Report 1

Andrei Petrea
Faculty of Automatic Control and Computers
National University of Science and Technology POLITEHNICA Bucharest

February 6, 2026

Abstract

This report examines container image minimization in the context of modern day use-cases. It provides an overview of the state-of-the-art tools and techniques for minimizing container images, as well as a detailed description of my work in developing a new tool for this purpose. The report concludes with a comparison of my tool with existing solutions and a discussion of potential future work in this area.

1 Introduction

In the era where cloud-based solutions have become the de-facto standard for enterprise infrastructure, with over 94% of enterprises using cloud services [6], containers have emerged as a pivotal technology, revolutionizing the way applications are developed, deployed and managed [1], but what exactly are containers?

Containers are lightweight, portable, standalone bundles of software that include everything needed to run an application, such as code, runtime, system tools, libraries, and settings [3]. They provide a consistent and isolated environment for applications, ensuring that they run reliably across different computing environments.

However, as the AI revolution continues to reshape the technological landscape, the demand for more efficient storage solutions has surged [9]. Although containers offer, on average, smaller sizes compared to traditional virtual machines, their size can balloon to several gigabytes when packaging complex application with numerous dependencies. This is owing to the fact that containers not only include the application code, but also other components that make the developer experience smoother, such as shell, package managers, and various libraries. The removal of these non-essential components can lead to a significant reduction in container size, amongst other benefits like faster deployment times and reduced attack surfaces [5].

In this report, we will explore the current state of affairs regarding the minimization of container images, examining existing techniques and tools that shrink container sizes, also highlighting my own contributions to this field through the development of a novel tool aimed at optimizing container images.

2 State of the Art

There are two main approaches to minimizing container images:

- **Use Smaller-sized Libraries:** This approach involves replacing larger libraries with smaller alternatives that provide the same functionality.
- **Removing Unnecessary Components:** This approach focuses on stripping away non-essential components from the container image.

Let us explore some of the ways these approaches are implemented in practice.

Slim Debian

Slim Debian is a stripped-down version of the standard Debian image. It removes unnecessary components and libraries, resulting in a smaller image size compared to the full Debian image (29 MB vs 47 MB). However, the reduction is negligible compared to the original image, and it still includes a lot of components that may not be necessary for all applications, such as package managers and shells.

Alpine Linux

One of the most popular base images for minimizing container size is *Alpine Linux.* Instead of the standard *glibc* library, which is 2.2MB in size, it uses the *musl* library (approximately 650KB). As such, any library that depends on *glibc* can be replaced with a version that relies on *musl*, resulting in a significant reduction in size. Most popular frameworks and runtime environments, such as Python, Node.js, and Java, have Alpine-compatible versions that utilize *musl.* In order to further reduce the size, Alpine Linux employs *BusyBox*, a software suite that provides several Unix utilities in a single executable file. The main disadvantage of using Alpine Linux is compatibility. In order to use it, developers must ensure that their applications and dependencies are compatible with musl, which may prove impossible for complex applications.

Minimal Images

Minimal images, frequently referred to as *distroless* images, are streamlined container environments that exclude non-essential system components such as package managers, shells, and standard operating system distributions, retaining only the application and its runtime dependencies [2]. As they only include the bare essentials required to run the application, they have the most reduction in size compared to the other approaches, however they are the most difficult to create and maintain, as they require a deep understanding of the application's dependencies and careful management of updates and security patches.

Building Minimal Images

Building minimal images is intrinsically linked to finding the dependencies of the application and extracting them into a new container image. This process can be achieved through various techniques, such as:

- **Static Analysis:** analyzing the application's code and configuration files to identify its dependencies

- **Dynamic Analysis:** executing the application and monitoring its behavior to identify the dependencies it uses at runtime
- **Brute Force:** iteratively removing components from the container image and testing if the application still functions correctly, until a minimal set of dependencies is identified

These techniques can be done manually, but they are time-consuming and error-prone, especially for complex applications with many dependencies. As such, there is a need for automated tools that can streamline this process and generate minimal container images efficiently and accurately.

Let us examine some of these tools.

Bazel

Bazel is a build automation tool created by Google that supports building and testing software applications [7]. It includes support for building Docker images through its *rules_oci* extension and it made the first distroless images. It uses a file called *BUILD* to define the build rules for the application, including the dependencies and the steps required to create the Docker image. The main disadvantage of Bazel is that it doesn't automatically identify the dependencies of the application, requiring developers to manually specify them in the *BUILD* file.

Dockershrink

Dockershrink is an AI-powered tool that optimizes Docker images by analyzing the application and generating a Dockerfile or optimizing an existing one [4]. As far as I could determine, it only looks at Dockerfile misconfigurations and does not actually analyze the application to find its dependencies and it currently only supports NodeJS applications.

Slim

Slim, previously known as DockerSlim, is an open-source tool that automatically minimizes Docker images by analyzing the application and its dependencies, as well fixing common security issues [8]. It works by running the application in a container and monitoring its behavior using ptrace, fanotify, etc to identify the dependencies it uses at runtime, then it generates a new Docker image that only includes those dependencies. Additionally, it can also execute commands inside the testing container to ensure that all code paths are exercised, further refining the dependency list. As Slim takes an already existing image as input, it can be used with any application, regardless of the programming language or framework used. However, Slim has some limitations, such as difficulty in handling applications that require complex setup or configuration, and it may not always produce the smallest possible image due to its reliance on dynamic analysis, as well as potential issues with incomplete code path coverage. This however remains, as far as I could tell, the best tool available for minimizing container images at the moment.

3 My Work

Considering the disadvantages of existing tools for minimizing container images, I developed a new tool called *dockerminimizer* that aims to address these issues.

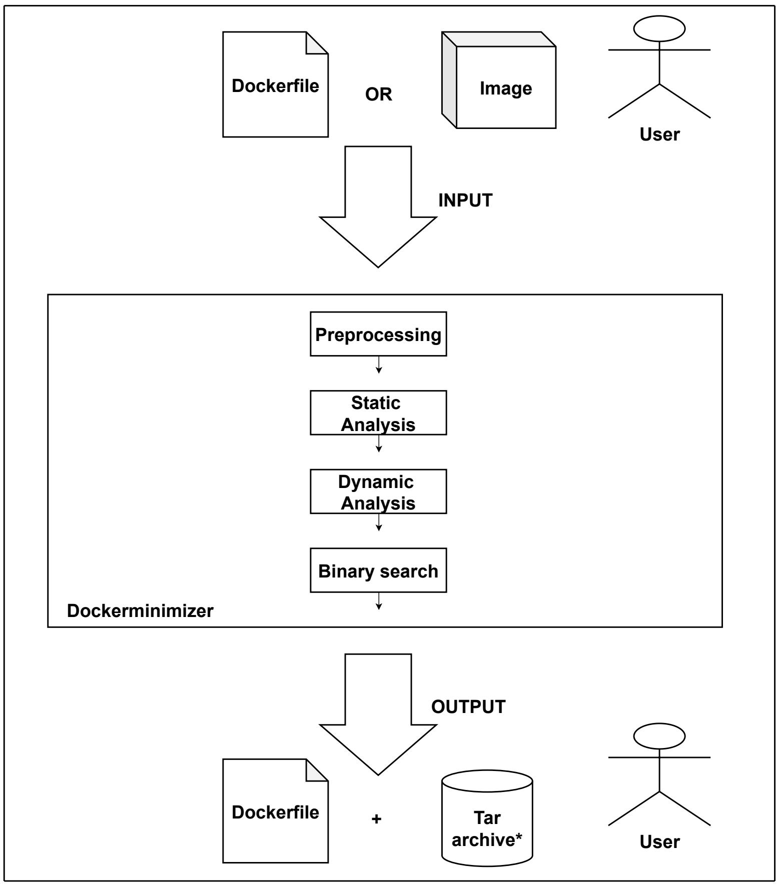

Figure 1: High-level architecture of the application

My tool is implemented in *Go* and is designed to be modular and extensible. It takes as input either a Dockerfile or an existing Docker image and performs static, dynamic analysis and binary search to minimize the application containerized, outputting the Dockerfile and if needed the tarball of all dependency files. It takes the following command-line 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

The tool expands from Slim by firstly performing static analysis to conserve processing resources and has a brute-force binary search approach in order to ensure that the minimization process terminates successfully even in cases where dynamic analysis fails.

Binary Search

Since this is a novel approach, I will briefly describe how the binary search procedure works.

The binary search procedure is a fail-safe mechanism that is used when the dynamic analysis fails to yield any results. It works by splitting the original image's filesystem into two halves and checking if the application can run successfully with only one of the halves. If it can, then the other half is split into two again and added to the first half. Once the application successfully runs, the files that remained in the second pile are discarded and the new files become the new baseline. This process is repeated until the maximum number of iterations is reached or the image cannot be minimized any further.

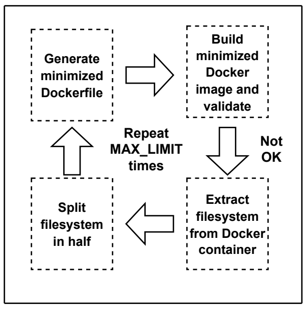
Generate minimized Dockerfile
Build minimized Docker image and validate
Repeat MAX_LIMIT times
Not OK
Split filesystem in half
Extract filesystem from Docker container

Figure 2: Binary search procedure for minimizing the application

4 Conclusion and Further Work

My tool is still a work in progress and there are several features that are missing or need improvement. I have created a comparison table between my tool and Slim in order to highlight the differences and similarities between the two tools.

Table 1: Comparison between Dockerminimizer and Slim

Feature	**Dockerminimizer**	**Slim**
Implementation Language	Go	Go
Analysis Technique	Static Analysis (‘ldd‘) Dynamic Analysis (‘strace‘) Binary Search (Brute Force)	Dynamic Analysis (Sensors/Probes) Static Metadata Analysis
Output Format	Minimized `Dockerfile` + Filesystem Tarball	Minified Docker Image (‘.slim‘) (Opaque Binary)
Fallback Mechanism	**Binary Search** (Automated recovery of missing files)	**Manual Configuration** (User must manually include paths)
Base Image	`FROM scratch`	Custom Minified Distro
Transparency	**High** (User can inspect/edit the resulting Dockerfile)	**Low** (Black-box image generation)
Additional Features	**None**	**Full Suite** (`xray`, `lint`, `debug`, `merge`, `registry`, `vulnerability`)

The main advantage of my tool is the transparency of the resulting Dockerfile, which allows users to inspect and modify the minimized image as needed and the automated binary search procedure that ensures that the minimization process terminates successfully even in cases where dynamic analysis fails. However, in order to ensure the viability of the tool, I need to either implement some of the additional features that Slim offers or improve on the minimization process in order to achieve better results than Slim.

I have also identified several areas for further work, including:

- **Dynamic Analysis Improvements**: In addition to *strace* (which uses ptrace), I could explore other dynamic analysis techniques such as eBPF or systemtap to capture more comprehensive runtime information and potentially improve the accuracy of the minimization process.
- **Binary Search Optimization**: The current brute-force binary search approach can be time-consuming, especially for larger images. I could investigate more efficient search algorithms or heuristics to reduce the number of iterations needed to find the minimal set of files.
- **Security Analysis Integration**: Integrating security analysis tools to identify and remove vulnerable dependencies could enhance the security posture of the minimized images.
- **User Interface Enhancements**: Developing a more user-friendly interface, such as a web dashboard or CLI improvements, could make the tool more accessible to a wider range of users.

- **Support for Additional Container Formats**: Extending support to other container formats (e.g., OCI images) could broaden the applicability of the tool in different container ecosystems.
- **Application Path Analysis**: Implementing a more sophisticated application path analysis to better understand the dependencies and interactions between files could lead to more effective minimization.

References

[1] Atmosly. Docker container vs virtual machine: Which should you use in 2026? https://atmosly.com/blog/docker-container-vs-virtual-machine-which-should-you-use-in-2025, December 2025.

[2] Docker. Minimal or distroless images. https://docs.docker.com/dhi/core-concepts/distroless/.

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

[4] R. Dua. dockershrink. https://github.com/duaraghav8/dockershrink.

[5] A. 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.

[6] Flexera. Rightscale 2019 state of the cloud report from flexera™. https://resources.flexera.com/web/media/documents/rightscale-2019-state-of-the-cloud-report-from-flexera.pdf, 2019.

[7] Google. Bazel. https://bazel.build/.

[8] K. Quest. Slim. https://slimtoolkit.org/.

[9] Technologent. Ai is causing cloud storage cost overruns. here's what to do. https://blog.technologent.com/ai-is-causing-cloud-storage-cost-overruns.-heres-what-to-do, August 2025.