DOCKERCLEANER: Automatic Repair of Security Smells in Dockerfiles

Quang-Cuong Bui
Institute of Software Security
Hamburg University of Technology
Hamburg, Germany
cuong.bui@tuhh.de

Malte Laukötter
Hamburg University of Technology
Hamburg, Germany
jan.laukoetter@tuhh.de

Riccardo Scandariato
Institute of Software Security
Hamburg University of Technology
Hamburg, Germany
riccardo.scandariato@tuhh.de

***Abstract*—Docker is a widely adopted platform that enables developers to create lightweight and isolated containers for deploying applications. These containers can be replicated from a single blueprint specified by a text file known as a Dockefile. The Dockerfile smells might not only hinder the performance of containers but also potentially introduce security risks. State-of-the-art scanning tools, such as Hadolint and KICS, are available to efficiently detect Dockerfile smells. Still, there is a lack of approaches focusing on resolving these issues. Therefore, we present DOCKERCLEANER, an automated repair tool that suggests fixes for eleven Dockerfile security smell types. Our tool employs the repair actions inspired by the best security practices for writing Dockerfiles. The evaluation results show that DOCKERCLEANER can remove the artificially injected security smells from 92.67% of the Dockerfiles and guarantee the buildability for 99.33% of them. Specifically for security smells in real Dockerfiles, DOCKERCLEANER outperforms the state-of-the-art repair tool by a wide margin. Finally, we leveraged the fixes generated by DOCKERCLEANER to propose improvements to twelve official Docker images. Eight pull requests have been accepted and merged by the developers.**

***Index Terms*—Docker, Security Smells, Automatic Repair**

I. INTRODUCTION

Nowadays, due to the need for fast-paced software development, much attention has been directed toward infrastructure services and tools that facilitate rapid development, testing, and deployment of applications by developers. One of the platforms that cater to this need is Docker [1], which offers a lightweight and transportable containerized environment for running applications. According to a recent survey conducted by Stack Overflow with over 70,000 developers in 2022,[1] Docker was ranked first as the most loved and wanted platform and second as the most popular tool used by professional developers as a fundamental tool. Docker images can be considered as prototypes for Docker containers, as each container is an instance of the image. A source code-level file containing instructions, referred to as a Dockerfile, is needed for the specification of the Docker image. Analogous to source code in programming languages, a Dockerfile has the potential to contain code smells. The smells in a Dockerfile not only impede the maintainability of Dockerfile artifacts but can also pose potential security threats.

[1]https://survey.stackoverflow.co/2022/

There exist a large number of academic works and practical tools that have focused on detecting Dockerfile security smells, such as Hadolint [2], KICS [3], Docker Bench for Security [4], and Binncle [5]. These tools showed their efficiency and effectiveness in identifying various smell types in Dockerfiles. Therefore, developers have widely used them to assess the quality of their Dockerfiles in software development and maintenance. Nevertheless, none of them can suggest fixes or remediations for these issues. Only a few techniques and tools [6]–[9] have been proposed to fix specific types of smells or have not focused on security-related smells. To fill the gap, we propose a set of repair actions that automatically generate fixes and remediation suggestions for eleven types of security smells in Dockerfiles. Our repair techniques have been implemented into a tool, namely DOCKERCLEANER. Our tool employs *language-docker* [10], which is extracted from Hadolint, to parse Dockerfiles and perform *pretty-printing* for the suggested patches. The repair actions have been based on the best practices in writing Dockerfiles from CIS Docker Benchmark [11] and OWASP Docker Security Cheat Sheet [12]. These are reliable sources for deriving our repair actions as security experts have well-maintained them. Not all repair actions are straightforward to implement. For example, the repair action *"Use Version Pinning"* requires sophisticated communication with DockerHub API and package repositories of the package managers to retrieve the correct versions for the pinned packages.

To evaluate DOCKERCLEANER, we first prepared a dataset of 910 Dockerfiles with known smells. This dataset was generated from 91 official Docker images with the smell-fixed versions. We also assessed the impacts of our proposed fixes on the maintainability of the Dockerfiles. To this end, we rebuilt the Dockerfiles after the repair and checked whether the builds broke due to the inserted changes or not. In an extended evaluation on a dataset of 4794 real-world large-scale Dockerfiles, we compared the performance of our tool against the baseline repair tool, PARFUM [8]. Finally, we evaluated the practical benefits of DOCKERCLEANER to the community. We selected the suggested repairs from our tool for the original Dockerfiles (that we collected from official images) for preparing the pull requests to the developers to obtain their feedback.

The results show that DOCKERCLEANER is able to repair security smells for 92.67% of the injected Dockerfiles. Moreover, only six Dockerfile builds are broken after the repair, which means the build degradation is 0.67%. In the extended evaluation, our tool significantly outperforms PARFUM in fixing security smells. That is, for the four security smells supported by both of the tools, DOCKERCLEANER can remove them from 82.81% of the Dockerfiles while PARFUM can address them for 46.70% of the Dockefiles. In the practical evaluation of our tool, we have received much positive feedback from the developers for the proposed fixes. As a result, nine out of twelve pull requests have been accepted, and eight of them have been merged into their Dockerfiles.

In summary, this paper makes the following contributions:

- A set of repair actions to fix eleven kinds of security smells in Dockerfiles; and a tool that has them implemented, namely DOCKERCLEANER.
- A dataset of 91 actual Dockerfiles (DS1) containing security smells, where each file has been manually cleaned (hence, each file is available in a clean vs. insecure version); and a ground truth dataset of 910 Dockerfiles with known smells (DS2), where the files are synthesized from the manually curated dataset.
- Contributions to the open-source community by submitting pull requests of fixes suggested by DOCKERCLEANER to the projects of the official Docker images.

We have also made the artifact of our tool implementation and the evaluation available in a publicly-accessible repository [13] to support the Open Science movement. To evaluate the DOCKERCLEANER's efficiency in the repair of Dockerfile security smells, we formulate the following research questions:

- **RQ1.** How effective is DOCKERCLEANER in repairing known security smells?
- **RQ2.** How effective is DOCKERCLEANER, compared to the state-of-the-art, in repairing security smells in real-world, large-scale Dockerfiles?
- **RQ3.** Do developers of the official Docker images acknowledge security smells and accept repairs suggested by DOCKERCLEANER?

II. BACKGROUND

In this paper, we focus on the Dockerfile smell types that impact to the security of the Docker images and containers. The fixed smells should not only improve the Docker build performance, such as reducing image size and build time; they should also make the Docker images and containers more secure.

To this end, we selected the security smell types for our study based on: (1) the prevalence of the smell type in Dockefiles, which were reported in recent studies [5], [14]–[16]; (2) the detection of the smell type is supported by the state-of-the-art scanning tools (e.g., Hadolint, KICS); and (3) the smell type is covered in the best security practices in CIS Docker Benchmark [11] and OWASP Docker Security Cheat Sheet [12] for writing Dockerfiles. The analysis of Dockerfile smell types is included in our replication package [13]. As a result, we come up with eleven security-related smell types that are suitable for our study. We briefly introduce our selected smells and their rationale as follows.

Use `apt-get update` alone. Using `apt-get update` solely in a single Dockerfile instruction causes a cached layer for the build of the Docker image. Therefore, the latest updates for packages that may contain security patches could not be fetched for the later builds.

Use no `--no-install-recommends`. Unnecessary *apt* packages for the Docker image will be added during the installation without this option. This may increase the attack surface of the Docker image.

No Version Pinning (includes five child smell types, specifically for *apt*, *apk*, *pip*, *npm*, and *gem* packages). Installing packages without having their versions pinned causes a similar issue as *"Use* `apt-get update` *alone"*. The version pinning helps force the build of Dockerfile to fetch particular package versions regardless of what is in the previous Docker layer cache [17].

Use `ADD` *(instead of* `wget` *and* `COPY`*). `ADD` instructions can download and unpack remote files from URLs without any checks, which potentially introduces security risks. They should be replaced by equivalent `wget` commands or `COPY` instructions.

Last user is root. Running as a privileged user in the Docker containers may allow privilege escalation attacks. Thus, the root user should be replaced with a non-root user in Dockerfiles.

Have secrets. Secrets stored in the Dockerfiles can cause huge risks, as they are transferred to the Docker images during the build and visible to every image user.

Have no `HEALTHCHECK`. The `HEALTHCHECK` commands should be included for monitoring the health of Docker containers, which is related to the security control of availability.

III. METHODOLOGY

In this section, we describe the approach used to conduct our study. Figure 1 depicts the overview of our evaluation methodology. From the three smelly datasets that we have prepared for the study, we ran our tool to repair the security smells for each of the Dockerfiles. The creation of these datasets is presented in detail in Section V. At a high level, DOCKERCLEANER combines two stages to repair a Dockerfile. First, our tool tries to parse the Dockerfile into an Abstract Syntax Tree (AST) for better manipulation of the content of the Dockerfile. We follow the idea of phased parsing, which was presented by Henkel et al. [5] to create the enriched AST for Dockerfiles. We used *language-docker* [10] for parsing Dockerfile-syntax and *ShellCheck* [18] for parsing embedded shell scripts. Second, the repair actions employed by our tool apply the fixes on the AST and then convert the fixed AST back to the Dockerfile. The detailed repair actions for DOCKERCLEANER are further explained in Section IV. Across the experiments, we use scanning tools as our oracle references to verify if the smells are truly removed after the repair.

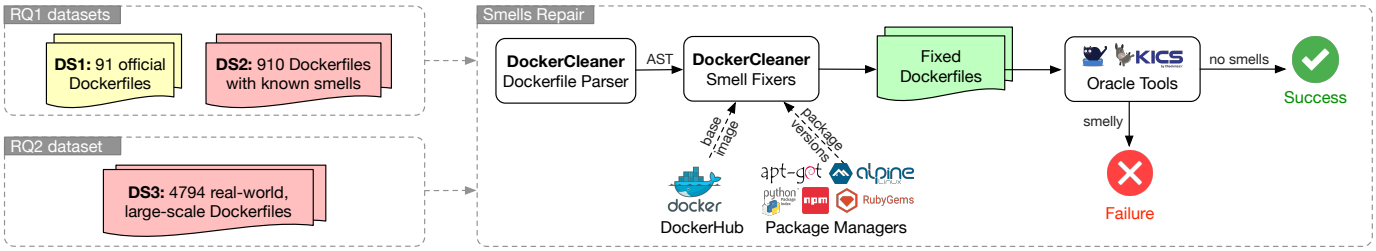

Fig. 1. Overview of our research study.

A. Oracle tools for smell detection

The detection of the aforementioned smells has been supported by plenty of smell scanning tools such as Hadolint and KICS. Hadolint is a well-known linter for Dockerfile, while KICS, which supports various smells and IaC (Infrastructure as Code) platforms, is not yet as established as Hadolint. Therefore, in our study, we use Hadolint to verify the presence of most of the considered smells in Dockerfiles, except for *"Use* `apt-get update` *alone"* and *"Have secrets"*, as they are currently not covered by Hadolint. For these two smell types, we use KICS instead to detect them.

It should be noted that we are acknowledging that the tools we selected as oracles are not perfect. In this study, we discovered two issues related to Hadolint, which may cause false positives[2] for *"No Version Pinning"* and true negatives[3] for *"Last user is root"*. These issues have been reported but have not been resolved by Hadolint developers yet. Also, several issues have been found for KICS, which we then reported to the KICS developers. These problems have been recognized and remediated quickly. For conducting the evaluation, we use the newer versions of KICS that include the improvements addressing our reported issues. The list of issues we reported to KICS developers is available in our replication package.

B. Evaluation metric

As we rely on the results of the oracle tools, we propose a metric named *repair effectiveness* to assess the performance of the repair tool based on these smell detection results. For a smell type S, given that D^S_{before} is the number of Dockerfiles flagged by the oracle tools as containing the smell before the repair, and D^S_{after} is the number of Dockerfiles flagged after the repair, the repair effectiveness of the smell repair tool is calculated as in the Equation 1.

$$RepairEffectiveness(S) = \frac{D^S_{before} - D^S_{after}}{D^S_{before}} \quad (1)$$

IV. Repair and Injection Actions

In this section, we explain in detail our proposed actions for fixing and injecting each of the security smell types considered in our study. We include injection actions as they are used to automatically generate the dataset of Dockerfiles with known smells (DS2). In most cases, we can reverse the repair actions to reintroduce the smells. However, in certain situations, accomplishing this is not a trivial task. We describe these specific actions in the subsection IV-B.

A. Repair actions

Do not use **`apt-get update`** ***alone.*** To fix the smell *"Use* `apt-get update` *alone"*, we first identified the `RUN` instructions with an `apt-get install` that are after a `RUN` instruction containing an `apt-get update`. The `apt-get install` commands might only be able to run under certain conditions. Therefore, we then added the command `apt-get update` right before each `apt-get install` command. Finally, we removed the original `apt-get update` that no longer served any purposes.

Use **`--no-install-recommends`.** The smell *"Use no* `--no-install-recommends`*"* can be fixed by adding a missing `--no-install-recommends` for every `apt-get install` command. This was done by changing the `noInstallRecommends` field of every `AptGetInstall` AST to `False`. A known issue was acknowledged for this repair action, i.e., in some cases, the fix could break the Dockerfile build process when a recommended package was removed. For example, the recommended package `ca-certificates` should be installed along with `wget` if we want to make a request to an SSL-enabled URL. We leave this for future work.

Use Version Pinning. A version must be specified for every installed package in the Dockerfile to completely fix the version pinning smells. Table I shows the list of sources we used to extract the package versions for each package manager.

In contrast to other package managers, the versions of *apt* and *apk* packages rely on the OS release versions of the base image defined in the Dockerfile. For example, it has been observed that the `git` package versions in the *apt* package manager database differ between Ubuntu 22.04 (`1:2.34.1-1ubuntu1.8`) and Ubuntu 20.04 (`1:2.25.1-1ubuntu3.10`) simultaneously. Using package versions from different OS release versions might lead to incompatible issues, therefore, potentially breaking the build. We found that *apt* and *apk* package managers are frequently used to install packages in Debian-based and Alpine Linux images (account for 94.5% of Dockerfiles in the original dataset). Therefore, to reduce overhead costs on duplicated requests to the version source websites for the same packages and versions, we wrote scripts to automatically download all the

[2]https://github.com/hadolint/hadolint/issues/329
[3]https://github.com/hadolint/hadolint/issues/328

TABLE I
SOURCES OF PACKAGE VERSIONS FOR PACKAGE MANAGERS.

Package Manager	Version Sources
apt	http://archive.ubuntu.com, https://archive.debian.org
apk	https://dl-cdn.alpinelinux.org
pip	https://pypi.org
npm	https://registry.npmjs.org
gem	https://rubygems.org

```
1  FROM alpine:3.17
2
3  - ADD https://example.com/tar.xz /opt
4  + RUN which wget &> /dev/null || apk add --no-cache
   ↪  wget=1.21.3-r2
5  + RUN wget -q -P /opt https://example.com/tar.xz
```

Fig. 2. An example of a patch generated by the repair action *"Use* `wget` *or* `COPY` *instead of ADD"*.

version information for *apt* and *apk* packages and store them in local databases for future queries while repairing the smells. The following steps were followed during the repair to obtain versions of *apt* and *apk* packages that correspond to the OS versions.

(i) Obtain the base image digest via DockerHub API.[4] Currently, we only support the images with *amd64* as the platform architecture.

(ii) Retrieve the base image based on the digest and check whether it is an OS image or not. The OS image must be a Debian-based or an Alpine Linux image. If the requirement is fulfilled, extract the OS name and the release version from the base image; otherwise, retrieve its parent images via Docker Scout API[5] and apply the same checks to them until a satisfied OS image is found. If the `scratch` image is reached, the repair should terminate here with no fixes returned.

(iii) Use the OS image name and version to find the corresponding versions of the installed packages in the local version databases. By default, the latest versions of the packages are returned. However, if a specific date is provided (such as the modified date of the Dockerfile), the versions released right before the given date are returned.

For other package managers, that are *pip*, *npm*, and *gem*, the sources in Table I provide API endpoints from which the latest versions of the packages can be fetched. However, since these package managers are less commonly used (only seven Dockerfiles in the original dataset), their package versions are fetched by the corresponding fixers on-the-fly when a repair is performed.

***Use* `wget` *or* `COPY` *instead of* `ADD`.** The `ADD` instruction can add files to the Docker image from multiple sources, such as system files on the current machine or external files from remote URLs. To fix this smell, the `ADD` instruction with normal file paths is replaced with a similar `COPY` instruction. Transformation into `COPY` instructions is not possible for URLs, as they do not support downloading remote files. Therefore, `RUN` instructions with `wget` are used for the fix to fetch the URLs. To this end, `wget` should be installed before we invoke it as the alternative to the `ADD` instruction. An additional `RUN` instruction for checking the presence of `wget` (and installing it if needed) is added right before the location of the original `ADD` instruction. Figure 2 illustrates an example of replacing an `ADD` instruction with the corresponding `wget` command in a Dockerfile using Alpine Linux as the base image. The installed version of `wget` package is provided by the *Version Pinning* repair action.

Have a non-root user. The smell *"Last user is root"* can exist in two scenarios: (1) no user is added in the Dockerfile, the `root` user is, therefore, implicitly used; and (2) `root` user is explicitly used via `USER` instruction. In both of the cases, the creation of a new user and switching to it via `USER` instruction at the end of the Dockerfile is deemed sufficient to address the smell. To verify that the last user is the `root` user, we check whether the argument of the last `USER` instruction is indeed `root` (username) or `0` (user ID).

Do not have secrets. Secrets can exist inside the `ENV` and `RUN` instructions. Currently, we support the detection and fix suggestion of secrets only in the `ENV` instructions. It should be noticed that using keywords or basic regular expressions to identify the secret names could produce false positive results. For example, if we consider `ENV` variables containing the keyword `KEY` as the secrets, `GOOGLE_API_KEY` is therefore included in the positive set, which is widely distributed to client applications and obviously not a secret. To avoid the false positive results of smell detection, as well as our effort in implementing complicated regular expressions, we currently consider 145 known variable names (divided into three groups: secret keys, secret tokens, and passwords) that are commonly used in the Binnacle dataset [5] as the database for finding secrets. It is not possible to completely fix this smell through the proposed repair action, as the secrets still need to be provided for the build context from outside. Therefore, secret variables are first removed from Dockerfiles. Moreover, to suggest the complete solution to the tool user, a comment is also added, which states that these secrets should be provided via `--secret` options of the Docker build command [19].

***Have a* `HEALTHCHECK`.** It often requires in-depth familiarity with the running application in the Docker container to write the command to check its health. Yet, we can create the basic `HEALTHCHECK` commands for several kinds of applications, such as web-based ones. For this purpose, we first checked if the Dockerfile used our considered applications (including *apache*, *nginx*, *node*, *php*, and *tomcat*) as the base images or not. If yes, we then added an `curl` command to test the accessibility to the defined URLs for health checking. The base images are fetched similarly in *Version Pinning*, and the `curl` package is installed in the same way in *Use* `wget` *or* `COPY` *instead of* `ADD`. For other base images, we added a comment to ask the tool users to write the `HEALTHCHECK` commands on their own.

[4]https://hub.docker.com/v2/

[5]https://api.dso.docker.com/v1/

B. Injection actions

Use `apt-get update` alone. To inject the smell, first, the existing `apt-get update` command in the same `RUN` instruction as `apt-get install` commands is replaced by a no-op (`:`). Then, a new `RUN` instruction with a single `apt-get update` is inserted before the original `RUN` instruction.

Last user is root. The `RUN` instructions may depend on the context of the running user. Therefore, it is possible to remove the existing user from the Dockerfiles. To inject this smell, we add a `USER` instruction at the end of the Dockerfile to switch the user to the root user. The argument for `USER` instruction is chosen to be either `root` or `0`.

Have secrets. The smell is injected by generating `ENV` instructions that typically contain the secrets. To this end, we randomly picked the variable names from the database we created for the repair action of this smell. Although the value of several secrets starts with a prefixed token (e.g., `SLACK_TOKEN` can start with "`xoxb`" or "`xoxp`" or "`xapp`"), we generate their values completely randomly. `ENV` instructions can exist in any place in the Dockerfile; therefore, we pick a random line number and insert `ENV` instructions with secrets inside that line.

V. Datasets of Smelly Dockerfiles

As shown in Figure 3, we have prepared three datasets for the experiments in our study, which are described in the following subsections.

A. (*DS1*) Original Dockerfile dataset

A dataset of *smell-free* Dockerfiles (i.e., containing no smells) is needed for our controlled experiments on injecting and fixing known smells. There exist multiple available Dockerfile datasets [5], [15], [20] with diversity in domains. However, none of these are smell-free. One possible option was to filter one of the existing datasets for smell-free Dockerfiles. We set the following criteria when selecting Dockerfiles to create our smell-free dataset:

- **C1. Non-trivial:** The Dockerfile should have a sufficient amount of instructions to create a high-quality dataset. As there are not many locations to inject the smells if the Dockerfile contains too few instructions.
- **C2. Reproducible build:** The Dockerfile should be buildable, as we want to check if our injecting and fixing actions affect the buildability of the Dockerfiles or not.
- **C3. Commonly used:** The Dockerfile is recommended to be selected from the commonly used Docker images on DockerHub as these Dockerfiles are well-written by experts and are more likely to contain less smell.

We evaluated the existing Dockerfile datasets and found that none of them met our defined requirements. For example, in the Binnacle dataset [5], we found that 33,056 (18.52%) of the Dockerfiles did not have the detected smells. Nonetheless, 21,454 (64.90%) of them contained only the `FROM` instructions and no other content. The complexity of these Dockerfiles is much lower than the general Dockefiles, and they, therefore, violated **C1**. Even the Binnacle's Gold Set (selected Dockerfiles from *docker-library* organization in

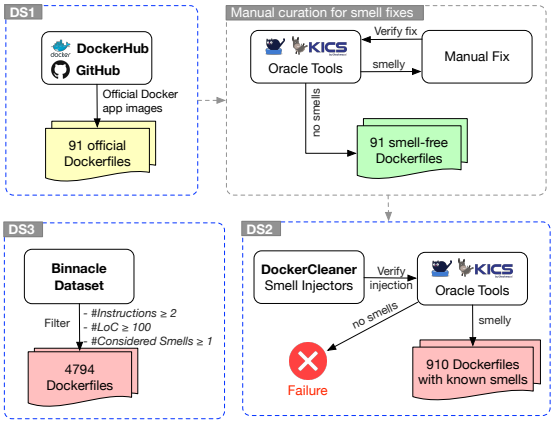

Fig. 3. The datasets we used and the approaches to create them.

Binnacle dataset [5]) violated our criteria. These Dockerfiles are sufficient in length and commonly used. However, they are different Dockerfiles for only 40 Docker images, and many of them are not reproducible for builds as they are deprecated or their build context files are missing. The build context files are required when we want to add external files to the Docker images with the `COPY` instructions. As a result, we were not able to use this Gold Set because it did not meet **C2**.

Therefore, we decided to find a dataset of Dockerfiles on our own that is able to be built, non-trivial, and well-written to create our dataset of smell-free Dockerfiles. To this end, we first found 173 official images from DockerHub[6]. These official images have seen widespread use, especially with more than one million pulls. From these 173 images, we excluded: (1) 29 deprecated images; (2) two images that are the old versions of *hello-world* image; (3) three images containing only the `FROM` instructions. After this step, we were left with 138 images. We further discarded the images for operation systems (OS) and programming languages (PL) as they are not necessarily checked for several of the security smells we consider in our study. For example, OS and PL images do not need a `HEALTHCHECK` instruction as they do not provide an actually running application whose health is to be checked. As a result, we retained 91 application images as our final Docker image list.

To find Dockerfiles for the chosen Docker images, we found 91 associated GitHub repositories via the image description texts. We then searched for all the existing Dockerfiles in the linked repositories[7]. In total, we found 866 Dockerfiles existed that could create different versions of these images (on average, 9.5 Dockerfiles existed in each repository, with the standard deviation being 15.93). To avoid bias toward specific images, a buildable Dockerfile was randomly selected with its context files from each repository. Consequently, the final dataset included 91 Dockerfiles that have been used to create 91 Docker images.

[6]https://hub.docker.com/u/library. Accessed on 01-03-2023.

[7]We performed GitHub searches to find Dockerfiles with the pattern "https://github.com/<org>/<project>/search?l=Dockerfile".

B. Manually-curated smell-free Dockerfile dataset

As we previously mentioned, in order to create the dataset of injected Dockerfiles (DS2), we need a smell-free dataset to start from. To this end, the security smells in the original Dockerfiles (DS1) must be removed. This was done by manually fixing every smell in each Dockerfile, which was detected by the smell detection tools. For most of the smells, it was straightforward to identify them and apply the corresponding fixes. For some smells, though, it was not obvious to synthesize their fixes. Note, we also fixed most of the smells detected by Hadolint and KICS rules which were not included in this study (e.g., DL3003: Use `WORKDIR` instead of `cd`, DL4006: Use `-o pipefail` for the shell, etc.). This helps make the smell-free dataset more usable for other studies. We further explain the curation procedures for the smells with non-trivial fixes as follows.

Have no `HEALTHCHECK`. Most of the Dockerfiles in the original dataset do not contain any `HEALTHCHECK` commands. To fix this kind of smell, we first went to the official Docker's healthcheck repository[8] to look for the related healthcheck scripts of the applications running in the Dockerfiles. For the applications which are not listed in the repository, if they serve any web services, we then used the command `curl` to check whether we could reach the web service or not. Otherwise, we wrote the `HEALTHCHECK` commands based on our knowledge.

No Version Pinning. We extracted the package versions from the build logs of Dockerfiles and bound them to the packages in the installation commands. To this end, we scanned the build logs to find the lines matching our defined patterns from which we could retrieve the names and versions of the installed packages. For example, one of the patterns we used to obtain versions of *apk* packages is `Installing <package_name> (<version>)`. This step was done automatically and generated a `csv` file containing information on packages and versions. Next, for each Dockerfile, we then selected the corresponding versions of the packages and manually pinned them in the installation commands. If we found a package was installed with its alias name, we then replaced it with the official name of the package. For the known false positives of Hadolint, we substituted the shell variables with their values if it was possible to infer them in the Dockerfiles. Otherwise, we moved the *falsely detected* installation commands into a new `RUN` instruction, and we then added a Hadolint's pragma to ask the tool to ignore version pinning checks on it. This ensured our dataset was truly smell-free to the oracle tools.

Use no `--no-install-recommends`. We first added the flag `--no-install-recommends` to the `apt-get install` commands, then we built the Dockerfile. If the build was broken, which was mostly due to the removal of suggested packages, we manually identified and added them to the installation commands.

[8]https://github.com/docker-library/healthcheck

TABLE II
DS2: THE NUMBER OF ATTEMPTS TO INJECT SMELL TYPES AND THE SUCCESS RATES.

*These smell types belong to the same smell type *addInsteadOf*{*Wget,Copy*}, however, are generated with different injectors.

Smell Type	#Attempts	#Succeed	
noVersionPinningAptGet	428	233	54.44%
noVersionPinningApk	431	151	35.03%
noVersionPinningPip	442	13	2.94%
noVersionPinningNpm	454	11	2.42%
noVersionPinningGem	403	5	1.24%
noAptGetInstallRec	451	224	52.21%
useAptGetUpdateAlone	429	248	54.99%
addInsteadOfWget*	460	9	1.96%
addInsteadOfCopy*	425	348	81.88%
lastUserIsRoot	473	280	59.20%
haveSecrets	450	450	100%
haveNoHealthcheck	441	441	100%

C. (*DS2*) Synthetic Dockerfile dataset of known smells

After obtaining 91 smell-free Dockerfiles, we tried to inject the smells into to create the smelly dataset.[9] Particularly, for each Dockerfile, we used ten different random permutations on all the smells to create ten injected Dockerfiles with different smell lists. This increased the variety of smell types in the injected Dockerfiles. We obtained a total of 910 Dockerfiles with known smells. Table II shows how often each smell type was attempted to inject into the Dockerfiles and the successful attempts rate. For most smells, it was successful for most of the attempts; however, not for some smells. For instance, several package managers had very few successful injections for their version pinning smells. The reason is that these package managers have been used in a few Dockerfiles (one original Dockerfile for *gem*, two for *pip*, and four for *npm*). The smell *addInsteadOfWget* had only nine successful injections as the `ADD` instruction currently does not support many features that `wget` offers (e.g., adding specific HTTP headers for authentication). Only one original Dockerfile could be injected with this smell. All the injected smells were detected correctly by the detection tools, and further manual checks on a sample of 200 random Dockerfiles verified that there were not any false positive or false negative results from the detection tools.

D. (*DS3*) Large-scale real-world Dockerfile dataset with smells

For the extended evaluation of DOCKERCLEANER, we reused the real-world Dockerfiles from Binnacle dataset [5]. To this end, we selected 4957 Dockerfiles, each of which contains at least two instructions and at least 100 lines of code, as we wanted to have the Dockerfiles that are sufficient in complexity and more likely to have the smells than the simple Dockerfiles. We further discarded 160 Dockerfiles that could not be parsed by *language-docker* and three Dockerfiles that were not detected with any considered smells by the oracle tools.

[9]Hereafter, we use the terms "synthetic dataset" and "injected dataset" interchangeably to refer to this dataset.

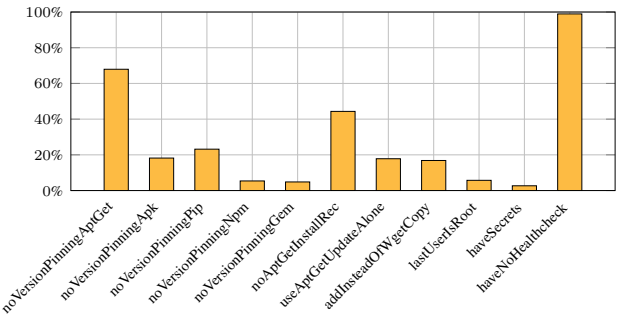

Fig. 4. DS3: The smells prevalence in the large-scale Dockerfile dataset.

After this step, we are left with 4794 Dockerfiles as our final dataset. Figure 4 shows how prevalent each smell type exists in this dataset. Interestingly, the smell *noAptGetInstallRec* is simple to avoid in practice; however, it stays in 44.36% of the real-world Dockerfiles in this dataset.

VI. Results

In the upcoming subsections, we describe the results of the conducted experiments to answer the three research questions that we posed.

A. RQ1. Repair performance on DS1 and DS2

To answer the first research question, we used DockerCleaner to automatically repair smells in the injected dataset and the original dataset. We included the original Dockerfiles for the evaluation as we wanted to check if our tool could fix the smells in their natural forms rather than their artificially injected versions. For each injected Dockerfile, a smell was only attempted to be fixed if we knew that it was successfully injected into the Dockerfile before. Meanwhile, all the smells were attempted to be fixed for the original Dockerfiles. We then checked if the smells were completely addressed by analyzing the results with the oracle tools. For each smell type, we evaluated the repair effectiveness by calculating the number of smelly Dockerfiles detected by the oracle tools before and after the repairs. We considered a repair successful only if it removed all the instances of the smell types in the Dockerfile. Table III shows repair results of DockerCleaner on these two datasets.

Synthetic dataset performance (DS2). Most of the smells were fixed successfully in the dataset. The overall repair effectiveness is 92.67%, which means that the smells were removed completely from 92.67% of the injected Dockerfiles. Nonetheless, it is not surprising that *haveNoHealthcheck* is the least repaired smell type by far. It requires a deep understanding of container applications to synthesize the proper `HEALTHCHECK` commands. This smell still exists in 339 Dockerfiles after the repairs. For a number of Dockerfiles, the *No Version Pinning* smells for *apt* and *apk* was not fully fixed. We found that the *apt* and *apk* packages, which could not have their pinned, are stored in different version sources than the ones we used as in Table I. For instance, *mongo* developers store *mongo*-related *apt* packages on their own repository.[10] The addition of new version sources in our tool will resolve this issue.

Original dataset performance (DS1). The original Dockerfiles violated only six smell types in our smell list. There is a similar trend for fixing smells in the original and injected datasets. The percentage of original Dockerfiles that are smelly to *No Version Pinning* for *apt* and *apk* remains higher than in the injected datasets after repair. This can be explained by two reasons. First, 22 original Dockerfiles used shell variables, such as `$buildDeps` and `$runDeps`, as the proxies for the list of installed packages. Our tool did not support repairing the smells in these cases, as well as Hadolint detected the wrong smells for them. Second, several Dockerfiles tried to install the packages with their alias names. For instance, *couchbase*'s Dockerfile used the alias name `man` to install the *apt* package `man-db`. DockerCleaner currently allows pinning versions for only the official package names.

Impact on buildability. After the repairs, we built the Dockerfiles to make sure that our suggested fixes did not break the builds. Table IV shows the number of successful builds before and after applying the smell injection and repair actions on our datasets. It should be noted that 20 out of 910 injected Dockerfiles were unable to build before the repairs. If we take only the buildable Dockerfiles into account as the input, DockerCleaner-suggested fixes break the builds for 0.67% of injected Dockerfiles and 5.49% of original Dockerfiles. By analyzing the build logs, we found that *language-docker* could not parse correctly the Dockerfiles of two images, which caused the build degradation of 2.19% for the original Dockerfile repairs. This was reflected for the smell injection, i.e., the Dockerfiles seeded from these two original Dockerfiles was not buildable. The remaining degradation of the original Dockerfile builds was caused by suggested fixes to *noAptGetInstallRec* in cases where it removed recommended packages that were necessary for the builds. An issue related to the repair action *Use Version Pinning* in *mariadb*'s Dockerfiles caused the build degradation for the repair of the injected dataset. These Dockerfiles fetched *apt* packages from both the Debian repository and the mariadb repository, with a preference for the latter. While the *mariadb*-related packages should be obtained from the mariadb repository, DockerCleaner pinned them with the incompatible versions from the Debian repository.

Answers to RQ1: DockerCleaner is able to repair most of the security smells with the overall repair effectiveness of 92.67% in the DS2 dataset. *"Have no* `HEALTHCHECK`*"* is by far the less repaired the security smells. We found that only 0.67% of Dockerfiles were broken due to the suggested fixes from our tool. A similar repair trend is confirmed by the repair results in the DS1 dataset.

B. RQ2. Repair performance on the large-scale dataset (DS3)

In this research question, we followed the same approach we used to evaluate DockerCleaner on the dataset of original

[10] https://repo.mongodb.org/apt/

TABLE III
THE NUMBER OF SMELLY DOCKERFILES DETECTED BEFORE AND AFTER THE REPAIR OF THE ORIGINAL (DS1) AND INJECTED (DS2) DOCKERFILE DATASETS.

"–" denotes that the smell type does not exist in the mentioned dataset.

Smell Type	#Smelly Dockerfiles							
	Injected dataset (DS2)				Original dataset (DS1)			
	Before repair		*After repair*		*Before repair*		*After repair*	
noVersionPinningAptGet	233	100.00%	15	6.44%	48	52.75%	12	13.19%
noVersionPinningApk	151	100.00%	7	4.64%	32	35.16%	13	14.29%
noVersionPinningPip	13	100.00%	0	0.00%	1	1.10%	0	0.00%
noVersionPinningNpm	11	100.00%	0	0.00%	1	1.10%	0	0.00%
noVersionPinningGem	5	100.00%	0	0.00%	–	–	–	–
noAptGetInstallRec	224	100.00%	0	0.00%	17	18.68%	0	0.00%
useAptGetUpdateAlone	248	100.00%	0	0.00%	–	–	–	–
addInsteadOf{Wget,Copy}	357	100.00%	0	0.00%	–	–	–	–
lastUserIsRoot	280	100.00%	0	0.00%	–	–	–	–
haveSecrets	450	100.00%	0	0.00%	–	–	–	–
haveNoHealthcheck	441	100.00%	339	76.87%	89	97.80%	69	75.82%

TABLE IV
THE BUILD ERROR RATE AFTER APPLYING SMELLS INJECTION AND REPAIR ACTIONS IN DS1, DS2, AND THE SMELL-FREE DATASET.

*The size of the smell-free dataset was multiplied up to 910 Dockerfiles before we injected the smells.

Action: Input dataset → Output dataset	#Buildable *Before*	#Buildable *After*	Build degradation
Smell Injection: Smell-free* → DS2	910	890	2.20%
Smell Repair: DS2 → Fixed	890	884	0.67%
Smell Repair: DS1 → Fixed	91	86	5.49%

Dockerfiles. We excluded the buildability verification step as it might not be possible to build many Dockerfiles in this dataset (as we mentioned in V-A). We also included the repair results from PARFUM [8], which is the only available tool for fixing Dockerfile smells in the literature, to compare with our tool. To this end, we first obtained the source code of PARFUM (checkout version: c40db3e) and executed it to repair our considered smells in this dataset. Table V presents the smell repair results of DOCKERCLEANER and PARFUM on the large-scale dataset. The symbol "–" denotes that PARFUM has not run on that kind of smell. According to PARFUM's documentation, this tool supports the detection and repair of four smell types in our study.[11]

Overall, DOCKERCLEANER has the repair effectiveness as 64.04% on the large-scale dataset across all considered smell types. For the common smells, on average, the repair effectiveness of DOCKERCLEANER and PARFUM are 82.81% and 46.70%, respectively. This indicates that our tool outperformed PARFUM by 36.11% in terms of fixing these smell types. Our results showed that PARFUM's performance is inferior in repairing *lastUserIsRoot* and *useAptGetUpdateAlone* smells. Inspecting PARFUM's source code revealed that this tool supports detection; however, not yet repair for *lastUserIsRoot*. Surprisingly, Hadolint could not detect this smell type in four smelly Dockerfiles after we repaired them with PARFUM. Further manual checks disclosed that this phenomenon was caused by the issue with the repair rule "*Use COPY Instead Of ADD*" of PARFUM. This rule generated the wrong Dockerfile syntax in some cases, as illustrated in the first example case in Figure 5. Hadolint cannot parse and scan smells in a Dockerfile that contains syntax errors. Therefore, no smells were detected by Hadolint in these Dockerfiles. By adopting the AST parser from Hadolint, which is well-developed and actively maintained, our tool could mitigate the risks of generating Dockerfiles with syntactic errors and successfully fix this kind of smell. For *useAptGetUpdateAlone* smells, PARFUM does not support the repair if there were multiple `apt-get update` and `apt-get install` commands in the Dockerfile. This is demonstrated as the second example case in Figure 5. While PARFUM could not propose any fixes, DOCKERCLEANER-suggested a fix that passed the KICS checks by inserting `apt-get update` right before each of `apt-get intall` commands and removing the original `apt-get update` command. A repair that combines both the `apt-get update` command and the `apt-get intall` commands into a single instruction would be the better solution. We leave this for our future work. Regarding *"Use ADD instead of wget and COPY"* smells, the lower repair performance of PARFUM compared to our tool is due to the fact that PARFUM cannot fix the `ADD` instructions with URLs while DOCKERCLEANER supports that.

Answers to RQ2: DOCKERCLEANER is able to repair security smells for 64.04% of the large-scale Dockefiles in the DS3 dataset. Especially to a group of four security smells, DOCKERCLEANER significantly outperforms the baseline repair tool by 38.11% in terms of repair effectiveness. We discussed the reasons why our tool is superior to the baseline in repairing these types of smells.

C. RQ3. Developer feedback on submitted repairs

In this final experiment, we evaluated the practical benefits of DOCKERCLEANER to the developers. To this end, we

[11] https://github.com/tdurieux/docker-parfum/blob/c40db3e/README.md

TABLE V
THE NUMBER OF SMELLY DOCKERFILES DETECTED BEFORE AND AFTER THE REPAIR OF THE EXTENDED DATASET OF 4794 DOCKERFILES (DS3).
"–" denotes that the mentioned tool does not support the repair of the smell type.

Smell Type	#Smelly Dockerfiles					
	Before repair		*After repair*			
			DOCKERCLEANER		PARFUM	
noVersionPinningAptGet	3259	67.98%	2702	56.36%	–	–
noVersionPinningApk	873	18.21%	696	14.52%	–	–
noVersionPinningPip	1112	23.20%	435	9.07%	–	–
noVersionPinningNpm	258	5.38%	19	0.40%	–	–
noVersionPinningGem	231	4.82%	1	0.02%	–	–
noAptGetInstallRec	2127	44.37%	193	4.03%	73	1.52%
useAptGetUpdateAlone	855	17.83%	470	9.80%	822	17.15%
addInsteadOf{Wget,Copy}	808	16.85%	0	0.00%	122	2.54%
lastUserIsRoot	275	5.74%	13	0.27%	271	5.65%
haveSecrets	128	2.67%	32	0.67%	–	–
haveNoHealthcheck	4746	99.00%	4378	91.32%	–	–

```
# Smell case 1: Use wget or COPY instead of ADD
ADD . /home/dataman/n6
# Parfum's patch, breaks the Dockerfile-syntax
# (At least two arguments are required for COPY instructions)
COPY ./home/dataman/n6
# DockerCleaner's patch, passes the checks of Hadolint
COPY . /home/dataman/n6
----------
# Smell case 2: Do not use apt-get update alone
RUN apt-get update
RUN apt-get intall -y --no-install-recommends wget
RUN apt-get intall -y --no-install-recommends git

# Parfum's patch
## No changes

# DockerCleaner's patch, passes the checks of KICS
RUN :
RUN apt-get update && apt-get intall  -y --no-install-recommends wget
RUN apt-get update && apt-get intall -y --no-install-recommends git
```

Fig. 5. Patches generated by PARFUM and DOCKERCLEANER.

utilized the DOCKERCLEANER-suggested fixes for the Dockerfiles of official Docker images and submitted pull requests to their corresponding GitHub repositories. The main goal is to obtain and analyze the developer's feedback on the fixes generated by our tool. *No Version Pinning*, *noAptGetInstallRec*, and *haveNoHealthcheck* were the three smells detected and repaired. However, we were able to make pull requests for only *noAptGetInstallRec*. The reason is that, according to the policies for creating official images, explicit `HEALTHCHECK` are not added to the official image for several reasons, such as the expectation that end users of the images will add them [21]. For *Version Pinning*, some package managers, such as *apk*, only retain the latest package versions in their repositories. Therefore, before each build of the Dockerfiles, the versions of the installed packages need updating by this repair action (with the database of the latest versions). In these cases, the *Version Pinning* should be performed by the developers that build the Docker images. From the 17 images that *noAptGetInstallRec* smells were fixed by DOCKERCLEANER, we excluded two images that the developers have already addressed the smells in their latest versions of Dockerfiles. We further discarded three images that the proposed fixes did not remove any unnecessary packages. We then manually created twelve pull requests in which we explained the smells and the fixes created by our tool. As a result, we handed in 12 pull requests to the developers. In cases of *kong*, *tomee*, and *php-zendserver*, we extended the originally generated fixes to make the Dockerfiles buildable by adding the missing packages.

Table VI shows the pull requests that we have sent. The first two columns contain the image names and their total pull counts on DockerHub. Most of the images have millions of pulls as they are official images and are widely used. The next two columns show the number of fixed smells and fixed Dockerfiles, followed by the manual changes for the fixes if needed. The last column indicates the status of the pull request, which can be categorized in: *Merged*, *Accepted*, and *Open*. Developers could review and accept the proposed fixes soon; however, it might take time to merge the pull request due to the policies of the contribution process in the projects. For instance, we had to file an issue on the Jira system of *solr* before we submitted the pull request to its repository. At the time of writing, we have received developer responses for nine pull requests. Nine pull requests were accepted, and eight of those were further merged. There was no negative feedback towards our proposed fixes. In several cases, the developers even asked us to include the fixes for other Dockerfiles in the repositories. We then ran DOCKERCLEANER to repair them and updated the pull requests. Developers commented on the pull requests that *"That looks great, thanks for the improvement!", "I was expecting that one :-D Thanks for your work.", "Interesting... I hadn't heard about this ... Thanks for the contribution", "Thank you for the PR ... I am fine with the --no-install-reccomends as it makes sense to me."* Overall, the results showed that the developers of official Docker images were open to improvements in fixing security smells suggested by our tool; they also accepted the pull requests and integrated the suggested fixes into their Dockerfiles.

TABLE VI
THE PULL REQUESTS SUBMITTED TO THE PROJECTS OF DOCKER OFFICIAL IMAGES. BASED ON THE PROPOSED FIXES BY DOCKERCLEANER IN DS1.

PC = Pull Count, #FS = #Fixed Smells, #FD = #Fixed Dockerfiles, Human interv. = Human intervention was needed.

Official image	PC	#FS	#FD	Human interv.	Status
backdrop	6.8M+	1	1		Merged
couchbase	83M+	1	1		Accepted
couchdb	179M+	2	1		Open
hitch	363K+	2	1		Merged
kong	308M+	2	1	Add ca-certificates	Open
mysql	3.6B+	1	1		Open
php-zendserver	4.1M+	2	1	Add ca-certificates,patch	Merged
rethinkdb	73M+	1	1		Merged
silverpeas	1.7M+	2	1		Merged
solr	139M+	2	2		Merged
tomee	21M+	43	43	Add dirmngr	Merged
varnish	13M+	6	3		Merged
Total (12 pull requests)		65	57	8 Merged, 1 Accepted, 3 Open	

Answers to RQ3: Twelve pull requests have been sent to the developers of the official Docker images. Nine of them have been approved, from which eight have been merged into the Dockerfiles without receiving any negative feedback. This proves that the developers have acknowledged the effectiveness of the fixes provided by DOCKERCLEANER.

VII. THREATS TO VALIDITY

A threat to external validity is that the smells we selected for our study may not represent all security-related smells in Dockerfiles and thus may affect our study's generalizability. To mitigate this risk, we elected the smells based on the best security practices in writing Dockerfiles that are synthesized by the security experts. The chosen smells are well-acknowledged by many scanning tools for security, showing that they are prominent in practice. Additionally, the official Dockerfiles we collected may not be representative of all the Dockerfiles in general. We mitigated this risk by selecting only the Dockerfiles that are non-trivial and span over multiple applications. We also extracted large-scale Dockerfiles from the Binnacle dataset to extensively evaluate our tool. Another threat is that our injected dataset may not be realistic as the real-world Dockerfiles. The evaluation showed that the trends in repair results are similar between our synthetic dataset and the real one (original Dockerfiles). Moreover, the verification of smell presence in our study relies on the results of Hadolint and KICS. We found several issues with the tools during our study and reported them to their developers. To present the current results in our paper, we used the newer versions of the tools, which include the improvements based on our reports. Still, we acknowledge a few limitations of Hadolint. Potential threats to our internal validity refer to the errors in our implementation and experiments. We have carefully examined our implementation of DOCKERCLEANER and experiments to mitigate these risks. Likewise, we reassessed the results of the manual curation of smell-free Dockerfiles and the creation of injected Dockerfiles with known smells.

VIII. RELATED WORK

Since its initial launch in 2013, Docker has become one of the most prominent tools for virtualization over the years. There have been a plethora of off-the-shelf tools developed for assessing the quality and security of Dockerfiles, Docker images, and Docker containers. As well as a growing number of studies have been carried out to improve the maintainability and security of projects using Docker.

Best practices and tools for Docker security. There exist the best practices for better usage of Docker, such as CIS Docker benchmark [11] and OWASP Docker Security Cheat Sheet [12] that have been edited by the security experts. In its latest version, v1.5.0, CIS Docker benchmark [11] contains 117 security recommendations which are systematically categorized into seven different groups of Docker configurations and usages, including the creation of Dockerfiles. Many available scanning tools, including both open-source tools [2]–[4], [22], [23], and commercial tools [24]–[27], can detect smells and vulnerabilities in multiple levels of Docker's artifact. Several tools such as Dockle [22] and Docker Bench for Security [4] support the checks based on the recommendations in CIS Docker benchmark [11]. Recently, Docker developers have integrated their own security tool, namely Docker Scout [28], to help search and fix vulnerabilities in the Docker images. Regarding Dockerfile linting, Hadolint is the most established tool and is widely used in practices and research studies.

Detection and analysis of Dockerfile smells. Xu et al. [29] proposed two detection techniques based on static and dynamic analyses to identify a type of smell they found in Dockerfiles, namely *"Temporary File Smell"*. Henkel et al. [5] introduced a static analysis tool, namely BIANNCLE, to contain 23 smell detection rules which they obtained from a frequent sub-tree mining tool that they created. The evaluation on 6,334 Dockerfiles of Wu et al. [30] showed that smells are very common and that co-occurrences exist between different smells. Lin et al. [15] ran Hadolint on over 3.3 million Docker images (98.38% of images hosted on DockerHub) and found that the prevalence of smells in Dockerfiles causing larger size for images is reducing over the years. By analyzing 9.4 million Dockerfiles from *World of Code* dataset, Eng et al. [14] reconfirmed the previous studies about the downtrend of smell prevalence in recent Dockerfiles.

Repair of Dockefile smells. The work of Durieux et al. [8], which introduced a repair tool, namely PARFUM for Dockerfile smells, is the closest to our work. PARFUM derived their fix actions for general smells mostly based on the rules proposed by Henkel et al. [5], while our work focused on developing repair actions for security-related smells. There are four kinds of security smells in our study that, according to its documentation, PARFUM supports in detection and repair. Therefore, we have included PARFUM in our evaluation, and the results showed that our tool is superior to PARFUM in repairing security smells. Rosa et al. [9] proposed a preliminary approach to fix the smells in Dockerfiles; however, the implemented

tool has not yet been available to be compared with our tool. Zhang et al. [6] recommended the base images for Dockerfiles by leveraging deep configuration comprehension about the Dockerfiles via a neural network. SHIPWRIGHT, a repair tool presented by Henkel et al. [31], targeted to the repair of broken Dockerfiles. This tool employed 13 repair rules which have been mined and curated from the build logs of broken Dockerfiles. Hassan et al. [7] proposed a technique to recommend the updates for the Dockefiles by performing change impact analysis for environment-related code scopes.

IX. CONCLUSION

In this paper, we have presented DOCKERCLEANER, an automated tool for repairing eleven types of security smells in Dockerfiles. By leveraging the AST parser from Hadolint, the most well-known linter for Dockerfiles, our tool can efficiently parse the Dockerfiles and repair the existing smells. We evaluated DOCKERCLEANER on a dataset of 910 synthetic Dockerfiles with known smells. The repair results show that the smells are fully addressed in 92.67% of these Dockerfiles, and the proposed fixes only break 0.67% of the Dockerfile builds. In the extended evaluation, we compared our tool against the state-of-the-art repair technique on a dataset of 4794 large-scale Dockerfiles. Experiments show that DOCKERCLEANER significantly outperforms the state-of-the-art tool by 36.11% in terms of repair effectiveness. Regarding the practical benefits of our tool, we found that the developers highly acknowledge the proposed fixes of our tools for their Dockerfiles.

In the future, we plan to improve the repair actions that require more complicated fixes, such as *"Have a* `HEALTHCHECK`*"* and *"Use Version Pinning"*. We also would like to extend our tool to support more types of security smells in Dockerfiles and other IaC platforms such as Terraform, Ansible, etc.

REFERENCES

[1] "Docker - develop faster. run anywhere." https://docker.com/, accessed: 2023-04-12.

[2] "Dockerfile linter, validate inline bash, written in haskell," https://github.com/hadolint/hadolint, accessed: 2023-04-12.

[3] "Find security vulnerabilities, compliance issues, and infrastructure misconfigurations," https://github.com/Checkmarx/kics, accessed: 2023-04-12.

[4] "The docker bench for security - a script that checks for dozens of common best-practices," https://github.com/docker/docker-bench-security, accessed: 2023-04-12.

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

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

[7] F. Hassan, R. Rodriguez, and X. Wang, "Rudsea: recommending updates of dockerfiles via software environment analysis," in *Proceedings of the 33rd acm/ieee international conference on automated software engineering*, 2018, pp. 796–801.

[8] T. Durieux, "Parfum: Detection and automatic repair of dockerfile smells," *arXiv preprint arXiv:2302.01707*, 2023.

[9] G. Rosa, S. Scalabrino, and R. Oliveto, "Fixing dockerfile smells: An empirical study," *arXiv preprint arXiv:2208.09097*, 2022.

[10] "Haskell dockerfile parser, pretty-printer and edsl," https://github.com/hadolint/language-docker, accessed: 2023-04-12.

[11] "Cis docker benchmark," https://www.cisecurity.org/benchmark/docker, accessed: 2023-04-12.

[12] "Owasp docker security cheat sheet," https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html, accessed: 2023-04-12.

[13] "The replication package for dockercleaner: Automatic repair of security smells in dockerfiles." https://github.com/tuhh-softsec/DockerCleaner, (the artifact of our evaluation study).

[14] K. Eng and A. Hindle, "Revisiting dockerfiles in open source software over time," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 449–459.

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

[16] 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.

[17] "Dockerfile best practices," https://docs.docker.com/develop/develop-images/dockerfile_best-practices/, accessed: 2023-04-12.

[18] "Shellcheck, a static analysis tool for shell scripts," https://github.com/koalaman/shellcheck, accessed: 2023-04-12.

[19] "Docker buildkit," https://docs.docker.com/engine/reference/commandline/buildx_build/, accessed: 2023-04-12.

[20] Y. Ma, C. Bogart, S. Amreen, R. Zaretzki, and A. Mockus, "World of code: an infrastructure for mining the universe of open source vcs data," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 143–154.

[21] "Faq for official docker images," https://github.com/docker-library/faq#healthcheck, accessed: 2023-04-12.

[22] "Dockle - container image linter for security, helping build the best-practice docker image," https://github.com/goodwithtech/dockle, accessed: 2023-04-12.

[23] "Cis docker benchmark - inspec profile," https://github.com/dev-sec/cis-docker-benchmark, accessed: 2023-04-12.

[24] "Providing a centralized service for inspection, analysis, and certification of container images," https://anchore.com/, accessed: 2023-04-12.

[25] "Aqua security stops cloud native attacks across the application lifecycle," https://www.aquasec.com/, accessed: 2023-04-12.

[26] "Quay [builds, analyzes, distributes] your container images," https://quay.io/, accessed: 2023-04-12.

[27] "Snyk is a developer security platform," https://snyk.io/, accessed: 2023-04-12.

[28] "Docker scout," https://www.docker.com/products/docker-scout/, accessed: 2023-04-12.

[29] 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.

[30] Y. Wu, Y. Zhang, T. Wang, and H. Wang, "Characterizing the occurrence of dockerfile smells in open-source software: An empirical study," *IEEE Access*, vol. 8, pp. 34 127–34 139, 2020.

[31] 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.