University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

DIPLOMA

Minimizing Dockerfiles

Scientific Adviser:
Prof. Răzvan Deaconescu

Author:
Andrei Petrea

Bucharest, 2025

I want first and foremost to thank my advisor, Prof. Răzvan Deaconescu, for his guidance and support throughout this project. His expertise and insights have been invaluable in shaping my understanding of the subject matter and in helping me navigate the challenges I faced during this endeavor.

I also want to thank my friends and family without whom these 4 years would have been so much harder.

Contents

Acknowledgements **i**

1 Introduction **1**

2 Background **2**
2.1 A brief history of containers 2
2.2 The anatomy of a container 2
2.3 Related work 4

3 Motivation and Objectives **6**

4 Use Cases **7**

5 Building Blocks **9**

6 Architecture Overview **11**
6.1 Static Analysis 11
6.2 Dynamic Analysis 11
6.3 Brute Force 12

7 Implementation Details **13**
7.1 Overview 13
7.2 Project Structure 13
7.3 Preprocessing 14
7.4 Static Analysis 15
7.5 Dynamic Analysis 17
7.6 Binary Search 19

8 Performance Evaluation **24**

9 Status and Planned Work **27**
9.1 Status 27
9.2 Planned Work 27

List of Figures

2.1 Containers vs VMs 2
2.2 Steps to containerize an application from a Dockerfile 3
2.3 Container layers 4
2.4 Docker architecture 4

4.1 High-level comparison of the software components of traditional VMs (a), containers (b), containers in VMs (c) with unikernels (d). 7

5.1 High-level architecture of the application 9

6.1 Brute force approach to finding the minimal Dockerfile 12

7.1 Language execution speed comparison [7] 13
7.2 Dynamic analysis using strace to determine the files used by the command . . . 17
7.3 Adding a tarball to the Dockerfile 22

8.1 Distribution of the results by stage of minimization 24
8.2 Size reduction per stage of minimization 25
8.3 Size reduction per application 25

List of Listings

2.1 Sample Dockerfile . . . 3
2.2 Sample Multi-stage Dockerfile . . . 3
4.1 Sample Kraftfile . . . 7
6.1 ldd command . . . 11
6.2 strace command . . . 11
7.1 Project Structure . . . 13
7.2 Go DockerConfig structure . . . 14
7.3 Dockerfile.minimal.template . . . 15
7.4 ldd parser function . . . 15
7.5 File util functions . . . 16
7.6 File util functions . . . 17
7.7 File util functions . . . 18
7.8 Filesystem parsing . . . 19
7.9 Binary search function . . . 20
7.10 Splitting the filesystem . . . 22

Chapter 1

Introduction

"But ... it works on my machine..." is a phrase no developer wants to hear. It is a phrase that highlights both the frustration of software engineers and the complexity of the software development life cycle. The most common cause for this issue arising lies in the mismatch between the developer's local environment and the production environment where the code is run [17]. On traditional deployments on a physical server or virtual machine, it is up to the developer to ensure that all the necessary configurations are met for the application to run, which can be a tedious process, especially when the the number of dependencies is high or when we want to upgrade to a newer version.

The solution? ... Using containers. Containers are software that package up code, runtime, libraries, config files, everything needed for an application to run [14] which can then be executed across different machines without additional setup. This, alongside faster boot times, lower resource usage and ease of scale, especially when coupled with orchestration tools like Kubernetes, made containers grow in popularity, even surpassing virtual machines in some cases [22].

However, the comparison with virtual machines, where containers are usually much better, often paints the picture that these metrics are irrelevant when it pertains to them, one such metric being size of the container itself. This is not true. The size of the container is directly proportional to the number of binaries and packages that are installed and as such reducing the size has the following benefits [9]:

- **Security** - smaller the size, smaller the attack surface a hacker has to work with
- **Performance and Efficiency** - smaller images are faster to deploy and, in general, use fewer system resources
- **Maintainability** - smaller images have fewer dependencies, making it easer to maintain and update them

As such, the goal of my project is to create a tool than can strip a container of all its unnecessary files and packages, leaving only the files needed for the application to run properly and export it as a *Dockerfile*, the recipe used to create the container.

Chapter 2

Background

2.1 A brief history of containers

The idea of containerization is not a new one. The concept has it's roots since the late 70s, with *chroot*, a Unix command that allows a process to change its root directory, effectively isolating it from the rest of the system [3], creating so called *chroot jails*. Over the decades, this concept grew and evolved, with the introduction of *FreeBSD jails* and *Solaris Zones* around the turn of the millennium adding support for multiple isolated environments within the same OS instance [3].

The next major milestone happened in 2008 with the introduction of *LXC* (Linux Containers), adding kernel-level support for containers, by leveraging two Linux kernel components: *cgroups*, which provides ways to group processes in order to better manage resources and *namespaces*, which provide isolation.[3]

In 2013, the launch of *Docker* radically changed the landscape of containerization by providing a developer-friendly way to create, manage and deploy containers. Docker introduced the concept of container images, files which store the data needed for the container to run, *Docker Hub*, a public repository for sharing container images, and a powerful command-line interface for managing containers [3]. In the years that followed, multiple platforms sprung up such as *Apptainer* and *Podman*, which are both open-source alternatives to Docker, but none of them managed to gain the same level of popularity. As such, Docker became synonymous with containerization, holding an overwhelming market share of *87.85%*[1] thus cementing themselves as the de facto standard in the industry.

2.2 The anatomy of a container

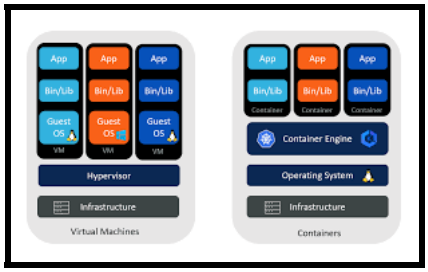

Figure 2.1: Containers vs VMs

As opposed to virtual machines, which run a full operating system on top of a hypervisor, containers share the host operating system and run as isolated processes making them much more lightweight and efficient by only having to package the application and its dependencies into a so-called container image. Currently, there are over 10000 container images available on Docker Hub, ranging from simple *hello-world* apps, to *databases* and even full-blown Linux distributions like *Ubuntu*.

Now, how do we create our own container image? By creating a *Dockerfile*, a simple text file that contains all the necessary steps to build the image.

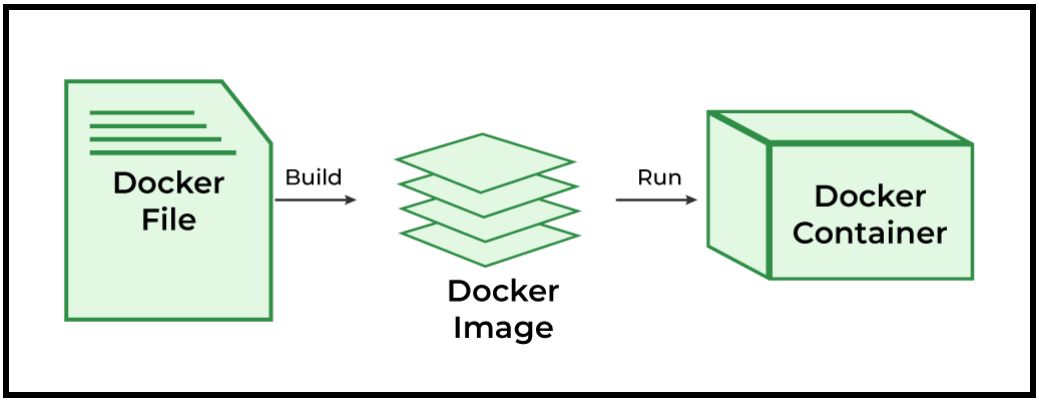

Figure 2.2: Steps to containerize an application from a Dockerfile

```
1 FROM python:3.10.0-slim
2 COPY requirements.txt .
3 RUN pip install -r requirements.txt
4 COPY . .
5 CMD ["python", "adapter.py"]
```

Listing 2.1: Sample Dockerfile

The format of a Dockerfile is quite simple, with each line representing a command that will be executed in order, the specifications for what each command does can be found in the official documentation[1] As each command is executed, a new layer is applied on top of the previous one, the container image being represented as a stack of these layers. A layer represents a change to the base image layer such as adding or removing files, installing packages or changing environment variables.

All Dockerfiles require a base image, which is specified by the keyword *FROM*. The base image can be represented by either an existing image, either local or remote , or if you want to create your own, you can use ***FROM*** *scratch* to start from an empty filesystem.

Additionally, Docker supports building the container image in multiple stages, by using the *FROM* command multiple times in the same Dockerfile.

```
1     FROM alpine:latest AS builder
2     RUN apk --no-cache add build-base
3
4     FROM builder AS build1
5     COPY source1.cpp source.cpp
6     RUN g++ -o /binary source.cpp
7
```

[1] https://docs.docker.com/reference/dockerfile/

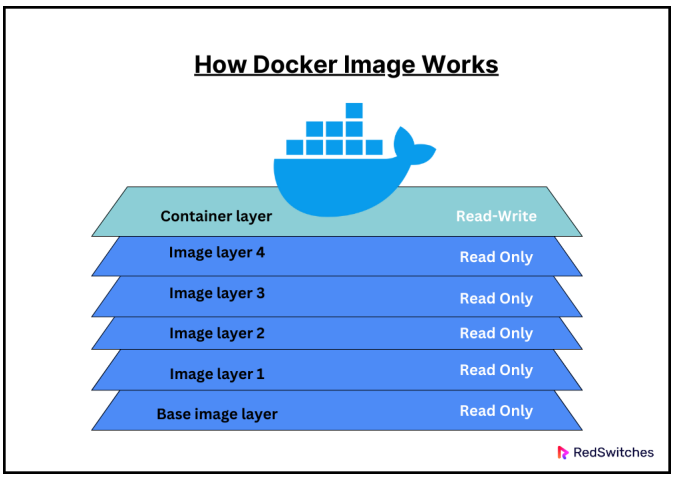

Figure 2.3: Container layers

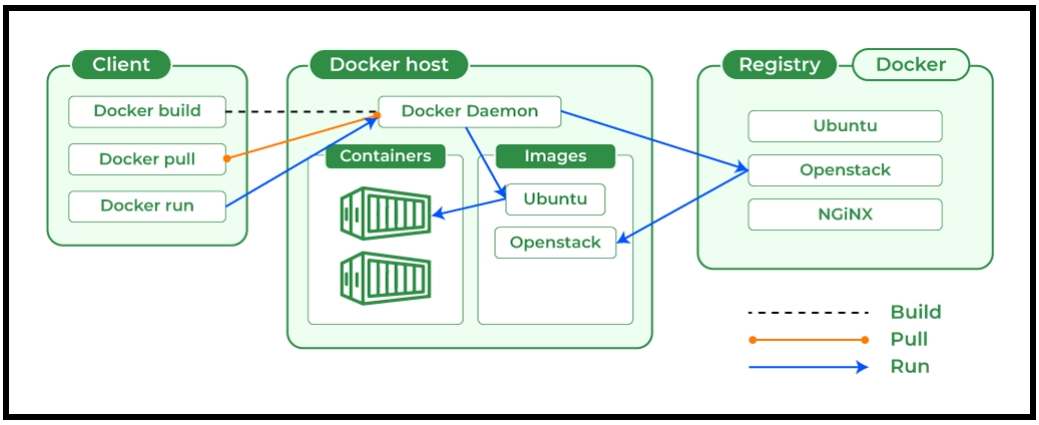

Figure 2.4: Docker architecture

```
8     FROM builder AS build2
9     COPY source2.cpp source.cpp
10    RUN g++ -o /binary source.cpp
```

Listing 2.2: Sample Multi-stage Dockerfile

This coupled with the *scratch* base image serves as the mechanism that allows for the creation of minimal container images, by defining first a builder image and then copying the required files to the empty image.

2.3 Related work

Before proceeding further, it is important to acknowledge the work done by my peers in the field of container image minimization.

The earliest mention of building minimal container images is in a blog post dated 4th of July 2014 by Adriaan de Jonge on the site Xebia [6] and is corroborated by a 2015-02-03[1] München

[1] not sure if it's the 2nd of March or 3rd of February as I could not find this presentation and the only mention of this is in his GitHub repository [12]

talk by Brian Harrington [12] which describe using *tar* to create the *scratch* image and then copying the required files to it, or by using tools like *buildroot*[1] or *debootstrap*[2] to create a minimal Linux distribution and then copying the required files to it.

In 2016, the first Alpine Linux image was published, which had a compressed size of 1.86MB[3] as it was built using *musl* and *busybox*, two lightweight alternatives to the standard C library and coreutils respectively. Around the same time, the *Distroless* philosophy i.e images containing only the application and its dependencies, was introduced by Google, with the first of these images being build using the open source tool *Bazel* for languages such as Java, Go and Python. Recently, Canonical, the company behind Ubuntu, has embraced distroless with their *Chiseled*[4] ubuntu images.

However, none of these tackle the fundamental issue of using minimal container images, being that the developer has to manually determine the dependencies of the application, which can be a arduous task, especially for large applications. The only project that I could find that is tangentially related to this is *dockershrink* by developer Raghav Dua [8]. This tool utilizes Artificial Intelligence in order reduce the size of the container image by generating a new Dockerfile updated to use slimmer base images and removing unnecessary files. [8] However, this tool is still in beta and it only works for NodeJS applications, as well as having to provide it with a OpenAI API key in order to access the full functionality of the tool. Additionally, it does not utilize the *scratch* image, which means that the final container image generated by the shrunk Dockerfile is not the most minimal possible.

[1] Buildroot
[2] debootstrap
[3] Oldest Alpine Linux image on Docker Hub
[4] Chiseled

Chapter 3

Motivation and Objectives

The main advantages of minimal containers are their enhanced security and small image size. Their improved security comes from the inherent properties of being minimal, meaning:

- **minimized attack surface** - by having only the required dependencies for the application to run, the only attack vector a hacker has is the application itself and not other components
- **clear dependency tree** - with only the required dependencies, it is easier to identify and audit them in case of a vulnerability being discovered, [2]

Additionally, their small size means that they are faster to deploy and use fewer system resources like *CPU* and *memory*, which is especially important in a cloud environment where the cost is directly proportional to the resources used and shaving a couple of seconds off the deployment time can lead to hours, even days given the size of the cluster. [16].

By creating a tool that can automatically detect an application's dependencies and create the Dockerfile which produces the minimal container for that app, we can save developers the time and effort of having to do it themselves, which can be tiresome and frustrating process.

Chapter 4

Use Cases

A real world example for the need to generate these minimal Dockerfiles and the reason that spawned this project is *Unikraft*[1].

Unikraft was envisioned as a faster and more secure alternative to running applications in containers or virtual machines, by leveraging the power of ultra-lightweight virtual machines knows as unikernels [21].

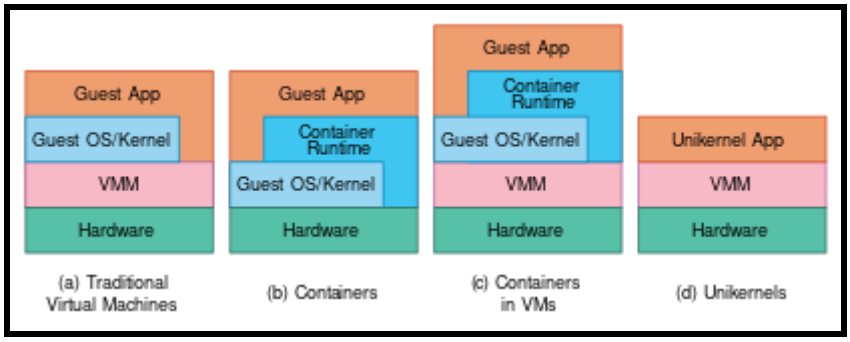

Figure 4.1: High-level comparison of the software components of traditional VMs (a), containers (b), containers in VMs (c) with unikernels (d).

Unikernels are specialized operating systems that are built as single-address space binary objects, meaning that they don't have user space-kernel space separation. They combine both the complete execution isolation and hardware access typical of VMs with the size of containers, which makes them ideal for cloud deployments [21].

Unikraft provides the tools needed to build, run and manage unikernels, in the form of a command-line program called *kraft*[2]. It features an interface similar to that of Docker's, with it being able to run pre-build unikernels or create new ones from a file called *Kraftfile*[3].

```
1    spec: v0.6
2    runtime: base:latest
3    rootfs: ./Dockerfile
4    cmd: ["/helloworld"]
```

Listing 4.1: Sample Kraftfile

[1] https://unikraft.org/
[2] https://unikraft.org/docs/cli
[3] also *kraft.yaml* or *kraft.yml* for legacy support

Kraftfiles are written in YAML[1], which follows an attribute-value structure. For my thesis, the only relevant attribute is *rootfs*, which describes the source from which the unikernel's root filesystem is built. Although optional [20], since most applications do require one, it is usually specified. As unikernels operate in resource-constrained specialized environments where efficiency is crucial, so do their filesystems need to strike the balance between providing the necessary storage requirements and also adhering to their lightweight philosophy.

There are many ways to define a root filesystem, one of which being highlighted in the Kraftfile example above, where a Dockerfile is used to build a container whose filesystem is later extracted to be used by the resulting unikernel. Thus, the Dockerfile should be as minimal as possible. So far, this process was done manually[2], which explains the rather low number of applications that have been ported to Unikraft[3], which need constant maintaining as different versions of the same app may require different Dockerfiles.

[1] https://yaml.org/
[2] https://unikraft.org/docs/contributing/adding-to-the-app-catalog
[3] https://github.com/unikraft/catalog

Chapter 5

Building Blocks

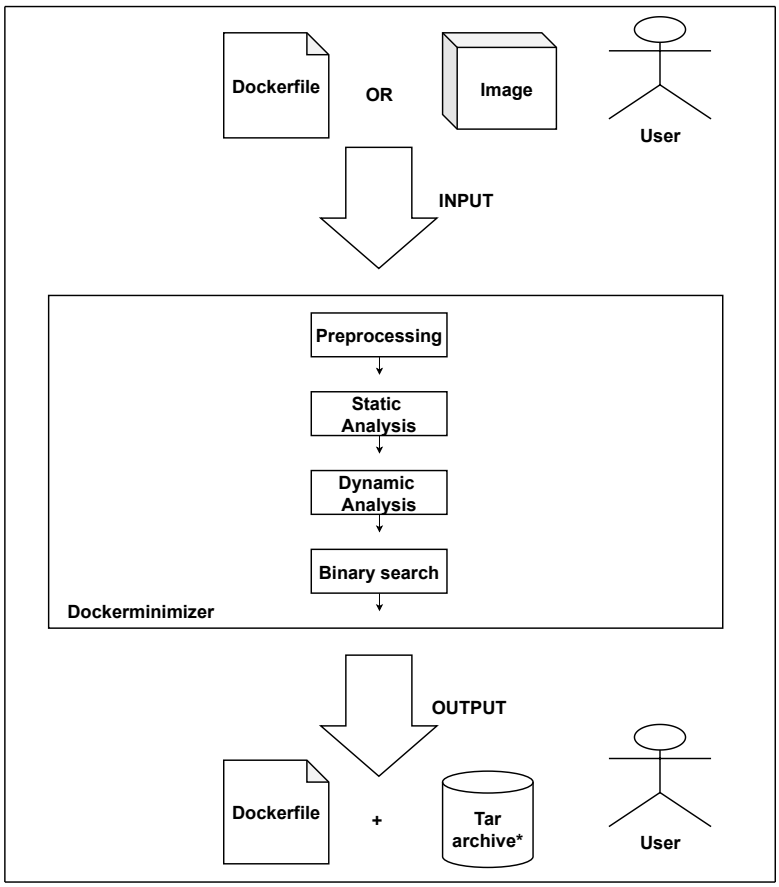

Figure 5.1: High-level architecture of the application

My solution is a command-line tool that automates the process of determining the runtime dependencies of a given application and generating the minimal Dockerfile.

For finding the runtime dependencies, I will be following the steps outlined here [19] and here [18] and will be a three-pronged approach, consisting of:

- **Static analysis** - inspecting the executable file for any linked libraries.
- **Dynamic analysis** - running the application and tracing its system calls [1].
- **Brute force** - a fail-safe incase the other two fail to yield any results.

The handling of the interaction between the program and Docker is done through the the *Docker SDK*[1], which provides a programmatic way to interact with containers. It is available both for Python and Go, both very popular languages.

Go is a statically-typed, compiled language known for its simplicity and efficiency. It was built by Google in 2007 for use in networking and infrastructure services, being designed to be efficient, readable and high-performing. [4]

Python is a high-level, interpreted language known for its simplicity and readability. It was created by Guido van Rossum in the late 1980s and has since become one of the most popular programming languages in the world. [10]

Although both languages are capable of achieving the same results, Go's performance superiority over Python, coupled with its concurrency and low-level capabilities, make it the clear choice for this project. The choice of using Go is also motivated by the fact that Docker itself is written in Go [13], which means that the SDK is better integrated with the rest of the Docker ecosystem and has better performance than the Python SDK.

[1] system calls, *syscalls*, are the mechanism used by applications to request services from the kernel, like I/O, spawning processes, etc.

[1] https://docs.docker.com/reference/api/engine/sdk/

Chapter 6

Architecture Overview

6.1 Static Analysis

Static analysis is the process of analyzing a program's code without executing it. In our context, we will be using it to identify the libraries that the application is dynamically linked using the *ldd* command.

```
~ > ldd /usr/bin/man
    linux-vdso.so.1 (0x00007ffe831f5000)
    libmandb-2.9.1.so => /usr/lib/man-db/libmandb-2.9.1.so (0
        x00007f068b572000)
    libman-2.9.1.so => /usr/lib/man-db/libman-2.9.1.so (0
        x00007f068b52f000)
    libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1 (0x00007f068b502000)
    libpipeline.so.1 => /lib/x86_64-linux-gnu/libpipeline.so.1 (0
        x00007f068b4f1000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f068b2ff000)
    libgdbm.so.6 => /lib/x86_64-linux-gnu/libgdbm.so.6 (0
        x00007f068b2ef000)
    libseccomp.so.2 => /lib/x86_64-linux-gnu/libseccomp.so.2 (0
        x00007f068b2cb000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f068b59b000)
```

Listing 6.1: ldd command

The output of the command is a list of shared libraries that the application depends on, along with their paths. This approach works well for simple applications but it quickly becomes insufficient for most production applications.

6.2 Dynamic Analysis

Dynamic analysis is the process of analyzing a program's behavior during its execution. Our use case is to identify the other files that the application accesses during its execution, as well any spawned processes and their dependencies. This is done using the *strace* command, which uses the *ptrace* system call to trace the system calls made by a process.

```
~ > strace -fe execve,openat echo "Hello, World\!"
execve("/usr/bin/echo", ["echo", "Hello, World!"], 0x7fff43b04ce8 /* 36
    vars */) = 0
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
```

```
4       openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC)
            = 3
5       Hello, World!
6       +++ exited with 0 +++
```

Listing 6.2: strace command

For our scenario, we are interested in the *openat*, which means that the application is trying to open a file, and *execve*, which means that the application is trying to execute another program. The *-f* flag is used to allow strace to also trace the child processes spawned by the application, thus ensuring that we do not omit any dependencies. Additionally, it can trace processes that are already running using the *-p* flag, which will be used to trace the process running inside the container, this however needing root privilege since the process inside the container is in another namespace.

6.3 Brute Force

The brute force approach is a last resort method that is used should both the static and dynamic analysis fail in generating the minimal Dockerfile. Using the Docker SDK, we can extract the filesystem from the container and use it for this step. Given that in the containerized environment, the application is running normally and in a *FROM scratch* environment it is not (which does not contain any files), therefore there exists a point where removing a file from the filesystem causes the application to fail. Since removing files one at a time and building a new container each time is not feasible, we have to attempt a more efficient approach, that being a *binary search*.

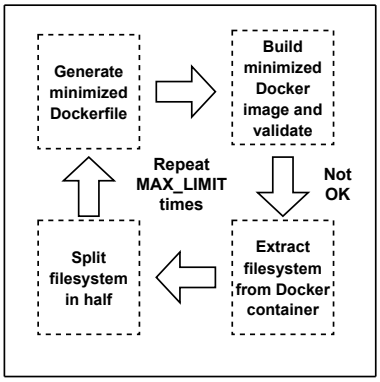

Figure 6.1: Brute force approach to finding the minimal Dockerfile

We start by splitting in half the filesystem into two buckets - *used* and *unused*. The decision of which files land in what bucket will be left to a coin toss in order to ensure fairness. We then build the Dockerfile with the files in the *used* bucket and run the application. If it fails, then we split the *unused* bucket in half and repeat the process. If the app succeeds, we repeat the cycle by splitting the shrunken filesystem in half again and follow the same steps. This will repeat at max MAX_LIMIT times, a predefined limit we impose on the number of iterations. This sifting process is a greedy one and thus will not always generate the minimal Dockerfile, but it will offer a good approximation of it, especially if the limit is set to a high value.

Chapter 7

Implementation Details

7.1 Overview

As stated previously, the implementation is written in Go, a statically typed, compiled language designed for simplicity and efficiency. The choice of Go was influenced by its support for both low-level systems programming and high-level programming paradigms, making it suitable for building robust and efficient applications. Additionally, Go is one of the faster languages in terms of execution speed and features a rich ecosystem of community-built modules [11], which can be leveraged to speed up development.

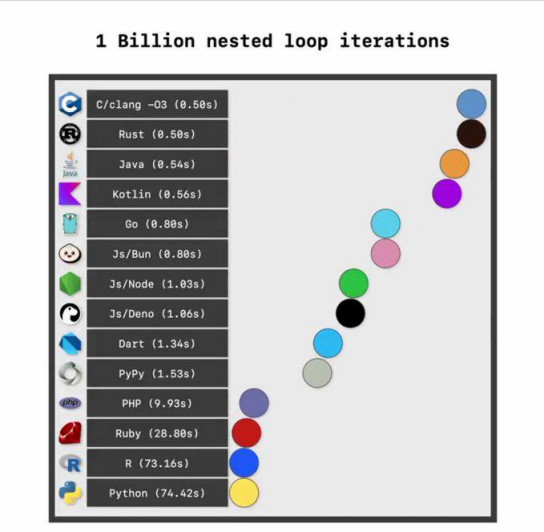

Figure 7.1: Language execution speed comparison [7]

7.2 Project Structure

The project is organized into multiple packages, each serving a specific purpose, as outlined in the architectural section.

```
1 .
2     |-- binary_search
```

```
3           |-- binary_search.go
4       |-- cmd
5         |-- main.go
6       |-- dockerminimizer
7       |-- dockerminimizer.go
8       |-- go.mod
9       |-- go.sum
10      |-- install.sh
11      |-- ldd
12        |-- ldd.go
13      |-- logger
14        |-- logger.go
15      |-- preprocess
16        |-- preprocess.go
17      |-- README.md
18      |-- strace
19        |-- strace.go
20      |-- types
21        |-- types.go
22      |-- utils
23        |-- utils.go
```

Listing 7.1: Project Structure

The executable created is *dockerminimizer*, which takes the following arguments:

- **–file**: The path to the Dockerfile to be minimized.
- **–image**: The name of an images from a Docker registry to be minimized
- **–max_limit**: How many times should the binary search procedure be run for
- **–debug**: Enable logging of actions
- **–timeout**: How long should the minimal Docker container run before being declared as successful
- **–strace_path**: The path to a statically-linked strace binary
- **–binary_search**: Decide whether to continue the minimization process with a binary search if dynamic analysis failed

7.3 Preprocessing

After the arguments are parsed, the program will create a temporary directory where the necessary files for the minimal Dockerfile will be stored under */.dockerminimizer/<unique_id>*. The unique ID is generated using a *MD5* hash of the current timestamp in order to ensure the uniqueness of the directory name should multiple instances of the program be run at the same time. The following steps are the building of the Docker image and the extraction of its filesystem to the directory. There are multiple ways of extracting a Docker container's filesystem [23], but the one we want to use is the one that preserves the original filesystem, without adding any additional files. This can be achieved by passing the *-o* flag to the build command, which will extract the unmodified filesystem to a desired location and type (the one we are interested in is tarball). The docker image is then parsed for its metadata, which is stored in a JSON file, which contains information about the image's environment variables, entrypoint, command, exposed ports and user. This metadata is stored as a custom type defined in the *types*.

```
1   type DockerConfig struct {
2       User         string                    `json:"User"`
```

```
        ExposedPorts map[string]map[string]any `json:"ExposedPorts"`
        Env          []string                  `json:"Env"`
        Cmd          []string                  `json:"Cmd"`
        WorkingDir   string                    `json:"WorkingDir"`
        Entrypoint   []string                  `json:"Entrypoint"`
}
```

Listing 7.2: Go DockerConfig structure

This is then used to create a template Dockerfile, which serves as a base for all the other Dockerfiles that will be generated.

```
FROM node:20.9.0 as builder

WORKDIR /app

COPY app.js /app/app.js
COPY package.json /app/package.json
COPY package-lock.json /app/package-lock.json

RUN npm install
EXPOSE 3000
CMD ["node", "app.js"]


FROM scratch

ENV PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
ENV NODE_VERSION=20.9.0
ENV YARN_VERSION=1.22.19
WORKDIR /app
EXPOSE 3000/tcp
ENTRYPOINT ["docker-entrypoint.sh"]
CMD ["node", "app.js"]
```

Listing 7.3: Dockerfile.minimal.template

This is then used to create the Dockerfile containing only the entrypoint/command files in order to check if it is already a minimal Dockerfile. Should it not be the case, the program will proceed to the static analysis phase.

7.4 Static Analysis

In the static analysis phase, the base Dockerfile default running command (i.e first entrypoint/-command in the image configuration) is run through *ldd* in order to determine the libraries that are required for the program to run. In order to ensure that ldd finds the correct libraries, the command is run utilizing *chroot* in the context of the extracted filesystem. The output of the command is then parsed and stored into two dictionaries, one for the normal files and one for the symbolic links, which need to be resolved to the actual files in order to be copied accurately into the Dockerfile.

```
    func ParseOutput(output []byte, rootfsPath string) (map[string][]string,
         map[string]string) {
        files := make(map[string][]string)
        symLinks := make(map[string]string)
        scanner := bufio.NewScanner(bytes.NewReader(output))
        for scanner.Scan() {
```

```
6               line := scanner.Text()
7               if strings.Contains(line, "=>") {
8                   parts := strings.Split(line, "=>")
9                   lib := strings.Split(strings.TrimSpace(parts[1]), " ")[0]
10
11                  if strings.Contains(lib, "not found") {
12                      continue
13                  }
14                  utils.AddFilesToDockerfile(lib, files, symLinks, rootfsPath)
15
16              } else if strings.Contains(line, "not found") {
17                  continue
18              } else {
19                  lib := strings.Split(strings.TrimSpace(line), " ")[0]
20                  utils.AddFilesToDockerfile(lib, files, symLinks, rootfsPath)
21              }
22          }
23          return files, symLinks
24      }
```

Listing 7.4: ldd parser function

The reason why we are storing the files in a dictionary where the key is the directory name is due to the Docker's limit of 127 build stages, meaning that there can be at most 127 *COPY* commands in the minimal Dockerfile. Fortunately, we can circumvent this by putting all the files that belong to the same directory in the same *COPY* command.

The way this is achieved is by utilizing Go's *filepath* package, which allows for easy manipulation of file paths.

```
1  func RealPath(path string) string {
2          realPath, _ := filepath.Abs(path)
3          return filepath.Clean(realPath)
4  }
5
6  func CheckIfFileExists(file string, envPath string) bool {
7          info, err := os.Stat(envPath + "/" + file)
8          return !os.IsNotExist(err) && !info.IsDir()
9  }
10
11 func CheckIfSymbolicLink(file string, envPath string) bool {
12         info, err := os.Lstat(envPath + "/" + file)
13         if err != nil {
14                 return false
15         }
16         return info.Mode()&os.ModeSymlink != 0
17 }
18
19 func ReadSymbolicLink(file string, envPath string) string {
20         link, _ := os.Readlink(envPath + "/" + file)
21         resolved := link
22         if !filepath.IsAbs(link) {
23                 resolved = filepath.Join(filepath.Dir(envPath+"/"+file),
                       link)
24         }
25         return strings.TrimPrefix(resolved, envPath)
26 }
27
```

```
28 func AddFilesToDockerfile(file string, files map[string][]string, symLinks
       map[string]string, rootfsPath string) {
29         file = RealPath(file)
30         if CheckIfFileExists(file, rootfsPath) {
31                 if CheckIfSymbolicLink(file, rootfsPath) {
32                         symLinks[file] = ReadSymbolicLink(file, rootfsPath)
33                 } else {
34                         files[filepath.Dir(file)] = AppendIfMissing(files[
                               filepath.Dir(file)], file)
35                 }
36         }
37 }
```

Listing 7.5: File util functions

With the two dictionaries populated, they are used to create the Dockerfile for this stage, by adding the *COPY* commands for each directory to the template Dockerfile, which is then tested to see if it can build and run the container successfully. If it it not, the next step is the dynamic analysis of the command.

7.5 Dynamic Analysis

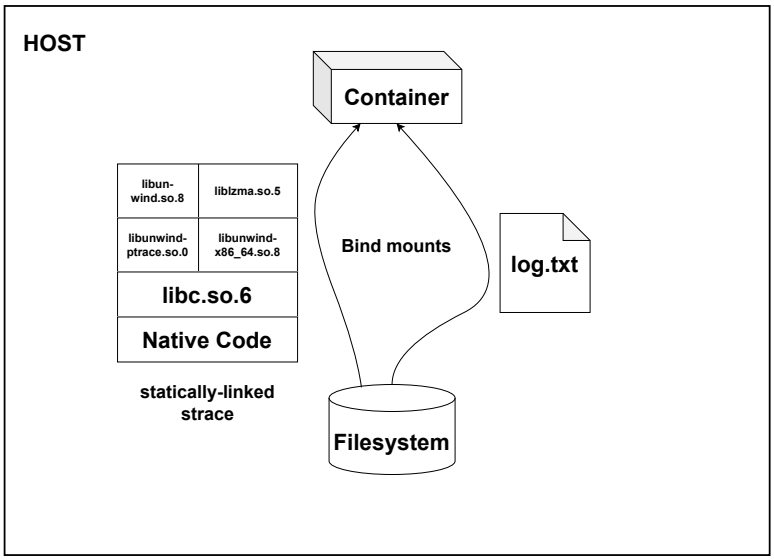

Figure 7.2: Dynamic analysis using strace to determine the files used by the command

For this stage, we attach a *strace* probe to the docker running command, in order to determine the files used and processes that are spawned by the command. In order to do this, we utilize Docker's volumes to mount a statically-linked *strace* binary into the container, as well as a logfile to save the output of the command. The reason for using a statically-linked binary is so that the libraries used by strace don't interfere with the libraries used by the command. To determine the runtime dependencies of the command, we will trace the *open* and *exec* familly of syscalls [18] as well as passing the *-f* flag to trace the child processes as well.

```
1     func parseOutput(output string, syscalls []string, files map[string][]
          string, symLinks map[string]string, envPath string) {
2         regexes := make(map[string]*regexp.Regexp)
```

```
3           for _, syscall := range syscalls {
4               regexes[syscall] = regexp.MustCompile(syscall + `
                    \([^"]*?"([^"]+)"`)
5           }
6           for line := range strings.SplitSeq(output, "\n") {
7               for _, syscall := range syscalls {
8                   if regexes[syscall].MatchString(line) {
9                       match := regexes[syscall].FindStringSubmatch(line)
10                      if len(match) > 1 {
11                          utils.AddFilesToDockerfile(match[1], files, symLinks
                                , envPath+"/rootfs")
12                      }
13                      break
14                  }
15              }
16          }
17      }
```

Listing 7.6: File util functions

Parsing the output is made trivial by using regexes to match the syscalls and extract the first text between "".

A problem that was quickly encountered was that in dealing with commands that are run via shebangs. A shebang is a special comment that is used to indicate the interpreter that should be used to run the script, and are always put at the top of the file, starting with the characters *#!*. When stracing the command, only the command itself is being shown as executed, and not the shebang. This is due to how the operating system handles the shebang. Finding the interpreter is done internally by the kernel in the *load_script* function, which is not visible to the strace command [5]. To fix this and make this step more robust, we firstly determine if a shebang is present in the base command, and afterwards we follow the minimization steps of static and dynamic analysis on the interpreter as well.

```
1       func getSheBang(command string, rootfsPath string) string {
2           file, err := os.Open(rootfsPath + "/" + command)
3           if err != nil {
4                   log.Error("Failed to open file:", command)
5                   return ""
6           }
7           defer file.Close()
8           var firstLine []byte
9           buf := make([]byte, 1)
10          for {
11                  n, err := file.Read(buf)
12                  if n > 0 {
13                          if buf[0] == '\n' {
14                                  break
15                          }
16                          firstLine = append(firstLine, buf[0])
17                  }
18                  if err != nil {
19                          break
20                  }
21          }
22          return string(firstLine)
23  }
24
25  func parseShebang(imageName string, containerName string, syscalls []string,
```

```
26          files map[string][]string, symLinks map[string]string, envPath
                string, metadata types.DockerConfig, timeout int) (map[string][]
                string, map[string]string) {
27          command := utils.GetContainerCommand(imageName, envPath, metadata)
28          hasSudo := utils.HasSudo()
29          shebang := getSheBang(command, envPath+"/rootfs")
30          regex := regexp.MustCompile(`^#!\s*([^\s]+)`)
31          if !regex.MatchString(shebang) {
32                  log.Error("Failed to find shebang in file:", command)
33                  return files, symLinks
34          }
35          match := regex.FindStringSubmatch(shebang)
36          if len(match) < 2 {
37                  log.Error("Failed to find interpreter in shebang:", command)
38                  return files, symLinks
39          }
40          interpreter := match[1]
41          lddCommand := hasSudo + " chroot " + envPath + "/rootfs ldd " +
                interpreter
42          log.Info("Running command:", lddCommand)
43          lddOutput, err := exec.Command("sh", "-c", lddCommand).
                CombinedOutput()
44          if err != nil {
45                  log.Error("Failed to run ldd command\n" + err.Error())
46          }
47          files, symLinks = ldd.ParseOutput(lddOutput, envPath+"/rootfs")
48
49          output := getStraceOutput(imageName, envPath+"/strace", envPath+"/
                log.txt", syscalls,
50                  containerName, interpreter, envPath, metadata, timeout)
51          parseOutput(output, syscalls, files, symLinks, envPath)
52          return files, symLinks
53  }
```

Listing 7.7: File util functions

Finally, we create the Dockerfile for this stage and validate it like in the static analysis phase.

7.6 Binary Search

Should the dynamic analysis fail to produce the minimal Dockerfile, we proceed to the *binary search* phase, so called because of the splitting in half of the search space, in our case being the files of the Docker image. The first step of this phase is to create into memory the filesystem structure as a dictionary, aptly named *unusedFiles*.

```
1   func parseFilesystem(rootfsPath string) (map[string][]string,
2           map[string][]string, error) {
3           info, err := os.Stat(rootfsPath)
4           if err != nil {
5                   log.Error("Error reading rootfs path:", err)
6                   return nil, nil, err
7           }
8           if !info.IsDir() {
9                   log.Error("Rootfs path is not a directory")
10                  return nil, nil, errors.New("rootfs path is not a directory"
                        )
11          }
12
```

```
13          usedFiles := make(map[string][]string)
14          unusedFiles := make(map[string][]string)
15          err = filepath.WalkDir(rootfsPath, func(path string, d fs.DirEntry,
                err error) error {
16                  if err != nil {
17                          log.Error("Error walking directory:", err)
18                          return err
19                  }
20                  relPath := strings.TrimPrefix(path, rootfsPath)
21                  if relPath == "" {
22                          relPath = "/"
23                  }
24                  unusedFiles[filepath.Dir(relPath)] = utils.AppendIfMissing(
                        unusedFiles[filepath.Dir(relPath)], relPath)
25                  return nil
26          })
27          if err != nil {
28                  log.Error("Error walking directory:", err)
29                  return nil, nil, err
30          }
31          return usedFiles, unusedFiles, nil
32  }
```

Listing 7.8: Filesystem parsing

For a maximum of *MAX_LIMIT* or until the *unusedFiles* dictionary is empty, we will run the following steps:

1. Split the *unusedFiles* dictionary in half, creating a new dictionary called *usedFiles*
2. Create a tarball from the *usedFiles* dictionary and add it to the Dockerfile
3. If the image runs successfully, the *usedFiles* dictionary becomes *unusedFiles* dictionary for the following iteration.
4. If the image fails to run, repeat step 1.

```
1   func binarySearchStep(envPath string, context string, timeout int, step int,
        usedFiles map[string][]string,
2           unusedFiles map[string][]string) (map[string][]string, map[string][]
                string, error) {
3           for {
4                   if len(unusedFiles) == 0 {
5                           return nil, nil, errors.New("no unused files or
                                symbolic links left to process")
6                   }
7                   usedFiles, unusedFiles = splitFilesystem(usedFiles,
                        unusedFiles)
8                   filename := fmt.Sprintf("Dockerfile.minimal.binary_search.%d
                        ", step)
9                   tarFilename := fmt.Sprintf("%s/files.tar", envPath)
10                  if err := utils.BuildTarArchive(usedFiles, tarFilename,
                        envPath); err != nil {
11                          log.Error("Error building tar archive:", err)
12                          return nil, nil, err
13                  }
14                  err := utils.AddTarToDockerfile(filename, "Dockerfile.
                        minimal.template", envPath)
15                  if err != nil {
16                          log.Error("Error adding tar to Dockerfile:", err)
```

```
17                          return nil, nil, errors.New("error adding tar to
                                Dockerfile")
18                  }
19                  utils.CopyFile(tarFilename, context+"/files.tar")
20                  err = utils.ValidateDockerfile(filename, envPath, context,
                        timeout)
21                  os.Remove(context + "/files.tar")
22                  if err != nil {
23                          exec.Command("docker", "rmi", "-f", "dockerminimize-
                                "+filepath.Base(envPath)+":"+fmt.Sprint(step)).
                                Run()
24                  }
25                  if err == nil {
26                          log.Info("Binary search step ", step, " succeeded.")
27                          utils.CopyFile(envPath+"/files.tar", "files.tar")
28                          return make(map[string][]string), usedFiles, nil
29                  }
30          }
31  }
32
33  func BinarySearch(envPath string, maxLimit int, context string, timeout int)
        error {
34          log.Info("Starting binary search...")
35          usedFiles, unusedFiles, err := parseFilesystem(envPath + "/rootfs")
36          if err != nil {
37                  log.Error("Error parsing filesystem:", err)
38                  return errors.New("error parsing filesystem")
39          }
40
41          step := 0
42          var lastErr error
43          for step := 1; step <= maxLimit; step++ {
44                  log.Info("Binary search iteration:", step)
45                  usedFiles, unusedFiles, lastErr = binarySearchStep(envPath,
                        context, timeout, step,
46                          usedFiles, unusedFiles)
47                  if lastErr != nil {
48                          break
49                  }
50          }
51
52          if lastErr != nil {
53                  log.Error("Binary search failed at step", step, "with error:
                        ", lastErr)
54                  return lastErr
55          }
56
57          if step > maxLimit {
58                  log.Info("Reached maximum limit of binary search iterations:
                        ", maxLimit)
59                  return errors.New("reached maximum limit of binary search
                        iterations")
60          }
61
62          log.Info("Binary search completed successfully.")
63          utils.CopyFile(envPath+"/files.tar", "files.tar")
64          return nil
65  }
```

Listing 7.9: Binary search function

Splitting the *unusedFiles* dictionary is done by simulating the randomness of a coin flip, where *heads* or in our case *0* means add the file to the ones that will be used.

```
1         func splitFilesystem(usedFiles map[string][]string,
2         unusedFiles map[string][]string) (map[string][]string, map[string][]
              string) {
3         cpyUsedFiles, _ := deepcopy.Anything(usedFiles)
4         cpyUnusedFiles, _ := deepcopy.Anything(unusedFiles)
5         usedFiles, _ = cpyUsedFiles.(map[string][]string)
6         unusedFiles, _ = cpyUnusedFiles.(map[string][]string)
7         for dir, files := range unusedFiles {
8                 originalFiles := slices.Clone(files)
9                 for _, file := range originalFiles {
10                        flag, _ := rand.Int(rand.Reader, big.NewInt(2))
11                        ok := flag.Int64()
12                        if ok == 0 {
13                                usedFiles[dir] = utils.AppendIfMissing(
                                      usedFiles[dir], file)
14                                unusedFiles[dir] = utils.RemoveElement(
                                      unusedFiles[dir], file)
15                        }
16                }
17                if len(unusedFiles[dir]) == 0 {
18                        delete(unusedFiles, dir)
19                }
20        }
21        return usedFiles, unusedFiles
22 }
```

Listing 7.10: Splitting the filesystem

This is done to ensure fairness in the selection of the files, assuming that all files are equally likely to be used by the application.

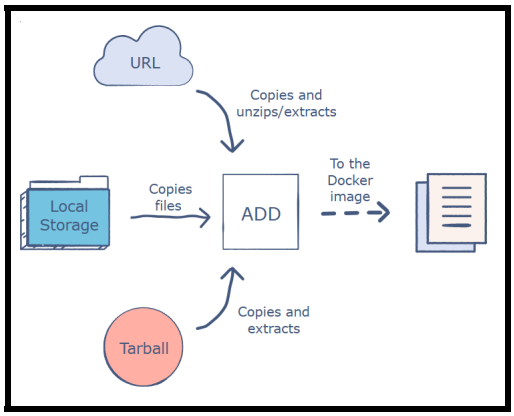

Figure 7.3: Adding a tarball to the Dockerfile

In order to check if these are the files that will produce a running container, we need to add them to the Dockerfile and build the image. However, we cannot simply follow the same steps as in the static and dynamic analysis phases, since of the sheer number of files that have to be copied which run the risk of exceeding Docker's hard limit of *127* image layers [15]. In

order to circumvent this, we will create a tarball (tar archive) of the files used and add it to the Dockerfile using the *ADD* command, which will automatically extract the files into the container's filesystem.

Finally, we validate the Dockerfile by building the image and running it to check if the command executes successfully.

Chapter 8

Performance Evaluation

In order to evaluate the minimization process, I tried minimizing a wide range of applications, from simple *Hello World* programs to complex software like *Wordpress*, *Chromium* and *MariaDB*, summing a sample size of 38. For each application, I calculated the size reduction of the minimal Docker image obtained by the minimization process compared to the original Docker image as well as the stage it reached in the minimization process (*Static Analysis*, *Dynamic Analysis* or *Binary Search*[2]). The results are as follows:

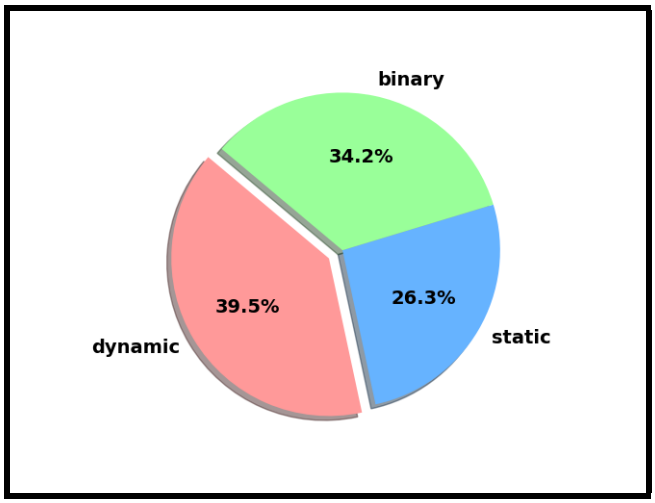

Figure 8.1: Distribution of the results by stage of minimization

The chart above shows that the program was able to correctly minimize applications in all three stages and that the majority of the applications were minimized in the *Static Analysis* and *Dynamic Analysis* stages, with only the more complex applications requiring the *Binary Search* stage.

In terms of the size reduction per stage, the results are the following:

As displayed in the chart above, the applications that benefited the most from the minimzation process were those that were able to be minimized in the *Static Analysis* stage, with an average size reduction of 96%. The applications that required the *Dynamic Analysis* stage had an average size reduction of 80%, while those that required the *Binary Search* stage had an average size reduction of 17% (with 3 iterations). The average size reduction across all applications was

[2] I chose to limit the steps of the binary search to 3, due to time constraints and fear of hardware failure

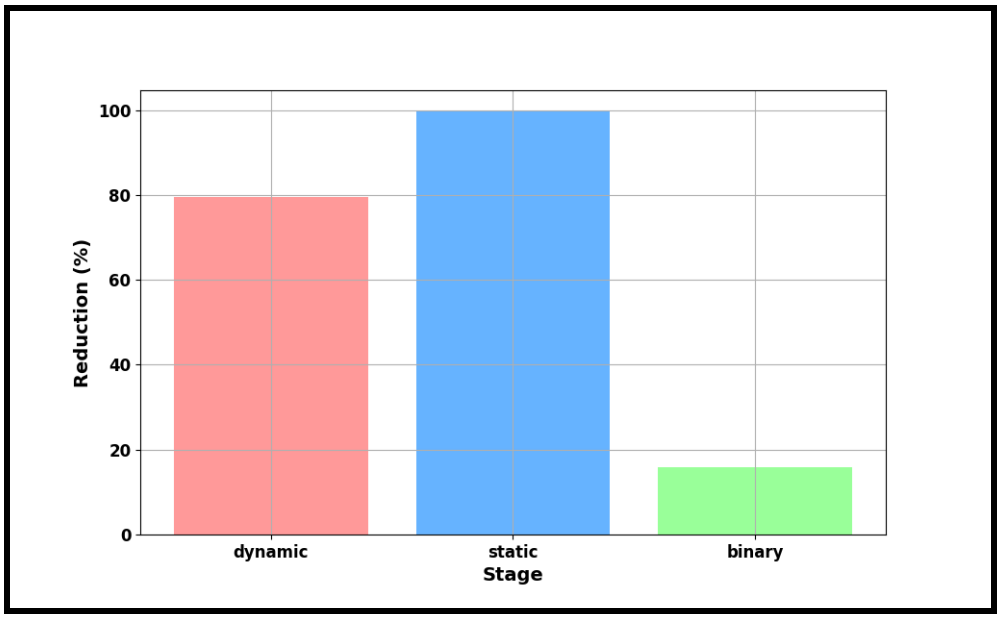

Figure 8.2: Size reduction per stage of minimization

70%, which is a significant improvement over the original Docker images. For a full breakdown of the size reduction per application, see the chart below:

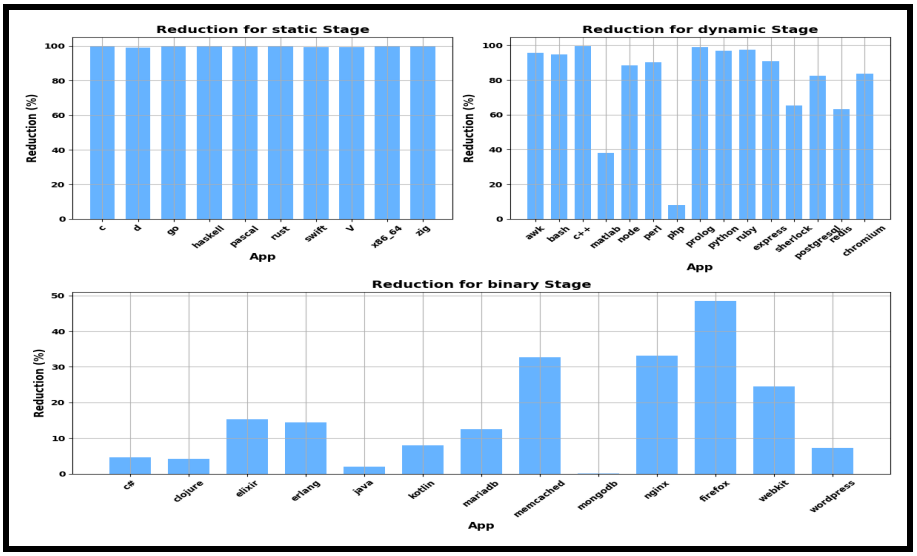

Figure 8.3: Size reduction per application

The main issues that I identified with the application were related to the *Dynamic Analysis* stage, where the application was not able to correctly identify all the files that were not used by the application. This was due to the fact that some applications require a more complex analysis of their dependencies and usage patterns, which *ptrace* that *strace* uses is not able to provide. Another issue was the relatively high execution time of the *Binary Search* stage, which can take a long time to complete for larger applications, owing to the archiving of many files and building of the images per each iteration, which is the reason why I limited the number of iterations to 3. Overall, the application was able to successfully minimize a wide range of

applications, with an average size reduction of 70%.

Chapter 9

Status and Planned Work

9.1 Status

The application has been successfully implemented and is mature enough to be able to be able to minimize a plethora of applications. I was able to implement all the functionalities that I set out to do, including the static analysis part, the dynamic analysis part, and the binary search phase. Although the dynamic analysis is not fully reliable and the binary search is computationally intensive with a high execution time, as seen in the performance evaluation, the application was able to minimize over 38 applications, with an average size reduction of 70%.

9.2 Planned Work

In the future, I plan to improve the following aspects of the application:

- **Dynamic Analysis**
- **Binary Search**
- **Resolve CVE[1] and Security Issues**
- **Improve User Experience**

The dynamic analysis part of the application is not fully reliable, as it does not always correctly identify all the files that are not used by the application. One way to improve this is to use other tracing ways besides *strace*, such as *ftrace*[2] or *extended Berkeley Packet Filter (eBPF)*[3]. can provide more detailed information about the application's usage patterns. Additionally, an application can be run multiple times with different inputs to ensure that all the files that are used by the application are identified.

The binary search part is computationally intensive due to the archiving of many files and building of the images per each iteration. The assumption made in the implementation is that the files needed by the application are all equally likely to appear in the whole filesystem, which is not always true. A more heuristic approach can be used to identify the files that are more likely to be needed by the application. Furthermore, the random selection of files can lead to the situation where the file that is needed by the application is skipped multiple times due to

[1] Common Vulnerabilities and Exposures
[2] https://www.kernel.org/doc/html/v5.0/trace/ftrace.html
[3] https://www.kentik.com/kentipedia/what-is-ebpf-extended-berkeley-packet-filter/

landing *tails* in the coin toss. This can be mitigated by analyzing the exit code of the container and add the missing file without needing to run the iteration again.

A new feature that I plan to implement is the ability to resolve any security issues that the Dockerfile I am minimizing has, such as vulnerabilities, exposed secrets and misconfigurations, inspired by *trivy*[4].

Finnaly, I plan to add a graphical user interface (GUI) to the application, which will make it easier to use and more user-friendly. The GUI will allow users to easily select the Dockerfile/image they want to minimize, view the progress of the minimization process, and see the results of the minimization.

[4] https://trivy.dev/

Bibliography

[1] 6sense. Market share of docker. https://6sense.com/tech/containerization/docker-market-share.

[2] 8grams Tech. Distroless: Using minimal container image for kubernetes workload. https://blog.8grams.tech/distroless-using-minimal-container-image-for-kubernetes-workload, February 2024.

[3] Chris Aubuchon. The shortlist: History of containers. https://cycle.io/blog/2024/07/shortlist-history-of-containers.

[4] William Boyd. What is go? an intro to google's go programming language (aka golang). https://www.pluralsight.com/resources/blog/cloud/what-is-go-an-intro-to-googles-go-programming-language-aka-golang.

[5] Bruno Croci. Demystifying the #! (shebang): Kernel adventures. https://crocidb.com/post/kernel-adventures/demystifying-the-shebang/.

[6] Adriaan de Jonge. Create the smallest possible docker container. https://xebia.com/blog/create-the-smallest-possible-docker-container/.

[7] Ben Dicken. Speed comparison of programming languages. https://benjdd.com/languages/.

[8] Raghav Dua. dockershrink. https://github.com/duaraghav8/dockershrink.

[9] Ashan Fernando. Why it's important to keep your containers small and simple. https://hackernoon.com/why-its-important-to-keep-your-containers-small-and-simple-618ced7343a5.

[10] Python Software Foundation. What is python? executive summary. https://www.python.org/doc/essays/blurb/.

[11] Google. Go packages. https://pkg.go.dev/.

[12] Brian Harrington. Minimal containers 101. https://github.com/brianredbeard/minimal_containers.

[13] Docker Inc. The underlying technology. https://docs.docker.com/get-started/docker-overview/#the-underlying-technology.

[14] Docker Inc. Use containers to build, share and run your applications. https://www.docker.com/resources/what-container/.

[15] Knowledge. Docker layers - failed to register layer: max depth exceeded. https://support.domino.ai/support/s/article/Docker-Layers-failed-to-register-layer-max-depth-exceeded-1718868032014.

[16] Ogo Ozotta. Small is beautiful: How container size impacts deployment and resource usage. https://www.fullstack.com/labs/resources/blog/small-is-beautiful-how-container-size-impacts-deployment-and-resource-usage.

[17] J. A. Pardo. But... it works on my machine.... https://medium.com/@josetecangas/but-it-works-on-my-machine-cc8cca80660c.

[18] Adam Rehn. Identifying application runtime dependencies. https://unrealcontainers.com/blog/identifying-application-runtime-dependencies.

[19] Unikraft. Adding applications to the catalog. https://unikraft.org/docs/contributing/adding-to-the-app-catalog.

[20] Unikraft. Concepts. https://unikraft.org/docs/concepts.

[21] Unikraft. Filesystems. https://unikraft.org/docs/cli/filesystem.

[22] Lionel Sujay Vailshery. Adoption rate of container technologies in organizations worldwide from 2016 to 2021, by development stage. https://www.statista.com/statistics/1104543/worldwide-container-technology-use/, February 2024.

[23] Ivan Velichko. How to extract container image filesystem using docker. https://labs.iximiuz.com/tutorials/extracting-container-image-filesystem.