnerability Database (NVD) contains all relevant information regarding vulnerabilities and their analyses, ensuring public accessibility [13]. This allows individuals to enhance their understanding of the risks associated with various vulnerabilities and take appropriate measures to protect their systems.

2.9 Related work

The research on container image vulnerabilities and technical lag is moderately extensive. This section provides a concise overview of relevant related research and emphasizes our unique contribution to the existing body of knowledge in this field.

2.9.1 Vulnerability in open-source software

In a thesis, Hu et al. [14] introduced a knowledge graph-based algorithm to analyze the propagation of vulnerabilities in open-source software systems. First, they extracted the software system's information and built a knowledge graph, including software components, dependencies, and relationships. In the second step, they used a vulnerability scanner to identify the vulnerabilities in the software system, then used a modified version of the PageRank algorithm to rank the vulnerabilities based on their impact and the likelihood of being propagated to other components in the system. Our study expands upon this previous research by focusing on vulnerabilities in open-source software, explicitly concerning container images. We achieve this by analyzing the software components, also known as layers, present within each container image.

While operating a vulnerability assessment tool, which they developed, Ponta et al. [15] manually curated a dataset of fixes to vulnerabilities in open-source software (OSS) projects by analyzing the vulnerabilities and their corresponding fixes from several sources, including the National Vulnerability Database (NVD) and vulnerability trackers of popular OSS projects. The dataset can improve the accuracy and

completeness of vulnerability databases such as CVE (Common Vulnerability and Exposure) by providing more detailed information on the vulnerabilities and their corresponding fixes. In addition, this research assisted our investigation of how we can use vulnerability databases such as CVE to point out vulnerabilities within container images.

2.9.2 Vulnerability of container images

Martin et al. [16] made a vulnerability analysis of the Docker ecosystem, focusing on Docker images and containers. They identified and analyzed the vulnerabilities present in the Docker ecosystem using a combination of manual analysis and automated vulnerability scanning tools; the paper further discussed the challenges associated with vulnerability management in the Docker ecosystem, such as the rapid pace of image and container updates and the lack of transparency in vulnerability reporting. We built on this research by examining vulnerabilities in the Docker ecosystem but instead analyzing the vulnerabilities within the ARM and AMD architectures.

Wist et al. [17] presented a comprehensive study of the security vulnerabilities in Docker images hosted on Docker Hub, a popular architecture for sharing and distributing Docker images. The authors analyzed 2500 Docker images to uncover the prevalence of security vulnerabilities and to identify the most common types of vulnerabilities present in these images, their methodology for conducting the vulnerability analysis involved using various security tools (Clair and Anchore) to scan the images for vulnerabilities. The study results show that many Docker images hosted on Docker Hub contain vulnerabilities, with an average of 9 vulnerabilities per image. Furthermore, the most common types of vulnerabilities found in the images were related to outdated packages and software components, missing security patches, and poor coding practices.

Also, Kaur et al. [18] in a paper examined the security vulnerabilities that exist in container images used for scientific data analysis; they analyzed a dataset of over 10,000 container images from the Docker Hub registry used in scientific data analysis, focusing on the vulnerabilities that exist in the software packages and dependencies included in these images.

These papers relate to our research in our methodology design on how to conduct vulnerability research for container images. They also guide the analysis of data for our research.

2.9.3 Vulnerability scanning tools

Berkovich et al. [19] investigated the architecture and workflows of container image scanners which includes: Trivy, Anchore, Clair, and Microscanner to understand their behavior and share their findings to help the DevSecOps community choose the best scanner for their environment. In further research [20], they proposed a benchmarking methodology called Ultimate Benchmark for Container Image Scan-

ning (UBCIS) that evaluates container image scanning tools based on several factors, including the number of vulnerabilities detected, the severity of those vulnerabilities, the time it takes to scan an image, and the false positive rate.

Javed and Toor [21] in their paper evaluated various container security vulnerability detection tools and compared their effectiveness in detecting vulnerabilities in container images. They conducted experiments using a set of vulnerable container images and analyzed the results of the scans performed by each tool, and the tools include Clair, Anchore, and Microscanner. They concluded that multiple tools should be used in combination to achieve comprehensive vulnerability detection in container images.

Our research further built on these papers by employing a combination of vulnerability scanning tools for static analysis. It reinforced our reasoning that scanning for the vulnerability of container images is more effective with a multi-dimensional approach.

2.9.4 Technical lag

Zerouali et al. [22] presented the concept of "technical lag" as the increasing lag between upstream development and the deployed system when no corrective actions are taken to quantify how outdated a deployed software package release compared to the "ideal" situation. The study proposed a formal framework that allows them to calculate technical lag for any component in a repository. Finally, they applied this framework to analyze technical lag in the npm repository.

Zerouali et al. [23], in another research, investigated the relationship between outdated Docker containers, severity vulnerabilities, and bugs by analyzing over 600,000 Docker images from the Docker Hub registry. Docker images were categorized based on the age of the image, with images less than 1 year old considered "fresh," images between 1 and 2 years old considered "stale," and images older than 2 years considered "expired." The study results suggest that outdated Docker containers are associated with higher vulnerability and bug rates.

Building on previous research on Technical Lag in which, they have proposed a set of metrics to measure technical lag, including the time between the release of a new version of a component and its first use in a downstream project, as well as the proportion of downstream projects using the latest version of a component. Zerouali et al. [2] used these metrics to analyze technical lag in a sample of Debian-based Docker images to identify factors that contribute to technical lag, such as the size and complexity of images, the popularity of components and the availability of security updates. They collected a unique dataset of Debian-based Docker images, analyzed what versions of packages are being used in these images, and looked up the versions to the vulnerability database.

Zerouali et al. [22] proposed a technical lag measurement framework that provides a formal approach to capture the various viewpoints and methods of measuring technical lag. This framework was developed and validated through case studies

focused on npm packages. Our formula was developed based on the formula proposed in the article, further elaborated in the following paragraph.

The technical lag framework F can be described as a collection of different elements. It consists of a set called C, which includes all the component releases. There is also another set called ℓ, which represents the possible lag values.

Within this framework, a function called ideal is denoted as ideal: $C \to C$, which helps identify the most preferred component release.

Additionally, there is a function called Δ, denoted as $\Delta : C \times C \to \ell$, which calculates the difference in terms of lag between two component releases.

Lastly, there is a function called agg, denoted as agg : $\wp(\ell) \to \ell$, which takes a set of lag values as input and provides an aggregated result.

Zerouali et al. [2] investigated and compared the technical lag of Debian-based Docker images using the above framework with five dimensions (package lag, time lag, version lag, vulnerability lag, and bug lag). For our formula, we have identified three dimensions within this article that are relevant to our work. However, we have reduced the original five dimensions mentioned in the article to three dimensions in our study. This modification was necessary because the article specifically analyzed open-source Debian-based Docker images, allowing access to the repository to check packages and issues. In contrast, our thesis focuses on the entirety of Docker Hub, where we do not have direct access to those dimensions mentioned in the article.

3

Methods

This chapter outlines and describes the methodology used to research to answer the relevant research questions mentioned in Chapter 1. This research aims to identify the security risks associated with technical lag and architecture differences in container images and propose solutions to mitigate these risks. The chapter describes the research design, data collection, analysis methods, the tools and techniques used in the study; for easier understanding, the research method is illustrated in Figure 3.1, Figure 3.3 and Figure 3.4 to answer research question 1, 2 and 3 respectively.

3.1 RQ1

This section describes how data was collected and the process of scanning container images to identify vulnerabilities. Figure 3.1 visually represents the steps undertaken to address research question one.

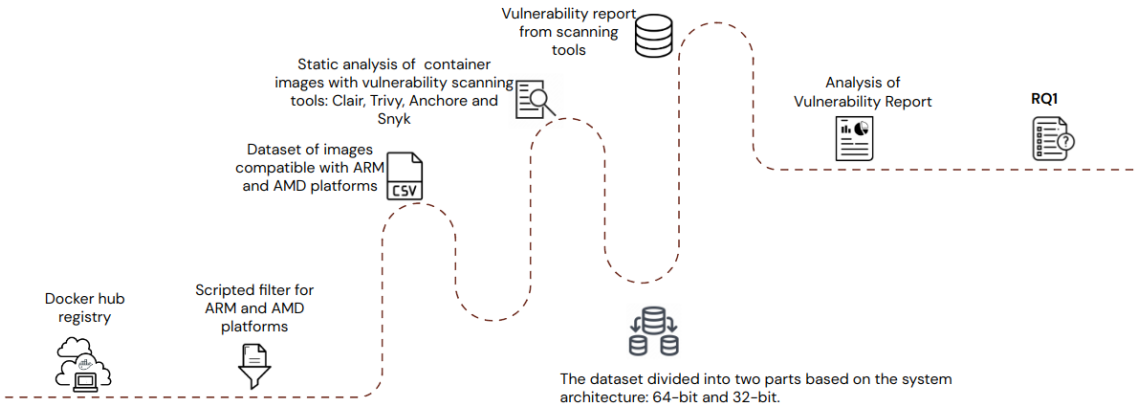

Figure 3.1: Pathway illustrating the steps used to collect and analyze data for Research Question 1. The process includes data collection from DockerHub, filtering the images based on their architecture (ARM and AMD), data storage, static analysis of the container images, and storage for analysis and analysis of the resulting vulnerability report

3.1.1 Data collection

For an empirical study, we used a quantitative research approach. We created a program that collects information on 40,852 commonly used Docker images from DockerHub, which currently contains over 100,000 container images [24]. The program uses the official Docker Hub API to retrieve the images based on their number of pulls. We assumed that images with a higher number of pulls are popular. Docker Hub serves as a cloud-based repository where container community developers, open-source projects, and independent software vendors (ISVs) can build and distribute their code in containerized form [4]. This process can be seen as the first step in Figure 3.1.

Table 3.1: Overview of the data

	Count	**Unique values**
Image	40,852	170 unique images
Architecture		'amd64', 'arm64', 'arm'
Variant		nan, 'v8', 'v7', 'v5', 'v6'

Our script was made to collect details about images from Docker Hub into a CSV file which includes: the image name, version, architecture, variant, date released, and digest of each image, respectively. Since our research focused on comparing ARM and AMD architectures vulnerabilities, we conducted a rigorous filtering process to gather the top 10 versions of each container image that support both ARM and AMD architectures. As a result, a dataset consisting of 2,119 container images that suitable for analysis. This process can be seen as the second step in Figure 3.1.

To ensure a comprehensive comparison, we partitioned the dataset into two subsets - one for 64-bit architectures such as amd64, arm64, and armv8, and the other for 32-bit architectures such as armv5 and armv7. This segregation was necessary to detect potential vulnerabilities that might exist in 64-bit libraries but not in their 32-bit counterparts. Table 3.2 and Table 3.3 below present a preview of the dataset's header.

Table 3.2: Sample container images for 32-bit architecture

Image	**Version**	**architecture**	**Variant**	**Date Released**	**Digest**
debian	testing-slim	arm	v5	2023-02-09T02:28:08	sha256:98425f...
debian	testing-slim	arm	v7	2023-02-09T06:44:58	sha256:3fbc93...
debian	testing-backports	arm	v5	2023-02-09T02:28:17	sha256:bc2b3c...
debian	testing-backports	arm	v7	2023-02-09T06:46:39	sha256:cf5b2c...
debian	stable-slim	arm	v5	2023-02-09T02:27:30	sha256:f22577...

Table 3.3: Sample container images for 64-bit architecture

Image	Version	architecture	Variant	Date Released	Digest
centos	latest	amd64	none	2021-09-15T18:38:28	sha256:a1801b...
centos	latest	arm64	v8	2021-09-15T17:56:15	sha256:65a4aa...
centos	centos7.9.2009	amd64	none	2021-09-15T18:38:13	sha256:dead0...
centos	centos7.9.2009	arm64	v8	2022-02-14T19:55:27	sha256:73f11af...
centos	centos8.3.2011	amd64	none	2020-12-08T00:39:12	sha256:dbbace...

The box plot in Figure 3.2 displays the range of release dates based on architecture, revealing that our date range spans from 2020-12-06 to 2023-03-12. The plot highlights a broader range for the amd64 architecture, likely due to the more significant number of AMD architectures available in the Docker Hub repository compared to ARM. For the first research question, it is worth noting that some images in our dataset only support one architecture and do not provide multi-architecture support. Consequently, we filtered our data only to include images supporting ARM and AMD.

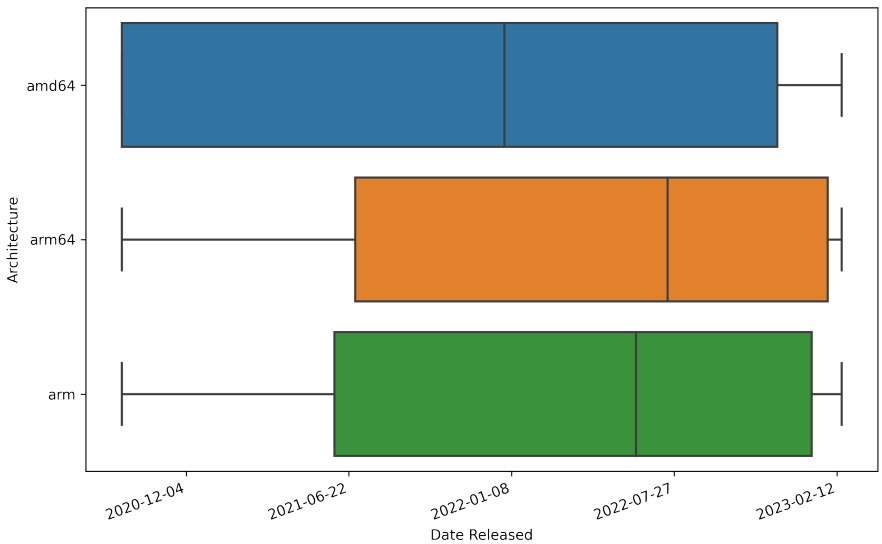

Figure 3.2: Box plot illustrating the relationship between architecture and date of release.

3.1.2 Scanning container images for vulnerabilities

Learning from our literature review, Javed and Toor [21] recommended combining multiple tools to achieve comprehensive vulnerability detection. Hence, we decided to try various tools for our analysis: Clair, Anchore, Trivy, and Snyk. However, on initial test analysis, we found that Anchore and Snyk required a paid tier to be able to scan all our container images. Hence, it left Clair and Trivy as available options. However, on further comparison, Clair has a more extensive vulnerability database and can scan for more vulnerabilities than Trivy [25]. Thus, we settled on using Clair for all our vulnerability scans.

We followed a systematic approach to investigate the impact of different architectures on container image security. We started by using our retrieved set of Docker images in CSV files as the input for our analysis. We then used the Clair tool, an open-source vulnerability scanner specifically designed for container images, to scan each image and detect any security issues.

Clair scans for known vulnerabilities in package dependencies and libraries, providing a comprehensive view of the security posture of each image. Clair works by fetching the layers of the scanned container image, scans the content of these layers, and returns an index Report; it matches this index report with a database of correlating vulnerabilities affecting the container image and returns a list of these vulnerabilities and their severity [26].

We then calculated each image's total number of vulnerabilities, without considering the severity status, as part of the first research question. This allowed us to estimate the number of vulnerabilities that might be present in each image and identify images that need further inspection, as shown in Table 3.4 below.

Table 3.4: Sample analysis of clair reports for 64-bit architecture

Image	**Version**	**architecture**	**Variant**	**Total vulnerabilities**
centos	latest	amd64	none	142
centos	latest	arm64	v8	133
centos	centos7.9.2009	amd64	none	45
centos	centos7.9.2009	arm64	v8	45
centos	centos8.3.2011	amd64	none	171

3.2 RQ2

This section describes how data was collected and the process of scanning container images to identify vulnerabilities. Figure 3.3 visually represents the steps undertaken to address research question one.

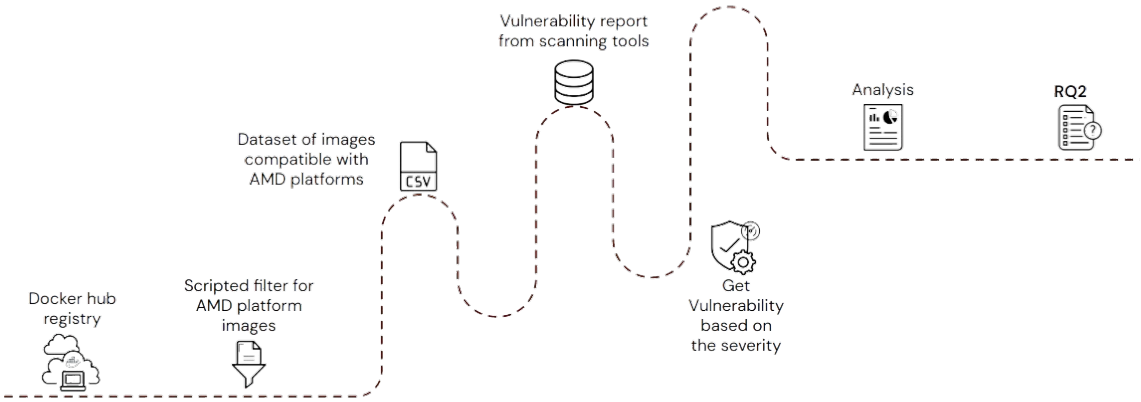

Figure 3.3: Pathway illustrating the steps used to collect and analyze data for Research Question 2. The process includes data collection from DockerHub, filtering the images based on AMD architecture, data storage, static analysis of the container images, and storing reports based on the severity and analysis of the resulting vulnerability report

3.2.1 Data collection

For the second research question, we used the dataset gathered from RQ1 and selected only the top 5 versions of each container image that supported AMD architecture. We chose AMD based on the results of the first research question, where we established that there is no significant difference in vulnerabilities between both architectures. Therefore, selecting ARM or AMD is the same regarding vulnerability, especially considering that we have gathered more data on AMD architecture and our machine was AMD. This filtering process resulted in a unique dataset of 1,875 container images for our analysis. This process can be seen as the second step in Figure 3.3.

3.2.2 Scanning container images for vulnerabilities

For the purposes of our technical lag analysis, we have incorporated severity status information into our data collection process. Our script scans the results of the scanner and extracts the number of vulnerabilities based on their respective severity levels. As a result, our final dataframe for technical lag analysis is structured as shown in the Table 3.5 below.

3.2.3 Dimensions of technical lag

Our goal in this study is to investigate and compare the technical lag of Docker images, using various methods for measuring such lag. To achieve this, we have

Table 3.5: Clair report for Technical lag data

Image	Critical	High	Medium	Low	Negligible	Unknown
nginx-stable-perl	0	1	7	6	57	19
nginx-stable-alpine-perl	0	0	0	0	0	3

identified three key dimensions:

- **Time lag**: the time difference between Docker image releases;
- **Version lag:** measures the number of missed versions between Docker image releases;
- **Vulnerability lag:** evaluates the difference in the number of vulnerabilities.

These dimensions align with the technical lag framework F=(C, L, ideal, delta,agg) presented in Section 2.9.4 and fully defined in Zerouali et al. [2]. To operationalize these dimensions, we introduce a range of functions.

We have established a series of functions $\Delta\alpha$ with $\alpha \in \{\text{time}, \text{vers}, \text{vuln}\}$ to calculate the variance between two image releases.

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

The function calculates the time gap, in months, between the release dates of two image versions, p, and q.

$$\Delta_{\text{vuln}} : \mathcal{P} \times \mathcal{P} \to \mathbb{Z} : (p, q) \to \sum_{j=1}^{n} w(j) \cdot (vuln(j, \text{q}) - vuln(j, p))$$

This function calculates the difference in the number of security vulnerabilities between two image releases.where n is the number of severity levels, $w(j)$ is the weight for severity level j,

$$\begin{aligned}\Delta_{\text{vers}} : \mathcal{P} \times \mathcal{P} \to \mathbb{Z} : (p, q) \to \\ & w_{\text{major}}(q_{\text{major}} - p_{\text{major}}) \\ & + w_{\text{minor}}(q_{\text{minor}} - p_{\text{minor}}) \\ & + w_{\text{patch}}(q_{\text{patch}} - p_{\text{patch}})\end{aligned}$$

This function computes the difference in the number of versions between two image releases where w is the weight for each version level.

3.2.4 Weight calculation

This section explains the weight calculation methodology used in the Vulnerabilities lag and Version lag formula.

3.2.4.1 Vulnerabilities

We rely on the CVSS Scoring Metrics, explained in Section 2 of this thesis, to assign weights to the severity levels in the Vulnerabilities lag calculation. Specifically, we use the maximum possible score for each severity level, as defined by the CVSS standard. For instance, we assign a weight of 10 to the "Critical" severity level, 9 to "High", 7 to "Medium", and 4 to "Low". These weights reflect the relative importance of each severity level in terms of the potential impact of the corresponding vulnerabilities on the system's security.

3.2.4.2 Version

We assigned different sets of weight values to the major, minor, and patch components of the version number format, respectively. The weights are based on the belief that each component of the version number signifies a distinct level of importance. We selected multiple sets of weight values using the following sets of weight values for major, minor, and patch components of the version number format: [3, 2, 1], [10, 5, 1], and [20, 5, 1] in increasing order to examine if different weight values would produce different results and to explore the impact of version lag in our overall analyses. The major component is typically used to denote major alterations, such as significant new features or breaking changes, while the minor component is used for minor improvements or bug fixes. The patch component is generally reserved for minor bug fixes and minor changes.

3.2.4.3 Example

Table 3.6 presents information on four distinct versions of the nginx image, and the three final columns in the table provide details on the vulnerability lag, version lag (using [3, 2, 1] as importance weights for major, minor, and patch components), and time lag for each version.

Table 3.6: Clair report for nginx perl

Image	Version	Date	Vuln Lag	Vers Lag	time lag
nginx	1.23.2-perl	2022-12-06	0	0	0
nginx	1.22.0-perl	2022-10-05	6	4	2
nginx	1.23.1-perl	2022-10-05	6	1	2
nginx	1.23.0-perl	2022-07-12	103	2	5

3.3 RQ3

The third research question investigates the extent of technical lag in Docker images used in industrial settings, focusing on vulnerabilities, versioning, and time lag. Specifically, we will apply the technical lag framework to a real-world setting, such as a company that uses Kubernetes for deploying their applications, to determine the magnitude of technical lag in practice. After obtaining results from the technical lag formula, we conducted expert interviews to gain further insights into our findings. The insights from these interviews provide valuable additional context to our assessment of technical lag in Docker images. The methodology used to answer the third research question is illustrated in figure 3.4 below for better understanding.

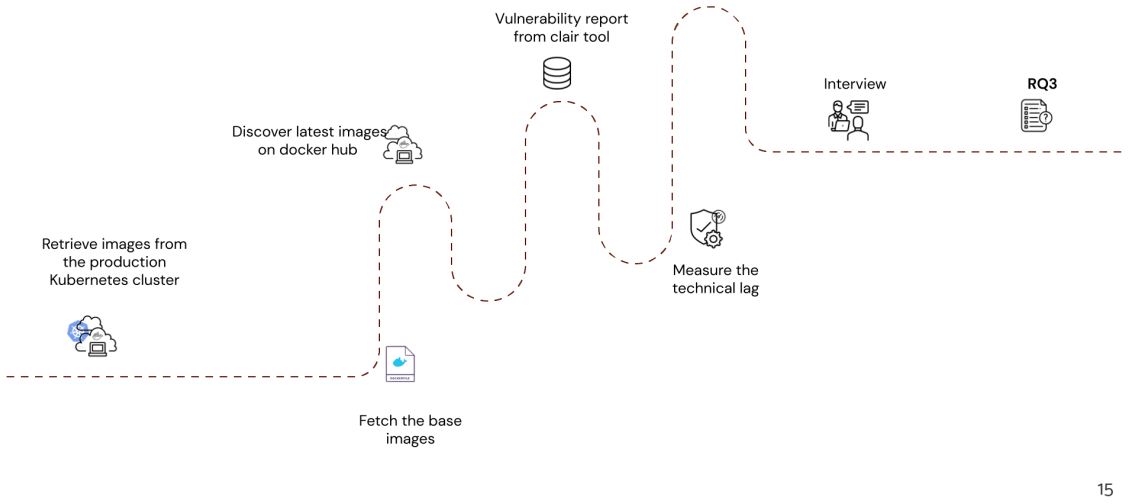

Figure 3.4: Pathway illustrating the steps used to apply technical lag in the industry for Research Question 2. The process includes retrieving images from Kubernetes production, fetching base images, getting the latest image of base images, scanning images with the Clair scanning tool, applying the technical lag framework, and interviewing experts to evaluate the Technical Lag framework

3.3.1 Applying the technical Lag framework

To apply the Technical lag framework to the company's container images, we followed a step-by-step process illustrated in figure 3.5 below for easier understanding. To achieve this, we begin by collecting all images in the Kubernetes environment using a script that gathers all images used within the company's Kubernetes infrastructure.

In order to successfully apply the technical lag formula to the collected images, we must identify the base image used in production for each image in the Kubernetes environment and also identify the ideal image, which represents the version of an

image with the latest tag; Identifying the base images will involve using a script to check the Dockerfile of each image in the Kubernetes infrastructure and recording which container image is used as a base image into a text file.

To determine the ideal image version for each base image used in the Kubernetes environment, we employed the same reasoning used for our previous research questions regarding vulnerability, time, and version lag, i.e., the ideal image is typically the latest version. But, finding the latest image version that we can use in our formula requires further steps.

As an example, consider a base image used in one of the images in the Kubernetes environment, such as nginx:18.2.1. To identify the ideal image for this base image, we first need to find the latest version of the nginx container image in Docker Hub, which is usually tagged as "latest". But, using this "latest" tag would result in us losing the version lag dimension.

However, since we can assign multiple tags to a single image, as mentioned in the background section 2.4.1, Docker Hub displays multiple entities for a single digest with different tags. We solve the problem of losing the version lag dimension by finding another tag for the same image.

To find another tag for the same image, we need to obtain the digest of the "latest" tag and locate another image with the same digest. For instance, in our example, the digest of nginx:latest is the same as that of nginx:20.1.2. We can use the nginx:20.1.2 image version to calculate the technical lag in our formula.

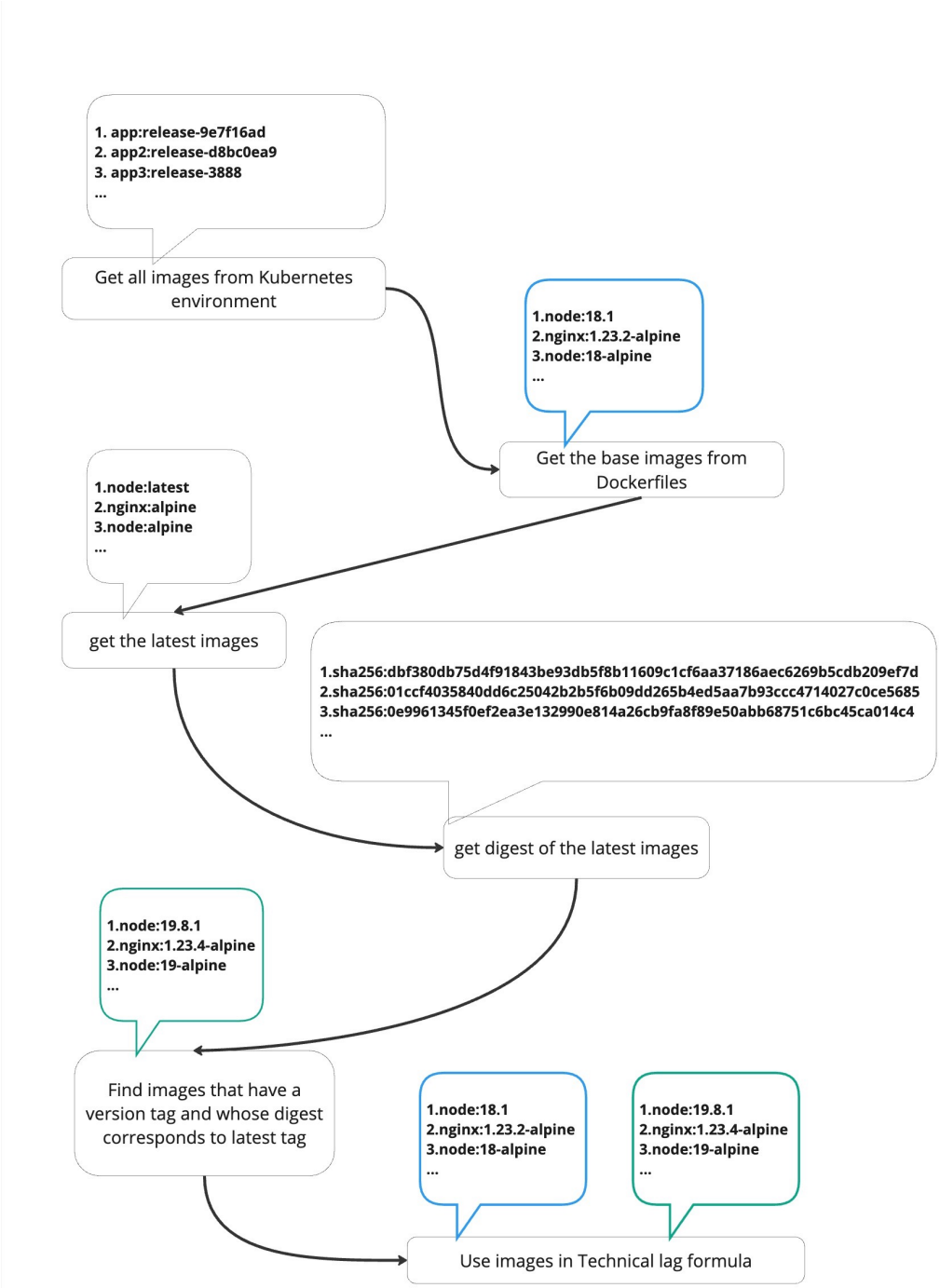

Figure 3.5: Process chart illustrating the steps involved in obtaining base images and ideal images for the technical lag formula in an industrial setting.

3.3.2 Evaluating technical lag in docker images for industrial use: Insights from expert interviews

This section describes our approach to conducting and analyzing expert interviews on evaluating technical lag in Docker images for industrial use. The interviews provided valuable insights into our results from applying the technical lag formula to the company. We explained the procedures we followed in selecting experts, how we conducted the interviews, and how we analyzed and interpreted the data obtained.

3.3.2.1 Participants

A total of five interviews were conducted for this study. The interviewees were part of the deployers team and developers team. Purposive sampling was employed for selecting interviewees based on their experience and expertise in deploying and developing Docker images in industrial settings. Table 3.7 presents a list of the interviewees, along with their experience working with Docker images (measured in years) and their current role in the company (as indicated by the 'role' column). The interview guide used in this study is included in Appendix A, providing readers with a detailed overview of the questions and prompts used during the interviews. These questions aimed to elicit information about the interviewees' experiences, challenges, and best practices in securing the container infrastructure.

During the interviews, the primary goal was to gather feedback and insights from the developers and deployers regarding the outcomes achieved. It was gratifying to observe that the interviewees expressed satisfaction with the results, indicating that the study positively impacted improving the security of the container infrastructure.

Table 3.7: Table of interviewees

Subject	Role	Experience
Interviewee 1	Cloud Engineer	16
Interviewee 2	Cloud Engineer	12
Interviewee 3	Cloud Engineer	10
Interviewee 4	Software Engineer	7
Interviewee 5	Software Engineer	5

3.3.2.2 Procedure

The interviews were conducted in person using a semi-structured approach. The interview questions were designed to explore the technical lag in Docker images used in industrial settings regarding vulnerability, version, and time lag. The interviews were conducted in a private setting, and notes were taken during the interviews to record the responses of the participants. Each interview lasted between 45 to 60 minutes.

3.3.2.3 Data analysis

The notes taken during the interviews were used to analyze the responses of the participants. A qualitative content analysis approach was employed to identify and categorize themes and patterns related to the research questions.

3.3.2.4 Ethical considerations

The study was conducted in accordance with ethical guidelines for research involving human participants. All participants were provided with an information sheet outlining the purpose and nature of the study, and they provided written informed consent prior to the interview. Confidentiality and anonymity were ensured throughout the study, and participants were given the option to withdraw from the study at any time.

4

Results

This chapter presents the results of the research method conducted on the data collected, as described in Chapter 3. The analysis aimed to answer the research questions and objectives, focusing on architecture vulnerability and assessing technical lag in container images. The chapter provides a detailed account of the findings, presented clearly and concisely, supported by tables, graphs, and other visual aids where applicable.

4.1 RQ1

What potential compatibility issues and vulnerabilities can arise from choosing different architectures(ARM and AMD) for Docker images?

In this section, we use the results of our research to understand the potential risks associated with two different architectures (ARM and AMD), which are essential for deployers who want to ensure there are no different security issues between architectures.

4.1.1 Vulnerability analysis of container images

In Section 3.1.1, we collected a dataset of container images from Docker Hub, consisting of 2,119 entries, which were subsequently used as input for our vulnerability scanning tools.

We were particular about using multiple scanning tools that were open source and free to use; this is to discover a wider variety of vulnerabilities for each image [20]. It was noted that some vulnerability scanners do not support some container images because vulnerability scanners like Clair rely on a set of known package managers and operating systems to perform their vulnerability scans. For example, suppose an image is built with an uncommon package manager or operating system that Clair does not support. In that case, it may not be able to perform a complete scan of the image, which means the scan of these container images returns an error message or reports that it has no vulnerability even though vulnerabilities may exist. For successful analysis encompassing 880 entities, the scanners generated results as a JSON file, as depicted in Figure 4.1 below. This file includes information such

as Vulnerability ID, Severity level, Package name and version, and vulnerability description.


```
"vulnerabilities": [
        {
            "featurename": "cyrus-sasl-lib",
            "featureversion": "2.1.27-5.el8",
            "vulnerability": "RHSA-2022:0658",
            "namespace": "centos:8",
            "description": "The cyrus-sasl packages contain the Cyrus implementation of Simple Authentication and
Security Layer (SASL). SASL is a method for adding authentication support to connection-based protocols. Security
Fix(es): * cyrus-sasl: failure to properly escape SQL input allows an attacker to execute arbitrary SQL commands
(CVE-2022-24407) For more details about the security issue(s), including the impact, a CVSS score, acknowledgments,
and other related information, refer to the CVE page(s) listed in the References section.",
            "link": "https://access.redhat.com/errata/RHSA-2022:0658",
            "severity": "High",
            "fixedby": "0:2.1.27-6.el8_5"
        },
        {
            "featurename": "bind-export-libs",
            "featureversion": "32:9.11.20-5.el8",
            "vulnerability": "RHSA-2021:1989",
            "namespace": "centos:8",
            "description": "The Berkeley Internet Name Domain (BIND) is an implementation of the Domain Name System
(DNS) protocols. BIND includes a DNS server (named); a resolver library (routines for applications to use when
interfacing with DNS); and tools for verifying that the DNS server is operating correctly. Security Fix(es): * bind:
An assertion check can fail while answering queries for DNAME records that require the DNAME to be processed to
resolve itself (CVE-2021-25215) For more details about the security issue(s), including the impact, a CVSS score,
acknowledgments, and other related information, refer to the CVE page(s) listed in the References section.",
            "link": "https://access.redhat.com/errata/RHSA-2021:1989",
            "severity": "High",
            "fixedby": "32:9.11.26-4.el8_4"
        },
        ...
```


Figure 4.1: Vulnerability analysis report for centos8.3.2011

Through data analysis, we retrieved the vulnerability count for each scanned image from the vulnerability report; Table 4.1 below shows the total vulnerability count of 4 variants of the Debian container image. Next, we compared the vulnerability report of the container images between the two architectures we are observing (ARM and AMD) to identify any significant differences in the count by categorizing our results according to which architecture had the most vulnerabilities in the image versions by using a script to retrieve the maximum vulnerability count according to architectures. Our analysis identified some images that showed a slight disparity in vulnerability count between the ARM and AMD architectures.

Table 4.1: Total vulnerability count for debian container image

Image	architecture	Variant	Digest	Total vuln.
debian	arm	v5	sha256:98	120
debian	arm	v7	sha256:is29	119
debian	amd64	none	sha256:87j	118
debian	arm64	none	sha256:38j	118

On further analysis, while looking into the reason for the slight difference in vulnerability between the architectures, namely: amd64 and armv7 or armv5, we investigated what CVE(vulnerabilities) exist on 64-bit libs but do not exist on 32-bit.

Hence, we split the data into two categories, i.e., 32-bit and 64-bit. In 32-bit, we compared arm v5 and arm v7; for 64-bit, we compared amd64 with arm64 or armv8. Table 4.2 shows the result for the vulnerability count of 64-bit architecture images; this is illustrated by the bar chart in Figure 4.2 below.

Table 4.2: Difference in vuln. count for 64-bit architecture.

Architecture	Total images
amd64	8
arm64	2
No-diff	516

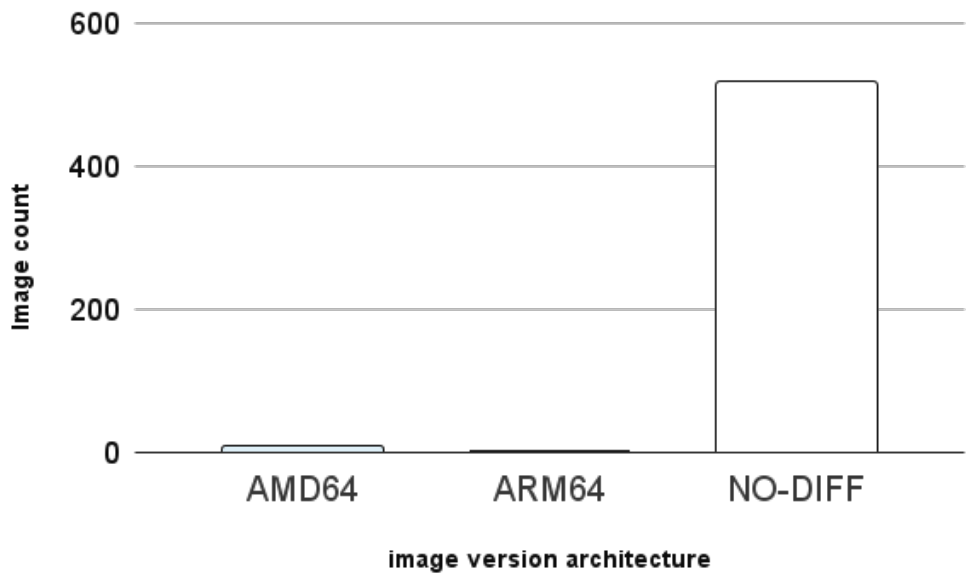

Figure 4.2: Difference in vuln. count for 64-bit architecture images.

Figure 4.2 shows that when comparing the difference vulnerability count for 64-bit architecture, out of a total of 536 images, 8 container images had more vulnerabilities in their amd64 versions, 2 images had more vulnerability in the arm64 versions while 516 images had no difference in vulnerability count between amd64 and arm64 architectures. The result of comparing the difference in vulnerability count for 32-bit architecture is shown in Table 4.3 and Illustrated in Figure 4.3 below, out of a total of 110 images, 1 container image had more vulnerabilities in its v5 version, no images had more vulnerability in their v7 versions while 108 images had no difference in vulnerability count between v5 and v7 32-bit architectures.

Table 4.3: Difference in vuln. count for 32-bit architecture.

Architecture	Total images
v5	1
v7	0
No-diff	108

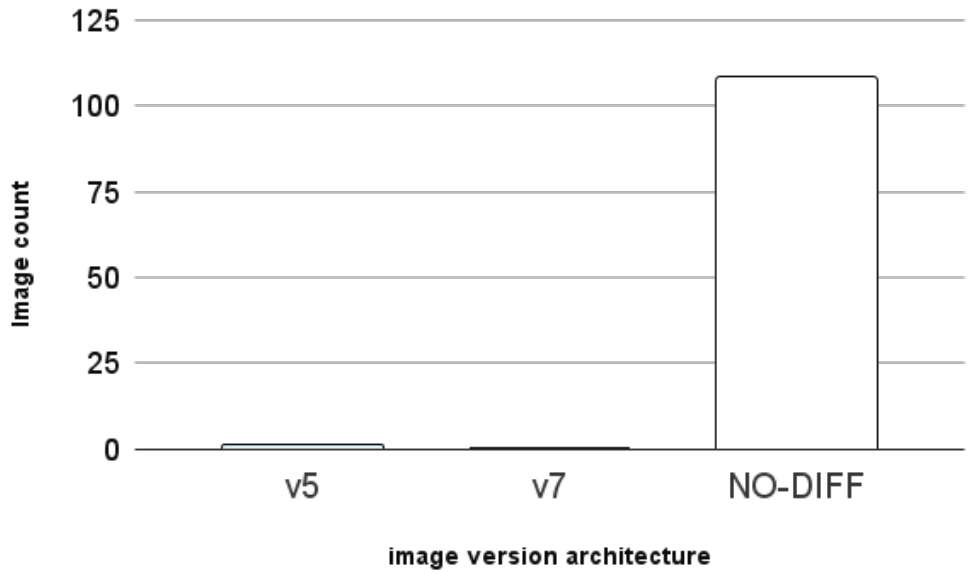

Figure 4.3: Difference in vuln. count for 32-bit architecture images.

We used various Docker commands and tools to understand the cause of these differences, including 'docker inspect', 'docker history', and 'dive'. We focused our investigation on differences introduced by the first layer of the images, which corresponds to the base image. Our approach involved a combination of automated scanning and manual inspection, which allowed us to gain deeper insights into the impact of different architectures on container image security. By using Clair to scan images and leveraging Docker tools to identify differences between the vulnerabilities, we were able to accurately and efficiently identify potential vulnerabilities in each image.

As we delved deeper into the images, we realized that the issue lies with the base layers. For instance, let's consider the 'crate' image. We discovered that this image is more susceptible to vulnerabilities on AMD architectures because the underlying base layer image 'centos-7' has a higher vulnerability rate on AMD systems. This discrepancy in vulnerability primarily stems from the latest version of the images, which is reliant on the base layer image. Therefore, instead of altering the crate image's version, we update its digest with a newer version of the base image as soon as the base image resolves the issue.

Regarding the first research question, which asks about potential compatibility issues and vulnerabilities associated with different architectures (ARM and AMD) for Docker images, our findings indicate that vulnerability levels in Docker images are only slightly different across architectures. Therefore, choosing an architecture based solely on vulnerability considerations is not necessary. Our research influenced our partner company that employs Kubernetes infrastructure, to opt for the ARM architecture due to its cost-effectiveness. However, we cannot conclude that either ARM64 or AMD64 is superior since the difference in vulnerability between both architectures is nearly negligible.

Our recommendation is to avoid using digests to specify Docker images whenever possible. This is because when a base layer image is updated to fix a vulnerability, the image version remains unchanged, but the digest is updated instead. If you specify a digest, your application will continue to use the old version, or you may encounter an error if the specified digest is no longer available.

4.2 RQ2

What is the extent of technical lag in Docker images in terms of vulnerability, version, and time lag?

To answer this research question, we started a filtered list of images for images that are compatible with AMD with 10 of their versions each, since from answering the first research question in Section 4.1, we now know the difference in vulnerability count between images of both ARM and AMD architectures is not statistically significant. This dataset comprises 2776 images, including 996 official Docker and 1780 community images.

To calculate the Technical lag for each image, we wrote a script to calculate vulnerability, version, and time lag using the formula as described in section 3.2.3. For time lag, the function calculates the time gap (in months) between each image and the latest available image for that particular image (i.e., the ideal image). To calculate version lag, our script computes the difference in the number of versions between image releases while assigning a weight w for each version level; section 3.2.4 explains the reasoning for the version weights.

4.2.1 Technical lag in terms of time lag

Statistical analysis revealed an average time lag of 5 months for community images and 3.4 months for official images when comparing the release dates of different versions and the latest version of each image. The time lag between older versions and the latest version of the amazon/aws-alb-ingress-controller community image is shown in Table 4.4 as an example, while Table 4.9 demonstrates the time lag for the nginx official image. These tables highlight that official images receive updates more frequently within shorter intervals than community images. Additionally, the

graph in Figure 4.4 illustrates that the time lag in official images follows a more linear pattern, whereas community images exhibit jagged peaks and drops, indicating variations in the time difference between releases of the latest versions.

Table 4.4: Time lag in amazon/aws-alb-ingress-controller image versions

Image	Version	Date	Time lag
amazon/aws-alb-ingress-controller	v2.4.5	2022-11-12	0
amazon/aws-alb-ingress-controller	v2.4.4	2022-09-23	2
amazon/aws-alb-ingress-controller	v2.4.3	2022-08-10	3
amazon/aws-alb-ingress-controller	v2.4.2	2022-05-25	6
amazon/aws-alb-ingress-controller	v2.4.1	2022-03-16	8
amazon/aws-alb-ingress-controller	v2.3.1	2021-12-08	13

Table 4.5: Time lag in nginx official image versions

Image	Version	Date	Time lag
nginx	1.23.2-perl	2022-12-06	0
nginx	1.22.0-perl	2022-10-05	2
nginx	1.23.1-perl	2022-10-05	2
nginx	1.23.0-perl	2022-07-12	5
nginx	1.21.6-perl	2022-05-28	7
nginx	1.20.2-perl	2022-05-17	7
nginx	1.21.5-perl	2021-12-29	12

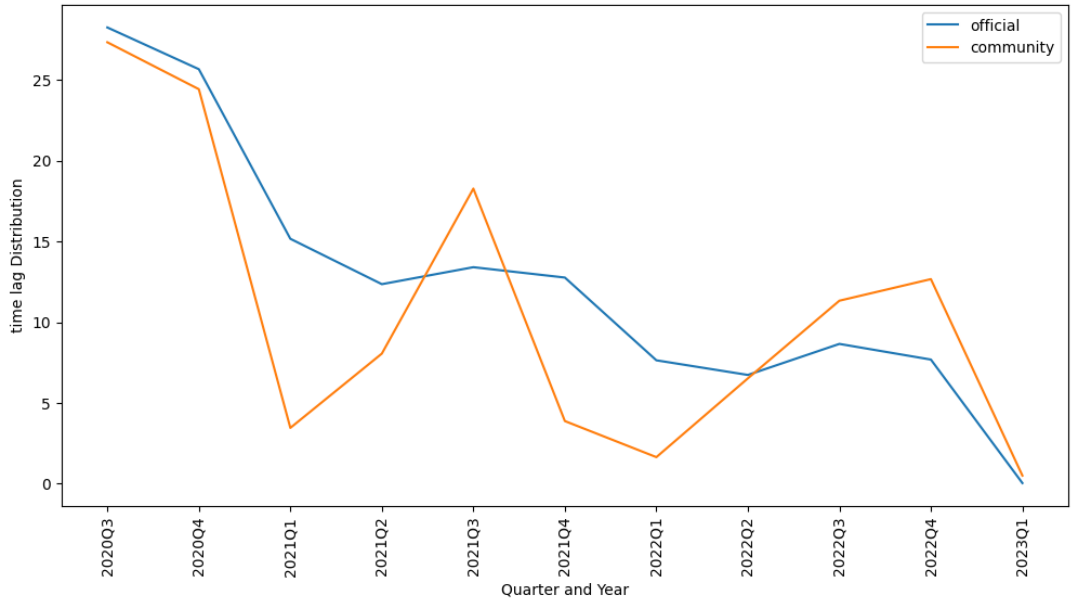

Figure 4.4: Mean Time Lag Distribution by Year and Quarter for Official and Community Images

Based on the statistical analysis, the research indicates that official images receive more frequent updates within shorter periods compared to community images. Emphasizing the significance of using official images in terms of receiving timely updates and ensuring the security and stability of the software becomes crucial.

4.2.2 Technical lag in terms of vulnerability lag

After previously establishing vulnerabilities in Docker images from 4.1, we used a function that computes the difference in the number of security vulnerabilities between image releases to analyze the technical lag in terms of vulnerability lag according to the formula defined in the section 3.2.3.

The plot in Figure 4.5 below illustrates the analysis of our results from comparing the technical lag between official images and community images; we observed more vulnerability lag in community images compared to official images. There is more vulnerability lag observed in community images because when vulnerabilities are introduced in community images, fixing them takes time since there are fewer updates to address them. This means the vulnerabilities accumulate until the next update, and the cycle repeats. Although there is less vulnerability lag in official images, which shows their stability in terms of vulnerabilities, the difference in vulnerability between the latest version and the old versions of official images is not large compared to community images.

In Figure 4.5, it is illustrated that, on average, there are peaks and drops of vulnerabilities lag occurring in two quarters, indicating that new vulnerabilities arise in two quarters and their respective fixes take place within the same duration as well.

The plot in Figure 4.5 below illustrates the analysis of our results from comparing the technical lag between official images and community images,

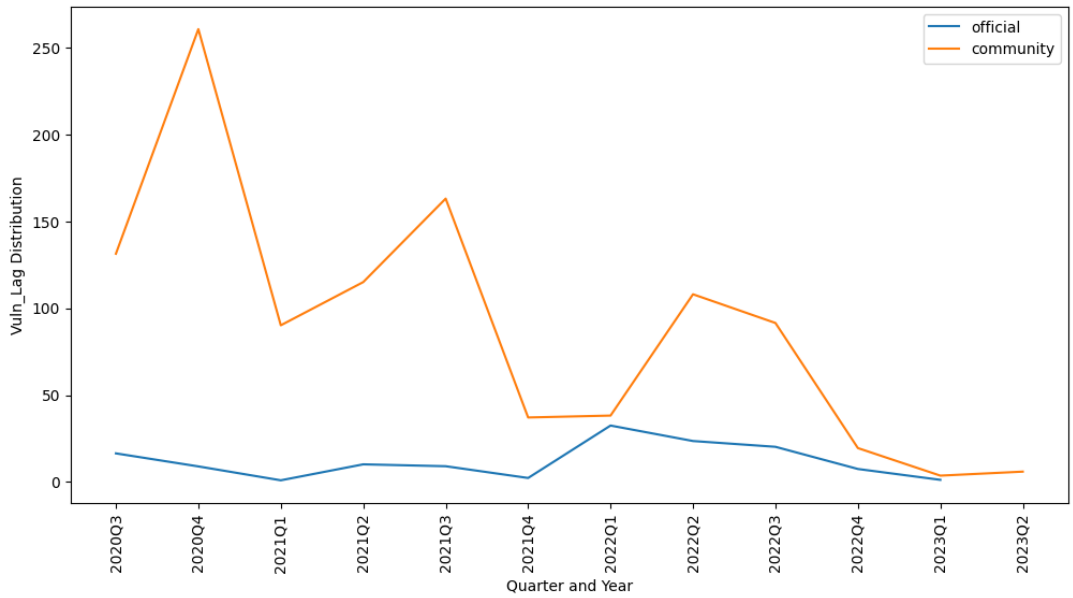

Figure 4.5: Comparison of vulnerability lag in Docker images over time.

Our recommendation is for developers or deployers to prioritize using official Docker images over community-contributed images whenever possible, based on the analysis of vulnerability lag. It has been observed that community images tend to demonstrate a higher vulnerability lag.

4.2.3 Technical lag in terms of version lag

Regarding version lag, the graph in Figure 4.6 below compares the number of missed versions between package releases. The figure reveals that the version lag for official images is greater than that of community images, indicating a higher frequency of updates for official images. These updates can include new features, bug fixes, or vulnerability patches.

During the period between 2020Q3 and 2021Q2, a notable disparity in vulnerability lag is observed between official and community images. This inconsistency is primarily driven by two specific images: "ubuntu" and "neurodebian." For instance, in the case of the "ubuntu" image, the version changed from bionic-33 to bionic-61, signifying a significant change of 28 major versions.

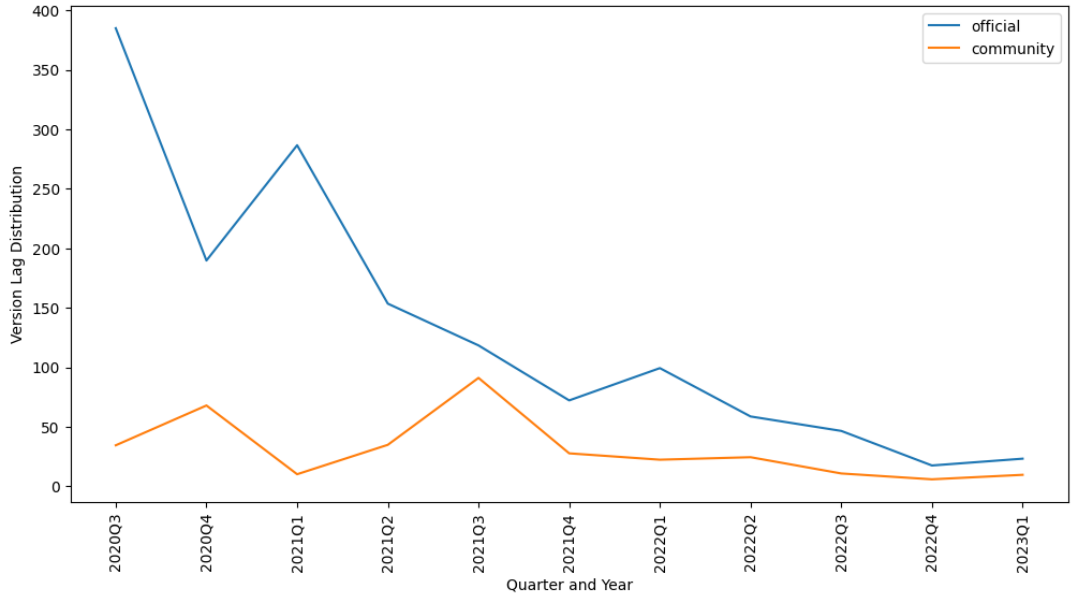

Figure 4.6: Comparison of version lag in Official and Community Docker images over time.

4.2.4 Correlation between technical lag dimensions

We also investigated if there exists a relationship between the different dimensions of technical lag we examined for container images. Using the Python correlation function corr(), we created a correlation matrix between time lag, version lag, and vulnerability lag; this is illustrated in the heatmap shown in Figure 4.7 below; In this particular context, the relationship between time lag and vulnerability lag is significantly stronger compared to other correlations involving different dimensions, but the remaining dimensions are almost independent of each other which means an increase in version lag does not imply a corresponding increase or decrease in vulnerability lag but an increase time lag is related to an increase in vulnerability lag.

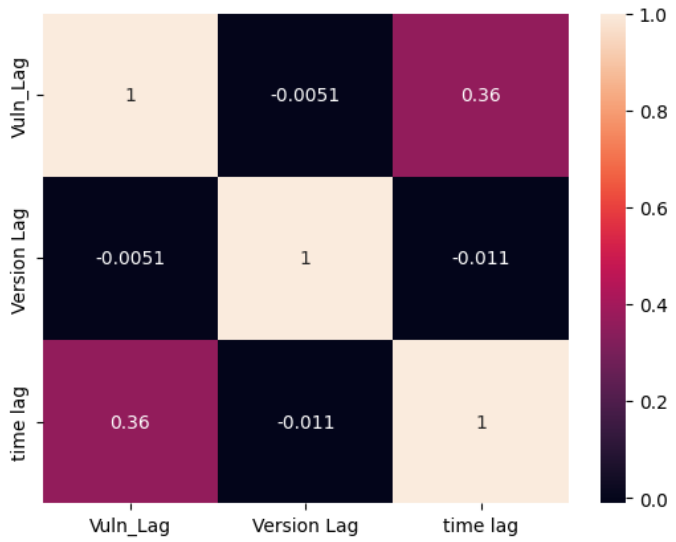

Figure 4.7: Correlation analysis of Technical Lag dimensions: A heatmap perspective

We also tried different importance weights for version lag as discussed in Section 3.2.4.2 and analyzed the correlation between version lag, vulnerability lag, and time lag for each of these importance weights, as shown in Table 4.6 below. We did this to ascertain if the change in importance weight for each version weight would impact the correlation between version lag and the other dimensions of technical lags being examined.

Table 4.6: Correlation between version lag importance weights (major, minor, patch), vulnerability, and time lag.

version lag weights	**[3, 2, 1]**	**[10, 5, 1]**	**[20, 5, 1]**
vulnerability lag	-0.0051	-0.0051	-0.0051
time lag	0.36	0.36	0.36

Based on our analysis, we found a weak correlation between version lag and time lag, as well as between version lag and vulnerability lag. Therefore, we recommend that deployers or developers check these dimensions independently.

Our findings reveal a strong correlation between time lag and vulnerability lag. Specifically, we observed that, on average, vulnerability is significantly reduced after two quarters. Therefore, we recommend that organizations monitor their technical lag at least every two quarters, which equates to a six-month interval. By doing so, they can proactively identify and address any vulnerabilities that may arise and thus enhance the security and reliability of their software systems.

4.3 RQ3

How much technical lag exists in Docker images used in industrial settings in terms of vulnerability, version, and time lag?

To answer this research question, We analyzed the technical lag of Docker images used in an industrial setting in terms of vulnerability, version, and time lag. In our study, we obtained a total of 251 container images from the company's Kubernetes infrastructure. 201 were unique images of these since some containers used the same image.

Out of 201 images, we discovered that most of the company's applications were built using node and go, which rely on the same underlying base images. As a result, we were able to gather 44 base images. Among those 44, we identified 33 of the latest images like "node:alpine" and "nginx:alpine", and then we found 33 images with a version number that corresponded to the latest image like "node:20.0-alpine" and "nginx:1.24.0-alpine". To help illustrate the process of narrowing down from 201 images to 33, please refer to Figure 4.8, which provides an example of reducing 5 images to 2.

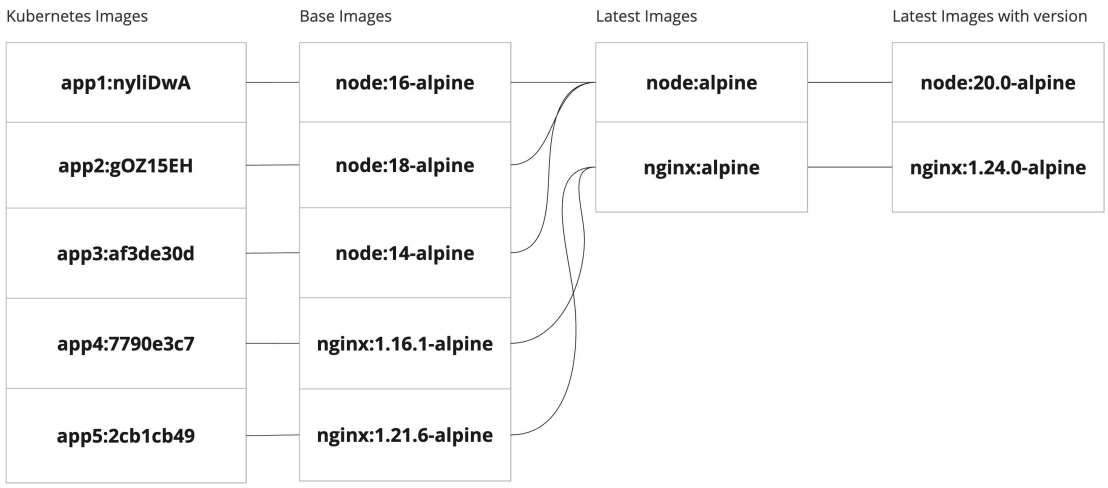

Figure 4.8: A simplified example of the process used to narrow down 201 images to a final selection of 33.

We performed a technical lag analysis to evaluate the security posture of the base images used in the company's applications. Table 4.7 presents one of the notable

results of this analysis, highlighting the vulnerability, version, and time lags of 5 base images. Our analysis revealed that the majority of the company's applications utilized Alpine Linux images as the base image. As a result, we identified only a small number of images that required updates based on the vulnerability lag dimension. However, our analysis identified 3 images with numerous vulnerabilities that urgently needed updating to more secure versions, as shown in Table 4.7.

Table 4.7: List of urgently vulnerable images requiring update

Image	Version	Vuln lag	Version lag	Time lag
nginx	1.21.6-alpine	102	6	14
nginx	1.16.1-alpine	90	17	38
node	19.1	1510	15	20
node	14-alpine	0	32	0
alpine	3.13	4	11	17

Additionally, we discovered that 10 images had not been updated in over a year, indicating a significant version and time lag. Therefore, these images require immediate updating to address any potential vulnerabilities that may have arisen during this period. The company must prioritize the timely update of these images to maintain a strong security posture and minimize the risk of any potential security breaches.

4.3.1 Findings from interviews

This section presents the findings from the interviews conducted for this study, the purpose of the interview was to validate our findings from applying the technical lag framework within an industry setting. The section describes the interviewees' views on technical lags in the industry, their use of the technical lag framework, and their concerns and priorities related to different types of technical lags.

4.3.1.1 Base image updating practices before the technical lag framework

The interviewees mentioned that they do not have a regular plan for updating their base images but instead update them when new features are needed, or significant vulnerabilities are identified. For example, one interviewee noted that they only update the image when a new feature is relevant to their use case. Another two interviewees mentioned that they rely on subscription blogs to inform them about significant vulnerability issues and will update their images accordingly. These findings suggest that the industry relies on reactive rather than proactive approaches to base image updates.

4.3.1.2 Prioritization of dimensions of technical lag

Regarding the prioritization of technical lags, the interviewees generally assigned the highest weight to vulnerability lag, indicating that this type of lag is a critical

consideration in the industry. The second highest weight was assigned to version lag, with some interviewees mentioning that they would prioritize it when new features are introduced. Finally, the time lag was generally less critical than vulnerability and version lag. Therefore, the results suggest that the industry prioritizes technical lags in the following order: vulnerability, version, and time lag.

Table 4.8: Weight and Vote for Prioritization of Technical Lags

Dimension	Vote
vulnerability lag	5
version lag	3
time lag	0

4.3.1.3 Comparing the importance weight of version lag in the technical lag framework

The interviewees assigned the third weight to version lag because it indicates a more significant change, mainly when there is a major version change.

Table 4.9: Weight and Vote for Version Lag Significance

Weight	Vote
[3, 2, 1]	0
[10, 5, 1]	0
[20, 5, 1]	5

4.3.1.4 Feedback from interviewees on technical lag results and suggestions for improvement

This section describes the interviewees' feedback on the technical lag results presented to them. The interviewees were asked for their thoughts on the results, and all expressed that the results were helpful and that they would consider using the framework in the future. They also mentioned that they would check their images and consider updating them based on the technical lag scores.

Furthermore, the interviewees suggested some improvements to the framework. Specifically, they recommended including more verbose information about the base images, such as the mean score for each dimension or the stability of the latest image. Additionally, they suggested incorporating a measure of the difference between the two versions to help identify significant changes. These suggestions could make the technical lag framework even more helpful and informative for industry professionals.

Our study revealed that there is currently no regular practice of checking images for the technical lag in the company. This suggests that deployers or developers may be unaware of the significant lags that exist between the current production version and the latest image. Therefore, we recommend the implementation

of the technical lag framework and running it periodically to monitor the lag between the base image and production.

5

Discussion

Docker's core promise of offering stable and consistent environments through its container images is a valuable feature that ensures the proper functionality of deployed images, even after years of their creation. The containerization model allows for the deployment of specific versions of images, making it easier to strike a balance between staying with a working version and upgrading to a newer one. However, the lack of up-to-date package versions in Docker images can lead to a technical lag, resulting in missing out on the latest functionality, security updates, and bug fixes required in most production environments. Thus, choosing to deploy a new version of a container image is a critical decision that requires a delicate balance between sticking with the old image and switching to a more recent one. Our technical lag framework offers a solution to this challenge faced by deployers, enabling them to navigate this balance easily.

5.1 ARM vs AMD

The investigation of different architectures on container image security revealed that there were no significant differences in vulnerability levels between ARM and AMD architectures, as shown in Section 3.

These findings indicate that developers and deployers can choose either architecture without concern for security risks for Docker images associated with one architecture over the other.

However, as reported in Section 3.1.1, the number of Docker images designed to run on multiple architectures was significantly lower than those designed for only ARM or AMD architectures. This observation has implications for deployers who want to use both ARM and AMD machines in their environments. Therefore, deployers must verify that the images they intend to use are available for both architectures.

5.2 Interest of incorporating multiple dimensions of technical lag

Incorporating multiple dimensions of technical lag is valuable for investigating and mitigating the impact of container image security. By considering time lag, version

lag, and vulnerability lag, we can understand the factors contributing to technical lag and the resulting security risks.

Analyzing each dimension of technical lag separately helps identify areas of weakness and prioritize efforts to mitigate security risks. For example, one container image may have a relatively short time lag but a significant version lag. In such cases, it may be necessary to investigate the compatibility between the current version and the latest version to ensure a smooth update process.

Incorporating multiple dimensions of technical lag is a valuable approach for investigating and mitigating the impact of container image security. By analyzing each dimension separately, we can understand the underlying causes of technical lag and take a more targeted approach to improve container image security.

As suggested by the interviewee in section 4.3.4, incorporating the multiple dimensions approach can be highly beneficial and provide more helpful information. Moving forward, exploring additional dimensions that could be incorporated to enhance the effectiveness of technical lag further may be worthwhile.

5.3 Community images vs official images

From our analysis of the technical lag in community and official images in section 4.2.2, we observed that the vulnerability lag in community images is almost five times that in official images. Also, as observed from the analysis of time lag in container images from Section 4.2.1, we noticed that official images get more updates within shorter periods than community images; this implies that updates address vulnerabilities are provided faster in official images than community images. Therefore, developers will make a wiser choice using official Docker images over community images if having fewer vulnerabilities in container images is a prioritized criterion.

5.4 On the correlation between technical lag dimensions

Our correlation analysis in Section 4.2.4 showed that among the dimensions of technical lag, we examined, i.e., version lag, time lag, and vulnerability lag, There exists a strong correlation between time lag and vulnerability lag with a correlation coefficient of 0.36 [27] which means an increase in time lag can be a corresponding increase in vulnerability lag. However, there is a weak correlation between version lag and other examined dimensions of technical lag (time lag and vulnerability lag), We tried to see if varying the importance weights of the version levels will have an impact by using 3 distinct sets of importance weights, but there was no impact at all. Hence, developers should examine these dimensions of technical lag independently.

5.5 Future work

While this study did not find significant differences between ARM and AMD Docker images based on vulnerabilities, it is important to consider other dimensions when choosing an architecture for Docker images. Performance, availability of pre-built images, and cost are all important factors that can influence the decision. Further investigation into these dimensions can provide valuable insights into which architecture is the most suitable for various use cases and scenarios. Moreover, while our study focused on ARM and AMD architectures, it is important to note that other architectures, such as MIPS or PowerPC, could exhibit different security vulnerabilities that may impact container image security.

This research analyzed the technical lag present in public Docker images by comparing them to an ideal release selected based on the latest image released by the provider. However, it is important to note that recent releases may not always be ideal for certain deployments due to various factors such as untested code, backward incompatibilities, or increased vulnerability to bugs, stability, and performance. As a result, alternative methods for selecting an ideal release should be explored within the technical lag framework. In particular, investigating the suitability of selecting an ideal release based on stability and performance could lead to more effective container management practices.

In addition, although measuring technical lag can provide valuable insights into image freshness, a more holistic approach is required to identify the specific areas that need improvement. This may entail examining the running container and evaluating metrics such as resource utilization, network connectivity, and availability to determine the underlying factors contributing to the technical lag.

Additionally, evaluating the effort required to reduce technical lag is critical, and incorporating a framework or tools that can provide this information would enhance the usefulness of our findings.

5.6 Threats to validity

Validity threats are potential factors that can compromise the validity or accuracy of research findings. They are important considerations for researchers to be aware of when designing, conducting, and interpreting research studies. These threats can arise from various sources, such as the study design and data analysis. By understanding and addressing these validity threats, researchers can ensure that their findings are reliable, trustworthy, and informative. In this section, we will explore some of the validity threats against our research.

5.6.1 Internal validity

Internal validity threats can compromise the accuracy of research findings by influencing the study results. One such threat is the potential for selection bias, which

may occur if our research filters for only the latest 10 versions of each container image based on certain criteria, such as supporting ARM and AMD architectures. These selection criteria may not accurately represent the entire population of container images on Docker hub, leading to biased findings.

The Docker hub's API that was used in this research only returns a limited amount of results which may not count as the total number of container images available in Docker hub, this poses a threat to validity in the sense that our dataset may not represent the entirety of container images on images on DockerHub

Another internal validity threat is related to the scanning tools used to analyze the vulnerabilities in container images. Changes to the scanning tools can impact the results of the scans, which may affect the validity of the study findings. Therefore, it is essential to ensure the consistency and reliability of the scanning tools used throughout the study to minimize the risk of compromised validity.

Furthermore, another internal validity threat is using one vulnerability scanning tool instead of multiple; since we were comparing differences in vulnerability between image versions, It made sense to apply only one vulnerability scanning tool, in this case, Clair. However, using multiple scanning tools can assist in the discovery of more vulnerabilities for container images and show more vulnerabilities that Clair may not discover.

5.6.2 Construct validity

Each of our scanning tools has a vulnerability database they use to identify the vulnerabilities in each container image they scan. The main threat to the accuracy of our research arises from potential inaccuracies in the data sources we used to identify vulnerabilities. Our analyses rely on the vulnerability database used by our vulnerability scanning tools as sources of complete and reliable information. However, these sources only capture known, reported, and disclosed vulnerabilities. Therefore, the actual situation of container images in terms of vulnerabilities may be underestimated in our analysis.

Also, another threat to construct validity is that not all container images are supported by the vulnerability scanning tools, this is because vulnerability scanners rely on a set of known package managers and operating systems to perform their vulnerability scans, If an image is built with an uncommon package manager or operating system that isn't supported by the vulnerability scanner, then it may not be able to perform a complete scan of the image which means the scan of these container images return an error message or it reports that the container image has no vulnerability even though vulnerabilities may exist.

5.6.3 External validity

External validity threats refer to factors outside of the research study that may limit the generalizability or applicability of the research study findings outside the scope

of the research. In the case of this research study, external validity threats may arise due to the limited scope of the study sample.

The study investigated vulnerabilities in ARM and AMD architectures; the findings may not be directly generalizable to other architectures commonly used in Docker images, such as 386, ppc64le, and s390x. Therefore, caution should be exercised when applying the results to architectures beyond the scope of our study.

Our analysis considers Docker images within a specific timeframe. The extent of technical lag and vulnerabilities may change over time due to updates, patches, or new vulnerabilities being discovered. Consequently, the findings may not remain valid for future Docker images or updates beyond the period covered in our study.

Also, this study's findings can be applied to other organizations utilizing similar container infrastructures. The company's specific request to develop a framework to bolster container security underscores their need for such measures, indicating that the results hold external validity within their organizational context. However, it is important to note that the interview sample size was limited to five participants. Expanding the sample to include a larger and more diverse group from various organizations would be beneficial. This broader range of perspectives would yield a more comprehensive understanding of the framework's effectiveness and bolster its generalizability to a wider array of settings.

6

Conclusion

In conclusion, this study investigated and analyzed the impact of technical lag and different architectures on container image security. Through a deep comparison and analysis of ARM and AMD architectures for a dataset of 2500 container images using vulnerability scanners, we measured and compared three dimensions of technical lag: time lag, version lag, and vulnerability lag. Our results indicate no significant difference in the context of vulnerabilities among different architectures, and official images consistently have a lower vulnerability lag than community images.

Based on our findings, we recommend regular monitoring and updating container images, focusing on official images, to minimize vulnerability lag. We also stress the importance of proactive measures to manage container image security in a production environment. In future work, it would be valuable to investigate other dimensions such as performance, availability of pre-built images, and cost when selecting a container image architecture. Furthermore, exploring alternative methods for selecting an ideal release within the technical lag framework, such as stability and performance, could lead to more effective container management practices. Additionally, a more holistic approach to identifying areas for improvement in container image security, such as examining running containers and evaluating metrics like resource utilization and network connectivity, could be beneficial. Lastly, evaluating the effort required to reduce technical lag and incorporating a framework or tools to provide this information would enhance the usefulness of the findings.

In conclusion, this study provides insights and recommendations for mitigating the impact of technical lag and different architectures on container image security, helping organizations make informed decisions in managing their container images.

Bibliography

[1] O. Bentaleb, A. S. Z. Belloum, A. Sebaa, and A. El-Maouhab, "Containerization technologies: taxonomies, applications and challenges," *The Journal of Supercomputing*, vol. 78, pp. 1144–1181, 2022. [Online]. Available: https://doi.org/10.1007/s11227-021-03914-1

[2] 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, 03 2021.

[3] M. S. Haq, A. Tosun, and T. Korkmaz, "Security analysis of docker containers for arm architecture," 12 2022, pp. 224–236.

[4] "Docker overview," Mar 2023. [Online]. Available: https://docs.docker.com/get-started/overview/

[5] "About storage drivers," Mar 2023. [Online]. Available: https://docs.docker.com/storage/storagedriver/

[6] "Manage repositories," Mar 2023. [Online]. Available: https://docs.docker.com/docker-hub/repos/

[7] May 2023. [Online]. Available: https://docs.docker.com/docker-hub/official_images/

[8] "Multi-platform images," Mar 2023. [Online]. Available: https://docs.docker.com/registry/spec/api/#content-digests

[9] "Multi-platform images," Mar 2023. [Online]. Available: https://docs.docker.com/build/building/multi-platform/

[10] "Everything you need to know about container scanning," Nov 2021. [Online]. Available: https://snyk.io/learn/container-security/container-scanning/

[11] Imperva, "Cve, cvss, vulnerability," 2021. [Online]. Available: https://www.imperva.com/learn/application-security/cve-cvss-vulnerability/

[12] T. M. Corporation. (2018, September) A look at the cve and cvss relationship. MITRE Corporation. [Online]. Available: https://cve.mitre.org/blog/September112018_A_Look_at_the_CVE_and_CVSS_Relationship.html

[13] N. I. of Standards and T. (NIST). (2021, March) Common vulnerability scoring system (cvss). National Institute of Standards and Technology. [Online]. Available: https://nvd.nist.gov/vuln-metrics/cvss

[14] W. Hu, Y. Wang, X. Liu, J. Sun, Q. Gao, and Y. Huang, "Open source software vulnerability propagation analysis algorithm based on knowledge graph," in *2019 IEEE International Conference on Smart Cloud (SmartCloud)*, 2019, pp. 121–127.

[15] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 383–387.

[16] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, "Docker ecosystem – vulnerability analysis," *Computer Communications*, vol. 122, pp. 30–43, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366417300956

[17] K. Wist, M. Helsem, and D. Gligoroski, "Vulnerability analysis of 2500 docker hub images," 2020. [Online]. Available: https://arxiv.org/abs/2006.02932

[18] B. Kaur, M. Dugré, A. Hanna, and T. Glatard, "An analysis of security vulnerabilities in container images for scientific data analysis," *GigaScience*, vol. 10, 06 2021.

[19] S. Berkovich, "What's in your pipeline? ups and downs of container image scanners," 10 2020.

[20] S. Berkovich, J. Kam, and G. Wurster, "UBCIS: Ultimate benchmark for container image scanning," in *13th USENIX Workshop on Cyber Security Experimentation and Test (CSET 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/cset20/presentation/berkovich

[21] O. Javed and S. Toor, "An evaluation of container security vulnerability detection tools," in *Proceedings of the 2021 5th International Conference on Cloud and Big Data Computing*, ser. ICCBDC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 95–101. [Online]. Available: https://doi.org/10.1145/3481646.3481661

[22] A. Zerouali, T. Mens, J. Gonzalez-Barahona, A. Decan, E. Constantinou, and G. Robles, "A formal framework for measuring technical lag in component repositories - and its application to npm," *Journal of Software: Evolution and Process*, vol. 31, 02 2019.

[23] A. Zerouali, T. Mens, G. Robles, and J. Gonzalez-Barahona, "On the relation between outdated docker containers, severity vulnerabilities and bugs," 2018. [Online]. Available: https://arxiv.org/abs/1811.12874

[24] "Docker hub is the world's largest library," May 2023. [Online]. Available: https://docs.docker.com

[25] Mar 2023. [Online]. Available: https://www.techbeatly.com/vulnerability-scanning-with-clair-and-trivy/

[26] CoreOS, "What is clair?" https://quay.github.io/clair/whatis.html, accessed 2023-03-13.

[27] J. Frost, "Interpreting correlation coefficients," Jul 2022. [Online]. Available: https://statisticsbyjim.com/basics/correlations/

A

Appendix 1

Interview guide

The following questions were asked during the interviews with the participants:

1. Have you experienced any security issues related to technical lag in Docker images? If yes, can you describe the issue?
2. How does the company ensure that Docker images are up-to-date and secure?
3. How do you manage project dependencies and ensure that they are up-to-date and secure?
4. What do you think about the concept of technical lag in software dependencies?
5. How do you think technical lag can be measured effectively?
6. What metrics or factors do you think should be considered when measuring technical lag?
7. Do you have any additional comments?