Politecnico
di Torino

Politecnico di Torino
Master of Science in Computer Engineering

Master Degree Thesis

Security Misconfigurations Detection and Repair in Dockerfile

Supervisor:
Prof. Paolo Ernesto Prinetto
Company Supervisors:
Dott. Riccardo Bortolameotti
Giuseppe Massaro

Candidate:
Lorenzo Antonio De Giorgi

Academic Year 2021/2022

Abstract

Containers offer a lightweight model for quick deployment of applications in cloud-based infrastructures, based on small, modular and transient services.
Several companies in IT industry adopt this mode of deployment instead of hypervisor-based infrastructures for different reasons: container images are portable in any locations without the need of being modified, containers offer near-native performance and permit a high degree of scalability.
Despite the numerous advantages, containerization technology raises different security concerns. In this regard, the most alarming factor is the minor layer of isolation between instances compared with hypervisor-based solutions. In this sense, the first barrier against several attacks is a container configured according to the most recent security best practices.
Unfortunately, manually hardening containers in a wide and complex environment is an error-prone and time-consuming activity. For this reason, numerous tools has been designed in order to help developers in this task.
Among the different containerization technologies, this work is focused on the de-facto standard in this field: Docker. In particular, this work aims to describe security misconfigurations that might affect Dockerfile and outline the current awareness of developers about them.
We perform a thorough evaluation of existing tools that identify misconfigurations in Dockerfile, using well-know and publicly available datasets. As a result of this evaluation, we discuss how common are security misconfigurations in Dockerfiles and the lack of tools that help to automatically repair such misconfiguration. In order to fill this gap, we develop a system which allows detecting and automatically repairing security misconfiguration in Dockerfile.

Contents

Chapter 1

Introduction

Over the last years, there has been an increasing use of virtualization technologies in order to improve the scaling ability quickly and effortlessly. Among all the different technologies, container-based solutions has been widely adopted from several companies to host distributed applications because of their flexibility and performance. Specifically, containerization is a virtualization technology that allows to package an application along with its dependencies in a self-contained unit which can run on different locations without any alteration [5].
Since containers are tightly integrates into the host operating system, they allow better performances compared with traditional virtual machines but, on the other hand, they offer a lower level of isolation between the virtualized environment and the host.
The de-facto standard in this field is Docker[1] which is an open source solution that ease the creation and the distribution of containers. Through the use of a textual file called Dockerfile, it is able to package application that can be easily deployed both in on-premise and cloud-based environments.
With the spread of Docker as virtualization technology, new threats have emerged. In fact, wrong configurations might provide several entry points for malicious acts. Furthermore, the tight bond between the container and the host allows the potential attack to jeopardize not only the application itself but the underlying environment as well [3].
Given the fastest growing of Docker, numerous new components have been added to it, such as Docker Swarm and Docker Compose. While these tools further simplify the developing and deploying phases, at the same time, they increase the attack surface, thus they require to be hardened in order to ensure a secure environment.

[1] https://www.docker.com/

1.1 Thesis objective

The work presented in this thesis has been accomplished in collaboration with the engineering division of ReaQta.
In particular, we have decided to focus on the basic constituent of each Docker container: the Dockerfile.
As a first goal, we want to estimate the extent of Dockerfile security misconfigurations by means of publicly available datasets. Due to this estimation, we can evaluate whether the developers are generally aware of all the best practices in Dockerfile hardening.
Detecting and removing all the possible source of insecurity manually is a complex and time-consuming task, for this reason the second step is to probe the presence of existing tools able to automate these operations. Once these tools have been identified, we want to know if they are actually effective.
Finally, we want to capitalize the previous findings by building a custom tool which might help the developers by complementing and enhancing already existing tools.
Specifically, the contributions of this work are the following:

- Assessing the extent of Dockerfile security misconfigurations by analyzing thousands of different Dockerfile coming from publicly available datasets.

- Investigating the presence of tools able to detect and repair security misconfigurations in these Docker artifacts.

- Implementing and evaluating a system which is able to detect and fix insecure Dockerfile.

1.2 Thesis description

After **Chapter 1** introduced the goals to achieve, the rest of the thesis is structured as follows:

- **Chapter 2** presents the basic knowledge that are useful for the better comprehension of this work. Specifically, it explains what the Docker ecosystem is composed of and why it has become the standard virtualization technology nowadays. Moreover, it outlines the components involved in the creation and deployment of containers. Finally, it highlights the security risks deriving from the use of insecure Dockerfile.

- **Chapter 3** discusses the works that have been written so far about security misconfigurations, taking into account both Dockerfile and other similar technologies. It also explores the most recent findings about automatic program repair, thus software able to fix other software, paying particular attention to the ones able to deal with Dockerfile.

- **Chapter 4** describes the process taken into account to assess the current level of awareness about Dockerfile misconfigurations. Secondly, it provides a comparison among different existing tools to detect misconfigurations in Dockerfile. We also discuss the results obtained, pointing out the gap underlying the current tools and the reasons that motivate us to build a new tool.

- **Chapter 5** describes the system that we have built, starting from the architecture overview and continuing with the description of each component. Furthermore, it uses a sample to delineate the system workflow and to specifically explain how an insecure Dockerfile is repaired.

- **Chapter 6** explains the tests we have conducted on our system in order to prove its validity and show the limitations. We also discuss the steps that could be taken in order to improve the tool effectiveness and which features could be introduced to enhance its capabilities.

- **Chapter 7** summarizes the results that we obtain in the previous chapters, in respect of the goals that we have established in the first place.

- **Chapter 8** reports technical details about the systems building that we omit in the previous chapters for sake of simplicity.

Chapter 2

Background

This chapter aims to provide knowledge about the main topics of this work. It begins with the definition of Abstract Syntax Tree in Section 2.1, which is an important way to represent source code and allows to easily manipulate it. In Section 2.2 is presented the containerization technology that we have decided to focus on: Docker. In Section 2.3 we describe more in detail Docker, focusing on its internal details. This accurate description is fundamental to understand the security risks behind Docker, pointed out in Section 2.4. In particular, Section 2.5 contains security best practices involving a fundamental Docker component: Dockerfile.

2.1 Abstract Syntax Tree

Repairing misconfigurations implies a modification of the original source code and in some cases, these modifications can deeply affect the code. Furthermore, detecting wrong configurations in large source code might be difficult due to the high number of instructions and parameters.
Therefore, working at text line granularity might increase the complexity of the detection and repairing processes. In order to overcome these limitations, it is possible to work at Abstract Syntax Tree level.
An Abstract Syntax Tree (AST) is a tree-based representation of source code, written according to a specific formal language.
The formal language specifies a set of grammar rules that determine how to translate different instructions into nodes belonging to the AST.
This representation captures solely the fundamental underlying concepts, while omitting unnecessary details [31].
Formally, an AST is a labeled ordered rooted tree. Let T be an AST, thus a set of nodes. A tree T has one node r that is root. Each node $t \in T$ has a parent $p \in T \cup \emptyset$. The only node which has $p \in \emptyset$ is r. Then each node $t \in T$ has a sequence of children. Each node has a label which correspond to the name of its rule in the grammar file, and might have a value which corresponds to the actual token in the code [11].

The conversion from source code to AST is obtained by means of a parser which analyzes each specific construct in the code and builds a new node out of it. It is worth mentioning that a source code could be parsed in different AST according to the specific grammar rules used: for instance, a node could encode a whole instruction or finer grain expressions.
AST are widely used by the compilers as intermediate representation of the source code [16]; moreover, they are used by numerous works to easily manipulate and compare different source codes [17][1][10]; indeed, we have chosen this representation precisely because it would be easier to manipulate the Dockerfile in order to fix them.

2.2 Docker

Docker is an open-source project to automate the deployment of applications into containers. It was released in March 2013 by dotCloud (later became Docker Inc.), a Platform-as-a-Service provider company. Over the years, it had gained public and media attention: by 2018 it had reached nearly 50.000 GitHub starts and over 14.000 forks. It has accepted a huge number of pull requests from companies like Red Hat, IBM, Google and Microsoft.
The reason behind its success is clear: it responded to a need for a high number of companies, thus building software in a flexible way and deploying it reliably and consistently in different environments.
The means to reach its goals is the use of containers, a small unit containing the software and all the needed dependencies and context to run it, regardless of the underlying environment. Even though the concept of container was not new when Docker was lunched, Docker has the credit for making a lightweight and fast workflow to produce and deploy containers as well as introducing some innovative features.

2.2.1 Container and virtualization technologies

Virtualization technologies allow multiple tenants to share the same physical machine with a high degree of isolation. This is a key requirement, especially in modern cloud-based infrastructures, where the same physical machine is shared in order to fully exploits the machine capabilities and lower the costs. Two categories of virtualization have been established so far: container-based and hypervisor-based solutions.
Containerization is a technology to virtualize applications in a lightweight way through the construction of containers: instances of images which holds the application itself and dependencies needed to run it. Over the years, it has been designed different container engines which allow this type of virtualization, but most of them share the same working principles. Under the hood, container engines like LXC [1], exploit the host Linux kernel mechanisms such as *namespaces* and *cgroups*. These

[1]https://linuxcontainers.org/

mechanisms were originally developed to enforce isolation between processes, then they have been leveraged to create this new type of lightweight virtualization. This close tied between containers and host is the essential aspect of this technology, which comes with both advantages and drawbacks. Using this Application-level virtualization allows each container to share the OS of the host machine, yielding to deployments that consume less resources compared with other virtualization solutions.
Hypervisor-based solutions are based on a component called hypervisor or Virtual Machine Monitor (VMM) which is in charge of managing the operations of a virtualized environment on top of a physical machine. The VMM spawns multiple virtual machines, which are isolated instances of the physical machine. The virtualization process can happen at different layers of the physical machine, according to the hypervisor capabilities and the needs of the machine owner. As opposed to container, virtual machines do not share the OS with the underlying machine. Moreover, the hypervisor provides an additional layer of isolation between the host and guest machine while adding a significant overhead on the virtualization. Generally, hypervisor-based solutions are considered more secure than container-based solutions because the hypervisor allows a higher degree of isolation between the host and the guest machine. On the other hand, container-based solutions are considered more effective due to the absence of overhead induced by the hypervisor.

2.2.2 Architecture

Docker adopts the client-server architecture illustrated in Fig. 2.1. Basically, Docker provides a set of commands that the client component can use to talks with the Docker host. The Docker host component can be on the same machine of the client or on a remote machine. The client component is in charge of directing the Docker host thought the respective commands sent by using a set of RESTful API.
The Docker host listens for API requests and, based on the commands received, instructs the Docker daemon. The Docker daemon is a fundamental component, which allows managing different objects such as images and container (further described in Section 2.3). Furthermore, the Docker daemon allows publishing the images in private and publish Docker registries.

2.3 Main components

In Section 2.2.2 are mentioned some components that have to be further described for the understanding of the possible threats.
In particular, we mention the Docker daemon as the component in charge of managing containers and image lifecycles. To be more precise, this component consists of different subcomponents which work together to reach the same goals while allowing the engine to be flexible and modular.
It is worth pointing out that the Docker architecture has been changing and improv-

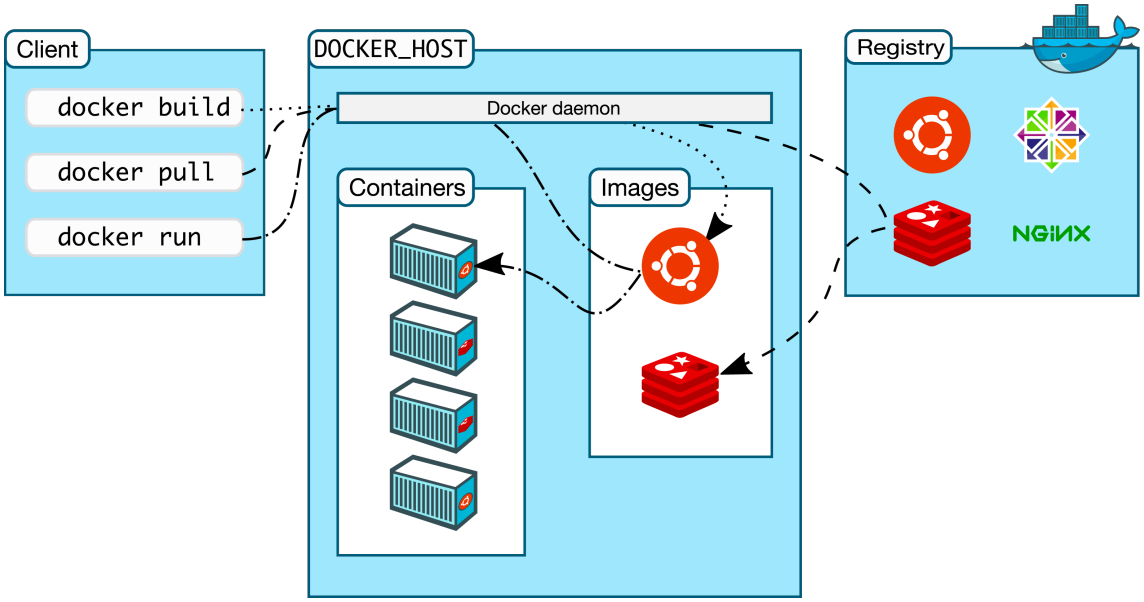

Figure 2.1: Docker architecture from Docker official website

ing over the years. In the first release, Docker engine had two major components: the Docker daemon and LXC [2]: the former was a monolithic library in charge of handling Docker client, Docker API, container runtime, images building process and more; the latter allows exploiting the mechanisms in the underlying kernel (such as namespace and cgroups) to build the containers.
The major problem with this architecture were two: first it was not flexible enough to allow changes, second, LXC is Linux-specific while Docker had aspirations of being multi-platform. Moreover, since the jobs of LXC was fundamental for Docker aims, relying on external tools was a huge risk. Over the years, the efforts of Docker Inc. has been aimed to break down the monolithic Docker daemon and to replace LXC technology.
Since Docker is a project in constant evolution, at the time of writing the major components are as described in the following sections [24].

2.3.1 Docker Engine

The Docker Engine has been built according to the Open Container Initiative (OCI) specifications [3]. The OCI is an open source community aiming to create open standards about container technologies. In particular, OCI defines two sets of specifications: Image specifications, which define a way to package applications in container images and Container runtime specifications, which define how containers work and how they run.
In order to implement these specifications, the Docker daemon lost some functional-

[2]https://linuxcontainers.org/
[3]https://opencontainers.org/

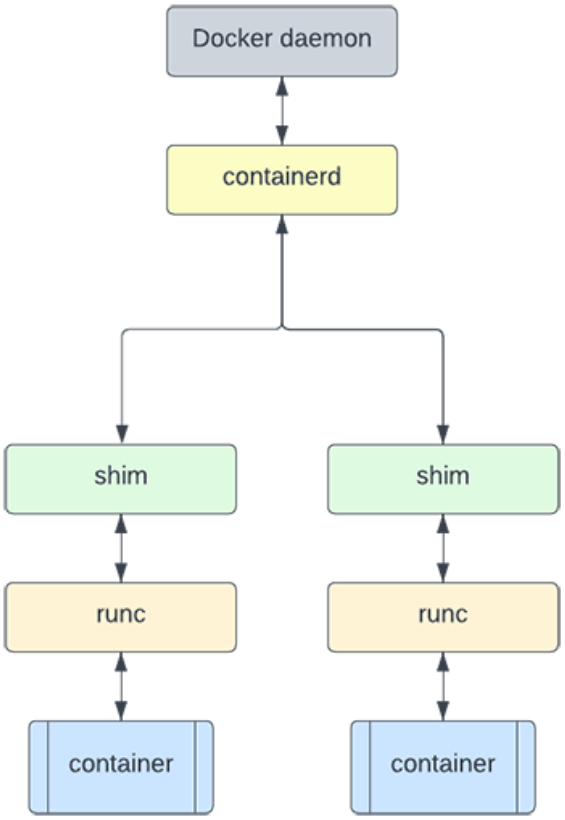

Figure 2.2: Docker Daemon architecture

ities that it had in the initial releases, and these functionalities have been assigned to new components. In particular, two new important components have been implemented: *runc* [4] and *containerd* [5]. *runc* has the job to create and run containers interacting with existing low-level Linux features, instead, *containerd* manages the container lifecycle operations.
Finally, another important component is *shim* which stands between *containerd* and *runc*. It is in charge of decoupling the running containers from the daemon (allowing for example to upgrade the daemon without disrupting running containers). The outlined modules are organized as the in the fig Fig. 2.2: *containerd* uses *runc* to create new containers by forking new instances of *runc*. Once a container is created, *runc* processes exit and *shim* processes becomes the container's parent. As container's parent, *shim* keep stdin and stdout streams open and reports the container's exit status back to the daemon.
The Docker daemon is still in charge of different tasks such as image management, image building, REST API, authentication, networking and orchestration[24].

[4]https://github.com/opencontainers/runc
[5]https://containerd.io

2.3.2 Docker Image

A Docker image is the starting point to create containers. In fact, the concept of image and container are strictly tied: it is possible to consider a container as a running instance of an image. This means that the image a container is created from, contains the operating system and the application file to run the required application or service.
Despite the number of file images should potentially contain, they are usually lightweight and fast. This is because of how they are built: as we mention, images do not contain the kernel, which is shared with the underlying OS. Specifically, an image is a set of read-only layers stacked on top of each other: every layer contains some file of the filesystem and, altogether, they compose the final and complete filesystem. In this sense, Docker daemon employs a storage driver which is in charge of presenting all the layers as a single unified filesystem.
The major difference between an image and a container is that the container adds a new top writable layer to store ephemeral data produced during the runtime. Once the container is deleted, this top layer is dismissed as well. This way to organize an image in layers is beneficial to reduce the container size on disk: since the only difference among containers sharing the same image is the top writable layers, different containers can share the same underling layers without duplicate them every time a new container is spawned. Then, when a container attempts to write a file in one of the underlying layers, that file is copied to the top writable layer and modified. This is known as copy-on-write (CoW) strategy [8].
Finally, it is worth mentioning that Docker images are stored in the so-called image registries. The most common registry is Docker Hub [6]. To be pointed out that other third-party registries with different features exists. These registries allow retrieving (this action is known as pulling) new images or to store new images (this action, instead, is known as pushing)[24].

2.3.3 Dockerfile

Dockerfile are text documents which contain instructions about how to assemble a Docker image. Each instruction is contained in one line and they are executed one after the other to create the final image. As we mention in Section 2.3.2, every new image is built by adding new layers on top of a previous image; in the same way, a Dockerfile starts with an instruction (FROM) which specifies the starting image, also called *base image*.
Having a valid Dockerfile it is not the only prerequisite to allows Docker to build an image: in fact, Docker need also a build context. A build context is a set of file at a specific directory to be added into the image to properly make it works. The build context is sent to the daemon and it is possible to refer to it in the Dockerfile through particular commands (such as ADD and COPY).

[6] https://hub.docker.com/

It is important to highlight that the way a Dockerfile is written has a huge impact both on the size and on the security of the image. We discuss security aspects in Section 2.4.
Regarding the size, during the Dockerfile writing, it should be considered that every instruction add a new layer in the image; then some of these instructions add some layer that actually increase the image size (ADD, COPY, FROM). Because of this reason, some best practices should be put in place in order to avoid creating images with huge size. One of the most famous is the builder pattern: briefly, it consists of producing two Dockerfile, one for the development and one for the production. The former could have a greater size, since it is in charge of setting up the development environment and producing the final artifact. The latter contains only the final artifact and the needed dependencies to make it works, but it does not contain developing tools. Obviously, the latter is the one that will be used in production or shared in a registry. This pattern is such important that Docker introduces new instructions to implement it without the effort of writing two distinct Dockerfile and this practice is known in Docker ecosystem as multi-stage build [24][7] .

2.4 Security risks

Docker has transformed the way in which containers are created and deployed, leading to a fast and huge adoption from several companies. Because of the wide use of this technology, it is mandatory to put in place mechanisms to ensure the security of the containers itself and of the host machine as well.
Security risks derived from the adoption of Docker containers might come from different layers, thus different kind of hardening processes should be performed. In this section, we report some security best practices in the use of Docker, following the recommendations reported by the Center for Internet Security (CIS) Docker Benchmark report v.1.3.1 ([4]). Since this work is focused on security aspects regarding Dockerfile, we discuss them apart in Section 2.5.

Host configuration According to CIS Docker Benchmark, one of the most important act before running Docker in production environment is to properly set up and harden the host machine. Their recommendations aim to logically separate containers and host space, while consenting only the authorized information to pass across the two partitions. Moreover, auditing processes which involve the different Docker components (containerd, shim, runc, etc.) should be in place.

Docker daemon configuration Having a vulnerable version or setup for the Docker daemon could lead to an easy exploit both of the running containers and the daemon itself. This is a fundamental aspect because, Docker daemon, by default, runs with *root* privileges, since it requires access to the Docker socket which is usually owned by the user *root*. Nevertheless, the same use cases allow running the Docker daemon in rootless mode. For this reason, it is worth analyzing the specific

case and, according to the "principle of least privileges" [28], run the daemon in rootless mode if it is possible, in order to mitigate potential vulnerabilities in the daemon and container runtime.
On the same page, overcoming the default behavior to allows all network traffic among different containers is needed in order to avoid unwanted disclosure of information to other containers. Still regarding networking mechanisms, since the Docker daemon can be available remotely over a TCP port, it is required to configure TLS authentication in order to restrict access to it.
When the container runs without a pre-defined non-root user, it is recommended to enable user namespace support in order to have a unique range of user and group IDS which are outside the traditional user and group range used by the host. It is important especially for the root user: in this way, the root user can have the expected administrative privileges inside the container but can effectively be mapped to an unprivileged UID on the host system. This means that if an attacker can break the container, he will not be a root user in the host system too. Unfortunately also this configuration, is not always possible since it is incompatible with a certain number of Docker features.

Container runtime Configuration There are some security implications in parameters the containers are started with. In fact, some runtime parameters can be exploited to compromise the host and the containers running on it. Thus, it is very important to verify which parameters are associated with the containers.
Using third-parties Linux application security system, such as AppArmor and SELinux to enforce security policies and augments the default Discretionary Access Control (DAC) model, is a great way to protect host and containers. In also important to avoid some common pitfalls in the container configurations: for instance, it is a common practice to use unnecessary privileged containers, which lifts some limitations enforced by the daemon. This leads to containers which have most of the rights of the underlying host. Another common misconfigurations is sharing sensitive directories of host (such as */etc* and */usr*) with the container: not only this misconfiguration could lead to data leaks but, if they are mounted in read-write mode, it could be also possible to make changes to files within them. It is also very common to run the SSH daemon within the container: this increases the complexity of security management by making difficult keys managements.

2.5 Best security practices in Dockerfile

After defining the security best practices involving different Docker components, we specifically focus on security best practices to be adopted during Dockerfile writing. Both the official Docker documentation [7] and CIS Docker Benchmarks, offer a useful list of Dockerfile best practices to follow with the purpose of preserving functionality

[7] https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

and safety.
In this section, we go through their recommendations, reporting those that concern the security of the host and the container.

Running Docker container in rootless mode

```
1 FROM base-image
2 RUN apt-get update
```

Listing 2.1: Misconfigured script for "Running Docker container in rootless mode"

```
1 FROM base-image
2 RUN apt-get update
3 USER baseusr
```

Listing 2.2: Correct script for "Running Docker container in rootless mode"

When we build Dockerfile, unless it is explicitly stated, the default user inside the container is the root user. Running a container with root user when the user does not need such privileges is a clear violation of the "least privilege principle".
The security risk is that the root user can do whatever it wants inside the container and, moreover, it ease operations of container breakout.

Using tagged minimal base images and multistage builds

```
1 FROM node:8 as build
2 WORKDIR /app
3 COPY package.json index.js ./
4 RUN npm install
5 EXPOSE 3000
6 CMD ["index.js"]
```

Listing 2.3: Misconfigured script for "Using tagged minimal base images and multistage builds"

```
1 FROM node:8 as build
2 WORKDIR /app
3 COPY package.json index.js ./
4 RUN npm install
5 FROM gcr.io/distroless/nodejs
6 COPY --from=build /app /
7 EXPOSE 3000
8 CMD ["index.js"]
```

Listing 2.4: Correct script for "Using tagged minimal base images and multistage builds"

The reason behind it is that the image size could be considered a proxy for attack surface measurement: the more packages or services are included into the image, the higher the probability that some of those are vulnerable, thus the higher the attack surface.
The best practice addresses this issue by using a tagged minimal base image, thus images containing only the bare minimum in terms of installed packages and services.
Another strategy could be using multi-stage builds, as showed in Listing 2.4. Multi-stage building consists of using multiple FROM instructions; each FROM instruction begins a new stage. It is possible to copy files from stage to another in order to leave only the necessary file in the final stage.
In Listing 2.4, after building the application in the first building stage (lines 1-4), that building stage is dismissed and the resulting image is composed only by the artifacts copied in the line 6.
Moreover, it is worth pointing out, that the final stage uses a *distroless* image[8] (line 5), thus an image that contains the application and its dependencies without any other features of a regular distribution. It is worth noting that, in any case, the base image should be tagged with an *immutable tag*. A common use case is to use the "latest" tag to pull the latest version of an image. This function is useful in some scenarios but it might constitute a security risk. Every time we use a mutable tag as "latest", we are performing a non-deterministic action, since that tag can point to different images in different times. It is preferable to refer to a base image with an immutable tag, as a specific image version.

Using COPY command with specific parameters

```
1 COPY . .
```

Listing 2.5: Misconfigured script for "Using COPY command with specific parameters"

```
1 COPY target/app.jar /app
```

Listing 2.6: Correct script for "Using COPY command with specific parameters"

This is a simple best practice consisting on using most specific parameters when it is needed to copy some file or directories inside the container filesystem.
It should be done because the COPY instructions is recursive inside the building context and this behavior may have two possible consequences:

- Copying unnecessary data and consequently increasing attack surface
- Accidentally copying sensitive data

Another solution is to use a *.dockerignore* file to exclude some file from the building process.

[8] https://github.com/GoogleContainerTools/distroless

Update and install packages in the same RUN instruction

```
1 FROM ubuntu:18.04
2 RUN apt-get update
3 RUN apt-get install -y \
4         package-bar \
5         package-baz \
6         package-foo=1.3.*
```

Listing 2.7: Misconfigured script for "Update and install packages in the same RUN instruction"

```
1 FROM ubuntu:18.04
2 RUN apt-get update && apt-get install -y \
3         package-bar \
4         package-baz \
5         package-foo=1.3.*
```

Listing 2.8: Correct script for "Update and install packages in the same RUN instruction"

Having two distinct RUN instructions for these commands can cause troubles with the cache: if new packages are added and the Dockerfile is rebuilt, Docker will use the packages cached in the layer generated with the instruction in the line 2. This may lead to outdated package installations.
For the same reason, version pinning for packages is a good practice as well.

Using COPY instead of ADD

The security risk behind this practice comes from the capabilities of the ADD: this instruction implements some more features compared to the COPY instruction. In particular, ADD supports downloading file from remote URLs with the associated risk of downloading a malicious file. Moreover, ADD supports auto-extracting features of some archive formats [9] and the user could accidentally open a zip-bomb.
Another very related best practice is to prefer using *curl* or *wget* shell commands to download the file because it is possible to add the *rm* shell command in the same instruction, avoiding increasing the layer size.

Don't leak sensitive info to docker images

```
1 FROM ubuntu:vivid
2 RUN apt-get update && apt-get -y upgrade
3 RUN apt-get -y install nginx && apt-get clean
4 RUN echo "daemon off;" >> /etc/nginx/nginx.conf
```

[9] https://docs.docker.com/engine/reference/builder/

```
5  ENV mysecret=secret
6  COPY file:0ed4cccc887ba42ffa23befc786998 in \
7  /etc/nginx/conf.d/example.conf
8  COPY file:15688098b8085accfb96ea9516b9 in \
9  /etc/nginx/ssl/nginx.key
10 COPY file:bacae426a125600d9e23e80ad12e17b5 in \
11 /etc/nginx/ssl/nginx.crt
12 CMD ["nginx"]
```

Listing 2.9: Misconfigured script for "Don't leak sensitive info to docker images"

```
1 FROM ubuntu:vivid
2 RUN apt-get update && apt-get -y upgrade
3 RUN apt-get -y install nginx && apt-get clean
4 RUN echo "daemon off;" >> /etc/nginx/nginx.conf
5 CMD ["nginx"]
```

Listing 2.10: Correct script for "Don't leak sensitive info to docker images"

Dockerfile's instructions may be used to store or move sensitive information, for instance through the ENV or COPY instructions, as in the example. Storing sensitive information or hard-coding them is a bad practice, especially if the container have to be deployed in a public register. Moreover, sensitive information may be cached in intermediate layers even if you deleted them on other layers. If sensitive information have to be shared with the container, the simplest and more effective way to do it is to share them leveraging a volume.

Removing unnecessary dependencies

```
1 FROM debian
2 RUN apt-get update \
3         && apt-get -y install \
4         Openjdk-8-jdk  ssh vim
```

Listing 2.11: Misconfigured script for "Removing unnecessary dependencies"

```
1 FROM debian
2 RUN apt-get update \
3         && apt-get -y install --no-install-recommends \
4         Openjdk-8-jdk  ssh vim
```

Listing 2.12: Correct script for "Removing unnecessary dependencies"

This practice refers to using the *–no-install-recommends option* (or related options) during package installation. This is a simple practice that, as highlighted in 2.5, aims to reduce the image size and consequently to reduce the attack surface.

Expose only the needed ports

It is a best practice to expose only the ports that the application running inside the containers needs. Moreover, it is considered a particular wrong practice to directly publish the port 22 on the container. Note that the EXPOSE instruction does not actually expose the port (it happens at runtime), but only indicates the ports on which a container listens for connections to the person who runs the container.

Using official image when possible

Using unofficial images increase the risks of employing a malicious image. It is also a good practice because official images usually adopt best practices, which consequently lowers the risk of having vulnerabilities.

Image should be pulled from the organization's private registry

If it is necessary to use custom images, it is better that such images have been pulled from a private and secure registry. Since anyone can publish images on public repositories, they could be malicious or vulnerable.

Chapter 3

State of the Art

In this chapter, we discuss detecting and repairing technologies applied to misconfigurations, with particular attention to those commonly found in Dockerfile scripts. Since Docker is under the umbrella of Infrastructure-as-Code (IaC) technologies, in Section 3.1, we firstly introduce the concept of security smells which describe antipatterns in IaC scripts that may inject some security risks. In Section 3.2 we analyze the techniques used by similar works for Dockerfile antipatterns detection and eventually rule learning. In particular, in Section 2.5, we list the main misconfigurations that may be present in Dockerfile scripts and, for each of them, discuss the security risks involved, and in Section 3.2.2 we list and compare existing commercial tools for configuration error detection. In Section 3.3 we discuss Automatic Program Repair which are a set of tools and techniques to repair errors in source codes.

3.1 Security smells in IaC scripts

According to [27], a code smell is a piece of code which does not respect well-known coding patterns and could yield to maintenance problems. Even if we can not assume with certainty that a code smell will cause problems, it still deserves attention.
In particular [27], focus on a specific class of code smell, named security smell. As for code smells, *a security smell is a piece of code which does not respect established coding patterns, risking the infrastructure security.* The concept of security smell could be very similar to the best known concept of vulnerability, but it is worth pointing out the differences among the two terms. Security smell is different from vulnerability because the latter is defined as a weakness that can be exploited instead the former don't describe how exploits can be enacted [25].
Authors conduct a study about security smell identification in IaC scripts, targeting the currently most famous IaC technologies: Ansible, Chef and Puppet. After evaluating 3.339 scripts, the report describes several recurring security weaknesses, which are related to the Common Weakness Enumeration (CWE)[23].
The security weaknesses are:

Administrator by default It refers to the pattern of specifying default users as administrative users. It is an antipattern because it could violate the principle of least privilege ([28]). The smell is related with "CWE-250: Execution with Unnecessary Privileges" [1].

Empty password It refers to the pattern of setting as password, a string of length zero. Having a length zero password does not always lead to security breach, but it makes it easier to guess the password itself. This smell is related to the "CWE-258: Empty Password"[2].

Hard-coded secret It is the practice to code secrets, such as usernames and passwords, directly in the scripts. Related weaknesses from CWE are: "CWE-259: Use of Hard-coded Password" and "CWE-798: Use of Hard-coded Credentials"[3] [4].

Unrestricted Internet Protocol IP It is the pattern of assigning the address 0.0.0.0 to a service. It can cause security concerns, because this particular address allows connections from every network. The smell is related to "CWE-284: Improper Access Control"[5].

Suspicious comment It refers to the practice of writing important information related to the scripts in the comments. It is the case of comments which reveal bugs that need to be fixed. The smell is related to "CWE-546: Suspicious Comment"[6].

Use of HTTP without TLS It is the pattern of using HTTP protocol without the Transport Layer Security (TLS) protocol. Unless other mechanisms are in act, information will go through the network without encryption. It may be an issue for privacy and data integrity. It is related to "CWE-319: Cleartext Transmission of Sensitive Information"[7].

Use of weak cryptography algorithms This smell refers to the practice of using weak algorithms, such as MD4 and SHA-1, for encryption purposes. It is related to "CWE-327: Use of a Broken or Risky Cryptographic Algorithm"[8] and "CWE-326: Inadequate Encryption Strength"[9].

No integrity checks It is related to the pattern of downloading contents without checking it via checksum and GNU Privacy Guard (GPG). It is a security

[1] https://cwe.mitre.org/data/definitions/250.html
[2] https://cwe.mitre.org/data/definitions/258.html
[3] https://cwe.mitre.org/data/definitions/259.html
[4] https://cwe.mitre.org/data/definitions/798.html
[5] https://cwe.mitre.org/data/definitions/284.html
[6] https://cwe.mitre.org/data/definitions/546.html
[7] https://cwe.mitre.org/data/definitions/319.html
[8] https://cwe.mitre.org/data/definitions/327.html
[9] https://cwe.mitre.org/data/definitions/326.html

smell because the developer can not be sure if the downloaded content has been tampered by an attacker. It is associated with "CWE-353: Missing Support for Integrity Check"[10].

Missing default in case statement It is the practice of not handling all input combination in a case-conditional structure. It is risky because an attacker can guess an unhandled input, trigger an error and possibly having access to system information in terms of stack traces and faults. It is associated with "CWE-478: Missing Default Case in Switch Statement"[11].

In order to derive information about the distribution and the frequency of these security smells, it is crucial to identify such smells from the collected scripts. For that purpose, the authors build a rule-based detection tool named Security Linter for Infrastructure as Code (SLIC)[25]. After manually writing the rules, the authors make the following remarks:

- Approximately 17.9 - 32.9% of the IaC scripts in the dataset include at least one security smell category.
- The most frequent security smell category is hard-coded secret: 6.8 - 24.8% of the collected scripts include at least one hard-coded secret.
- For every 1.000 IaC Line of Code (LoC), security smells appear in 13.3 - 31.5 LoC.

In addition, in a similar previous work [25], the authors state that a security smell can persist in a script for as long as 98 months.
As the author suggests, numerous studies try to characterize insecure IaC scripts in order to figure out if there are some correlations that could be used to prioritize inspection efforts. A representative example is [26], in which the authors try to characterize defective IaC scripts by means of Natural Language Processing (NLP) techniques.
After preprocessing IaC scripts in the form of text tokens, the authors apply Bag-of-Words (BAG) model and Term Frequency-Inverse Document Frequency (TFDF) techniques to build features vectors, and then, they use Principal Component Analysis (PCA) to select the most influencing tokens.
After this features selection phase, they have some text features which are correlated with defective IaC scripts, but since these features are tokens (i.e. string with one or few words), they might be pointless on their own. The authors address that problem by applying a qualitative analysis called Strauss-Corbin Grounded Theory (SGT) on the identified tokens. By means of SGT, it is possible to derive properties of defective IaC scripts from the identified text features.
The authors uncover three properties that characterize defective IaC scripts and that could be very useful to prioritize verification and validation efforts:

[10] https://cwe.mitre.org/data/definitions/353.html
[11] https://cwe.mitre.org/data/definitions/478.html

Filesystem operations are related with performing input and output tasks, such as setting permissions of files and directories.

Infrastructure provisioning mostly concerns setting up and managing infrastructures for special systems such as data analysis and database systems. Results indicate that the capability of provisioning via IaC tools can introduce defects.

Managing user accounts concerns about setting up accounts and user credentials.

As example, [26] reports the case of filesystem operations as the operations more susceptible to defects (23% in one of the dataset). In order to perform this operation, experts have to set up values, as file locations and permissions, but doing this, they may inadvertently make mistakes.
In addition to define properties of defective IaC scripts, [26] builds a machine learning model by means of Random Forest technique and text-features previously extracted, in order to predict if a given IaC script is defective.
A similar work has been proposed by [6] which analyze 104 Ansible projects and leverage different machine learning techniques to predict defective IaC scripts.
It uses 108 features to train the model which cover different aspects of the developing process instead of just focusing on the scripts. These metrics have been classified into three categories:

- **IaC-oriented (ICO) metrics**: structural metrics derived from the analysis of the IaC source code. They belong to previous works which focuses on IaC scripts and more traditional code metrics that can be adapted to IaC scripts.
- **Delta metrics**: 46 metrics which capture the amount of changes in a file between two successive releases
- **Process metrics**: 16 measures that consider aspects concerning the development process rather the code (e.g. number of developers that change a file).

The work points out, after experimenting with different machine learning algorithms and different combination of metrics, that Random Forest is the best predictor, and that the best performance are reached with the ICO metrics alone.

3.2 Dockerfile misconfigurations

Being part of Infrastructure-as-Code tools, Dockerfile do not come without risks, and they may be affected by the presence of security smells and misconfigurations as well [32]. This problem has worsened by the opportunity to embed Bash code.
In recent times, different studies emphasize the need to write Dockerfile following

the Docker official documentation's best practices [12] in order to not risk containers and hosts security.
In [34], the authors study the differences among Dockerfile evolutionary trajectories basing on different variables that affect both quality and build latency time. A Dockerfile evolution is the changing of the Dockerfile according to the project requirements.
This work can be considered as an attempt to characterize a Dockerfile according to different metrics, as has been done with other IaC technologies in [26].
From the previous work [33], the authors cluster six different categories of evolutionary trajectories:

Increasing and holding Developers added new instructions to the Dockerfile in the project's early period, but after reaching a certain size, they just update the environment variables or change instruction locations.

Constantly growing Developers keep adding to the Dockerfile, causing its size to continually increase.

Holding and increasing Dockerfile was stable in the early periods, but then it increases by adding new instructions.

Increasing and decreasing Developers kept adding until it reach a large size, then they remove useless plugin/services or move settings to additional script files.

Holding and decreasing Developers changed Dockerfile during the project's early period, and then they try to reduce its size.

Gradually reducing Developers keep removing useless instructions or change the base image to continually reduce image layers.

In the Table 3.1, it is possible to find the percentages of projects belonging to the six categories and the identifications (ID) of each cluster.
Authors compare the clusters assuming C-2, which is the one with the most occurrences, as baseline. C-1 and C-6 seem to help reduce quality issues, instead C-4 has more quality issues. The reasonable justification is because of the pre-accumulation quality issues in the earlier period.
In particular, the subsequent study consists on analyzing 2.804 projects which contains 76.925 versions of Dockerfile by means of 2 linear regression models: the quality issue model and the build latency model.
The research demonstrates that the most positive factor influencing quality issues is the number of layers in the image: more image layers could bring more quality issues. A quality issue in this sense may be the one reported by [20].
A counterintuitive result is the following: quality issues may decrease if image layer

[12] https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

size increases. The authors explain this result, stating that large layer size may mean that instructions have detailed and complete parameters.
Another finding is that additional scripts may reduce quality issues: it can be easily explained because additional scripts reduce the burden on the original Dockerfile. In this case, it is important to not relocate possible issues to the additional scripts. On the other hand, a greater number of different instructions tend to have less quality issues since it assumes that the developers used the right Dockerfile instructions for the intended purposes.

ID	Name	% of projects
C-1	Increasing and holding	28
C-2	Constantly growing	31.6
C-3	Holding and increasing	19.2
C-4	Increasing and decreasing	10.2
C-5	Holding and decreasing	9.5
C-6	Gradually reducing	7.7

Table 3.1: The six different clusters of [34]

Another relevant study is Wu et al. [32] which focuses on a particular class of quality issues related to Dockerfile named Dockerfile smells.
Formally, a Docker smell is defined as an instruction of a Dockerfile that violates the recommended best practices and potentially affect a project's quality negatively.
In [32], authors conduct an extensive study on 6.000 GitHub open-source projects containing Dockerfiles and belonging to the Top-10 popular programming languages. The authors identify two different smell categories:

- **DL-smell**: refers to the issues against the Dockerfile best practices.
- **SC-smell**: refers to the quality issues against the Bash scripts practices.

Then, according to the presence or not of smells, a project can be classified as Smell Project (S-Project) or a Health Project (H-Project).
The study collects and processes 6.000 projects and then analyzes them by means of the analyzer named *Haskell Dockerfile Linter*[13] (further explained in Section 3.2.2). *Haskell Dockerfile Linter* parses the Dockerfile and try to match its rules against the scripts in order to report smells (DL-Smell). In turn, *Dockerfile Linter* stands on the shoulders of *ShellCheck*, a tool to analyze and detect bash best practice violations (SC-Smell).
The quantitative analyses reveal alarming findings: 83.8% of the projects have at least one smell in Dockerfile code, 61.5% have DL-smells, 0.5% have SC-smells and

[13]https://github.com/hadolint/hadolint

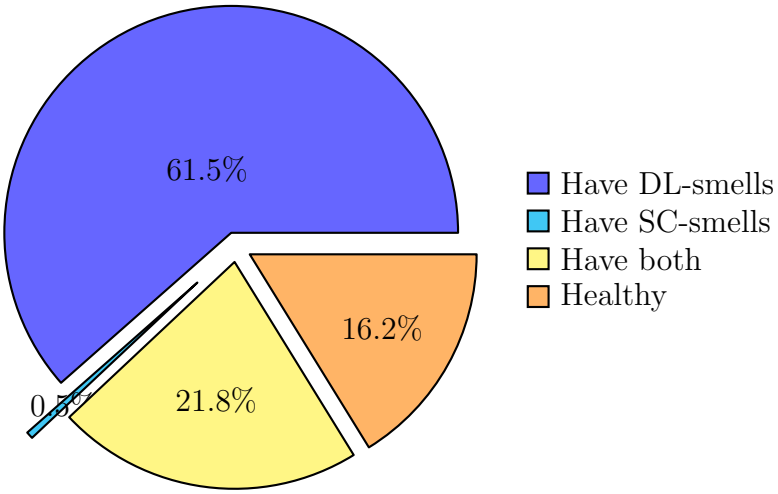

Figure 3.1: Percentage of different smells classes according to [32]

21.8% have both. This means that S-projects have an average 5.6 smells in their Dockerfiles.
The study reveals also that DL-smells and SC-smells are correlated; in fact a project that show presence of large number of DL-smells (or SL-smells), moderately indicates the presence of large number of SC-smells (or DL-smells). Moreover, as expected, popular and young projects tend to have fewer Dockerfile smells.
A qualitative analysis, conducted randomly selecting 60 S-projects, shows also some interesting examples of incorrect practices in relation with the languages used. For example, Ruby projects tend to use ADD instruction, especially when it does not require its tar auto-extraction capabilities, instead Shell projects tend to use *apt-get update* and *apt-get install* in the RUN instruction which is prone to go against the best practices.

In [20], authors focus on a particular class of smells called Temporary File Smell (TF smell). A TF smell occurs when the developer delete a file from the image outmost layer, but that file had been added in previous layers. In this case, the file will be deleted from the outmost layer but not from the previous ones, yielding a pointless image size increase.

```
1 FROM centos:7
2 COPY jdk-8u171-linux-x64.tar.gz
3 RUN tar zxvf jdk-8u171-linux-x64.tar.gz
4 RUN rm -f jdk-8u171-linux-x64.tar.gz
```

Listing 3.1: "TF smell example"

For instance, in the Listing 3.1, developers add a new layer with the deletion of *jdk-8u171-linux-x64.tar.gz* (line 4) and it seems to completely remove the file from the image. Conversely, it removes only the file from that layer, but it is still present

in the layer in which the file has been copied (line 2).
TF smells can occur through different patterns, such as using *wget* bash command inside a RUN instruction and the *rm* command in another RUN.
The main challenge is to be able to correctly identify all the TF smells. The authors develop two different approaches for this purpose: one based on dynamic analysis and one based on static analysis.

Dynamic analysis

The approach based on dynamic analysis consists of injecting logging code into the file system and rebuilding the kernel in order to trace the temporary file creation and deletion. Through an automated analysis of the resulting log, it is possible to effectively classify TF smells instances. The disadvantage of this approach is that it can generate numerous logs and significantly reduce system performance. According to authors experiment, a performance lost of 37.1% occurred.

Static analysis

The second approach is based on static analysis. The major challenge using static analysis is to correctly identify the path of the different file since they can be found in different instructions, with absolute path or path relative to the current working directory. Moreover, the massive use of wildcard characters makes it very difficult to correctly know if, after the creation, a file has been deleted in the following instructions.
After building the AST of the Dockerfile, the main idea is to use a State Table (ST) to store the variables that represents the current status of resources. In this way, the State Table is designed to dissolve ambiguities. In particular, ST stores the *CurrentDirectory* variable which is used to record the absolute directory path of the current working directory, and it is updated as the tool go through the AST. It also stores the type (file or directory) of the paths, and these variables are update during the AST examination as well. Finally, it stores the environment variables.
Thanks to the ST, during the analysis phase, a list of created file and a list of delete file are set up. On these lists, a file matching is carried out to discover if a file has never been deleted.

3.2.1 Detection and Repairing Techniques

The problem of security misconfigurations detection and repair concerning Dockerfiles can be split into two different sub-problems: the problem of detection and the problem of repairing.
At the best of our knowledge, Dockerfile misconfiguration repairing problem has not been addressed in literature as intended in the Chapter 1. In the same way, far

too little has been said about the problem of misconfigurations detection applied to Dockerfiles.
Nevertheless, in recent times, different commercial tools come out to help developers writing Dockerfile in accordance with best practices. Since these tools use the same rule-based approach, they are covered apart in Section 3.2.2.

Detection Techniques

Regarding misconfigurations detection, the work which is most similar to ours is [14], where the authors aim to build a tool to support developers during Dockerfile writing.
The tool, named *Binnacle*[14], consists of two main components: the rule miner and the rule enforcer. The rule miner is in charge of automatically detecting recurrent rules by analyzing hundreds of thousand of Dockerfiles. The rule enforcer, instead, is in charge of enforcing the previously discovered rules in a new Dockerfile.
Dockerfile are transformed into Abstract Syntax Tree (AST) by means of a phased parsing, which is very convenient when it comes to deal with languages that allow embedded scripting.
The phased parsing approach consists of building the AST through different phases; in each phase, the AST is progressively enriched.
Moreover, both components make use of an abstraction process which replace every literal value with either zero, one or an abstract node. Specifically, the abstract node type is assigned based on a regular expression matching. This abstraction process helps *Binnacle*, by giving the necessary vocabulary to reason about properties of interest.
The rules discovered from the rule miner and checked by the rule enforcer consist of an antecedent, which is a sub-tree that may be matched in the Dockerfile's AST, and a consequent, which is a sub-tree representing the code that should be appeared along with the antecedent. According to the position of the consequent with the respect of the antecedent, it is possible to have different classes of rules.
A strong limitation of this work is that the miner is only capable of returning a specific class of rules named Local Tree Association Rule (local TAR). In the Local TARs the consequent sub-tree exists within the antecedent sub-tree. This limitation is given by the inability to arbitrary find rules from a corpus of hundreds of thousands of trees. It is worth highlighting, that this limitation affects only the miner and not the enforcer.
Moreover, another limitation is given by the fact that not all the Bash commands can be parsed because the authors choose to support only the commonly used command-line tool (which cover over 80% of command-line tool invocations though).
The process of Local TAR mining among a dataset of AST-parsed Dockerfile is the following for a given node type of interest:

[14]https://github.com/jjhenkel/binnacle-icse2020

1. Identification of the set of all sub-trees rooted in a node of the given type.

2. On the identified sub-trees, employing a frequent sub-tree mining algorithm to recover a set of likely consequent.

3. Returning rules in which the antecedent is the root of a sub-tree (where the type matches the input node type) and the consequent is a sub-tree identified by the frequent sub-tree algorithm.

The process of enforcement is the following.
Given a Dockerfile and a set of TARs, for each rule:

1. The Dockerfile is parsed into AST and the enforcement engine attempts to match the TAR's antecedent. If no matches, it continues with the next rule.

2. Depending on scope and location, the enforcement engine chooses a region of the input tree where it looks for the consequent sub-tree.

3. The enforcement engine attempts to align the consequent of the TAR with a sub-tree, within the chosen region. If there are no matches, the rule has been violated.

Another contribution made by [14] regards the creation of 15 rules named Gold Rules. Gold Rules are rules manually extracted from the so-called Gold Dataset which is a dataset of 400 Dockerfile coming from the official *docker-library* repository and that presumably have a higher standard of quality. These Gold Rules are used as reference for *Binnacle* evaluation.

Repairing Techniques

The most significant work concerning Dockerfile repairing tools is [15]. Although it differs in the motivation that lead to the repair, it is worth analyzing this work because of the repairing process.
The authors develop a tool named *Shipwright* which is a human-in-the-loop system to automatically repair or suggest to the developers.
In this case, the repair action is needed not because of the presence of misconfigurations but due to broken Dockerfile, thus Dockerfiles that fail to be built.
Shipwright builds Dockerfile with their context file and retrieve build logs. On those build logs, a token splitting (which consist of split on snake and camel case and on several operators) and a string normalization are applied.
The preprocessed data are passed to *BERT*, a pre-trained transformer-based neural model which output the embedded vectors. The vector space composed by the vectors of all the build logs is clustered with HDBSCAN.
Then a human expert obtains, for each cluster, a likely root cause of failure and if possible, offer a repair or a suggestion.

The process of finding a repair is a step taken by a human expert with the assistance of *Shipwright* which builds a search string from log's keywords and reports a list of top-5 URLs as output for human inspection.
Focusing on the fixes, they consist of a pattern and a repair. The pattern is a regular expression that matches a string either in the error logs or in the Dockerfile. The repair is a transformative function which describes how to fix the Dockerfile.
Even if there are no significant details about the implementation of these transformative functions, authors specify the functions have been manually written because it is not a time-consuming error-prone task.
The clustering approach can be helpful because it gives the developers the chance to generalize a repair, thus to repair Dockerfiles which have similar causes of failure with the same repair pattern. Due to this approach, 13 repairs has been created from the dataset.
Specifically, *Shipwright* has been able to cluster 34% of the 5.405 broken Dockerfile in 144 clusters and for 36.5% of these clusters the author confirms that they consist of Dockerfile that have the same root cause. Of the 34% clustered Dockerfile, Shipwright was able to find a repair to 20.84% and suggestions to an additional 69.63%. For non-clustered Dockerfile, *Shipwright* is able to produce automated repairs in 18.18% of the file and to provide suggestions in 46.63% of the file.

3.2.2 Rule-based detection tools

This section is focused on a particular class of existing tools which implements a static rule-based approach for Dockerfile misconfigurations.
We have chosen the following tools as representative of this class for the variety of languages used, scope of the tool, number of policies and embedding code support.

Haskell Docker Linter [15] This tool has been already used in [34] to find weaknesses in the code. It is an open-source project written in Haskell which parses the Dockerfile into AST and check human-written rules on top of it. Moreover, it leverages ShellCheck to lint Bash code.

Checkov [16] It is a static code analysis for different IaC tools written in Python and developed by BridgeCrew [17]. It counts over 1000 built-in policies but having a wide scope on the different IaC technologies, only 8 of those regard Dockerfile and none of these regards Bash code.

Trivy [18] is a scanner for vulnerabilities in different IaC files. It has been written in Go and developed by Aquasecurity [19]. Currently, it supports 50 different rules regarding Dockerfile and Bash.

[15] https://github.com/hadolint/hadolint
[16] https://github.com/bridgecrewio/checkov
[17] https://bridgecrew.io
[18] https://aquasecurity.github.io/trivy/v0.22.0/
[19] https://www.aquasec.com

Kics [20] is a static tool analyzer developed by Checkmarx [21] to find security vulnerabilities and compliance issues in different IaC scripts. Currently, it supports more than 50 policies for Dockerfile, and it also includes Bash code analyses.

Docker-bench-security [22] is a shell script which attempts to automatize Dockerfile best-practice of the CIS Docker Benchmark v1.3.1[23]. It does not support embedded Bash code.

We test the capabilities of these tools against an ad-hoc written Dockerfile which contains all the misconfigurations listed in 2.5

```
1 FROM pmietlicki/apache-php5.6.30
2 RUN apt-get update
3 RUN apt-get --allow-unauthenticated install -y nano
4 ENV mysecret=secret
5 COPY . .
6 COPY important_key /var/www/html
7 ADD Context_test/test.tgz /var/www/html
8 ADD https://tmpfiles.org/dl/170022/test.tgz /var/www
9 EXPOSE 8000 80 443 22
```

Listing 3.2: Misconfigured Dockerfile

We define the associations among misconfiguration and its code location in Table 3.3.

Table 3.2: Comparative table of rule based tools

	Haskell Docker Linter	Checkov	Trivy	Kics	Docker-Benchmark-sec
Container in rootless mode	With an explicit USER instruction				
Use minimal base images					
Use multistage building					

Continued on next page

[20] https://github.com/Checkmarx/kics
[21] https://checkmarx.com/
[22] https://github.com/docker/docker-bench-security
[23] https://www.cisecurity.org/benchmark/docker/

Table 3.2: Comparative table of rule based tools (Continued)

	Haskell Docker Linter	Checkov	Trivy	Kics	Docker-Benchmark-sec
Restrict COPY command scope					
Update and install in the same RUN					
Use COPY instead of ADD					
Avoid COPY sensitive information					
Avoid ENV with sensitive information					
Use official images					
Image pull from organization private repository					
Install only verified packages					
Not Use ADD to download files					
Tag image version		Only 'latest' tag	Only 'latest' tag		

Misconfiguration	Line of Code
Root user in container	No line with USER instruction
No minimal image or multistage building	No multistage procedure
COPY without defined parameters	5
Apt-get update and install in the different run	2-3
Using ADD instead of COPY	6
Sensitive information leak	4
Install Unnecessary packages	3
No official image	1
Image not pulled from private registry	1

Table 3.3: Association misconfiguration and Listing 3.2's line of code

As it possible to notice from Table 3.2, almost all the tools can detect the same misconfigurations such as "Container in rootless mode" or"Apt-get update and install in the same RUN".
On the other hand, there are some misconfigurations, such as "Avoid COPY sensitive information" and "Not Use ADD to download file", which no tool can detect.
We formulate the hypothesis that some misconfigurations can be hard to detect due to the difficulty in coding a static rule to detect them. Taking as example "AVOID ENV with sensitive information", it requires to correctly and statically determine if the arguments of ENV instruction are private information.

3.3 Automatic Program Repair

In the following, we present the state of the art regarding automatic program repair (APR). As already said, none of these tools are able to deal with Dockerfiles or IaC scripts, but we believe that it is useful to discuss them in order to correctly be able to classify and compare our tool. Moreover, some approaches are still valid regardless of the specific implementation.
We distinguish among works which leverage machine learning (ML) techniques and works which adopt a static approach.

3.3.1 Static template based

Even if it has been developed for Java projects, [22] proposes *SpongeBugs*, which is the most similar tool with the respect of the approach used and the classes of errors addressed. In fact, it targets code smells, thus patterns that may indicate a poor programming practice as opposed to the rest of tools which focus mainly on behavioral bugs.
SpongeBugs fixes 11 distinct rules checked by *SonarQube* and *Spotbugs*, two static code analysis tools for Java projects. The 11 rules have been chosen among the

most 400 most violated rules in the two static analyzers, with the criteria of being possible to define a syntactic fix templates that are guaranteed to remove the source of warnings without changing the behaviors.
SpongeBugs is implemented in Rascal[24], a domain-specific language for source code analysis and the repairs are performed in the following three steps:

1. Fast and imprecise textual search based on conditions needed for rule triggering
2. Full and more expensive AST matching to the results of step 1
3. Execution and generation of a patch to fix the rule violation

It is worth highlighting the two phased approach for rule matching, which try to overcome the trade-off between computation effort and accuracy. Moreover, *SpongeBugs* does not use any type of learning in order to make the process fast and precise. The disadvantage of this approach is that it is not flexible, since it can not automatically learn new repairing templates.
Authors identify the following limitations to this work:

1. **Local Analysis**: the analysis scope is strictly limited to each method
2. **Contextual restrictions**: some rules are analyzed with additional restrictions on their context of applicability. For example, it does not check rules inside lambda expressions
3. **Typing information**: *SpongeBugs* analysis of typing information is incomplete
4. **Toolchain**: other limitations derived from the tools used to build *SpongeBugs*, in particular Rascal's Java 8 grammar, which is not complete

These limitations have as consequence a high rate of false negative during the evaluation: 15% of rule violations reported by *SonarQube* are not detected and hence not fixed by *SpongeBugs*. This result was predictable, since it is a consequence of design decisions in order to trade off detection capability for precision during the correction, which is about 85%.

Always leveraging *SonarQube* as Java code static analyzer, [10] builds *Sorald*.
As *SpongeBugs*, *Sorald* does not use any learning, but instead it uses human-written templates which transform a buggy AST into a fixed one.
Despite using the same tool, *Sorald* approach is different from *SpongeBugs*. *Sorald* employs *SonarQube* to detect rule violations into source: it means that the buggy line detection is performed by an external tool instead of using the two phase parsing of *SpongeBugs*.

[24]https://www.rascal-mpl.org/

Moreover, the transformation in *Sorald* is conducted on the AST and not on the code. As consequence of these two difference, *Sorald* developers do not write the detection rules, but they leverage those of the static analyzer and secondly, *Sorald* developers implement a process to transform the fixed AST into fixed source code preserving the formatting.
In order to recover the source code from the AST, at the very beginning, they separate the documentary structure from the AST, by building a Source Fragment Tree. At the end of the fix, they match unchanged AST nodes with the corresponding source fragment from the original source.
The study shows that during the evaluation, 80% of violations are detected and 94% of these are fixed.

3.3.2 Machine learning based

A complete different approach has been adopted by the following works, which use different ML algorithms to automate the repairing process.

The authors of [18] focuses on the developing of *DLFix* which is a tool for APR in Java projects.
There are some new key ideas behind *DLFix*. First, it differs from other similar works ([30], [12]) that use neural network techniques because it leverages tree-based Recurrent Neural Network (RNN) instead of a sequence-to-sequence Neural Machine Translation (NMT). The major problem with NMT for coding repair is that the knowledge on which part of the code should be fixed in not encoded in the model. The NMT-based model is trained with pairs of buggy methods and the corresponding fixing ones, thus the model must implicitly align the source code before and after the fix. An RNN-based model is beneficial to avoid incorrect alignment between the source code before and after the fix.
Another feature comes from the idea of training a different model for context learning.
Moreover, *DLFix* adopts a Convolutional Neural Network based model to push the correct fix onto the top-1 position among other fixing candidates.
The model that learns the code transformation is a two-layer model: the first one is called Context Learning Layer (CLL). It is in charge of context learning, and it is composed by a tree-based LSTM model and a tree-based encoder decoder model. As we said, the context is treated separately, and it is possible by comparing buggy and fixed AST and replacing the modified sub-tree with a summarized node. This phase outputs a vector which be used as further input for the next phase. The separation of context learning and transformation learning helps the tool to consider also the surrounding code into the fixing and to avoid incorrect alignments.
The second phase is called Transformation Learning Layer (TTL) where the before and after sub-trees are used for learning the code transformation. Moreover, the

context vector from previous layer is used as an additional input. It uses the same tree-based encoder-decoder model as the first layer.
After the model has been trained given a buggy sub-tree, it automatically generates the fixed sub-tree.
Then, a program analysis phase, in which DLFix tries to recompose the original fixed code, produces a list of final candidates.
At the end of the process, a CNN-based binary classificator is trained in order to push onto the top the perfect correction candidate.
The study points out that *DLFix* reaches 39% Top1 accuracy, where TopK is an evaluation metrics that indicates the number of times that a correct patch is in the ranked list of top K candidates. *DLFix* outperforms comparing with other tools which have a 2% - 15.8% Top1.

In, [1], the authors present *Getafix* which is an approach to produce human-like fixes based on a novel hierarchical clustering algorithm and which has been deployed on Facebook.
Getafix needs a set of changes that fix a specific class of bugs to be able to generate changes. The three main steps are the following:

1. Split set of example fixes into AST-level edit scripts

2. Learning recurring fix patterns from these edit steps basing on the hierarchical algorithm

3. Given an unseen bug, finding a suitable fix patterns and validating it against the static analyzer to ensure that the fix removes the warning

During the learning process, *Getafix* identifies changes at the AST level and generates concrete edits: pairs of sub-ASTs of the original before and after ASTs. Each AST node has a label, a value and child nodes. Furthermore, child nodes have also a description of their relationship to the parent node.
The next step is to generalize the concrete edits. To do that, it adds "holes" inside the concrete edits to represent part of the tree that may change according to the specific buggy code. This step is performed by applying the anti-unification algorithm, which merges nodes that have the same labels, the same values and the same number of children, into a combined sub-tree. Finally, it replaces them with a hole.
The major idea is to recursively combine pairs of concrete edit until all edits are combined into a single edit pattern. At each iteration, the algorithm picks a pair of edit patterns to combine, so that their anti-unification yields the least loss of concrete information.
This procedure yields a hierarchy of edit patters, where leaf nodes represent concrete edits and higher-level nodes represent increasingly generalized edit patterns. Note that the merge order is important to minimize the loss of concrete information.
Having a hierarchy of fixing patterns for a class of bugs may lead to have more

than one candidate: *Getafix* chooses the most suitable by ranking candidate with 3 scores.
As a result, given 6 class of bugs, the Top1 accuracy varies from 12% to 91% based on the type of bug.

A similar work has been done by [19]: the authors develop *AVATAR* APR system which exploits fix patterns of static violations for patch generation.
The pattern mining procedure starts by leveraging ASTs: the buggy code version and the fixed version are given to GumTree ([11]), an AST-based code differencing tool, to produce the AST edit script. The edit script describes the repair actions to be taken in order to implement the patch.
The main idea behind this work starts from the problem of grouping similar edit scripts in order to infer the common subset of edit actions. That has been done by leveraging CNNs deep learning techniques to learn edit script features, which are then used to cluster similar scripts. At the end, the largest common subset of edit script among all the scripts in a cluster is considered as the fix pattern.
Once that a suspicious fault location has been detected by an external tool, AVATAR iteratively attempts to match the location with a pattern from the database of mined fix patterns.

Chapter 4

Security misconfigurations assessment

The first goal of this chapter is to evaluate the awareness about Dockerfile security best practices by the developers. We perform these analyses by using datasets coming from real scenarios as proxy measure to judge if developers are mindful of the risks that come by using insecure Dockerfile.
The second purpose of this chapter is to evaluate the effectiveness of existing tools in detecting and repairing this kind of security misconfigurations.
In particular, in Section 4.1 we describe the datasets used in all the following analyses and the preprocessing phase exploited to ensure the testing validity.
In Section 4.2 we analyze in details the datasets to determine whether and to what extent the datasets are affected by security misconfigurations. In Section 4.3 we analyze existing detection tools to investigate whether they are equivalent in terms of misconfigurations found. We discuss the results of the analyses in Section 4.4.
In order to perform analyses, we use two datasets of Dockerfile. These datasets were already used in the analysis of [14] and are thoroughly described in [13]. We report here some important information for the purpose of our analyses.

4.1 Datasets overview and preprocessing

The choice of the datasets was basically dictated by the considerable number of unique Dockerfiles in them. As the authors of [13] claim, despite Dockerfile are one of the most prevalent DevOps artifact, there is a shortage of good and numerous datasets of Dockerfile.
Nonetheless, we consider the datasets chosen enough good for our analysis because of the features reported below.
For the purpose of our analysis, we consider the two datasets that [13] named respectively *Standard* and *Gold* dataset. In this work, we refer to the two datasets using the same name, thus Standard and Gold dataset.
The Standard dataset was collected by using GitHub's API to query every public

Table 4.1: Datasets filtering process

Dataset	Original file	Excluded file	Remaining file
Standard	178.505	16.167 (9.06%)	162.338 (90.94%)
Gold	435	248 (57.07%)	187 (42.9%)

Table 4.2: Misconfiguration in Standard and Gold dataset

	Standard	Gold
Sec. misconfig.	315.883	161
N. sec. misconfig/N. file	1.94	0.86
Affected file	158.498 (97.63%)	152 (81,28%)

repository with ten or more stars from January 1st, 2007 to June 1st, 2019. After different phases of filtering, the Standard dataset contains 178.505 unique Dockerfile.
The Gold dataset contains 435 Dockerfile and comes from the *docker-library* organization on GitHub. These files are particularly important because they belong to repositories managed and maintained by Docker experts and we expected to be better written and thereby having less security misconfigurations.
It is worth pointing out is the lack, in both datasets, of building context file (described in Chapter 2): in fact, the datasets come without any context file that is required to build the Dockerfile into an image. This factor, limits the analysis that we can carry out as described in Section 6.2.

The analyses have been conducted on the datasets after a filtering phase. For different causes listed in Section 6.2, we have not been able to correctly build an Abstract Syntax Tree from all the Dockerfile in the datasets, and this means that we can not further use our tool to analyze them. In order to speed up the analysis, we decide to rule out all these Dockerfile from the beginning. In we show Table 4.1 the number of Dockerfile involves in this process.

4.2 Misconfigurations assessment

Since the Standard dataset contains a large number of Dockerfiles coming from real scenarios, doing a misconfiguration evaluation on this dataset let us get a feel of the standard Dockerfile quality. It means that we might know whether the normal user tends to be aware of Dockerfile best practices and whether he carefully applies them.
Moreover, having the Gold dataset provides a lucky chance to us to test the awareness of the best practices on the part of experts.
Concretely, we perform these tests in order to answer different questions.

First, we want to know *how many Dockerfile in the datasets are affected by security misconfigurations, highlighting the distribution of such misconfigurations.*
Second, we want to *prove if the Gold dataset contains less security misconfigurations than the Standard one*, since the Dockerfile in it have been written by experts.
Third, we want to know if *any correlation exists among the different security misconfigurations.*
After discussing the security misconfigurations that might affect Dockerfile in Section 2.5, we pick some of them to be evaluated in the analyses. Specifically, we do not consider all the misconfigurations which can not be detected by at least one of the existing tools. Ultimately, the security misconfigurations that we are going to deal with in this chapter are:

1. Last user is root
2. Missing USER instruction.
3. Using 'latest' tag in the base image
4. Using ADD instead of COPY
5. Exposing port 22
6. Update and install packages in different RUN instructions
7. Installation of additional packages
8. Hard-coded secrets

We also introduce the ability to detect hard-coded secrets in the Dockerfile, since this security misconfiguration can be detected solely by our tool, as we discuss it in Chapter 6.
As we mention in Section 3.2.2, there are multiple existing scanners able to detect both security and not security misconfigurations. Among these, we have chosen Kics (3.2.2) to conduct the following analyses because it is the only tool able to detect all the security misconfigurations listed before while its results are still coherent with the ones of other tools as we point out in Section 4.4.

As first action, we run Kics on both datasets and collect all the resulting data in Table 4.2 to have a general overview about the number of misconfigurations found.
As preliminary remark, it is possible to notice that the number of affected file is higher on the Standard dataset than on the Gold one. To some degree, this is an expected result and might confirm that Gold dataset has been written with higher security standards. However, even the Gold dataset contains a very high percentage (81.28%) of affected Dockerfile.
Also, the ratio between the number of security misconfigurations and the total number of file in each dataset confirms that the Gold dataset contains fewer misconfigurations in relation to the number of file.

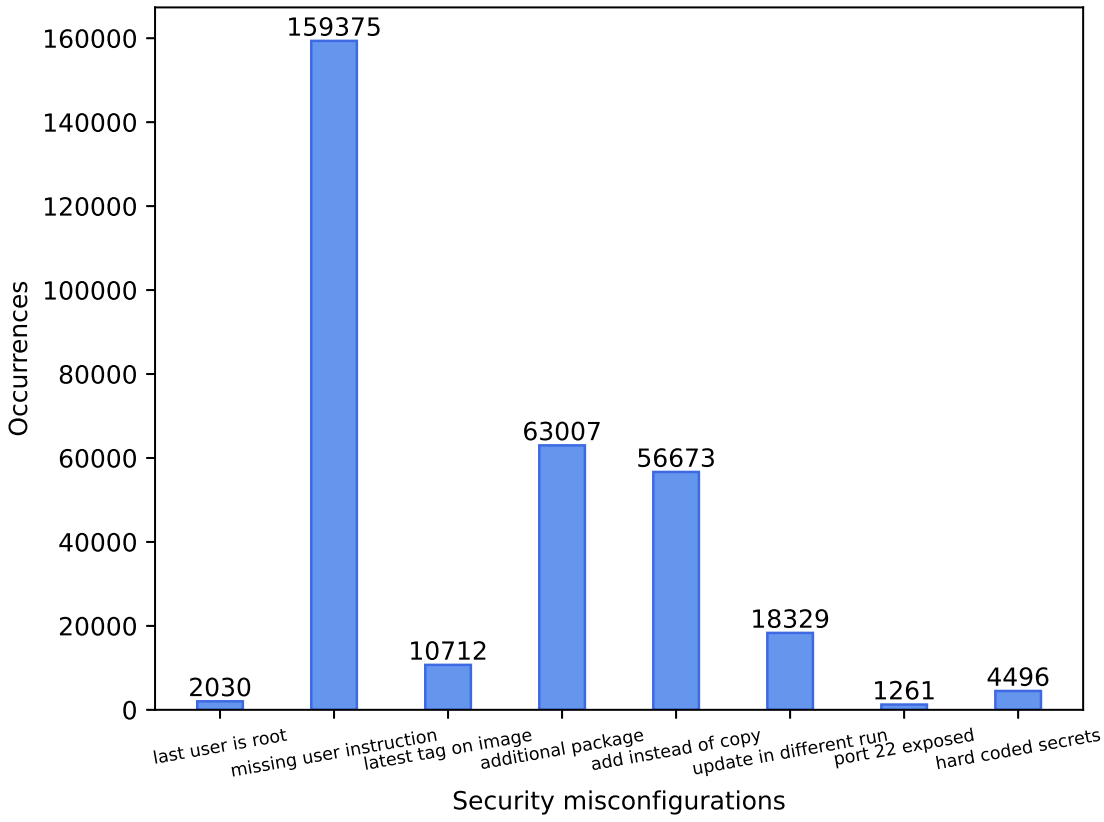

Figure 4.1: Security misconfigurations occurrences in Standard dataset

Then we move on analyzing the types of misconfigurations that we could find in the datasets in order to verify if there are some misconfigurations which occur far more frequently than other and if the results are the same for both the datasets.
From Fig. 4.1, we can highlight that the most frequent misconfigurations are respectively: "Missing USER instruction", "Installing additional packages" and "Using ADD instead of COPY". The first one is a serious misconfiguration, since it allows the resulting image to be run with root privileges, as described in Section 2.5. "Installing additional package" refers to the practice of increasing the attack surface by installing not needed dependencies and the last one refers to the practice of using ADD instruction instead of COPY one, which is more safe.
As shown in Fig. 4.2, the distribution is almost the same in the Gold dataset: the two most frequent misconfigurations still are "Missing USER instruction" and "Installing additional package" but none of the remaining ones are present.

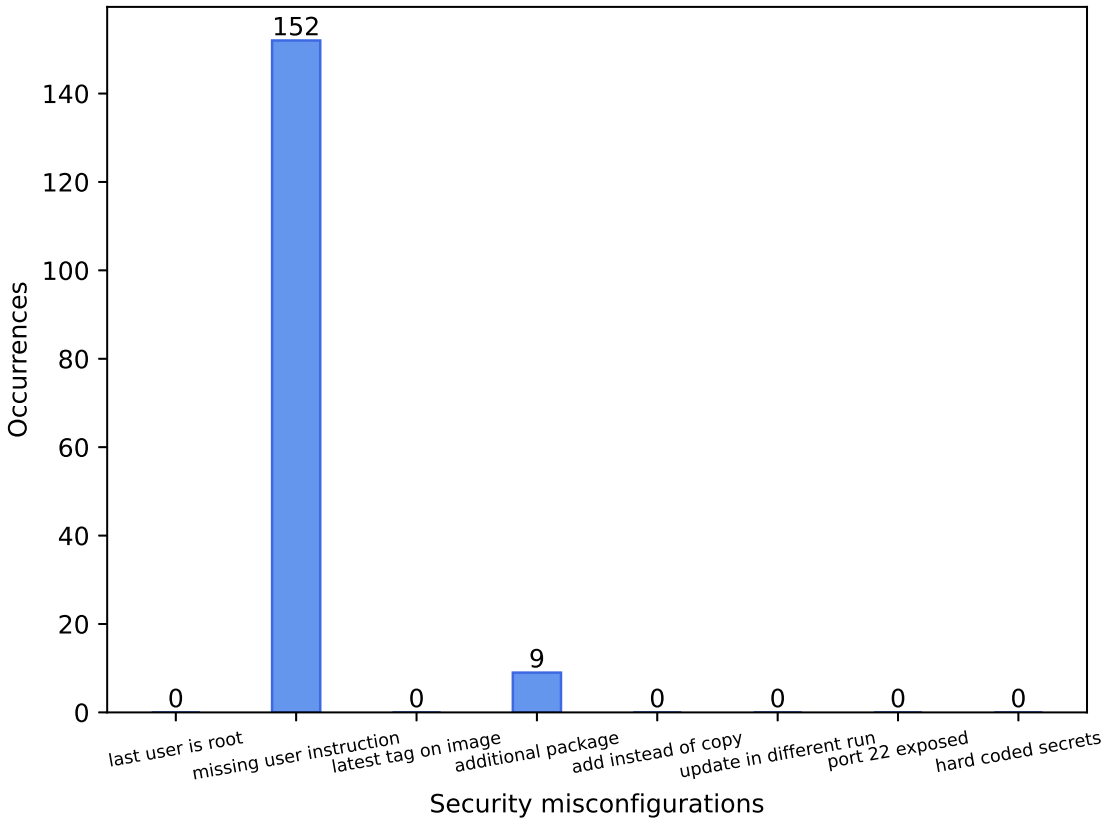

Figure 4.2: Security misconfigurations occurrences in Gold dataset

A different point of view is given by the Fig. 4.3 where we plot the security misconfiguration densities for each misconfigured Dockerfile in both datasets.
The densities are computed as:

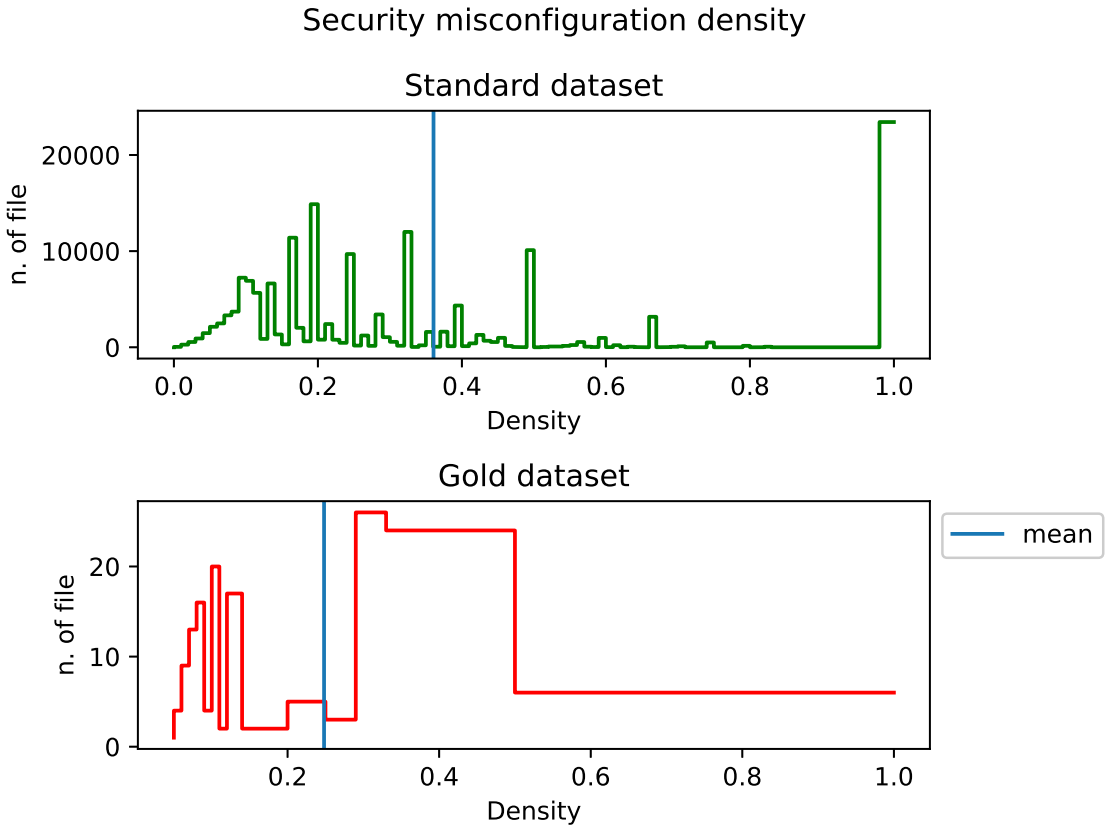

Figure 4.3: Security misconfigurations density in both datasets

$$\text{density} = \frac{\text{num. of instructions affected by a security misconfiguration}}{\text{num. of instructions}}$$

This measure give some information about how the misconfigurations are, in average, distribute inside a single Dockerfile. The distributions are quite different in the two datasets: in the Standard one it varies approximately between 0.1 and 0.5, instead in the Gold dataset, there are some peaks approximately between 0 and 0.09 and between 0.3 and 0.5.
It means that, on average, a Dockerfile in the Standard Dataset contains more insecure instructions than in the Gold Dataset.
An important value to check is the number of file having a density of 1.0: Dockerfile with that value have at least one misconfiguration in every line. This peak is salient in the Standard dataset: more than 20.000 Dockerfile are entirely insecure. On the other hand, there is no peak in the chart of the Gold dataset.

After discussing which and how many misconfigurations influence the datasets, we want to examine the number of completely secure Dockerfile, namely the Dockerfile which are not affect by any security misconfigurations.
This data can be easily inferred from Table 4.2 but since it is a very important point

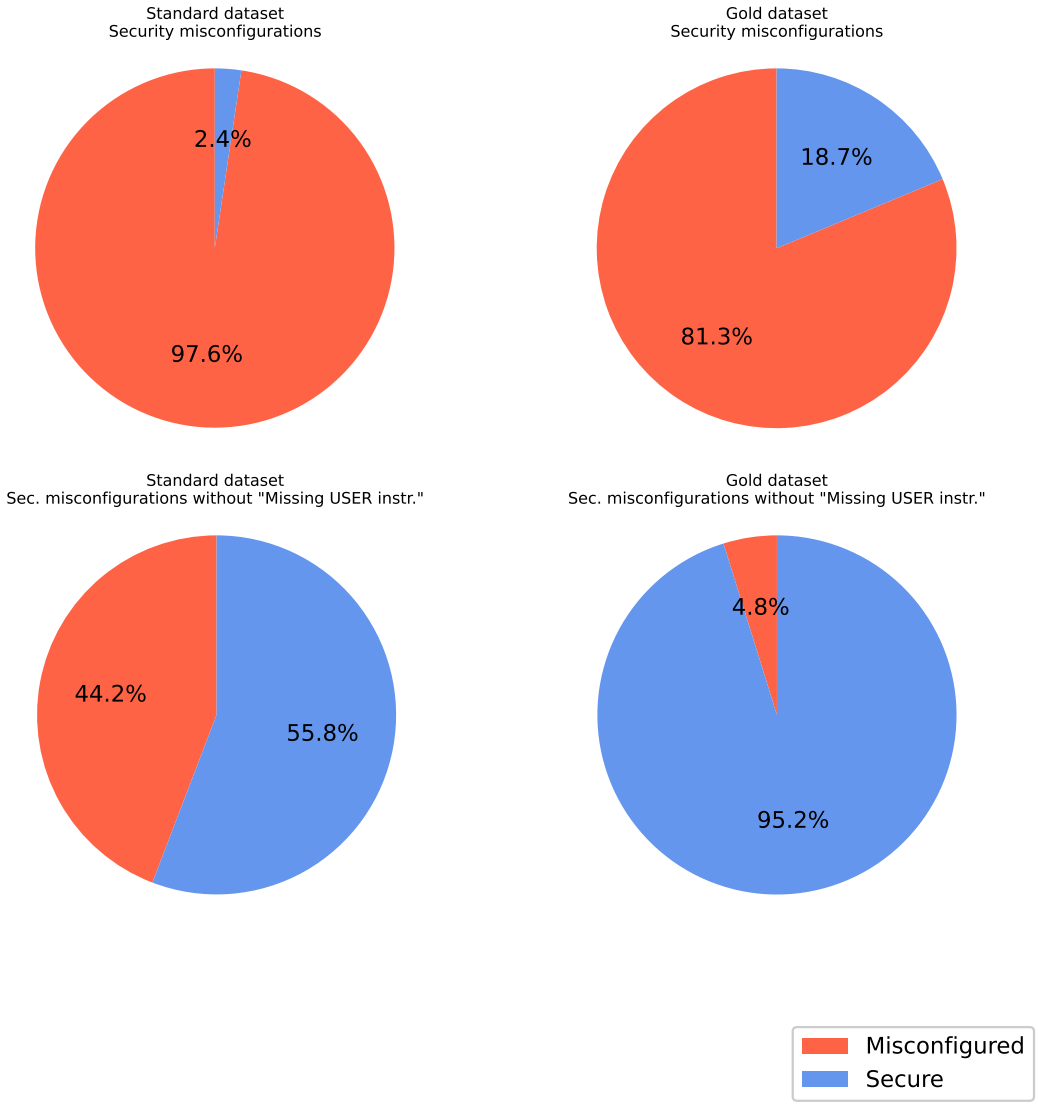

Figure 4.4: Percentages of secure and insecure Dockerfile with different set of misconfigurations

for this work, we want to make it more clear. Another reason behind discussing this data aside, is evaluating how much this value varies considering different set of misconfigurations.
Starting from the first row of Fig. 4.4, it results that almost all the Dockerfile are insecure in both datasets (97.6% for the Standard dataset and 81.3% for the Gold dataset). In this definition, we consider insecure the Dockerfile that contain at least one misconfiguration among the one listed in Section 4.1.
If we drop the most frequent misconfiguration ("Missing USER instruction"), we get a huge improvement in terms of secure Dockerfile: the Standard dataset contains 55.8% of secure file and the Gold dataset contains 95.2%. This shows that the most frequent issues is also responsible for the lack of security of the majority of Dockerfile.

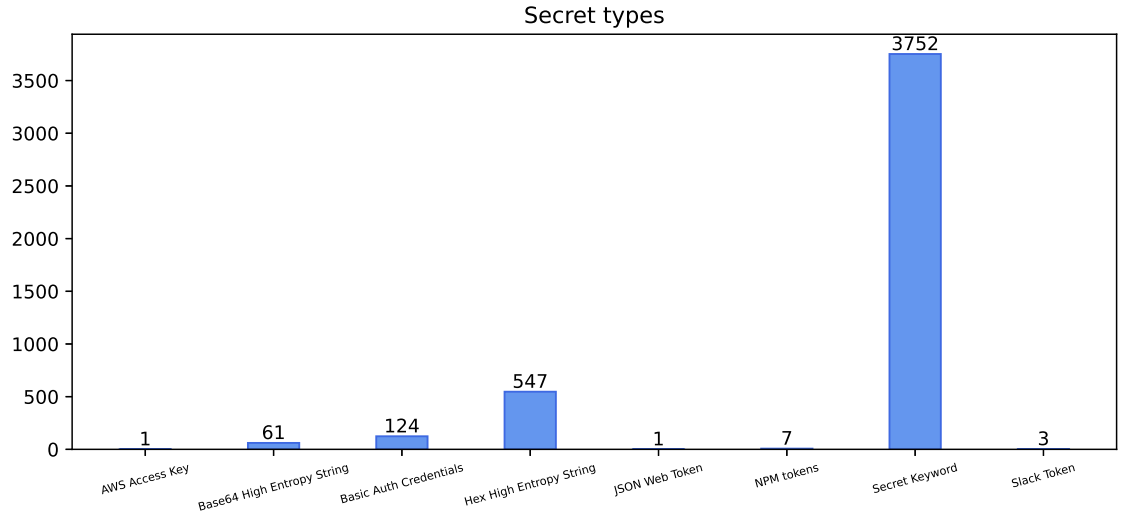

Figure 4.5: Secret types in Standard dataset

As last analysis, we want to know if there are some relevant correlations among the different misconfigurations. In order to perform this analysis, we compute the correlation matrix using the Pearson correlation coefficient on the Standard dataset and plot the results in Fig. 4.6.
Although there are no strong positive correlations, the matrix highlights a slight positive correlation between the act of updating the package manager in a RUN instruction alone and the act of installing additional packages. Both these misconfigurations affects Bash code embedded in the Dockerfile through the RUN instructions. It might mean low awareness of Dockerfile authors about how to correctly integrate Bash code in a Dockerfile.
Another remark is the expected negative correlation between the missing of USER instruction and the presence of "USER root" instruction, since both misconfigurations are mutually exclusive.

Hard-coded secrets As we said at the beginning of this section, we introduce the ability to detect hard-coded secrets in the datasets, by using the system described in Chapter 5. From the findings represented in Fig. 4.1, we uncover the presence of a fair amount of secrets in the Standard dataset. On the other hand, Fig. 4.2 shows no presence of secrets in Gold dataset.
We further analyze the types of secrets which have been embedded in the Dockerfile and summarize the results in Fig. 4.5. The "Secret Keyword" type is the most recurring one with 3.752 occurrences. It refers to the presence of specific keywords that might indicate the existence of hard-coded secrets; examples of such keywords are: "api_key", "password", "db_key" etc. The second most recurring category, even though with considerably fewer occurrences (547), is the "Hex High Entropy String": as mentioned in [21], high entropy is a fundamental feature of strong passwords,

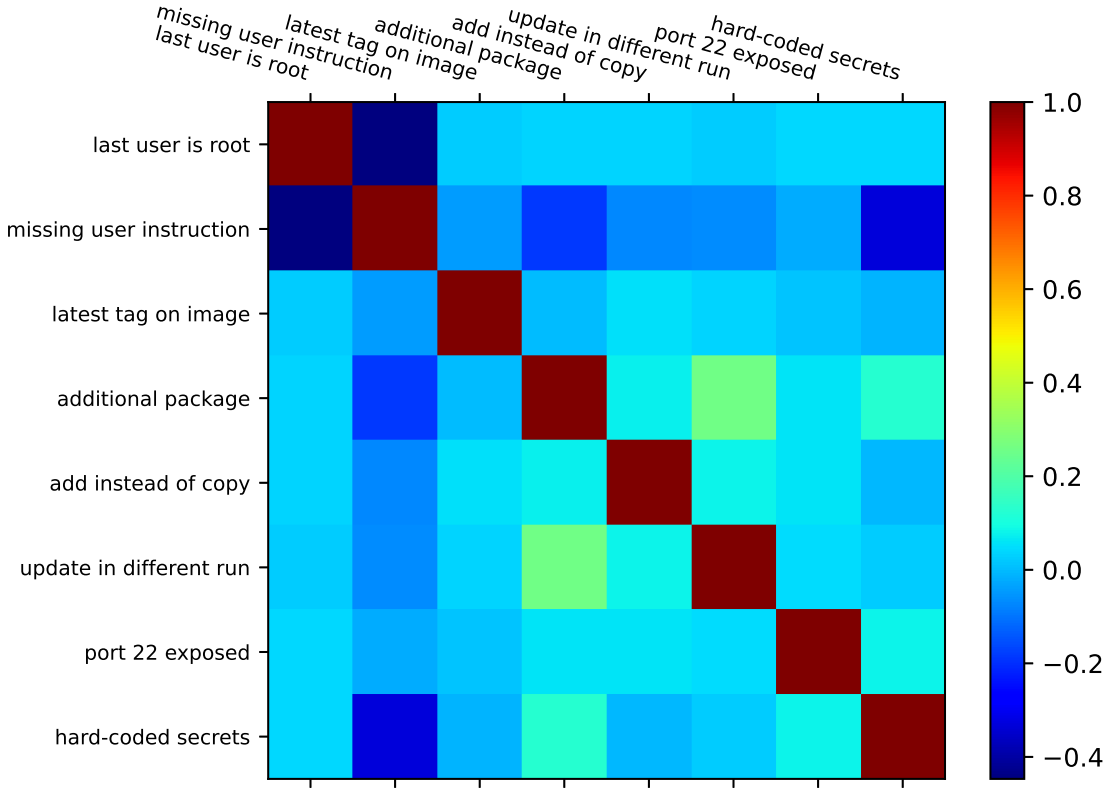

Figure 4.6: Correlation matrix misconfigurations on the standard dataset

thus having strings with high entropy might indicate the presence of hard-coded passwords. In this specific case, the secret is encoded according to the hexadecimal scheme. The same logic is applied to the "Base64 High Entropy String" category, where the secrets are encoded according to the Base64 scheme and which contribute to the total amount with 61 occurrences. Another relevant type is the "Basic Auth Credentials" that, by using regular expressions, catch authentication credentials in formatted URI [1].
The remaining categories have very few instances (between 1 and 7) and regards specific technologies such as AWS [2] and NPM [3].

HEALTHCHECK instruction missing During the assessment, we do not include the "HEALTHCHECK instruction missing" as a security misconfiguration, even if it has been reported in the CIS Docker Benchmark (version 1.3.1) [4].
Briefly, the HEALTHCHECK instruction indicates how to test a container to check whether it is still working. From a security prospective, the lack of this instruction may come under the definition of security misconfiguration because an unavailable

[1] https://datatracker.ietf.org/doc/html/rfc3986
[2] https://aws.amazon.com/
[3] https://www.npmjs.com/
[4] https://www.cisecurity.org/benchmark/docker

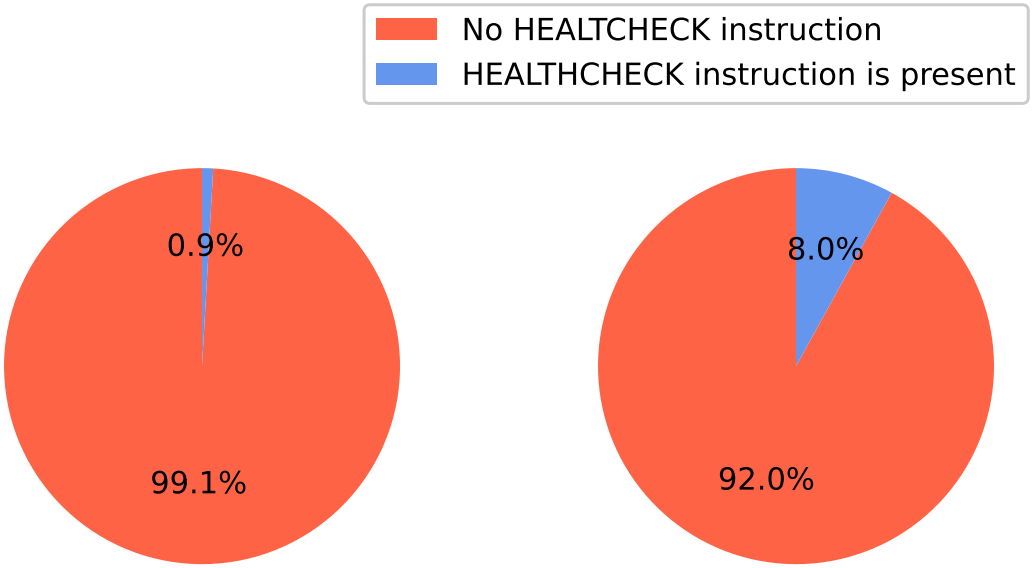

Figure 4.7: Percentage of Dockerfile with HEALTHCHECK instruction in Standard dataset

service violates the availability property in the CIA triad [29].
We come to the conclusion that, if we do not have any other information than the Dockerfile itself, it can not be considered a security misconfiguration a priori. This because the service availability might be checked at a higher level in the system, such as with an orchestration tool.
Despite this, for the sake of completeness, we briefly evaluate the presence of this instruction in the datasets. In Fig. 4.7 is clearly visible how the absence of the HEALTHCHECK instruction is highly prevailing in both datasets. If we had included this as a security misconfiguration, it would have been by far the most recurring one. This chart is also important for Section 4.3, where we highlight how these misconfigurations heavily affects the results of the current tools analysis.

4.3 Evaluation of existing tools

In this section, we analyze the selected tools described in Section 3.2.2 and compare their effectiveness on the datasets.
As a first step, we run the selected scanners on the datasets. The results are summarized in the Table 4.3 where the number of file affected by security related and not security related misconfigurations have been split respectively in the columns "Security" and "Other".
In order to further investigate the results in Table 4.3, we plot the number of security misconfigurations identified by the four tools on the Standard dataset in Fig. 4.8.

Table 4.3: Scanner results on both datasets

	Gold						Standard					
	Security			Other			Security			Other		
	n. of file	% of file	Avg	n.of file	% of file	Avg	n. of file	% of file	Avg	n.of file	% of file	Avg
Hadolint	6	3.21	0.26	138	73.8	2.57	61.040	37.6	0.73	125.176	77.11	3.66
Checkov	177	94.65	1.04	172	91.98	0.92	156.040	96.12	1.35	161.554	99.52	1.22
Trivy	177	94.65	0.95	27	14.44	0.18	154.395	95.11	1.36	71.901	44.29	0.78
Kics	152	81.28	0.86	183	97.86	4.04	158.184	97.44	1.92	161.608	99.55	5.78

A first consideration that could be done, looking at Table 4.3, involves how different scanners picture a different evaluation of the same dataset. As example, Trivy detects an enormous difference evaluating security and not security misconfigurations in both datasets, inducing the user to think that the file affected by a security misconfigurations are higher than the ones affected by other misconfigurations. Completely opposite results are given by Hadolint, while Checkov and Kics indicate a balance between the two categories. The evident fluctuation caused by the scanners used lead us to further investigate and determine which scanner is the most accurate.

First we take into account Trivy since it gives very different results: as we can notice in Table 4.3, it founds very little number of file affected by not security misconfigurations compared with other tools. We found out that this unusual behavior is determined by two causes: firstly, a lower number of policies involving not security misconfigurations (the number of policies for each tool has been reported in Table 4.4) and secondly, which not security misconfigurations it is able to detect.

Concerning the former, Trivy uses 19 policies against respectively 71 and 42 of Hadolint and Kics. An obvious claim is that Checkov has a low number of policies as well (6 policies), even less than Trivy, but its results are still comparable, in terms of affected file, with Hadolint and Kics. This claim lead us to on the latter case: it is possible to point out that the higher number of file affected by other misconfigurations reported by Checkov and Kics come from the "HEALTHCHECK instruction missing" misconfiguration. This issue can not be recognized with Trivy, so it detects less file affected by not security misconfigurations.

In this regard, it is also possible to remark that Hadolint can not detect the missing of the HEALTHCHECK instruction as well (actually it can, but it is deactivated by default). Despite this, its results for other misconfigurations are quite high. This because, even if it can not detect that misconfiguration, it has 71 other policies concerning different not security misconfigurations, more than any other.

Continuing to analyze Hadolint, it is worth mentioning the low number of file affected by security misconfigurations reported by it. This behavior can be easily explained once again by looking at the number of policies in place: security policies compose approximately the 5% of the entire policy list. The consequence is that the detection of not security misconfigurations occurs more frequently than the security related one.

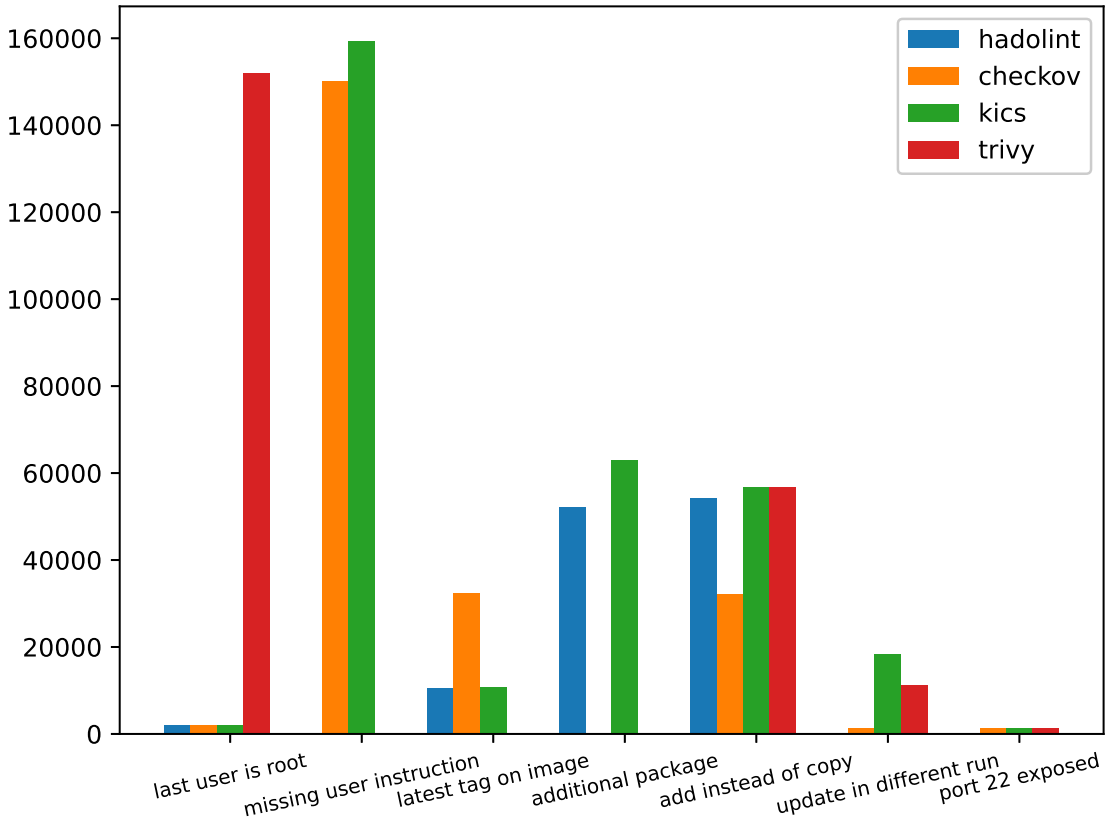

Figure 4.8: Misconfigurations occurrences

In Fig. 4.8 we can analyze how many misconfigurations can be detected with the different tools: Kics is the only one capable of catching all the misconfigurations we are taking into account. On the other hand, Hadolint is the one that is doing worst in terms of security misconfigurations recognized, as it misses half of the misconfigurations taken into account.
Almost all the tools but Hadolint, concur that the most frequent misconfiguration is "Missing USER instruction". Hadolint, not being able to detect it, shows "Using ADD instead of COPY" as the most recurrent issue. Kics, Trivy and Checkov agree regarding the less frequent misconfiguration as well, reporting "Port 22 exposed". Once more, Hadolint does not have a policy for detecting this, so it reports "Last user is root".
Previously commenting Table 4.3, we notice that there are some differences in the number of file affected by security misconfigurations. We justified it by highlighting the different number of policies in place for each tool. Analyzing Fig. 4.8, we can further investigate these differences by noticing which policies mostly contribute to this discrepancy.
As already said, "HEALTHCHECK instruction missing" heavily contribute to raise other misconfigurations count for each tool that can recognize it. The same can be said with "Missing USER instruction" for the security misconfigurations: Checkov

and Kics have higher number of warnings than Hadolint due to the detection of that misconfiguration.
Trivy detects both "HEALTHCHECK instruction missing" and "Last user is root" with the same policy: since we can not automatically distinguish the root cause, in Fig. 4.8 we put warnings belonging to both categories under the "'Last user is root" category and zero the "Missing USER instruction" category. This choice is still coherent with the ultimate consequence of not adding the USER instruction: if no user is explicitly specified, the image will run with root user.

Table 4.4: Scanner policies

Scanner	Security policies	Other policies	Total policies
Hadolint	4	71+	75+
Checkov	7	6	13
Trivy	4	19	23
Kics	8	42	50

4.4 Discussion

After the analyses on the misconfigurations affecting the datasets and the effectiveness of detection tools, in this section we put together all the results trying to picture the current situations about Dockerfile best security practices awareness and existing detection tools effectiveness.

4.4.1 Security best practices awareness

The analyses on the Standard dataset reveal that the majority of Dockerfile in it can be considered insecure: 97.6% of the Dockerfile contains at least one security misconfiguration.
The situation is even worse considering the fact that a Dockerfile in that dataset, in average, has a security density of 0.36; therefore 36% of the instructions are affected by at least one security issue. It is slightly better if we consider the Gold dataset: 81.3% of the dataset is misconfigured, with an average density of 0.24.
One of the intention of this study is to prove whether the Dockerfile in the Gold dataset are actually better written than the ones in the Standard dataset. Our analyses prove this hypothesis both by showing a lower percentage of affected file in the Gold dataset and a lower security density. Even if the Gold dataset percentages are quite closer to the one of Standard dataset (81.28% for the Gold and 97.63% for the Standard dataset), we should take into account the majority of issues in the Gold dataset come from the single misconfiguration "Missing USER instruction"; by not considering it, the percentages of secure Dockerfile increase to 55.8% for the

Standard and 95.2% for the Gold dataset. Since the Standard dataset has also a greater variety of misconfigurations in its Dockerfile than the Gold dataset, we can definitely affirm that Gold dataset has been written with higher security standards.

Concerning potential correlations among the misconfigurations, it is an interesting point noticing a positive correlation of the issues "Install and update in different RUN" and "Installation of additional packages": indeed both of them concern embedded Bash commands. Specifically, in order to recognize and avoid this kind of misconfigurations, Dockerfile authors should have an understanding of how Docker file system works since the former increases unnecessarily the image size and the latter is unsafe because of Docker caching mechanism. Therefore, we can affirm that the presence of these misconfigurations are an indicator of an unawareness about how to effectively embed Bash code in a Dockerfile. This can find support for its validity by considering the Gold dataset, where the presence of "Install and update in different RUN" misconfiguration does not appear and "Installing additional package" appears only in 9 Dockerfile out of 161.

If we use the Standard dataset as proxy for the current awareness of security best practice, we can draw the conclusion by asserting that, since Dockerfile misconfigurations found are numerous and various, it is still needed more awareness about Dockerfile security best practices and the risky consequences of bad written Dockerfile.

4.4.2 Existing tools limitations

Since an extensive number of Dockerfiles is affected by some security misconfiguration, different tools exist to audit them before they are used in production.
We have already mentioned in Section 3.2.2 that the majority of the rule-based scanners have been written to check file belonging to different IaC technologies. This is not a problem per se but, by looking at the policies in place, emerge the trend to underestimate the importance of a well-written Dockerfile. As example, we take in account Checkov which, at the time of writing, has approximately 900 policies for Kubernetes file and only 13 for Dockerfile. It is certainly true that auditing Kubernetes configuration file is far more complex than a single Dockerfile, but it does not mean that 13 policies are enough to completely secure a Dockerfile. We can compare Checkov with Hadolint, the only tool exclusively focused on Dockerfile: it uses more than 75 policies to completely check Dockerfile.
On the same page, it is even worse if we consider misconfigurations related to security. From our analyses, we determine that only one tool, Kics, is able to detect all the security misconfigurations chosen. In addition, it should be considered that the set of security misconfigurations analyzed in this work is only a subset of the total security issues, since some of them can be detected by no tools as represented in Table 3.2.

Besides the insufficient number of policies, the problem of secure Dockerfile is worsened by the fact that often different tools output different results based on how the policies have been built. Thus, before using a certain scanner, it is essential to comprehend how it works, which misconfigurations could be caught and how the results should be interpreted.
Another problem that affect all the tools we analyzed is how the user should fix security misconfigurations once the scanner reported them. In fact, to the best of our knowledge, it does not exist any detection tool which is also able to automatically correct security misconfigurations in a Dockerfile.
This is an evident limitation of the existing tools that we try to solve with the tool described in Chapter 5.

Chapter 5

System Implementation

As we point out in Section 6.2, one of the limitations of the existing tools is their inability to fix the Dockerfile once a misconfiguration has been reported, without involving the developer itself. A manual fixing procedure requires some kind of expertise and knowledge about the security best practices, while not excluding an ineffective or bad repair.
To the best of our knowledge, it does not exist any tool able to automatically fix security misconfigurations in Docker artifacts. As we consider in Section 3.2.1, *Shipwright* is the tool that comes closer to this idea, albeit it is focus on repairing errors which lead to the building process failure.
In the direction of filling this gap, we have worked to a system which is automatically able to detect and correct some security misconfigurations discussed so far, and we present it in this chapter. In particular, in Section 5.1 we outline a general overview about the tool, the technologies that we use to build it and a high-level view of the architecture. In Section 5.2 we describe in details each component and its purpose. In Section 5.3 we illustrate the workflow: how each element exchange information with others to produce the final outcome.

5.1 Architecture

A high-level overview of the tool architecture is represented in Fig. 5.1, where it is possible to identify the main components; in particular:

- **Parser**: it is responsible for parsing the Dockerfile and transforming it in AST (tree representation of the code described in Section 2.1). It makes use of a custom version of *tree-sitter*[1] parser to recognize Dockerfile instructions and their parameters. The resulting AST is further adjusted by a custom logic.
- **Detector**: this component is in charge of detecting potential security misconfigurations and appending their IDs to the AST's nodes. Since it already

[1]https://tree-sitter.github.io/tree-sitter/

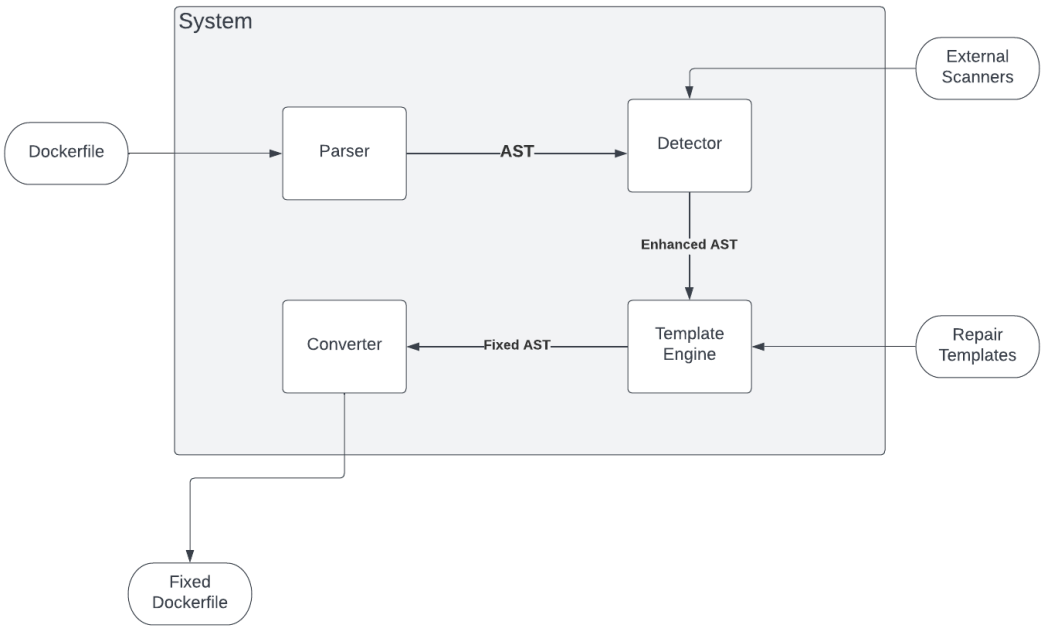

Figure 5.1: System architecture

exists valuable scanners for this scope, it makes use of them.

- **Template engine**: this element is actually able to secure the detected misconfigured instructions. All the repairs happen in the AST produced by the parser according to the directives given by the repairer. Internally, it exploits human written templates to fulfil its purpose, which indicates how to modify the AST still maintaining the right degree of generalization.

- **Converter**: it is a small component which go through the AST and convert it back into a valid Dockerfile.

During the developing phase, we focus on making the architecture as much expansible and flexible as possible. Namely, almost each component can be replaced or adjusted with little endeavor: for example, other templates can be written, and new scanners could be implemented.
The system has been built entirely in Python 3.8[2] because of this versatility and ease of use, which allow us to build a complete system in relative limited time.
Finally, despite the workflow is extensively described in Section 5.3, for better comprehension of the following sections, we give here a general insight.
The procedure starts by parsing the Dockerfile into an AST to ease the analysis. One or more scanners detect the misconfigurations in the original Dockerfile and append them to respective nodes in the AST. A template engine scans the AST

[2]https://www.python.org

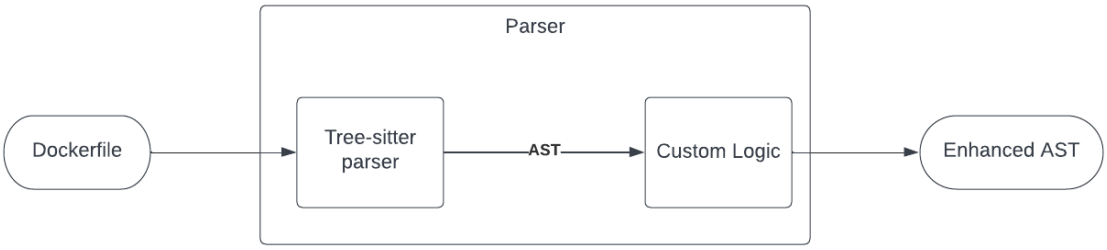

Figure 5.2: *Parser* component

looking for reparable issues. When it founds them, it modifies the original AST. Once all the misconfigurations have been fixed, the new AST is converted back into a Dockerfile.

5.2 Main components

After presenting a general overview of the architecture, we analyze in details each component in order to better understand its job inside the entire system. As we mention in the previous section, the system is built around the four components which are described in the following sections.

5.2.1 Parser

The final purpose of the *Parser* represented in Fig. 5.2 is to convert the Dockerfile into a proper AST, usable in the further analyses.
In the system, mostly of this job has been done by *tree-sitter* library. Briefly, *tree-sitter* is an open-source tool able to parse different artifacts into AST. The parsing process is made possible by particular file, the grammar file, which contain directives about how to build the AST.
In accordance with *tree-sitter* documentation[3], each node comes with different fields. Among all the fields, we have put particular attention to three of them because they are essential for the building of the final Dockerfile.
These fields are:

- **Type node**: it identifies the node type of the AST. In turn, the node type identifies the Dockerfile instruction a node is referring to. For example, the *run_instruction* type refers to the Dockerfile RUN statement.

- **Text**: it contains the text of the instruction. In most of the cases, each child node contains a portion of the parent node text. In this regard, it is worth

[3]https://tree-sitter.github.io/tree-sitter/using-parsers

noting that the root node contains the entire Dockerfile.

- **Line**: it identifies the line of the Dockerfile that a particular node comes from. It is useful to be able to assign the misconfigurations found by the scanners to the right nodes.

The AST produced by *tree-sitter* parser has been built according to specific grammar rules documented in the Dockerfile grammar file [9].
There is a major problem in using the straight AST obtained from *tree-sitter*: it does not parse the embedded Bash code. Parsing it might not be required but since, there are some misconfigurations involving just the Bash code, further actions are needed.
The problem has been solved implementing a custom logic (*Custom Logic* component in Fig. 5.2) which performs the following steps:

1. Detaching the unparsed Bash code from tree at *shell_command* node level.
2. Using *tree-sitter* with Bash grammar file to parse the code inside the *shell_command* nodes.
3. Attaching the Bash AST to the previous AST by means of *shell_command* node.

The *Custom Logic* component is also responsible for slight modifications in the AST structure.
Moreover, we have made different minor modifications, still described in Chapter 8, both on Dockerfile([9]) and Bash ([2]) grammar file of *tree-parser*. These adjustments on the grammar file and the ones made by the Custom logic, have been needed to overcome some limitations in the grammar rules, and to facilitate the conversion of the AST back to a Dockerfile.
Nevertheless, we have not been able to completely rearrange grammar rules for our purposes because of the complexity of this task, and this choice is the cause of some limitations described in Section 6.2.

5.2.2 Detector

The detector is the component in charge of detecting misconfigurations and append them to the right nodes of the AST produced by the parser.
In particular, it is an interface able to transform the outputs of existing scanners into useful information to be attached to the AST.
This design choice comes from the idea of making the system flexible enough to exploit already existing valid scanners, while not preventing other developers to create and use their own scanner.
The internal details of this element are represented in Fig. 5.3. As it is possible to notice, the component has four main subcomponents:

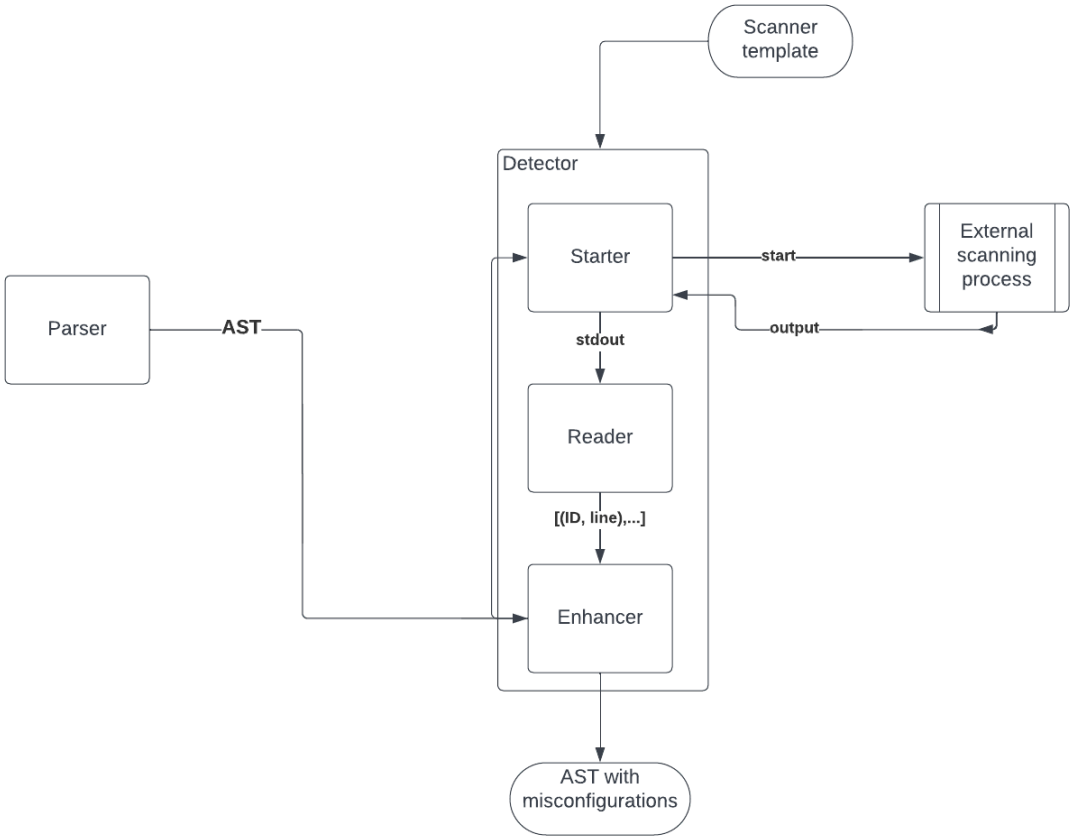

Figure 5.3: *Detector* component

- Starter: it is in charge of starting the external scanner process and collect its output. It is worth noting that the scanning process is done on the original Dockerfile contained into the root node.

- Reader: since different tools have different output format, this component transforms each output in a format understandable by the system. In particular, it passes to the next subcomponent a list of tuples with the misconfiguration ID (generated from the scanner itself) and the line in which the misconfiguration occurs.

- Enhancer: element which job is to enhance the AST nodes with corresponding misconfigurations produced from the reader.

- Scanner template: manually written template that give directives to the detector about how to configure other subcomponents. For example it indicates how the scanner should starts and how to convert its output format.

In our implementation, the system gives the opportunity of using simultaneously different scanners on the same Dockerfile in order to detect as many misconfigurations as possible and fix them in a single run.
In particular, for this work, we both use *Kics* scanner to detect the security misconfigurations listed in Section 4.1 and still develop our own scanner to detect a misconfiguration ("Hard-coded secret in the Dockerfile") that no other scanner can detect on Dockerfile.
Regarding the custom scanner, we integrate *detect-secret* project[4] as external scanner in order to detect different kind of secrets.

5.2.3 Template engine and converter

The template engine represented in Fig. 5.4 is the component that actually allows the system to repair misconfigurations. In particular, it scans the AST enhanced with the misconfigurations and, for each of them, it finds a corresponding *Repair template*. If a template exists, it starts the repairing process by modifying the AST. It is worth poiting out that, the templates have been written by human experts.
Each *Repair template* contains two sections: the context and the edit actions. The former indicates the misconfiguration topology, namely it denotes the misconfiguration as a particular node combination. The latter contains the set of actions needed to modify the current topology, thus to repair the AST. The *Repairer* structure is divided into two subcomponents, reflecting the template's sections:

- Context checker which double-check the presence of the issue in the AST with the context provided by the template and eventually return the nodes to be modified along with other information useful for the editor.

[4]https://github.com/Yelp/detect-secrets

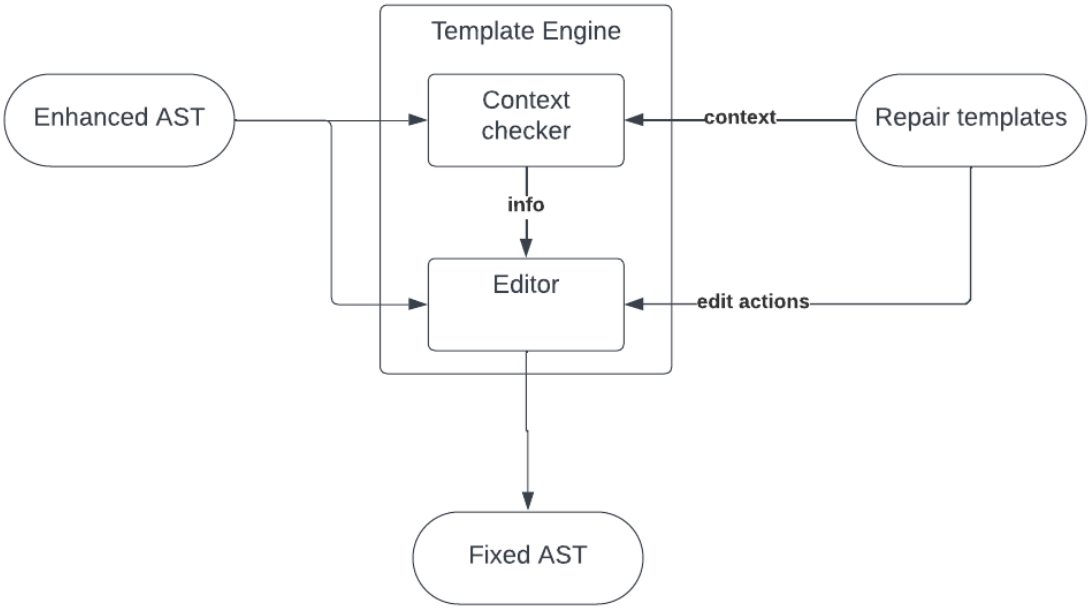

Figure 5.4: *Template-engine* component

- Editor which modifies the structure of the tree exploiting the context provided by the context checker and the action reported in the templates.

The *Converter* is in charge of converting back the AST into a Dockerfile. As we mention in Section 5.2.1, each node contains the *Text* field. Thus, the *Converter* iterate over the complete AST and put together the *Text* field of each node. Moreover, since some nodes could have the same text sequences, a custom logic inside the *Converter* avoid the presence of redundant information.

```
1 # My Dockerfile
2 FROM debian:stretch-slim
3 RUN apt-get update
4 RUN apt-get install -y gcc
5 WORKDIR /usr/src/hello
```

Listing 5.1: "Dockerfile sample"

5.3 Workflow

In this section, we illustrate the workflow behind the misconfiguration detection and correction; in particular, we cover the path of an insecure Dockerfile given in input to the system until the creation of new a secure Dockerfile.
For example, we consider the Dockerfile in Listing 5.1.

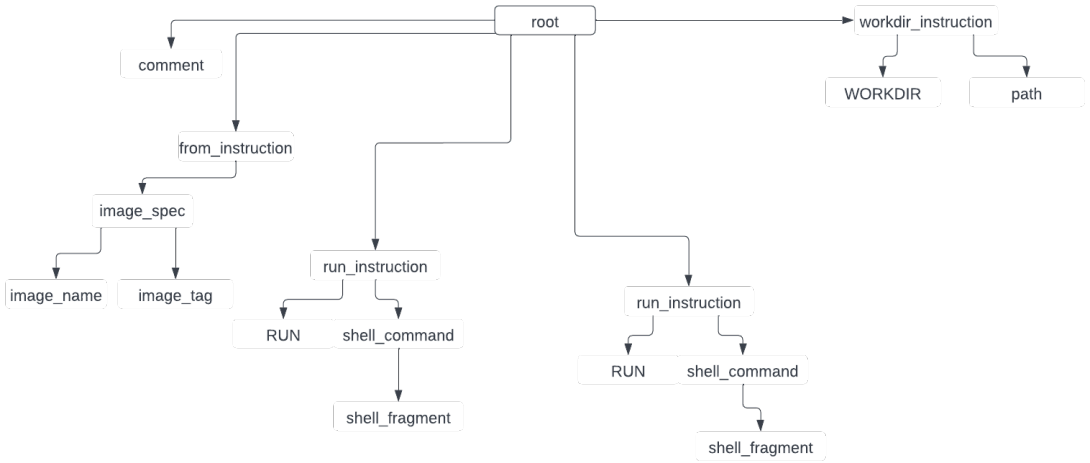

Figure 5.5: Example of AST generated by *tree-sitter* from Listing 5.1

From the Dockerfile to the AST The sample Listing 5.1 contains three security misconfigurations out of the ones listed in Section 4.2:

- **Updating and installing packages in different RUN instructions**: it uses "apt-get update" and " apt-get install" in two different RUN statements (lines 3 and 4).
- **Installing additional packages**: it does not prevent the package manager to install recommended packages with the "–no-install-recommends" option (line 4).
- **Missing USER instruction**: it builds an image which is going to start with root privileges.

When the Dockerfile is passed as input, the first step taken by the system is trying to convert it into an AST. Specifically, the *Parser* creates the AST in two phases: it derives a first draft of AST by using *tree-sitter*, then a custom logic enhances it with the AST of the embedded Bash codes and performs other minor modifications. The tree after the first parsing phase is showed in Fig. 5.5; instead, the final AST is represented in Fig. 5.6.
The final AST differs from the intermediate one because of the nodes highlighted in green, which represent the ASTs of the two Bash codes.

Looking for misconfigurations in the AST The final AST is passed to the *Detector* which allows running different external scanners on it and, as final result, enrich the nodes with possible misconfigurations.
In Fig. 5.7 each red node contains two information about the misconfiguration that affect it: the misconfiguration ID given by the scanner and the line corresponding to

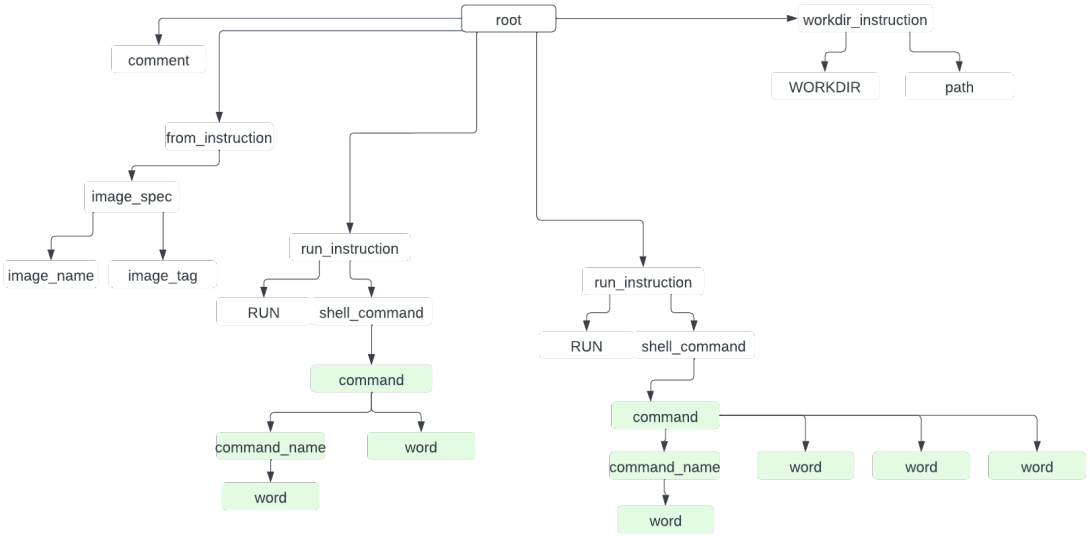

Figure 5.6: Example of final AST generated from Listing 5.1

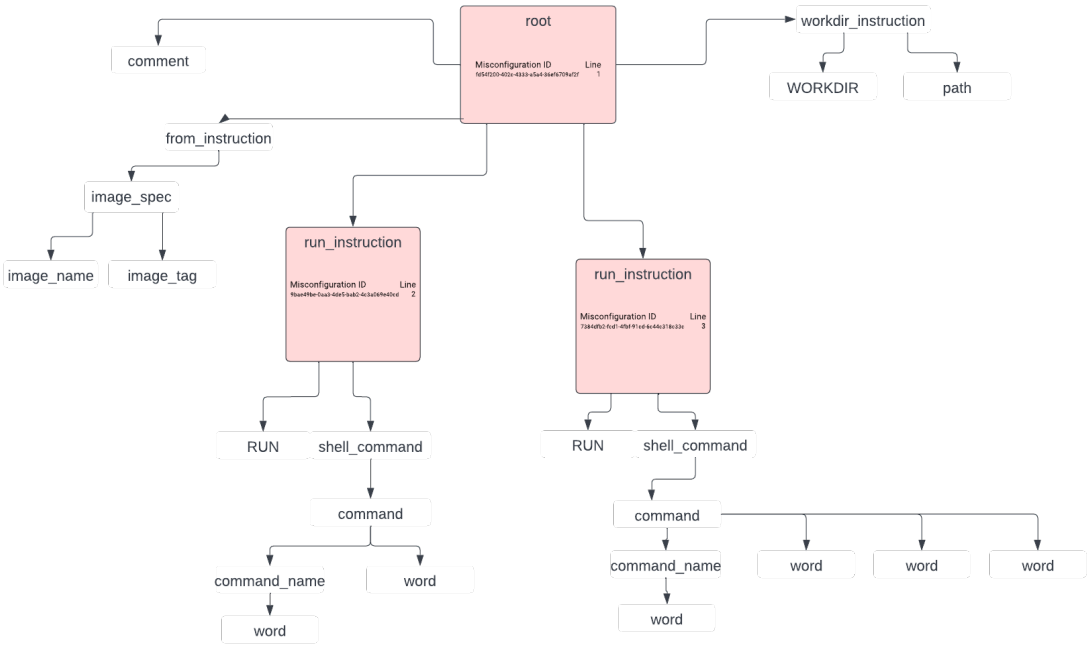

Figure 5.7: Example of AST enhanced with misconfigurations ID

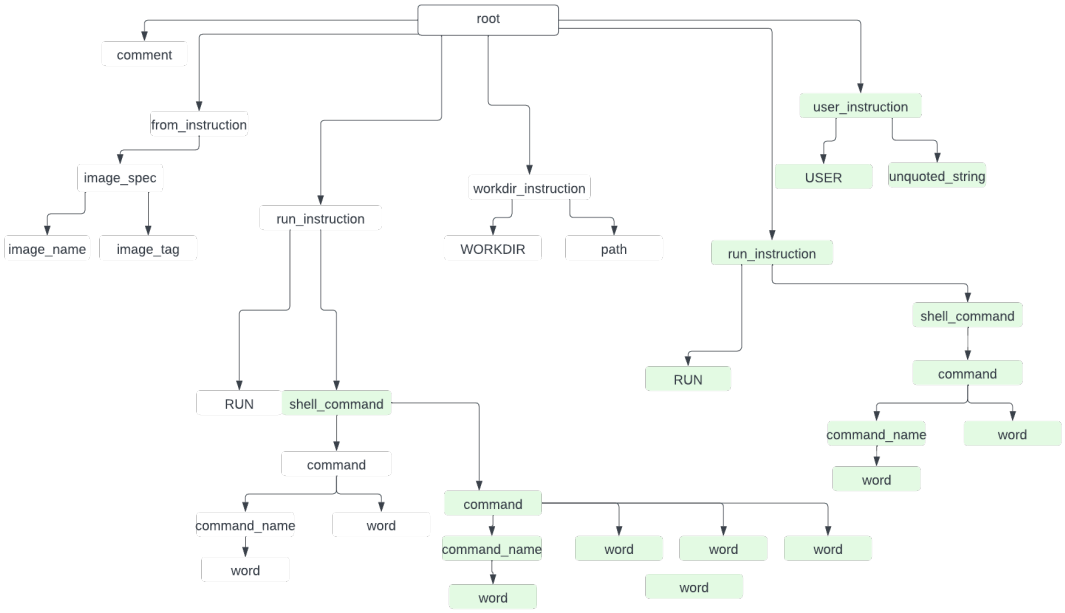

Figure 5.8: Example of fixed AST

that node. In particular, in this example, the root node is responsible for the missing of a USER instruction, its *run_instruction* left child is responsible for running *apt-get update* and *apt-get install* in different RUN statements and its right child for the installation of recommended packages.
The work of the *Detector* is made possible by the *Scanner templates* which describe how to interface external tools with the system.
In this example, we run as scanner *Kics*, which identify the three misconfigurations, and a custom scanner which is able to detect hard-coded secrets in the artifact but in this case does not produce any warning. Regarding *Kics*, the scanner launches the process and collect the resulting output in JSON format[5]. Then it transforms the output into (ID, line) tuples where the ID is the misconfiguration identification number given by *Kics* and the line is the line affected by that misconfiguration.

Misconfigurations corrections This resulting AST, enhanced with the misconfigurations found, is ready to be fixed by the *Template Engine*. Since the system work with an AST, the fixes have to be expressed in terms of node topologies as well. This particular expression is provided by the *Repair templates*, one for each misconfiguration. In particular, each template contains the misconfiguration topol-

[5]https://www.rfc-editor.org/rfc/rfc8259.html

ogy and the actions, in terms of operations to be done on the nodes, to fix the AST. The result of the *Template Engine* is a new, fixed AST represented in Fig. 5.6. As it is possible to notice, the nodes in green have been changed. In particular, the two branches that belonged to the two different RUN instructions have been merged together; instead, a new *run_instruction* node has been created to create the new user. Finally, a *user_instruction* has been added in order to set the new user as entry user.

From the AST back to the Dockerfile The last step is to convert the fixed AST back into a Dockerfile. This procedure is performed by the *Converter* which scans the nodes of the AST and, through the *Text* fields described in Section 5.2.1, it is able to write back each node with the right syntax. The resulting Dockerfile is in Listing 5.2 where the misconfigurations have been repaired in the following way:

- **Update and install packages in different RUN instructions**: the two RUN statements have been merged together (line 3).
- **Installing additional packages**: The "–no-install-recommends" option has been added to the *apt-get install* Bash command (line 4).
- **Missing USER instruction**: a new user has been created and set as entry user (lines 6-7).

```
1 # My Dockerfile
2 FROM debian:stretch-slim
3 RUN apt-get update && apt-get install -y
4 --no-install-recommends gcc
5 WORKDIR /usr/src/hello
6 RUN adduser userapp
7 USER userapp
```

Listing 5.2: "Fixed Dockerfile from Listing 5.1"

Chapter 6

Evaluation

After presenting the system we have built, we want to assess whether it could work effectively in real scenarios. For this reason, we set up different tests in order to prove its validity.
In this chapter, we discuss the tests we have performed on the system; in particular in Section 6.1 we describe the different tests, how we perform them and the corresponding results. In Section 6.2 we discuss the test results, along with the limitations that we encounter both in the system implementation and in the testing phase. Finally, in Section 6.3 we propose some ways to improve our system and enhance its capabilities.

6.1 Results

As a first test, we want to know how many misconfigurations, both in Gold and Standard dataset, the system is able to repair. Specifically, we consider a misconfiguration repaired if, after a scanning process, no warnings are raised by the scanner about it.
It is worth mentioning that the system is not able to fix all the security misconfigurations listed in Section 4.2 for the reason mentioned in Section 6.2. Specifically, the system does not have a *repair template* (component described in Chapter 5) for the misconfigurations: "Using 'latest' tag in the base mage" and "Hard-coded secrets"; therefore the misconfigurations that can be repaired are:

- Last user is root
- Missing USER instruction
- Using ADD instead of COPY
- Exposing port 22
- Updating in different RUN instructions
- Installation of additional packages

The steps that we take to run the tests are the following:

1. We analyze the Dockerfile and collect data about the misconfigurations found in them.

2. We run the system on the Dockerfile in order to repair the misconfigurations found and produce fixed versions of the original Dockerfile.

3. We analyze again the Dockerfile produced by the system and collect data about eventually not fixed misconfigurations.

4. We compare the data produced in point 1 with the ones produce in point 4 and summarize our findings in Table 6.2 and Table 6.1.

In Table 6.2 are highlighted the number of security misconfigurations found in both datasets by the system, the number of misconfigurations that we could repair and the number of misconfigurations that the system actually repaired. It is important to point out that with the term *Security misconfigurations* in Table 6.2, we refer to all the misconfigurations detectable by the system and listed in Section 4.2; instead with the term *Reparable misconfigurations* we refer to the misconfiguration that the system is able to detect *and fix*, which has been listed before.
We uncover that, with the *repair templates* that we have, we can potentially repair 95.19% of the misconfigurations in the Standard dataset and the 100% of the misconfigurations in the Gold one. After running the repairing process, the system has been able to actually repair 90.05% security misconfigurations in the Standard dataset and 78.26% misconfigurations in the Gold dataset. We discuss the results and the limitation in Section 6.2 but we can mention that the missed repairs are mainly due to imperfect grammar rules of the parser.

We further investigate problems occurred during the repairing phase and outline the findings in Fig. 6.1. In this chart, we list the reparable misconfiguration categories and, for each of them, we report the number of misconfigurations that the system has been able to repair. For the Standard dataset, the misconfiguration "Updating in different RUN instructions" has been most difficult to repair. On the other hand, almost every misconfiguration in the "Missing USER instruction" categories have been fixed.
Regarding the Gold dataset, it contains only two type of misconfigurations, as already mentioned in Section 4.2: "Installation of additional packages" and "Missing USER instruction". Only half of the former have been repaired, instead all the latter have been fixed.

We evaluate the repairing process considering the number of fixed Dockerfile as well, and summarize the findings in Table 6.1. In our findings, we define an *affected Dockerfile* as a Dockerfile which contains at least one of the misconfigurations detectable by the system and listed in Section 4.2. We refer to a *reparable Dockerfile*

	Standard	Gold
Affected Dockerfile	158.498 (100%)	152 (100%)
Reparable Dockerfile	157.758 (99.53%)	152 (100%)
Completely repaired Dockerfile	144.532 (91.18%)	119 (78.28%)
Partially repaired Dockerfile	7.268 (4.58%)	4 (2.63%)
No repaired Dockerfile	1.419 (0.89%)	0 (0.0%)
Broken Dockerfile	4.539 (2.86%)	29 (19.07%)

Table 6.1: Repaired Dockerfile in both datasets

	Standard	Gold
Security misconfigurations	315.883 (100%)	161 (100%)
Repairable misconfigurations	300.675 (95.19%)	161 (100%)
Repaired misconfigurations	284.469 (90.05%)	126 (78.26%)

Table 6.2: Repairable misconfigurations in both datasets

as a Dockerfile which contains at least one reparable misconfiguration. Furthermore, we consider a Dockerfile *partially* fixed if at least one security misconfiguration has been repaired; instead, we consider a Dockerfile *completely* fixed if all the security misconfigurations have been repaired.
Finally, we consider a Dockerfile *broken*, if, after the repairing phase, some syntax issues have been introduced.
The findings reported in Table 6.1 pointed out that the *repair templates* in place are able to potentially repair 99.53 % of Dockerfile for the Standard dataset and the entire Gold dataset (100%).
After the fixing process, in the Standard dataset, 91.18% of the artifacts do no longer contain none of the fixable misconfigurations while in the Gold dataset, 78.28% of Dockerfile have been completely fixed. Some Dockerfile have been partially repaired (4.58% for the Standard dataset and 2.63% for the Gold one); instead, 0.89% of the file in the Standard dataset contains the same fixable misconfigurations as before the repairing. The system shows major limitations with some Dockerfile: in fact, some of them have been irreversibly broken by the introduction of some syntax errors. In particular, this issue affects a significant part of the Gold dataset (19.07%).

Moreover, in order to further verify the validity of the repairs, we perform a test

	Standard	Gold
Analyzed Dockerfile	1.702 (100%)	152 (100%)
Dockerfile buildable before repairing	275 (16.15%)	38 (25%)
Dockerfile buildable after repairing	275 (16.15%)	38 (25%)

Table 6.3: Results about buildable Dockerfile

involving the building process.
The aim of the experiment, is to prove that the repairing process does not affect the semantic of the Dockerfile. In other words, we want to know whether the repairing process prevents a Dockerfile to be built. In this test, two factors have to be highlighted. First, we run the test only on the Dockerfile that can be built: as we mention in Chapter 4, both datasets come without building context (a component described in Section 2.2) and this reason does not allow us to build images out of all the Dockerfile. For some Dockerfile, context file are not needed, thus we can still build them in order to verify whether our repairs cause some issues. In this sense, a *buildable* Dockerfile is an artifact which can be built without any other file. Moreover, due to performance issues described in Section 6.2, we have not been able to perform this experiment for the entire Standard dataset. In that case, we randomly choose a certain amount of Dockerfile.
The results are summarized in Table 6.3: the *Analyzed Dockerfile* are the artifacts that we take into account. Specifically, we consider the entire Gold dataset (152 file) and 1702 file for the Standard dataset. Among the *Analyzed Dockerfile*, only part of them can be built without building context (Docker component described in Chapter 2), in particular 16.15% for the Standard and 38% of the Gold Dataset.
Finally, *Dockerfile buildable after repairing* represents the number of Dockerfile that, after the repairing process, are still buildable; in this case, all the previously buildable Dockerfile, are still buildable after the process.

6.2 Discussion and Limitations

One of the goal of this work, mentioned in Section 1.1, is to assess the effectiveness of the developed system described in Chapter 5. The tests conducted on the system show a good overall performance in the Dockerfile repairing process; as we outline in Table 6.2, approximately 90% of the reparable misconfigurations have been fixed. Moreover, approximately 9 Dockerfile out of 10 don't contain any of 5 security issues highlighted in Section 6.1. Clearly, the system reaches its initial goal of making Dockerfile more safe without involving any manual activities during the process.
The repairing process is proved to be not disruptive for generated Dockerfile: only 2.86% of the produced Dockerfile come with syntax issues and; also, their semantic have not been altered as proved by the number of still buildable Dockerfile.
Furthermore, the modular and flexible architecture described in Section 5.1, allows developers to integrate its custom detection and repairing processes, further increasing system capabilities and adapting it to particular needs.
Nevertheless the good effectiveness, the system shows the limitations that we summarize below.

Low number of reparable misconfigurations As we already mention in Section 4.4, the system is able to repair only a subset of all the security misconfigurations listed in Section 2.5. This limitation is strictly tied with the capabilities of existing

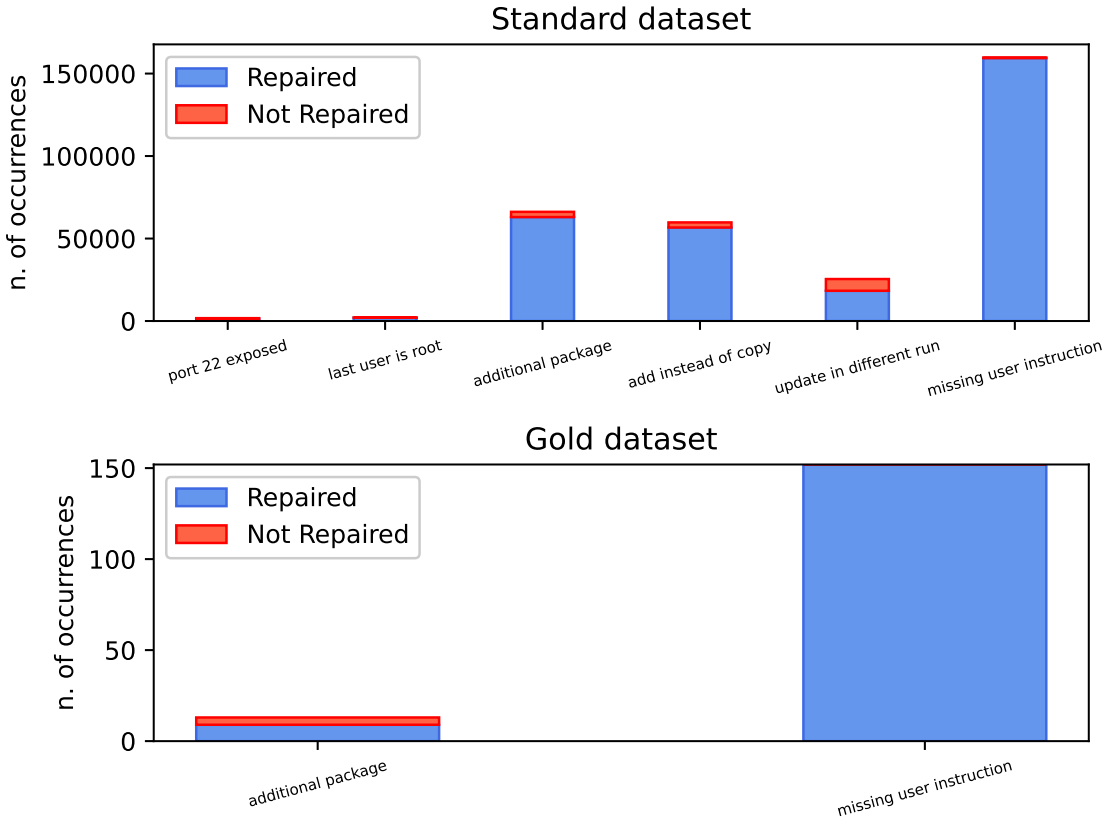

Figure 6.1: Repairing statistics in both datasets

tools: since the system uses external scanners to detect misconfigurations, if none of them is able to find a particular misconfiguration, similarly the system can not. We demonstrate that it is possible to overcome this limitation by implementing custom detection processes. We do this with the detection of custom secrets: in fact, none of the existing tools, specified designed for detecting misconfigurations in Dockerfile, have this capability; Despite this, the system is able to detect them due to a custom detection process.

Using human written templates It could be claimed that the system still need human experts to write the templates, so that the approach is not completely automatic. As we already mention in Section 3.3, the choice of using human written templates over a machine learning based system comes with benefits and drawbacks. Having static templates definitely contributes to have precise corrections without the effort of training AI models. However, the number of issues fixable might be lower than solutions which adopt some kind of ML system because of the absence of an active learning process.
In the concrete case of this system, aware of this, we have chosen the static template based approach considering two factors: the desired precision and the data that we have at our disposal. Regarding the latter, the Gold dataset could have been a good foundation for the learning system, but the number of file is too little. On the other hand, as we point out in Section 4.2, the Standard dataset contains a large amount of misconfigurations to be considered as training dataset. Regarding the former, the results obtained from the tests confirm that the system is precise in repairing misconfigurations, being able to get rid of the majority of misconfiguration while not disrupting the fixed version of Dockerfile.

Parsing issues One of the major limitation the system is affected of, concerns the parsing process: we already mention this issue in Chapter 4 where we state that we filter out some Dockerfile because of the inability of parsing them. The root of the problem lies in the grammar rules of the parser described in Section 5.2.1. While they can parse the majority of Dockerfile and embedded Bash code, there are still some specific patterns which are impossible to properly parse. This inability led to produce a wrong AST and thus to generate back a wrong Dockerfile. Moreover, some rules had been written in such a way that do not ease the process of getting back a new Dockerfile.
We try to limit the drawbacks by manually modified existing rules and adding new ones for our own purposes. Nevertheless, this job is complex and time-consuming since the rules are strictly tied.
It is worth pointing out that these limitations mostly affect the Gold dataset, since the Dockerfile in it are, on average, more complex in terms of number of instructions and embedded Bash code used.

Building issues When we set up the test described in Section 4.3 that should prove that the repairing does not affect the building process, we mention that we have not been able to conduct it on the entire Standard Dataset. This limitation comes from the fact that building a Dockerfile could be a time-consuming task, and it is unfeasible to perform it for thousands Dockerfile. Moreover, DockerHub registry puts a limit on the number of pull requests that could be done in limited amount of time. Despite this, we consider the results obtained enough to prove our point.

6.3 Future work

A good starting point for eventual future works on the system, should be overcoming the existing limitations described in Section 6.2. In particular, we mention the issues with the parser: complete and coherent grammar rules to be used inside the parser would be highly important in order to be able to use the system on Dockerfile particularly complex in terms of number of instruction and embedded Bash code. Furthermore, it would be fundamental in order to automate repairing process for large amount of Dockerfile as we do in the tests described in Section 6.1.
Future works could also focus on exploiting the system architecture and enhancing its capabilities; in fact, it can be easily integrated new scanners and new repair templates in it. A valuable aim in this sense should be implementing detecting and repairing processes in order to fully cover the security best practices exposed in Section 2.5.
It is also possible to replace the way it detects and repairs: the system could be a good framework to experiments new ways of detection and fixing by using machine learning based approaches as described in Section 3.3 instead of the current template based approach. In this sense, a viable approach is to mix the two techniques, thus being able to automatically generate new repair templates.
Apart from the system developed, given the low awareness about Dockerfile security best practices, proved in Section 4.2, we believe that the topic is open to further implement new systems and methodologies, in order to make the Dockerfile hardening process evermore easy and automatic. Unfortunately, with the growing use of this type of virtualization technology mentioned in Chapter 1, securing Docker artifacts is not an optional choice, but an important imperative.

Chapter 7

Conclusion

The growing usage of containers technologies, in particular Docker, represents a great possibility to ease development and deployment phases; but it does not come without security risks. In this sense, an important aspect for container security is the building of secure images.
This work took its cue from these considerations and proceeded with the intention to evaluate the most recent findings in security misconfigurations detection and repair in Dockerfile. Through different experiments, we were able to determine the extent of this kind of misconfigurations in real-world artifacts. Moreover, we were also able to compare these results with Dockerfile written by experts. Our findings show an overall low degree of awareness since the majority of Dockerfile analyzed do not follow the current security best practices compared with the ones written by experts. The consequence is to possibly expose containers to severe risks.
Unfortunately, our researches show that there are no tools able to replace the manual work of hardening Dockerfile, since every current tool is developed to just detect misconfigurations. Moreover, these tools lacks of the ability to detect most of the security issues that could affect Dockerfile.
With this work, we tried to fill this gap by developing a system which allow, not only to detect, but also to fix security issues in Dockerfile.
The tests performed on the system show an overall good performance, being able to repair the majority of the Dockerfile taken into account.
We strongly believe that the developed system could be a first step toward the creation of a more sophisticate and complete repairing software for Dockerfile. For this reason, as a final contribution, we proposed some works that could be done, in order to further ease the writing of secure Dockerfile by enhancing the tool developed.

Chapter 8

Appendix

In this chapter, we report technical details that we have previously omitted for the sake of simplicity.

8.1 Improvements in Bash and Dockerfile grammar rules

During the system implementation described in Chapter 5, we mention the limitations of the parsing library, *tree-sitter*, used to build the AST. In order to be able to correctly parse the Dockerfile and build the AST, we perform some adjustments to both Bash and Dockerfile grammar rules. Specifically, we modify the *grammar.json* file in the repositories [9] and [2].

8.1.1 Improvements in Dockerfile grammar rules

We modify the Dockerfile grammar rules for two purposes: to fix eventually issues and to make the AST easier to be converted back into a Dockerfile.

Making STOPSIGNAL command visible According to the original grammar rules, some node type is not visible in the final AST, STOPSIGNAL node type is one of these. In the grammar rule, we modify the properties of this node to make it visible in the final AST, thus, to be able to parse Dockerfile containing STOPSIGNAL instruction.

Creation of a new rule We create a new type in the grammar rule, named *unquoted_string_value* for the purpose of allowing variable expansion in the ENV instructions.

Limitations Nevertheless our efforts, modifying the entire grammar rules is a complex and time-consuming task, since the rules are strictly tied. For these reasons, some particular syntaxes and constructs in Dockerfile can not be corretly parsed.

For example, an instruction such as the one listed in Listing 8.1, can not be built because ONBUILD grammar rule does not expect a following RUN instruction.

```
ONBUILD RUN set -ex && pipenv install --deploy --system
```

Listing 8.1: Unquoted_string_value type

8.1.2 Improvements in Bash grammar rules

Bash grammar rules play an important role in the parsing of the Dockerfile, since they allow to further enhance the AST, with parsed Bash code. Despite Bash rules are more complete than Dockerfile ones, we perform some slightly modification to ease the conversion process.

Making *special_character* type visible In order to have special characters nodes during the creation of the new Dockerfile, we make visible (and thus usable) the *special_character* type, which is hidden by default. We do not consider these nodes as part of the original AST, since AST should not contain syntax elements, according to the definition in Section 2.1. Nevertheless, we use them to complement the conversion of the AST back to a Dockerfile.

Fix comment rule We fix an issue involving parsing comments in Bash. In the fixed version, we add of an end-line characters in the parsing sequence. This is because without that, the line following the comments is considered a comment line too.

Limitations Despite the completeness of this grammar, during our tests we hit some limitations by parsing certain Bash scripts. As example, the presence of comments inside the "apt-get install" Bash command, make it impossible to obtain the right AST. As we already discuss for the Dockerfile grammar rules, completely modify these rules is a complex task. Nevertheless, our tests confirm that these limitations are acceptable, since we are able to parse the majority of the Dockerfile in the datasets.

Bibliography

[1] Johannes Bader et al. "Getafix: Learning to Fix Bugs Automatically". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360585. URL: https://doi.org/10.1145/3360585.

[2] *Bash grammar rules for tree-sitter*. URL: https://github.com/tree-sitter/tree-sitter-bash (visited on 06/13/2022).

[3] Thanh Bui. "Analysis of docker security". In: *arXiv preprint arXiv:1501.02967* (2015).

[4] *CIS Docker Benchmark*. URL: https://www.cisecurity.org/benchmark/docker (visited on 07/10/2022).

[5] *Container and Kubernetes: market dynamics report, 2021*. URL: https://www.redhat.com/rhdc/managed-files/cl-containers-kubertnetes-market-dynamics-f29518-202106-en.pdf (visited on 06/13/2022).

[6] Stefano Dalla Palma et al. "Within-Project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics". In: *IEEE Transactions on Software Engineering* (2021), pp. 1–1. DOI: 10.1109/TSE.2021.3051492.

[7] *Docker documentation*. URL: https://docs.docker.com/ (visited on 07/10/2022).

[8] *Docker Storage Driver*. URL: https://docs.docker.com/storage/storagedriver/ (visited on 07/10/2022).

[9] *Dockerfile grammar rules for tree-sitter*. URL: https://github.com/camdencheek/tree-sitter-dockerfile (visited on 06/13/2022).

[10] Khashayar Etemadi et al. "Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations". In: *ArXiv* abs/2103.12033 (2021).

[11] Jean-Rémy Falleri et al. "Fine-Grained and Accurate Source Code Differencing". In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden: Association for Computing Machinery, 2014, pp. 313–324. ISBN: 9781450330138. DOI: 10.1145/2642937.2642982. URL: https://doi.org/10.1145/2642937.2642982.

[12] Hideaki Hata, Emad Shihab, and Graham Neubig. "Learning to generate corrective patches using neural machine translation". In: *arXiv preprint arXiv:1812.07170* (2018).

[13] Jordan Henkel et al. “A dataset of dockerfiles”. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020, pp. 528–532.

[14] Jordan Henkel et al. “Learning from, understanding, and supporting DevOps artifacts for Docker”. In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE. 2020, pp. 38–49.

[15] Jordan Henkel et al. “Shipwright: A Human-in-the-Loop System for Dockerfile Repair”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2021, pp. 198–199. DOI: 10.1109/ICSE-Companion52605.2021.00087.

[16] Joel Jones. “Abstract syntax tree implementation idioms”. In: *Proceedings of the 10th conference on pattern languages of programs (plop2003)*. 2003, pp. 1–10.

[17] Dongsun Kim et al. “Automatic Patch Generation Learned from Human-Written Patches”. In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE ’13. San Francisco, CA, USA: IEEE Press, 2013, pp. 802–811. ISBN: 9781467330763.

[18] Yi Li, Shaohua Wang, and Tien N. Nguyen. “DLFix: Context-Based Code Transformation Learning for Automated Program Repair”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE ’20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 602–614. ISBN: 9781450371216. DOI: 10.1145/3377811.3380345. URL: https://doi.org/10.1145/3377811.3380345.

[19] Kui Liu et al. “AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations”. In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2019), pp. 1–12.

[20] Zhigang Lu et al. “An Empirical Case Study on the Temporary File Smell in Dockerfiles”. In: *IEEE Access* 7 (2019), pp. 63650–63659. DOI: 10.1109/ACCESS.2019.2905424.

[21] Wanli Ma et al. “Password entropy and password quality”. In: *2010 fourth international conference on network and system security*. IEEE. 2010, pp. 583–587.

[22] Diego Marcilio et al. “SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings”. In: *Journal of Systems and Software* 168 (2020), p. 110671. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110671. URL: https://www.sciencedirect.com/science/article/pii/S016412122030128X.

[23] MITRE. *Common Weakness Enumeration*.

[24] Nigel Poulton. *Docker Deep Dive*. O’Reilly, 2020.

[25] Akond Rahman, Chris Parnin, and Laurie Williams. "The Seven Sins: Security Smells in Infrastructure as Code Scripts". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 164–175. DOI: 10.1109/ICSE.2019.00033.

[26] Akond Rahman and Laurie Williams. "Characterizing Defective Configuration Scripts Used for Continuous Deployment". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 2018, pp. 34–45. DOI: 10.1109/ICST.2018.00014.

[27] Akond Rahman and Laurie Williams. "Different Kind of Smells: Security Smells in Infrastructure as Code Scripts". In: *IEEE Security Privacy* 19.3 (2021), pp. 33–41. DOI: 10.1109/MSEC.2021.3065190.

[28] Jerome H. Saltzer and Michael D. Schroeder. *The Protection of Information in Computer Systems*. 1975.

[29] Spyridon Samonas and David Coss. "The CIA strikes back: Redefining confidentiality, integrity and availability in security." In: *Journal of Information System Security* 10.3 (2014).

[30] Michele Tufano et al. "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 832–837.

[31] David S Wile. "Abstract syntax from concrete syntax". In: *Proceedings of the 19th international conference on Software engineering*. 1997, pp. 472–480.

[32] Yiwen Wu et al. "Characterizing the Occurrence of Dockerfile Smells in Open-Source Software: An Empirical Study". In: *IEEE Access* 8 (2020), pp. 34127–34139. DOI: 10.1109/ACCESS.2020.2973750.

[33] Yang Zhang, Huaimin Wang, and Vladimir Filkov. "A clustering-based approach for mining dockerfile evolutionary trajectories". In: *Science China Information Sciences* 62 (2018), pp. 1–3.

[34] Yang Zhang et al. "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. DOI: 10.1109/COMPSAC.2018.00026.