Vessel: A Taxonomy of Reproducibility Issues for Container Images

Kevin Pitstick, Alex Derr, Lihan Zhan, Sebastián Echeverría
Carnegie Mellon Software Engineering Institute
Pittsburgh, PA USA
{kapitstick, aderr, lzhan, secheverria}@sei.cmu.edu

***Abstract*—Build reproducibility of container images is essential to ensure that deployed systems will work as expected and have not been tampered with. However, bit-by-bit reproducibility of container images is almost never achievable due to external factors, and it is also very slow and labor intensive to determine the causes and severity of reproducibility failures. In this paper, we present a taxonomy of reproducibility issues for container images, as well as a tool, Vessel Diff, to help automatically categorize the type and severity of reproducibility failures in container images. We analyzed a set of open source repositories where container images are built to find common patterns and configure our tool to properly categorize failures. Our analysis shows that approximately 87% of their reproducibility failures were automatically classified by the tool according to our taxonomy. However, the vast majority of these failures were caused by trivial issues and not non-trivial issues, which could cause noticeable changes in execution of container applications and are more difficult to detect. These results highlight the need for additional research and tooling to detect, classify, and fix reproducibility issues, especially those that can lead to major failures.**

***Index Terms*—containers, builds, Docker, reproducibility**

I. Introduction

A critical component of trusting the software supply chain is securing the build process from malicious tampering. The SolarWinds [1] and Codecov [2] hacks, where malicious code was inserted into a build environment, highlight the consequences of not having measures that can be used to detect unauthorized modifications. Build reproducibility is an essential method that provides strong guarantees that built capabilities have not been tampered with. Security frameworks such as Supply-Chain Levels for Software Artifacts (SLSA) [3] were developed in response to these attacks by industry members including Google, Red Hat, Intel, and Microsoft, and require reproducible builds to achieve their highest levels of assurance (Level 4). However, despite its recognized importance, a recent study revealed that build reproducibility is the least adopted practice by organizations due to not only organizational but also technical challenges [4].

A build is reproducible when, given the same build environment, any party can recreate bit-by-bit copies of all specified artifacts [5]. However, in practice builds are often not reproducible [6], [7], due to elements of the build environment that rely on external nondeterministic factors such as timestamps, filesystem file ordering, and unique ID generation.

Container images are increasingly being used as the main method for software deployment [8], so ensuring the reproducibility of container images, among others, is becoming a critical step in protecting the software supply chain as a key aspect of overall build reproducibility. Lack of trust, changing build behavior, and broken builds [6] can all result from build unreproducibility, and significantly hinder efforts to build and deploy services securely and reliably. Cito et al. [9] performed a study on container build files (Dockerfiles) on GitHub and reported that 34% of the containers in the study did not build and that most container quality issues come from missing version pinning (28.6%). Adding reproducibility to build source code (e.g., Dockerfiles) is a manual and time-intensive process of identifying and fixing reproducibility gaps, and the state-of-the-art lacks sufficient tooling for automation. Furthermore, there is no taxonomy that identifies the most common types of reproducibility issues for container images, nor their criticality.

Our contributions to address this container reproducibility problem include (1) a taxonomy of nine common reproducibility issues based on an empirical analysis of open source container build files, (2) an open source tool to automate the categorization of reproducibility failures in container images according to the taxonomy, and (3) empirical results and insights from applying our tool to 199 open source projects.

The paper is structured as follows. Section II presents related work. Section III introduces the taxonomy to categorize reproducibility issues. Section IV presents the reproducibility issue identification tool that was built based on the taxonomy. Section V describes the repositories studied and their analysis. Section VI discusses the results and observed limitations. Finally, Section VII concludes the paper and describes planned extensions to this work.

II. Related Work

As noted by Butler [6] and Wheeler [7], most builds in the software industry are not designed to be reproducible. Butler cites the high level of effort required, unclear direct benefits to the customer, lack of awareness, and technical difficulty as significant barriers to widespread adoption. In open source software, the Reproducible Builds project, a non-profit initiative to promote reproducible builds, maintains a list of projects that are working toward reproducible builds [10]. One notable example is Debian [11] which has achieved reproducibility in over 30,000 packages (95%) [12]. In industry, awareness of reproducible builds is low, with safety-critical and security domains seeing the highest adoption [6].

A key to achieving build reproducibility is to identify issues that exist in the software build process that cause its lack of reproducibility. Some tools exist that aim to lint or identify errors in Dockerfiles. Hadolint [13] is a linter that performs static analysis on a Dockerfile using a list of rules to identify violations of best practices [14]. Dockle [15] and Docker Bench for Security [16] are similar tools that perform linting for best practices and operate on the final container image rather than the Dockerfile. Binnacle [17] and DRIVE [18] are linters whose rules were extracted from a set of well-formed, existing Dockerfiles. Some tools suggest improvements and attempt to fix Dockerfiles directly; Shipwright [19] is an example that looks for internal problems and attempts to automatically provide fixes for those issues. FLAKIDOCK [20] uses large language models to fix Dockerfiles that are failing to build, but it does not focus directly on reproducibility. Although these tools can identify and fix some reproducibility issues, such as the lack of version pinning, they are not designed with the goal of reproducibility and do not identify other issues, such as volatile input sources or timestamps.

The Reproducible Builds [5] project consolidates software development practices and tools to help developers make their builds reproducible. Notable tools found in this portal include diffoscope, disorderfs, reprotest, and diffoci. Diffoscope [21] compares files or directories to present differences in a human-readable fashion. Disorderfs [22] is a filesystem that intentionally injects non-determinism to help find reproducibility issues with unstable inputs. Nondeterministic stripping [23] is a tool that removes nondeterministic information from files, such as timestamps, as a post-build step to make builds more reproducible. Reprotest [24] builds the same source code in different environments to identify reproducibility failures in the output binaries or packages. Diffoci [25] compares container images to provide a detailed comparison between manifests, configuration files, and metadata, and to identify files with differing checksums. These tools assist developers with creating reproducible builds of binaries and archives; however, with the exception of diffoci, they do not focus on containers and, therefore, do not address issues such as container-specific files and metadata or package manager version pinning. Diffoci helps bridge this gap but does not analyze or consider deltas between differing files.

The largest source of guidance that lists issues that prevent reproducibility is also the Reproducible Builds project [5], which maintains documentation on ways to help achieve deterministic builds. An associated paper by Lamb [12] discusses these issues as well, including build timestamps, build paths, filesystem ordering, archive metadata, randomness, and uninitialized memory. While many of these issues apply to container image builds, containers present some unique challenges because they encapsulate an entire runtime environment with software applications and all of their dependencies. The work presented in this paper aims to address this gap by sharing a taxonomy of reproducibility issues specific to container builds, and tooling to detect and help address these issues.

III. Reproducibility Issue Taxonomy

First, we define key concepts related to reproducible builds. We define a *reproducibility failure* as a difference between two OCI (Open Container Initiative) [26] images that are built using the same build process. We define a *reproducibility issue* as a characteristic of the container build process that is capable of triggering a reproducibility failure. Reproducibility failures occur when a reproducibility issue in the build process aligns with a change in external factors (e.g., external dependency change, current time change) that occurs between two builds.

Reproducibility issues may differ in their severity, and thus some organizations may not need to prioritize fixing inconsequential issues. For example, timestamp changes are nearly unavoidable without a large amount of manipulation of the build process and do not alter the behavior of the container. We separate issues into three categories based on the effect of the failures they can cause: *trivial* when failures resulting from the issue do not alter the functioning of a container, *non-trivial* when failures could cause noticeable changes in execution of container applications, and *unknown* when failures are detected but cannot be classified. An example of a non-trivial issue is: A Dockerfile installs the latest version of a system package (reproducibility issue). Between the first and second builds of the image, a new version of the package is released (external factor). This causes the two container images to have different packages installed with different files (reproducibility failure).

To start building a taxonomy of reproducibility issues, we performed a survey of existing tools and literature, using a keyword-based search plus snowballing approach, to identify reproducibility issues that impact container builds.

The following 8 tools were identified:

- Dockerfile error detection and repair tools: hadolint [13], binnacle [17], shipwright [19]
- Tools from the Reproducible Builds website [5], including disorderfs, diffoscope, reprotest, and diffoci

We also identified 11 papers in the following categories:

- Dockerfile specification [27]
- OCI image specification [26]
- Dockerfile analysis papers: [9], [28], [17], [29], [30], [31]
- Infrastructure-as-code (IaC) analysis papers: [32], [33]
- Docker container vulnerability analysis papers: [34]

As a result of the analysis of these tools and the literature, we defined a taxonomy of reproducibility issues to aid in identifying and understanding reproducibility failures in the container build process, shown in Table I. To the best of our knowledge, this is the first build reproducibility taxonomy targeting container image builds.

Unverified external dependency (EXT) [5], [19] reproducibility issues occur when a Dockerfile or build script pulls files from remote sources (e.g., the Internet) without checking that the file's checksum matches a known hash.

Lack of version pinning (VER) [5], [9], [19], [13], [34] reproducibility issues occur when a Dockerfile depends on a base container image or installs an unpinned package version.

TABLE I
REPRODUCIBILITY ISSUE CLASSES

Type	Abbreviation	Severity
Unverified external dependency	EXT	Non-trivial
Lack of version pinning	VER	Non-trivial
Use of current timestamp	TIME	Trivial
Installation of recommended dependency	REC	Non-trivial
Use of platform variable	PLAT	Non-trivial
Not clearing cache	CACHE	Situational*
Use of randomly generated data	RAND	Situational
Use of environment variable or build argument	ENV	Non-trivial
Reliance on filesystem ordering	FILE	Non-trivial

* Situational means that severity depends on the specific issue

Use of current timestamp (TIME) [5], [26], [27] reproducibility issues occur when a Dockerfile or a build script run by a Dockerfile includes a timestamp in its output.

Installation of recommended dependency (REC) [13], [17] reproducibility issues occur when a Dockerfile installs a package without specifying the package manager to not install weak or recommended dependencies.

Use of platform variable (PLAT) [5], [13], [19], [26] reproducibility issues occur when a Dockerfile does not specify a platform or uses platform information in its output.

Not clearing cache (CACHE) [13], [17] reproducibility issues occur when a Dockerfile does not clear the caches of programs (e.g., package managers) within the same layer where the cache was created and used.

Use of randomly generated data (RAND) [5] reproducibility issues occur when a Dockerfile or build script run by a Dockerfile uses of randomly generated data in its output.

Use of environment variable or build argument (ENV) [5], [27] reproducibility issues occurs when a Dockerfile makes use of build arguments in its output.

Reliance on filesystem ordering (FILE) [5] issues occur when a Dockerfile or build script run by a Dockerfile relies on the ordering of the filesystem to have consistent output.

These classes of reproducibility issues were identified using listed sources and refined through an analysis of GitHub repositories, as discussed in detail in Section V.

IV. VESSEL DIFF TOOL

Existing tools for analyzing container reproducibility do not provide a holistic view of build differences. To help in the analysis of reproducibility between images, we created the Vessel Diff tool [35] that automates the linking of reproducibility failures to reproducibility issues according to our taxonomy. The tool compares two container images (two builds of the same Dockerfile) at the file system level and categorizes differences between these to provide the user with a more manageable set of differences to investigate.

The tool takes two container transport paths to images as input, extracts the files in the image layers into a unified file system view, and then compares the two file systems. The first step is to pull the images with skopeo [36] and unpack the images with umoci [37]. After the images have been unpacked, they are compared using two methods: SHA256 checksums and the diffoscope tool. The checksum comparison identifies the file pairs that are different, and diffoscope generates human-readable unified diffs using specialized file comparators. The tool then analyzes the checksum comparisons and diffoscope output, using *flags* to categorize the differences (i.e., reproducibility failures). Each flag is a group of regular expressions used to match differences that occur due to a known reproducibility issue. There are regexes for file path, file type, command diffoscope used to find the diff, comment from diffoscope, and for the unified diff. These categorized failures are linked back to the files to explain the cause of their mismatched checksums. Checksum mismatches between files without associated diffoscope output are treated as unknown failures. The final result is a summary file that contains a list of different files associated with their unknown and classified reproducibility failures. It also contains a list of files that were only found in one of the images. If multiple failures are found in a file, they are reported as separate failures.

V. ANALYSIS PROCESS

To apply our taxonomy and create flags for the Diff tool, we analyzed reproducibility failures occurring in containers built from open-source repositories on GitHub. We selected 199 public repositories with Dockerfiles in their root directories and filtered them by activity metrics (>= 30 stars, >= 3 commits, updated in 2024 or 2025). We successfully built container images for 61 of them (using one Dockerfile from each) and compared two builds using the Diff tool.

To evaluate the representativeness of the repositories that were evaluated with the tool, the programming language distribution of files included in all the repositories that met the search criteria was compared with the distribution of files in the repositories that were built and compared successfully. As shown in Table II, the distributions are similar, ensuring that the subset of repositories is representative.

TABLE II
LANGUAGE DISTRIBUTION OF FILES IN OPEN-SOURCE DATASET

Language Distribution of Files	Overall Repositories	Compared Repositories
Shell	24.9%	34.4%
Dockerfile	34.8%	21.9%
Python	13.7%	10.9%
Go	5.8%	10.9%
JavaScript	5.6%	9.4%
TypeScript	2.1%	1.6%
Makefile	1.5%	1.6%
HTML	1.7%	1.6%
Ruby	0.6%	1.6%

Once the Diff tool was run on each pair of images, the results were manually analyzed to identify reproducibility failures that could be associated with reproducibility issues from our taxonomy. Flags were created for failures that could be recognized as being caused by a reproducibility issue through a pattern, so that the Diff tool could automatically categorize reproducibility failures in future runs. We repeated the comparisons to evaluate how effectively the Diff tool could categorize trivial and non-trivial issues, detect checksum differences, and identify unknown failures. During this analysis, we found that several of the classes of issues would be quite difficult to identify in an automated way when analyzing the resulting images. These classes include EXT, VER, REC, ENV, and FILE. This is because the failures caused by these issues are not consistent and can be widespread in unpredictable ways. We found that these classes of issues can be detected more easily in Dockerfiles before being built. A tool that searches Dockerfiles for these issues before the image is built would be much more effective than trying to detect the failures after being built. A tool like this would also be able to fix issues that fall under classes that the Diff tool is able to detect, such as CACHE and PLAT.

VI. Analysis Results

After running the Diff tool on images built from each repository in the dataset, we analyzed the number of failures that were categorized according to our taxonomy. We found that of the 415,526 reproducibility failures detected, 87% (359,517) were classified as being caused by known reproducibility issues in our taxonomy. We then reviewed the issue class distribution of failures and, as shown in Table III, the vast majority of classified failures (87.5%) fell under the TIME class, while the rest fell under the CACHE, RAND, or PLAT classes. A majority of these failures (81%) were TIME failures traced back to file modified time metadata, rather than changes in the file contents themselves.

TABLE III
Distribution of Failures by Reproducibility Issue Class

Issue Class	Total Percentage
TIME	87.5%
CACHE	11.9%
RAND	0.3%
PLAT	0.3%

To better understand failures that were not flagged, we manually reviewed files associated with the 13% (55,009) of failures that were unknown. We traced 81% (44,658) of them back to one outlier repository whose images each had 22,329 identical cache files buried in non-matching, randomly-generated folder names. Of the remaining 19% (10,351), we found that many of these failures were caused by external dependencies (EXT) and lack of version pinning (VER), which led to a large number of binary differences. Although these failures were not identified by the Diff tool, we determined that they do fit within our taxonomy and could be addressed more easily by linting and repairing the Dockerfile before being built, as mentioned earlier.

Because many of the identified failures were caused by timestamp metadata and not related to the contents of the files, the next stage of the analysis focused on investigating which reproducibility issues contributed to checksum mismatches in the image file system. We found that 91% of non-metadata, trivial failures were associated with byte-compiled Python files (.pyc). In contrast, non-metadata, non-trivial failures were mainly associated with ASCII text (41%), unknown data (48%), ELF executables (4%), and secret keys (3%).

Through this analysis, we find that the vast majority of reproducibility failures are caused by timestamps, especially modified file times, which can be easily detected and safely ignored using automated tooling. For files with actual checksum differences (not just metadata), cache files make up the large majority of them, which can either be ignored or used as feedback to clean up the container build process. In projects that have files with unknown reproducibility failures, we find that these can be largely attributed to changes in external dependencies and can be used as feedback to pin versions and verify checksums of dependencies.

The scripts used to perform this analysis, along with the full results, can be obtained from the following link: https://zenodo.org/records/15870502.

VII. Conclusions and Future Work

In this paper, we describe a taxonomy for reproducibility issues present in Dockerfiles, as well as a tool that can be used to easily automate the categorization of failures using this taxonomy. To configure and illustrate the use of the tool and the usefulness of the taxonomy, we applied the tool to a set of 61 open-source projects. We used these results to properly configure the tool and reviewed the categorization results it provided. We found that the Diff tool categorized 87% of the differences (reproducibility failures) in our dataset, showing that our taxonomy properly covers the majority of failure instances found in representative projects from the real world. Using the Vessel Diff tool will save time and effort for maintainers to focus on non-trivial reproducibility issues that need to be addressed.

Despite our work showing the ability to detect and classify a large number of trivial reproducibility failures between two images, we found that additional work is required to detect, classify, and fix non-trivial failures, especially related to unverified external dependency (EXT) and lack of version pinning (VER), as they account for significant differences when they occur. As part of our current and future work, we are extending the tool with Dockerfile linting functionality for finding reproducibility issues before a container image is built as well as functionality to perform automated repairs on Dockerfiles based on identified issues. A reproducibility tool that combines these diff, lint, and repair capabilities would provide organizations with the capability to build reproducible container images with greater software assurance and avoid build process exploits.

Acknowledgment

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. (DM25-0748)

References

[1] CrowdStrike Intelligence Team, "SUNSPOT: An Implant in the Build Process," 2021. [Online]. Available: https://www.crowdstrike.com/blog/sunspot-malware-technical-analysis/

[2] Sentry, "Post-Mortem / Root Cause Analysis," 2021. [Online]. Available: https://about.codecov.io/apr-2021-post-mortem/

[3] The Linux Foundation, "SLSA - Supply-Chain Levels for Software Artifacts." [Online]. Available: https://slsa.dev/

[4] L. Williams, S. Hamer, and Z. Nusrat, "Can the Rising Tide of Software Supply Chain Attacks Raise All Software Engineering Boats?" in *Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE) - Companion*, 2025.

[5] "Reproducible builds." [Online]. Available: https://reproducible-builds.org/

[6] S. Butler, J. Gamalielsson, B. Lundell, C. Brax, A. Mattsson, T. Gustavsson, J. Feist, B. Kvarnström, and E. Lönroth, "On Business Adoption and Use of Reproducible Builds for Open and Closed Source Software," *Software Quality Journal*, vol. 31, no. 3, pp. 687–719, 2023.

[7] T. L. Foundation, "Preventing Supply Chain Attacks like SolarWinds," Sep 2022. [Online]. Available: https://www.linuxfoundation.org/blog/blog/preventing-supply-chain-attacks-like-solarwinds

[8] W. M. C. J. T. Kithulwatta, W. U. Wickramaarachchi, K. P. N. Jayasena, B. T. G. S. Kumara, and R. M. K. T. Rathnayaka, "Adoption of Docker Containers as an Infrastructure for Deploying Software Applications: A Review," in *Advances on Smart and Soft Computing*, F. Saeed, T. Al-Hadhrami, E. Mohammed, and M. Al-Sarem, Eds. Singapore: Springer Singapore, 2022, pp. 247–259.

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

[10] Reproducible-builds.org, "Who is Involved?" [Online]. Available: https://reproducible-builds.org/who/projects/

[11] Debian, "ReproducibleBuilds - Debian Wiki." [Online]. Available: https://wiki.debian.org/ReproducibleBuilds

[12] C. Lamb and S. Zacchiroli, "Reproducible Builds: Increasing the Integrity of Software Supply Chains," *IEEE Software*, vol. 39, no. 2, pp. 62–70, 2021.

[13] Hadolint, "Hadolint: Dockerfile Linter, Validate Inline bash, Written in Haskell." [Online]. Available: https://github.com/hadolint/hadolint

[14] Y. Wu, Y. Zhang, T. Wang, and H. Wang, "Dockerfile Changes in Practice: A Large-Scale Empirical Study of 4,110 Projects on GitHub," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2020, pp. 247–256.

[15] GoodWithTech, "dockle." [Online]. Available: https://github.com/goodwithtech/dockle

[16] Docker, "Docker Bench for Security." [Online]. Available: https://github.com/docker/docker-bench-security

[17] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps, "Learning From, Understanding, and Supporting DevOps Artifacts for Docker," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 38–49.

[18] Y. Zhou, W. Zhan, Z. Li, T. Han, T. Chen, and H. Gall, "Drive: Dockerfile Rule Mining and Violation Detection," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–23, 2023.

[19] J. Henkel, D. Silva, L. Teixeira, M. d'Amorim, and T. Reps, "Shipwright: A Human-in-the-Loop System for Dockerfile Repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1148–1160.

[20] T. Shabani, N. Nashid, P. Alian, and A. Mesbah, "Dockerfile flakiness: Characterization and repair," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 1793–1805.

[21] diffoscope, "diffoscope." [Online]. Available: https://diffoscope.org/

[22] "Reproducible Builds / disorderfs." [Online]. Available: https://salsa.debian.org/reproducible-builds/disorderfs

[23] "Reproducible Builds / strip-nondeterminism." [Online]. Available: https://salsa.debian.org/reproducible-builds/strip-nondeterminism

[24] "Reproducible Builds / reprotest ." [Online]. Available: https://salsa.debian.org/reproducible-builds/reprotest

[25] "diffoci." [Online]. Available: https://github.com/reproducible-containers/diffoci

[26] The Linux Foundation, "Open Container Initiative." [Online]. Available: https://opencontainers.org/

[27] Docker, "Dockerfile Reference." [Online]. Available: https://docs.docker.com/reference/dockerfile/

[28] Y. Zhang, G. Yin, T. Wang, Y. Yu, and H. Wang, "An Insight Into the Impact of Dockerfile Evolutionary Trajectories on Quality and Latency," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 01, 2018, pp. 138–143.

[29] K. Yin, W. Chen, J. Zhou, G. Wu, and J. Wei, "STAR: A Specialized Tagging Approach for Docker Repositories," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, 2018, pp. 426–435.

[30] F. Hassan, R. Rodriguez, and X. Wang, "RUDSEA: Recommending Updates of Dockerfiles via Software Environment Analysis," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 796–801.

[31] J. Xu, Y. Wu, Z. Lu, and T. Wang, "Dockerfile TF Smell Detection Based on Dynamic and Static Analysis Methods," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, 2019, pp. 185–190.

[32] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does Your Configuration Code Smell?" in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 189–200.

[33] Y. Jiang and B. Adams, "Co-evolution of Infrastructure and Source Code - An Empirical Study," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 45–55.

[34] 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

[35] Vessel Team, "Vessel." [Online]. Available: https://github.com/cmu-sei/vessel

[36] Skopeo Contributors, "skopeo." [Online]. Available: https://github.com/containers/skopeo

[37] A. S. et al., "umoci - Standalone Tool For Manipulating Container Images," 2016. [Online]. Available: https://umo.ci/