National College of Ireland

Performance Enhancement and Security Integration in a Multi-Objective Optimization Framework for Docker Image Slimming

MSc Research Project
Programme Name

Clint Fernandes
Student ID: 23348089

School of Computing
National College of Ireland

Supervisor: Sean Heeney

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Clint Fernandes
Student ID:	23348089
Programme:	Cloud Computing **Year:** 2024 -2025
Module:	MSc Research Project
Supervisor:	Sean Heeney
Submission Due Date:	11/08/2025
Project Title:	Performance Enhancement and Security Integration in a Multi-Objective Optimization Framework for Docker Image Slimming
Word Count:	**3586 Page Count: 12**

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Clint Fernandes

Date: 11/08/2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Performance Enhancement and Security Integration in a Multi-Objective Optimization Framework for Docker Image Slimming

Clint Fernandes
Student ID: 23348089

Abstract

Modern software deployment has been revolutionized by containerization, with Docker being the most widely adopted containerization platform. However, if not properly inspected, Docker images often retain redundant and superfluous elements that increase the storage needs, transmission overhead, and attack surface. While existing image slimming approaches like δ-SCALPEL have shown great potential in reducing image sizes through static code analysis, their substantial performance bottlenecks prevent them from being widely used in CI/CD processes and real-world implementations. This paper offers an enhanced version of the δ-SCALPEL tool that decreases the execution time by 40% while incorporating vulnerability scanning for security assessment through Trivy. The focus was to improve static code analysis, implement efficient caching mechanisms, and integrate vulnerability scanning. The paper conducts comprehensive comparative experiments using the 20 NPM packages in the original tools' experiments. The experiments were performed on Amazon Web Services (AWS) cloud infrastructure to assess the security posture, slimming efficacy, and performance gains. The results conclude that the enhanced tool dramatically improves processing speed from an average of 8.3 minutes to 4.9 minutes while maintaining the original δ-SCALPEL's highest image size reduction of 61.4%. Furthermore, an integrated vulnerability assessment conducted by Trivy enabled the tool to identify and classify the various CVEs present in the slimmed image. These enhancements respond to growing demands for effective and secure container optimization in automated DevOps workflows and real-world production environments.

Keywords: cloud computing, Docker image slimming, static code analysis, performance optimization, vulnerability scanning, DevSecOps

1 Introduction

The proliferation of containerization technologies has significantly changed modern software development, testing, and deployment approaches. Docker is the leading platform for containerization that allows packing applications together with all their dependencies into a portable and lightweight container, which can be run easily on different platforms.

However, Docker's layered architecture can lead to bloated images, and, according to research, some popular containers may be bloated by over 60%. Such bloated images are the result of inheriting dependencies from the base images, unnecessary packages

downloaded during the build, temporary files and cache retained, and bad practices while composing the Dockerfiles.

The increase of redundant content in Docker images presents several critical challenges in recent software development practices. Images that are considered large images consume substantial storage resources in local development environments and cloud-based container registries. Pulling large images from repositories or transferring them across the networks consumes significant resources, increasing operational costs (Openja et al., 2022).

As Docker images continuously grow in size, software dependencies management, along with ensuring compatibility across projects, becomes difficult and ultimately hinders deployment pipelines and scalability in the cloud. On serverless platforms, such large images translate into longer data transmission times and higher response times, rendering Docker nearly unsuitable for such environments (Ching et al., 2024). In edge or cloud-edge environments where rapid processing is critical, these large images can quickly add to the increase in processing time. Reliance on a central cloud repository for storing these large images can result in high transfer times and resource constraints (Azzolini, Forti and Ielo, 2025).

1.1 Research Question

Research Question. The below research problem arises the following research question:

How does the performance-optimized δ-SCALPEL with integrated security assessment compare to the original δ-SCALPEL in terms of execution efficiency, slimming effectiveness, and security posture improvement when evaluated on real-world Node.js applications in cloud environments?

2 Literature Review

This section presents a critical review of existing literature on various Docker image slimming techniques, highlighting their strengths and weaknesses. It will also discuss the δ-SCALPEL image slimming tool and its cons. We will critique why the tool is not ready for production environments. The goal is to outline the strengths and weaknesses of each study, identify the gaps this research intends to address, and explain how this study will fulfill those needs.

2.1 Dynamic Analysis

Dynamic analysis methods monitor a container's runtime behavior to figure out what is essential for executing an application. Dynamic approaches track the files, libraries, and binaries accessed during the containerized application's execution, enabling targeted removal of unused components and generating smaller images.

A popular tool for dynamic image optimization is DockerSlim or SlimToolkit. It acts to automatically reduce the image size, further reducing the attack surface and resource utilization. The tool analyses the image, layers, and components. During the container runtime, the application is dynamically probed to identify the critical libraries and

dependencies. Any additional elements are removed, thus reducing the overall image size drastically (Liu et al., 2024).

A limitation of this approach to dynamic analysis stems from the quality of the tests or code coverage exercised. It monitors the container application on many different execution paths, but any path not executed during that monitoring session will remain unchecked, even though it may represent an important feature. An incomplete set of code paths will be constructed, leading to an incomplete and malfunctioning application.

2.2 Static Code Analysis:

Static code analysis is an application source code verification technique that examines the source code without executing it. It enables early detection of defects, vulnerabilities, and code quality issues. Static code analysis is a valuable technique for improving software quality and security without actually compiling the code (Rahaman and Ive, 2023).

Static code analysis finds bugs and errors early, hence reducing the cost and effort of debugging and fixing them later. It could identify errors that are difficult to detect at runtime such as uninitialized variable use, potential overflows, and logical inconsistencies, thus improving software reliability and maintainability (Solanki, 2024).

(Han et al., 2025) introduced δ-SCALPEL, which represents the most advanced static analysis approach for Docker image slimming. δ-SCALPEL employs several innovative techniques:

- Data Flow Analysis: Uses CodeQL for advanced data dependency analysis, tracking flows of data from when variables are declared, to when system commands are called.

- Command Linked List (CLL): A new data structure is suggested to model complex relationships between the system commands, symbolic links, and binary files.

- Multi-Stage Pipeline: The process follows a systematic three-stage approach consisting of the extraction of environment dependency, slim construction of rootfs, and slim building of images.

Its inherent technicality fails to lend itself to performance enhancement, in particular where improvements are demanded for CI/CD environments.

3 Research Methodology

3.1 Research Approach

The primary objective of this study is to understand the limitations of the δ-SCALPEL tool and why it is not ready for production environments and real-world scenarios. Founded on that study, improvements will be made to the tool to enhance it and make it eligible for production. The study will use quantitative experimental research designs

to assess the performance improvements and security enhancements of the enhanced δ-SCALPEL tool. The improved tool will be tested against the exact 20 NPM packages utilized to evaluate the original tool. The reason for selecting the exact NPM packages was to accurately measure the difference in metrics across both versions. The research will then follow a comparative approach in methodical measurement and comparison of the enhanced tool against the original δ-SCALPEL implementation. The study involves deploying both the original and enhanced versions of the tool on Amazon Web Services (AWS) infrastructure.

3.2 Equipment and Tools

For the experiments, various tools were employed to guarantee a clean, fair, and realistic setup. AWS was the preferred IaaS provider as it has secured its position as a market leader in cloud services. Opting for AWS would ensure users can relate easily and find it easy to replicate the experiments. It provided great flexibility and a variety of services and configurations to choose from. Most of the software development and testing occurred on a t3.micro EC2 instance that simulated the development environment.

The AWS Elastic Container Registry (ECR) service was used as a central repository to store the Docker images. Two separate AWS CodeBuild projects were prepared to run both versions of the tool, with each creating temporary containers with 2vCPUs and 4 GB RAM to run the workload.

When a 'fat' image was pushed to ECR, it would trigger the CodeBuild service to execute both projects. To listen for a successful push to ECR and trigger the CodeBuild projects, the AWS EventBridge service was used.

Since the original tool was designed to slim Docker images in particular, the containerization platform used was Docker. The Go programming language was installed in the development environment, as the tool itself was built using Go.

For versioning and integrated feature development, Git was used locally. GitHub was used as a central repository for the application code. Visual Studio Code (VS Code) was the code editor of choice as it provided all the basic features to assist in easy development. It delivered a seamless development experience with extensions for SSH, Git, and Docker.

3.3 Experimental Setup

The experiment setup involves configuring a few AWS services. The starting point would be a development environment that is isolated from all the other services. To do this, a t3.micro EC2 instance was started, and the Go programming language and tools like Docker and Git were installed.

Another service implemented was ECR to store the Docker images before the slimming process. Two AWS CodeBuild projects were also created and configured to build both versions of δ-SCALPEL and begin with their respective executions. AWS EventBridge will be set up to listen for successful pushes to the ECR. If detected, it will trigger the

CodeBuild projects. This setup aims to have a common starting point from which both the tool versions would start the slimming process.

3.4 Data Collection

The data and metric collection was a crucial step after the experiment implementation. Most of the metrics were to be collected from the builds of both the CodeBuild projects. These include timestamps of the build start and end times, the image sizes before and after slimming, and finally, the generated CVE scores after slimming the image. The data would have to be manually fetched from each CodeBuild build.

3.5 Data Evaluation

The metrics collected were evaluated and compared to assess the performance improvement of the enhanced tool over the original. Time taken for the execution would be calculated by subtracting the start time from the end time. If the total time taken by the enhanced tool was less than that of the original tool, that would indicate that the enhanced tool performed faster.

Another critical metric would be the CVE assessment data, which would find the number of CVEs present in the slimmed image. This metric would give an understanding of the security posture of the application and the image. It would mention the different categories of vulnerabilities. This allows for actionable steps to improve the overall security of the application and container in production environments.

4 Design Specification

4.1 Original Tool

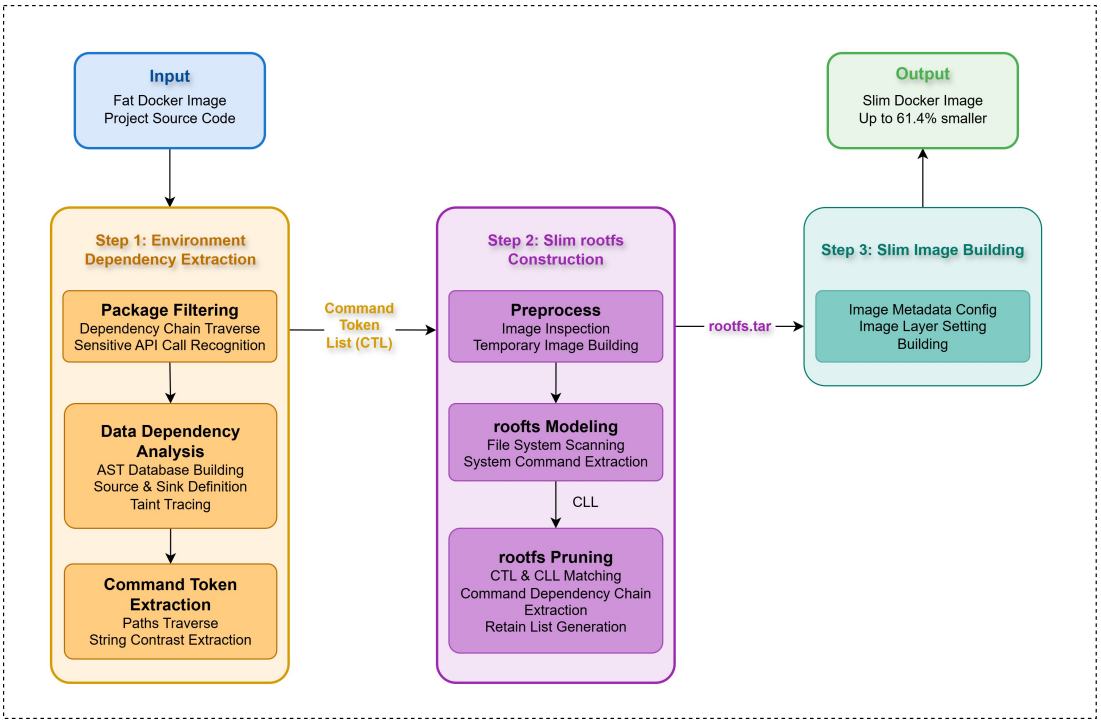


Figure. 1. The δ-SCALPEL framework.

The framework of the δ-SCALPEL tool is shown in Figure. 1. The tool is divided into three phases.

4.1.1 Environment Depedency Extraction

Uses static code analysis techniques with CodeQL to analyze JavaScript code. Filters NPM packages that make the "exec" API call. Performs taint tracing and data flow analysis, allowing data to flow from sources to sinks. Extracts command tokens to represent system dependencies.

4.1.2 Slim rootfs Construction

Analyzes the file system of the 'fat' Docker image. Creates a Command Linked List (CLL), which models command dependencies and symbolic links. Maps the Command Token List (CTL) with the CLL. Produces a slim root file system by removing redundant resources.

4.1.3 Slim Image Building

Takes the slim rootfs.tar and original image metadata. Builds the final slim Docker image. Behaves exactly like the original, even though reduced in size.

4.2 Proposed Enhanced Tool

The framework of the enhanced δ-SCALPEL tool is shown in Figure. 2. The enhanced version has an additional phase.

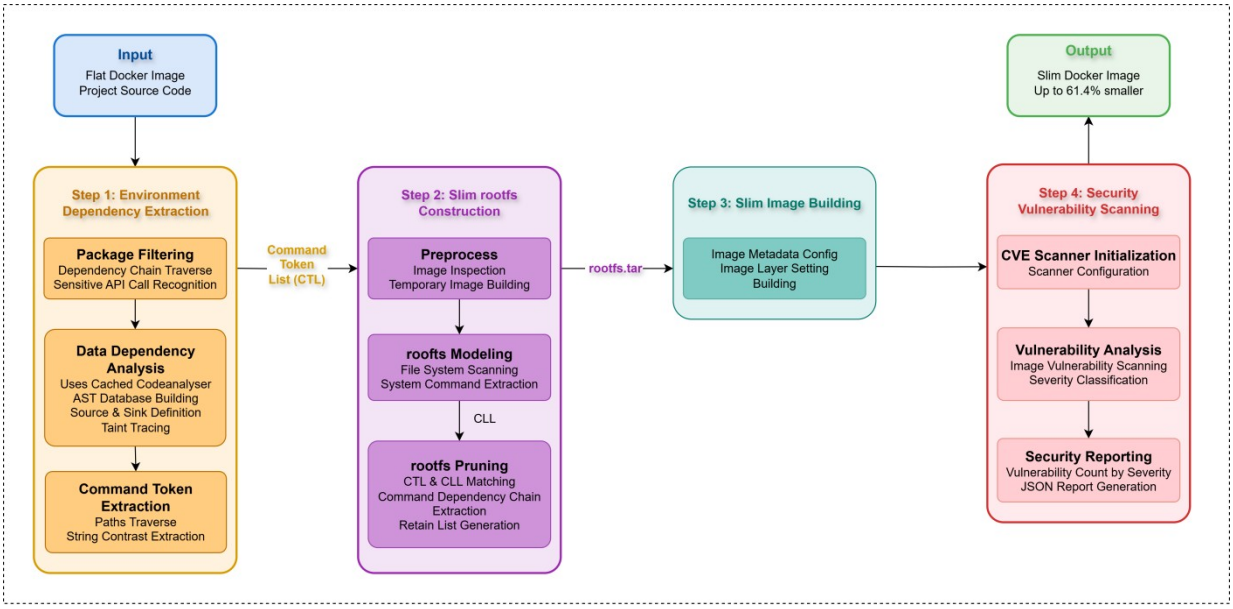


Figure. 2. The δ-SCALPEL2 Framework.

4.2.1 Caching Mechanism

Creation of a function to cache the code analyser image to avoid rebuilding it on subsequent runs. Creation of another function to utilize the cached image from previous runs for faster deployment. Expecting a substantial reduction in execution time in successive runs.

4.2.1 Security Vulnerability Scanning

Adding a new CVE scanner module for a complete security assessment system. Trivy integration for vulnerability detection. Will generate detailed reports with vulnerability severity categories (Critical, High, Medium, Low).

4.3 Overview

5 Evaluation

We evaluate δ-SCALPEL2 by finding the answers to the following three questions:

- **Q1:** Was there a performance improvement in δ-SCALPEL2?
- **Q2:** Did δ-SCALPEL2 provide vulnerability assessment of the slimmed image?
- **Q3:** Was there a decrease in the effectiveness of δ-SCALPEL2?

5.1 Experimental Setup

5.1.1 Dataset

To evaluate the enhanced version of the δ-SCALPEL tool, the experiments will be performed on the exact 20 NPM projects, that were selected to evaluate the original δ-SCALPEL tool. The researchers of the original tool randomly selected 20 NPM projects from the top 1000 most depended-upon packages listed on GitHub. For each project, the researchers downloaded its source code from the address specified in the NPM

repository and used it to create a Dockerfile that would form a Docker image. The researchers used Docker Hub's official node images; node:current and node:current-slim, to create the Docker images for each project.

This research opted to use the exact NPM projects so as to achieve near-exact results. This would benefit in performing a comparative analysis between the two versions. The data and metrics observed from δ-SCALPEL would act as a benchmark to base future experiment evaluation results on.

5.1.2 Implementation

All the experiments were conducted in a identical environment to minimize differences caused by different environments. The development environment was an AWS t3.micro EC2 instance with 2 vCPUs and 2GB of memory and Amazon Linux 2 OS. The tools were compiled an built on AWS CodeBuild. CodeBuild would create containers with 2 vCPUs and 4 GB of memory to execute the builds.

5.2 Q1: Performance

In the Dockerfile of the NPM projects, CMD or ENTRYPOINT can be used to specify the command to run right after the container is initialised. The projects in the dataset are broadly categorised as specifying the entry point and not specifying the entry point. Every project under these two categories is divided again based on the image's base image, which is either node:current or node:current-slim. This section evaluates the performance of *δ*-SCALPEL2. The original tool will be tested against the dataset to form the baseline. Then will test *δ*-SCALPEL2 against the dataset to find out the performance metric. Comparative analysis will be performed on both outcomes to determine the performance of *δ*-SCALPEL2 in two scenarios: specifying the entry point and not specifying the entry point.

NPM Project	Base Image With Entry Point					
	node:current-slim			node:current		
	δ-SCALPEL Time Taken (Minutes)	δ-SCALPEL2 Time Taken (Minutes)	Time Saved / Performance Increase	δ-SCALPEL Time Taken (Minutes)	δ-SCALPEL2 Time Taken (Minutes)	Time Saved / Performance Increase
semver	25:12	12:22	12:50 / 49.8%	29:53	12:23	17:30 / 58.6%
chalk	24:45	10:23	14:22 / 58.2%	28:20	10:38	17:42 / 61.8%
nodejs-websocket	12:55	06:20	06:35 / 50.6%	13:31	08:22	05:09 / 38.2%
lru-cache	17:28	09:03	08:25 / 47.7%	21:07	11:04	10:03 / 47.6%
minimatch	13:25	07:48	05:37 / 40.5%	20:26	09:38	10:48 / 51.7%
strip-ansi	22:24	09:46	12:38 / 55.7%	27:54	12:18	15:36 / 55.8%
node-glob	13:56	07:55	06:01 / 44.3%	15:47	09:30	06:17 / 39.9%
commander.js	12:24	07:07	05:17 / 42.2%	15:02	09:20	05:42 / 36.1%
yallist	14:07	07:45	06:22 / 44.2%	15:17	10:09	05:08 / **33.5%**
estraverse	23:44	08:40	15:04 / 64.1%	28:32	10:22	18:10 / **63.9%**
deepmerge	09:42	05:50	03:52 / **37.4%**	11:46	07:42	04:04 / 35.3%
node-fs-extra	16:28	07:50	08:38 / 51.5%	20:01	09:45	10:16 / 50.8%
node-jsonwebtokcn	19:10	08:54	10:16 / 53.2%	21:09	10:47	10:22 / 48.6%
node-which	23:06	09:32	13:34 / 57.8%	31:33	12:39	18:54 / 59.2%

prompt	11:37	05:19	06:18 / 54.6%	12:34	08:00	04:34 / 35.2%
shelljs	24:25	09:08	15:17 / 62.6%	27:44	11:53	15:51 / 56.5%
winston	14:24	07:52	06:32 / 44.4%	16:27	09:42	06:45 / 37.8%
ws	16:03	07:54	08:09 / 50.5%	23:12	09:09	14:03 / 60.7%
minimist	24:48	08:05	16:43 / **67.1%**	21:39	09:09	11:38 / 53.2%
node-portfinder	12:13	06:13	06:00 / 49.5%	13:27	07:30	05:57 / 41.8%

Table 1. Comparison of the image slimming performance between SCALPEL and SCALPEL2 when the entry point is explicitly specified. The highest and lowest performance increases have been highlighted.

The evaluation results are shown in Table 1. For images based on the node:current-slim base image, *δ*-SCALPEL2 outperforms *δ*-SCALPEL on all the projects. The highest performance increase observed was 67.1% (for minimist) over *δ*-SCALPEL, and the least performance increase observed was 37.1% (for deepmerge) over *δ*-SCALPEL.

For images based on the node:current base image, *δ*-SCALPEL2 again outperforms *δ*-SCALPEL on all the projects. The highest performance increase observed was 63.6% (estraverse) over *δ*-SCALPEL, and the least performance increase observed was 33.5% (yallist) over *δ*-SCALPEL.

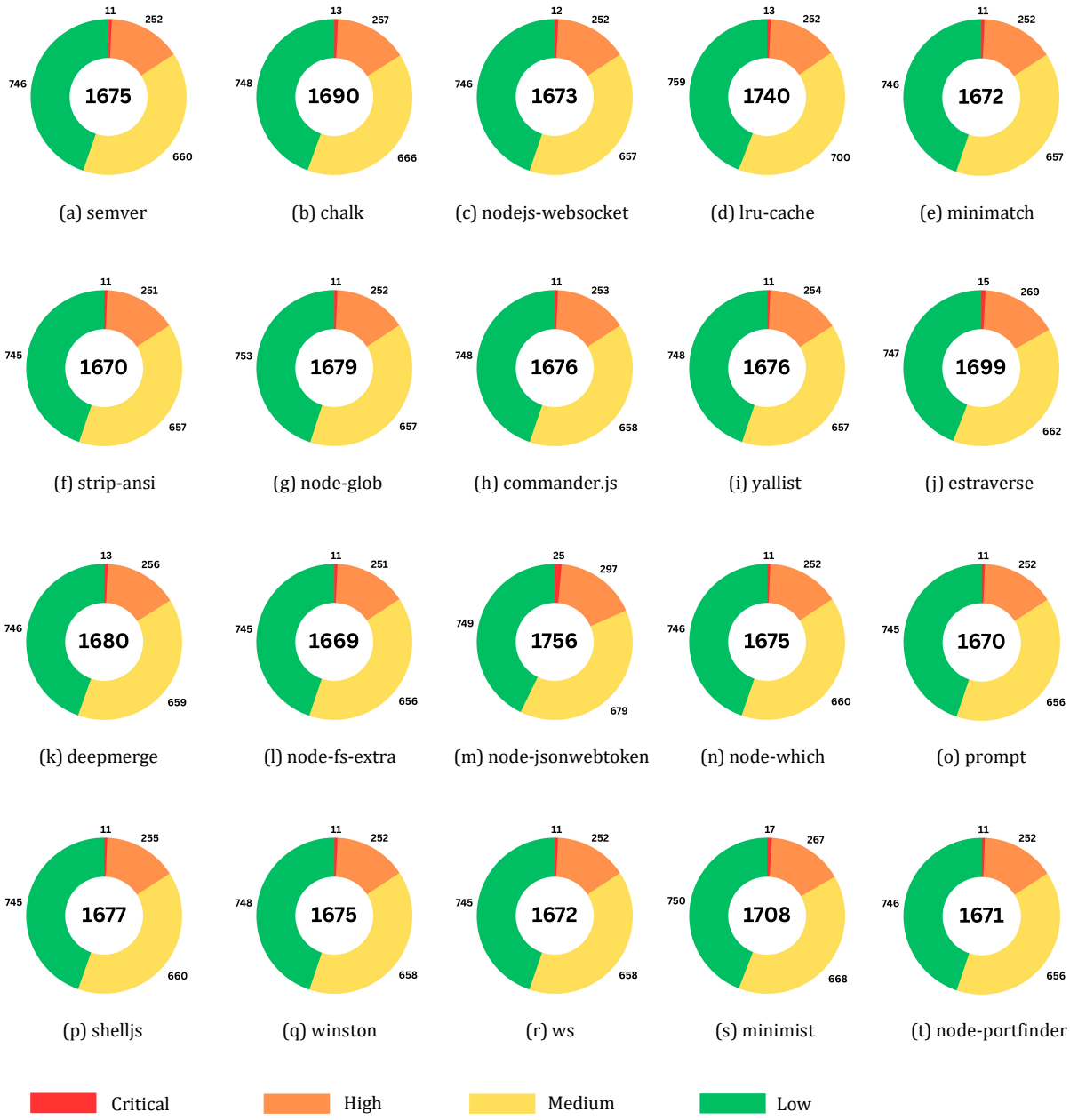
NPM Project	**Base Image Without Entry Point**					
	node:current-slim			**node:current**		
	δ-SCALPEL Time Taken (Minutes)	**δ-SCALPEL2 Time Taken (Minutes)**	**Time Saved / Performance Increase**	**δ-SCALPEL Time Taken (Minutes)**	**δ-SCALPEL2 Time Taken (Minutes)**	**Time Saved / Performance Increase**
semver	24:15	12:30	11:45 / 47.4%	26:07	13:18	12:49 / 47.9%
chalk	23:48	10:15	13:33 / 56.8%	25:36	12:29	13:07 / 51.5%
nodejs-websocket	12:43	06:37	06:06 / 48.8%	13:19	08:10	05:09 / 38.6%
lru-cache	19:27	09:15	10:12 / 52.5%	20:32	11:39	08:53 / 42.0%
minimatch	17:16	07:38	09:38 / 54.7%	16:48	09:39	07:09 / 43.0%
strip-ansi	24:29	09:53	14:36 / 59.1%	26:54	12:01	14:53 / 54.7%
node-glob	15:06	07:36	07:30 / 48.5%	15:43	10:02	05:41 / 35.1%
commander.js	12:53	07:09	05:44 / 43.4%	15:43	09:33	05:18 / 35.7%
yallist	13:57	07:40	06:17 / 45.5%	15:33	09:42	05:51 / 35.9%
estraverse	18:57	08:23	10:34 / 55.7%	26:12	10:57	15:15 / **58.0%**
deepmerge	10:08	06:30	03:38 / **33.5%**	11:32	07:38	03:54 / **31.3%**
node-fs-extra	16:28	07:12	09:16 / 56.3%	18:58	09:58	09:00 / 48.4%
node-jsonwebtokcn	23:21	08:52	14:29 / 61.6%	24:13	11:03	13:10 / 54.3%
node-which	24:42	09:43	14:59 / 59.7%	27:16	12:35	14:41 / 53.1%
prompt	10:58	05:32	05:26 / 49.7%	12:43	07:09	05:34 / 43.0%
shelljs	26:34	09:11	17:23 / 65.4%	28:57	12:49	16:08 / 56.3%
winston	13:08	07:53	05:15 / 39.4%	15:05	09:06	05:59 / 37.1%
ws	15:38	07:34	08:04 / 52.3%	22:06	09:26	12:40 / 56.2%
minimist	23:57	08:06	15:51 / **65.8%**	21:04	10:11	10:53 / 50.0%
node-portfinder	12:03	05:36	06:27 / 52.1%	13:33	07:47	05:46 / 41.0%

Table 2. Comparison of the image slimming performance between SCALPEL and SCALPEL2 when the entry point is not explicitly specified. The highest and lowest performance increases have been highlighted.

The evaluation results are shown in Table 2. For images based on the node:current-slim base image, δ-SCALPEL2 outperforms δ-SCALPEL on all the projects. The highest performance increase observed was 65.8% (for minimist) over δ-SCALPEL, and the least performance increase observed was 33.5% (for deepmerge) over δ-SCALPEL.

For images based on the node:current base image, δ-SCALPEL2 again outperforms δ-SCALPEL on all the projects. The highest performance increase observed was 58% (estraverse) over δ-SCALPEL, and the least performance increase observed was 31.3% (deepmerge) over δ-SCALPEL.

5.3 Q2: Vulnerability Assessment



(a) semver (b) chalk (c) nodejs-websocket (d) lru-cache (e) minimatch

(f) strip-ansi (g) node-glob (h) commander.js (i) yallist (j) estraverse

(k) deepmerge (l) node-fs-extra (m) node-jsonwebtoken (n) node-which (o) prompt

(p) shelljs (q) winston (r) ws (s) minimist (t) node-portfinder

Table 3: CVE Scores of the Projects

To enhance the application's security posture, δ-SCALPEL2 includes integrated vulnerability assessment. The tool has integrated Trivy for vulnerability detection with severity categories like Critical, High, Medium, and Low. Since δ-SCALPEL does not have this feature, there will be no baseline. The scores generated will be a fresh perspective into the security of the projects, and will provide actionable insights to the end users.

6 Conclusion

It has shown that δ-SCALPEL2 is capable of performance improvement and security assessment. The evaluation results demonstrate that the tool effectively reduces the image size while performing faster than the original tool, provides security assessment, without loss in effectiveness.

References

Han, J., Huang, C., Liu, J. and Zhang, T. (2025). An Effective Docker Image Slimming Approach Based on Source Code Data Dependency Analysis. [online] arXiv.org. Available at: https://arxiv.org/abs/2501.03736.

Durieux, T. (2024). Empirical Study of the Docker Smells Impact on the Image Size. doi:https://doi.org/10.1145/3597503.3639143.

Zhang, H., Alhanahnah, M., Leitner, P. and Ali-Eldin, A. (2023). The Cure is in the Cause: A Filesystem for Container Debloating. [online] arXiv.org. Available at: https://arxiv.org/abs/2305.04641v5

Rahaman, M.A. and Ive, J. (2023). Source Code is a Graph, Not a Sequence: A Cross-Lingual Perspective on Code Clone Detection. [online] arXiv.org. Available at: https://arxiv.org/abs/2312.16488

Solanki, R.K. (2024). C Analyzer : A Static Program Analysis Tool for C Programs. [online] arXiv.org. Available at: https://arxiv.org/abs/2403.12973

Han, J., Huang, C., Liu, J. and Zhang, T. (2025). An Effective Docker Image Slimming Approach Based on Source Code Data Dependency Analysis. [online] arXiv.org. Available at: https://arxiv.org/abs/2501.03736.

Openja, M., Majidi, F., Khomh, F., Chembakottu, B. and Li, H. (2022). Studying the Practices of Deploying Machine Learning Projects on Docker. The International Conference on Evaluation and Assessment in Software Engineering 2022. doi:https://doi.org/10.1145/3530019.3530039

Liu, Y., Aitor Hernandez Herranz and Sundin, R.C. (2024). RoboKube: Establishing a New Foundation for the Cloud Native Evolution in Robotics. [online] doi:https://doi.org/10.1109/icara60736.2024.10552996.

Ching, C.-W., Guan, B., Xu, H. and Hu, L. (2024). StraightLine: An End-to-End Resource-Aware Scheduler for Machine Learning Application Requests. arXiv (Cornell

University), [online] pp.5–10. doi:https://doi.org/10.1109/aiotc63215.2024.10748262.

Azzolini, D., Forti, S. and Ielo, A. (2025). Continuous reasoning for adaptive container image distribution in the cloud–edge continuum. Cluster Computing, 28(7). doi:https://doi.org/10.1007/s10586-025-05253-9.