δ-SCALPEL: Docker Image Slimming Based on Source Code Static Analysis

Jiaxuan Han*, Cheng Huang✉, *Member*, Jiayong Liu, and Tianwei Zhang, *Member*

***Abstract*—Containerization is the mainstream of current software development, which enables software to be used across platforms without additional configuration of running environment. However, many images created by developers are redundant and contain unnecessary code, packages, and components. This excess not only leads to bloated images that are cumbersome to transmit and store but also increases the attack surface, making them more vulnerable to security threats. Therefore, image slimming has emerged as a significant area of interest. Nevertheless, existing image slimming technologies face challenges, particularly regarding the incomplete extraction of environment dependencies required by project code. In this paper, we present a novel image slimming model named δ-SCALPEL. This model employs static data dependency analysis to extract the environment dependencies of the project code and utilizes a directed graph named command link directed graph for modeling the image's file system. We select 30 NPM projects and two official Docker Hub images to construct a dataset for evaluating δ-SCALPEL. The evaluation results show that δ-SCALPEL is robust and can reduce image sizes by up to 61.4% while ensuring the normal operation of these projects.**

***Index Terms*—Docker image, Image slimming, Static code analysis, Data dependency analysis, Command link directed graph.**

I. Introduction

As a lightweight virtualization technology designed to create isolated environments, containers differ from virtual machines (VMs) by relying on process-level isolation rather than operating system-level resource isolation [1]–[3]. Docker [4], as a mainstream tool for creating containers, has developed rapidly in recent years. Its advantage lies in allowing developers to package various applications and their dependencies into Docker images, which can then be installed and run on any physical device, such as Linux or Windows devices, to achieve virtualization. This allows applications to be completely decoupled from the underlying hardware, enabling flexible migration and deployment between physical machines [5], [6]. As a result, engineers are freed from complex environment configurations, significantly improving the efficiency and reducing potential risks during deployment [7].

Docker Hub[1] is one of the most popular Docker image registries. Similar to open-source package repositories like NPM[2], Maven[3], and PyPI[4], it provides a centralized platform where users can access, share, and distribute Docker images published by developers [8]–[10]. In the software development process, developers can define the image-building process using a Dockerfile and specify the base image with the `FROM` instruction. During the image building, Docker downloads the specified base image from Docker Hub and then builds the image according to the user's requirements based on that base image [11], [12].

Although users can extend a base image to build one that meets the project's operational requirements, the resulting image may include redundant resources, wasting server storage space and potentially introducing security risks [13], [14]. Image configuration defects, as highlighted by NIST SP 800-190 (Application Container Security Guide) [15], are a core container security risk. The presence of non-essential components can expose containers to unnecessary network threats.

For example, in Fig. 1, a Dockerfile is used to extend the base image `node:latest` as the environment for the `run.js` file. We use the Dive[5] tool to inspect the built image and find that it mainly contains two parts: the base environment part and the project code part. The base environment part takes up 1.1 GB, while the project code part only occupies 319 B. The inspection result shows that there is a great waste of resources in the image extended from the base image. At the same time, we analyze the detail page[6] of the `node:latest` image on Docker Hub and find that it introduces vulnerable `Git` and `Python` environments in the third layer, even though these environments are not required for the project code. Therefore, it is crucial to remove redundant binaries, commands, and privileges to reduce storage waste and minimize the attack surface.

The key of image slimming is how to determine the necessary environment for the project code. DockerSlim is a popular tool for reducing the Docker image size [16]. It creates a temporary container for the target image and hooks key files in the container's file system to identify the necessary environment for running the container. This allows for the

*This work was completed by the author when he was a visiting student at Nanyang Technological University.

Jiaxuan Han, Cheng Huang (corresponding author), and Jiayong Liu are with School of Cyber Science and Engineering, Sichuan University, Chengdu 610207, Sichuan, China (e-mail: zhanSxDrive30i@gmail.com, opcodesec@gmail.com, ljy@scu.edu.cn).

Tianwei Zhang is with College of Computing and Data Science, Nanyang Technological University, Singapore 639798 (email: tianwei.zhang@ntu.edu.sg).

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

[2]https://www.npmjs.com/

[3]https://mvnrepository.com/

[4]https://pypi.org/

[5]https://github.com/wagoodman/dive

[6]https://hub.docker.com/layers/library/node/latest/images/sha256-eb5bb667442cadcd1bb8e6b3d44b2d11bbc5beb280db7f022872d33177b61ca1?context=explore

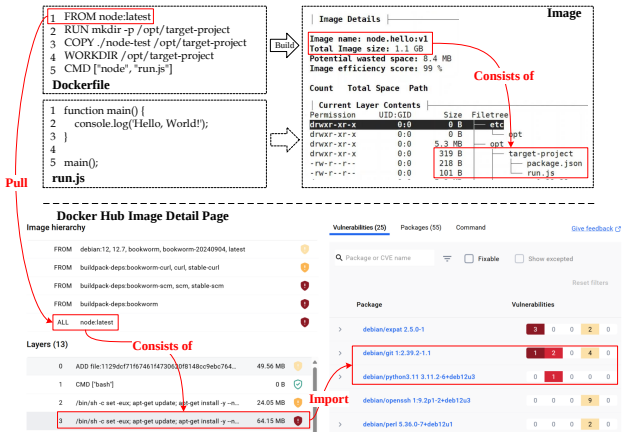


Fig. 1: An example of security risk caused by image resource redundancy.

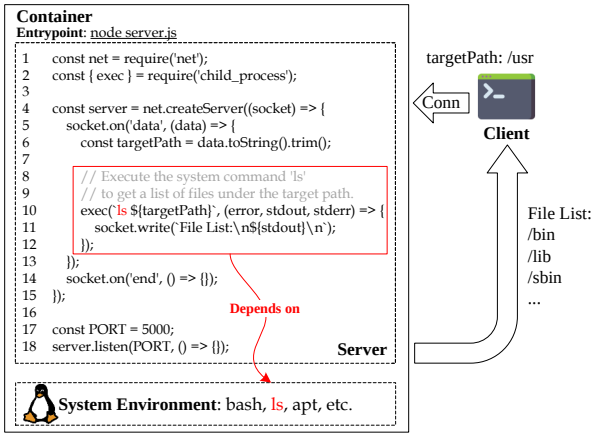


Fig. 2: An example that the code needs to depend on the system environment.

exclusion of content irrelevant to the project code, enabling efficient image slimming. The advantage of DockerSlim is that it can reduce the image size by up to 30 times without altering anything in the image. However, DockerSlim also exhibits two limitations:

Limitation 1: The extraction of the necessary environment for the project code operation is incomplete. Since DockerSlim determines the required system environment based on the runtime behavior of the project code, like many dynamic program analysis techniques, it still faces the issue of incomplete code coverage [17]–[20]. Fig. 2 is an example that the code needs to depend on the system environment. When the server is running, it first starts listening. Upon receiving a client connection, it calls the `exec` API (Application Programming Interface) to execute the system command `ls` that retrieves the file list from the folder specified by the client and returns the results. When using the DockerSlim tool to slim this container, since there is no client connection, the code from lines 5 to 14 will not be executed. As a result, the project's dependency on the system command `ls` cannot be detected, leading to the project code not running properly in the slimmed container.

Fig. 3: An example of project code operation failure caused by image slimming using DockerSlim.

Limitation 2: It is necessary to explicitly give the entry point of the image. This limitation is also due to DockerSlim's reliance on the project's runtime behavior. As shown in Fig. 3, when the image is built, the commands to be executed at runtime (i.e., the container entry point) are not explicitly specified. As a result, DockerSlim cannot identify the system environment required by the project code, leading to the erroneous removal of the Node.js base environment and the project code, which affects the normal operation of the container.

In order to solve these two limitations, we adopt static code analysis technology to extract the environment dependencies of the project code. We propose a novel Docker image slimming model, δ-SCALPEL. δ-SCALPEL extracts environment dependencies through static data dependency analysis of the project source code and its dependent software packages, determining the scope for image slimming. Simultaneously, it constructs a data structure called the command link directed graph to model the image's file system, which enhances the accuracy of extracting environment dependencies. This model can slim the image whether the entry point is explicitly specified or not, while ensuring the normal operation of the project code.

In general, the major contributions of this paper are summarized as follows:

- We propose δ-SCALPEL, a novel and robust static Docker image slimming model. By utilizing static code analysis, this model overcomes the limitations of the dynamic approach, which relies on runtime behavior and often results in unusable or incomplete slimmed images.
- We perform static code analysis to identify data dependencies (on the system environment) within the project source code. Concurrently, we construct a data structure called the command link directed graph to model the image's file system. These ensure the accuracy and reliability of the image slimming process.
- We conduct comprehensive evaluations of δ-SCALPEL

to assess its effectiveness and practical significance. The evaluation results show that δ-SCALPEL can reduce the image size by 61.4% at most, while ensuring that the slimmed image remains functional.

II. Related Work

A. Docker Security

Docker is a widely used technology for containerizing applications along with their dependencies, creating reproducible environments [29], [30]. Research on Docker security can be categorized into image & container security and ecosystem security:

Image & container security. Image and container security primarily refers to the security of the system and software bundled within the container generated by an image, i.e., application security [31]. The container is deployed under the guidance of an automated deployment chain [32], which usually contains third-party programs, software packages, and components. Introducing these third-party elements may bring security risks to the container [30]. Tak et al. [33] pointed out that packages included in images, such as perl, curl, and wget, may contain vulnerabilities. Therefore, software vulnerabilities, configuration defects, and malware are potential threat scenarios within this category [34]–[36].

Ecosystem security. Docker Hub is a central registry where developers can obtain and push images. In a detailed study of Docker Hub images, Tarun Desikan et al. [37] found that over 30% of images in the official repository were highly susceptible to various security attacks. For unofficial images, i.e., those pushed by users without any formal verification from an authoritative entity, this figure rises to approximately 40%. They highlighted that many official Docker Hub images contain packages with CVE (Common Vulnerabilities and Exposures) vulnerabilities, which are often unnecessary in certain cases. If not explicitly removed from the container, these packages may leave the container vulnerable to malicious attacks. Malicious images are also a key concern in the security of the Docker ecosystem. Spring et al. [38] revealed that 17 malicious images hosted on Docker Hub allowed hackers to earn $90,000 through cryptojacking, with these images being downloaded over 5 million times in a single year. The inheritance of Docker images can propagate security vulnerabilities from parent images to child images, thereby impacting the entire Docker ecosystem [29].

B. Image Debloating

Base images deliver standardized, pre-built environments (OS kernel, runtime, core libraries, tools). Developers use the `FROM` instruction to inherit this foundation, then focus solely on adding application-specific layers (i.e., dependencies, code, and configurations). This eliminates environment inconsistencies, avoids redundant rebuilds, and boosts deployment efficiency. Nonetheless, base images such as official generic versions (e.g., Ubuntu/Debian latest) prioritize broad compatibility by including libraries/tools for diverse use cases. For specific projects, many components become unnecessary at runtime. Direct usage of unoptimized base images wastes storage and may introduce security risks, making image debloating (as known as image slimming) essential for production efficiency and security.

Cimplifier [39], proposed by Rastigi et al., is an automated container debloating model that profiles containerized application resource usage through dynamic analysis of executable behavior. It operates without requiring application source code and remains independent of specific languages or runtime stacks (e.g., JVM), supporting diverse container types. Specifically, Cimplifier identifies processes' access patterns to files, IPC (Inter-Process Communication), and network objects during original container execution. Combining these observations with user-defined policies, it partitions the original container into multiple isolated containers, each containing only resources essential to its designated function. These isolated containers communicate minimally when necessary and are interconnected via RPE (Remote Process Execution). δ-SCALPEL relies on static source code analysis to create a single slimmed image, whereas Cimplifier utilizes dynamic analysis and does not require source code, partitioning the container into multiple independent containers glued together by RPE.

SummSlim [13], proposed by Zhang et al., is an automated container image debloating model. It takes an original image as input. During container initialization, the model employs dynamic analysis to monitor container processes (including child processes) using the Linux `strace` command, filtering file-related system calls and recording accessed files. Concurrently, it applies static analysis to each image layer, preserving files from `ADD/COPY` layers while debloating the `FROM` base layer (retaining only runtime-essential components). Finally, SummSlim synthesizes necessary files identified through both analyses to generate the debloated image.

DockerSlim [16] (now called MinToolkit/Mint, or SlimToolkit) is a widely adopted container optimization tool, enabling developers to inspect, debloat, and debug Docker containers. DockerSlim integrates dynamic and static analysis: It statically analyzes the content of the original image, then dynamically analyzes the application by launching a temporary container, monitoring its runtime behavior, and identifying the files, binary dependencies, and system calls actually utilized. Based on these, it generates a new Dockerfile containing only the necessary runtime dependencies and files, resulting in the construction of a minimized image. DockerSlim relies on dynamic runtime analysis, monitoring the application's actual behavior when running in a container to remove unused dependencies.

SummSlim and DockerSlim are both hybrid image slimming models, the core of which is to determine the project code's dependencies on the system environment by observing the container's runtime behavior. A limitation of this kind of approach is that it requires all code paths to be executed; otherwise, the resulting slimmed image can easily fail due to incomplete code dependency extraction. In contrast, δ-SCALPEL adopts the static code dependency analysis method, which extracts the project code's dependencies independent of the image, thus avoiding this problem.

III. Methodology

A. Overview

The framework of δ-SCALPEL is shown in Fig. 4, which is divided into three steps:

- **Step 1: Environment Dependency Extraction.** Create an image for data dependency analysis of the project code and its NPM dependencies, extracting environment dependencies (i.e., system commands executed during `exec` API calls) from the source code.
- **Step 2: Slim rootfs Construction.** Perform a static scan of the fat image to extract metadata and build a temporary image. Next, scan its file system, identify system commands, and determine which content should be removed based on the command token list extracted in step 1. Finally, construct a slim root file system (rootfs).
- **Step 3: Slim Image Building.** Create a slim image based on the `rootfs.tar` file constructed in step 2.

B. Environment Dependency Extraction

This step is divided into three modules: package filtering, data dependency analysis, and command token extraction. Before performing these modules, environment preparation is required. We choose Node.js as the default language and perform static code analysis using CodeQL[7]. In order to extract environment dependencies, we configure an image through a Dockerfile. Specifically, we set the base image to `node:current-slim`, and create the installation directory for NPM packages. Then, we configure the environment for CodeQL. Afterward, we initialize the target project with `npm install` command to install all its dependencies. Finally, an image for environment dependency extraction is generated.

1) Package Filtering: With the expansion of the software ecosystem, developers increasingly import open source packages to enhance development efficiency. Consequently, during code execution, both control flow and data flow extend into these imported packages. So environment dependency extraction is required for these packages. However, analyzing the entire dependency chain of the project code is challenging due to the large number of packages involved. For example, in the nodejs-websocket[8] project, the dependency chain includes 30 packages. Analyzing data dependencies of these packages individually would be highly time-consuming. To optimize this process, we filter the packages in the project's dependency chain. In particular, we traverse the project's dependency chain, analyze the JavaScript code of each NPM package, and use regular expressions to identify packages with `exec` API calls. In this way, we can filter out most packages unrelated to environment dependency extraction, thereby speeding up the execution of δ-SCALPEL.

2) Data Dependency Analysis: This is the core module of step 1, focusing on extracting string constants from `exec` API calls. Since developers do not always use string constants (i.e., the commands need to be executed) as parameters when calling the `exec` API, it is necessary to analyze data dependencies of the `exec` API calls. In this paper, we utilize CodeQL to achieve this. First, we use CodeQL to parse the target project and the NPM packages containing `exec` API calls, and build the AST database. Then, we define the source and sink nodes in the source code: sources include `VariableDeclarator`, `Assignment`, and `CallExpr` nodes, while sinks are the arguments of the `CallExpr` nodes named `exec`. Finally, we use the `TaintTracking` package of CodeQL to identify all paths from sources to sinks.

3) Command Token Extraction: After obtaining all the paths from sources to sinks, we traverse the nodes along these

[7]https://github.com/github/codeql

[8]https://www.npmjs.com/package/nodejs-websocket

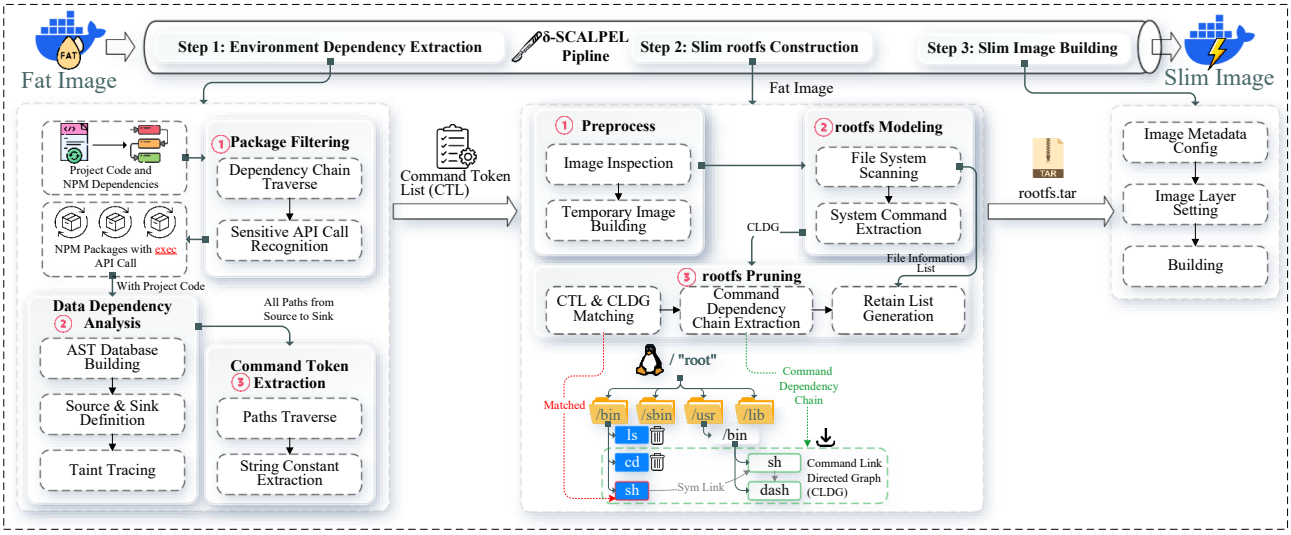


Fig. 4: The framework of δ-SCALPEL.

Algorithm 1: CLDG Construction

```
   Input: Commands supported by the system sysCmdList
   Result: Command link directed graph cmdLinkDiGraph
1  func buildCmdLinkDiGraph(linkNode):
2      if isCmdName(linkNode.nodeName) then
3          cmdAbsPath ← getAbsPath(linkNode.nodeName);
4      end
5      else if isSymLink(linkNode.nodeName) then
6          cmdAbsPath ← getRefPath(linkNode.nodeName);
7      end
8      if cmdLinkDiGraph.hasKey(cmdAbsPath) then
9          currentNode ← cmdLinkDiGraph.get(cmdAbsPath);
10     end
11     else
12         currentNode ← new(
13                 nodeName=cmdAbsPath,
14                 nextNode=null
15             );
16         cmdLinkDiGraph.append(currentNode);
17     end
18     if linkNode.nextNode == null then
19         linkNode.nextNode ← currentNode;
20     end
21     buildCmdLinkDiGraph (linkNode.nextNode);
22 return
23 init(cmdLinkDiGraph); // global variable
24 for sysCmd in sysCmdList do
25     if isBuiltInCmd(sysCmd) then
26         continue;
27     end
28     linkNode ← new(
29             nodeName=sysCmd,
30             nextNode=null
31         );
32     buildCmdLinkDiGraph (linkNode);
33 end
```

Algorithm 2: CLDG Expanding

```
   Input: Command link directed graph cmdLinkDiGraph
   Result: Command link directed graph cmdLinkDiGraph
1  for linkNode in cmdLinkDiGraph do
2      if !isPath(linkNode.name) then
3          continue;
4      end
5      allParentDirs ← getAllParentDirs(linkNode.nodeName);
6      for dir in allParentDirs do
7          isSymLink ← false;
8          hasSymLink ← false;
9          if isSymPath(dir) then
10             isSymLink ← true;
               /* dir is a symbolic soft link            */
11             refDir ← getRefPath(dir);
12             newCmdPath ← linkNode.nodeName.replace(dir, refDir);
13         end
14         else if hasSymPath(dir) then
               /* dir is not a symbolic soft link, but it
                  has related symbolic soft link         */
15             hasSymLink ← true;
16             linkDir ← getLinkPath(dir);
17             newCmdPath ← linkNode.nodeName.replace(dir, linkDir);
18         end
19         if hasFile(newCmdPath) then
20             if !cmdLinkDiGraph.hasKey(newCmdPath) then
21                 newNode ← new(
22                         nodeName=newCmdPath,
23                         nextNode=null
24                     );
25                 buildCmdLinkDiGraph(newNode)
26                 if isSymLink then
27                     linkNode.nextNode ← newNode;
28                 end
29                 else if hasSymLink then
30                     newNode.nextNode ← linkNode;
31                 end
32                 cmdLinkDiGraph.append(newNode);
33             end
34         end
35     end
36 end
```

paths to identify string constants and split them by spaces to extract command tokens. Additionally, the shell script files also have dependencies on the system environment. Therefore, we analyze all shell scripts in the project code folder and the `docker-entrypoint.sh` file, extracting tokens from them.

It should be noted that these tokens are not necessarily commands; they may also represent strings without command-related meaning, or command parameters. They are merely considered potential commands, and will be addressed further in Sect. III-C3.

C. Slim rootfs Construction

After completing the extraction of environment dependencies, the command token list (CTL) is generated. The next step is to construct a slim rootfs for the fat image. In this step, we design three modules: preprocess, rootfs modeling, and rootfs pruning, which are used for analyzing the fat image's file system, identifying the content that needs to be removed, and constructing a slim rootfs.

1) Preprocess: First, we perform a static inspection of the fat image to obtain metadata, including its configuration and architecture. Then, we construct a temporary image based on the metadata, which contains a component called image analyzer. The goal of the image analyzer is to model the fat image's rootfs (rootfs modeling, Sect. III-C2) and prune it (rootfs pruning, Sect. III-C3) to create a slim rootfs.

2) rootfs Modeling: In this module, we first collect the file information list by scanning the entire rootfs to retrieve all files, folders, and their permissions. At the same time, we extract the symbolic soft link relationships between these files and folders. Then we extract all the commands supported by the system to build the command link directed graph (CLDG).

The usage of commands can be categorized into two types: directly using the command name and using the absolute path of the command's binary file. So, we need to get the absolute path of a command. However, due to the symbolic soft link mechanism, one path can be linked to another, meaning the absolute path of a binary file obtained by the `which` command may not reflect its actual storage location. For example, running the `which sh` command returns the absolute path `/usr/bin/sh`. However, this path is a symbolic soft link to the `dash` command, whose absolute path is `/usr/bin/dash`. Additionally, `/bin` is a symbolic soft link to `/usr/bin`, meaning `/bin/sh` and `/usr/bin/sh` point to the same file. These indicate that for the two different ways of using commands, the paths that need to be included when generating the retain list are different. When the project code directly uses the `sh` command, the paths included in the retain list must be `[/usr/bin/sh, /usr/bin/dash]`, just as when `/usr/bin/sh` is used. However, when `/bin/sh` is used, the paths included in the retain list must be `[/bin/sh, /usr/bin/sh, /usr/bin/dash]`.

To this end, we design a data structure called command link directed graph to handle the above situation. First, we construct the command link directed graph using Algo. 1. Then, we extend it with Algo. 2 to handle the case of folder

symbolic soft links. For the `sh` command, we can construct a command link directed graph as illustrated in Fig. 5.

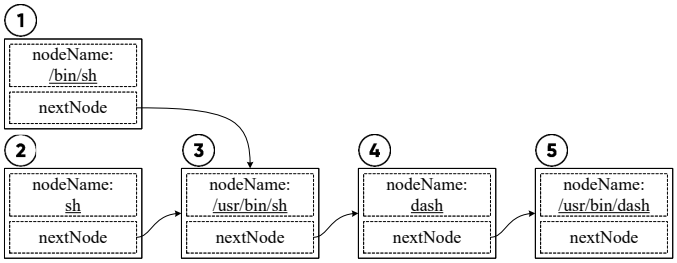


Fig. 5: A command link directed graph example for the `sh` command.

3) rootfs Pruning: After obtaining the file information list and the command link directed graph, we need to prune the fat image's rootfs according to the command token list generated in step 1. First, we match each token in the CTL with the node names in the CLDG. If a token t_i from CTL matches a node n_j in CLDG, we designate this node as the starting node and extract the file paths (i.e., the value of `nodeName`) from all subsequent nodes, and store them in the retain list.

For example, consider the CLDG shown in Fig. 5. If the token in the CTL is `/bin/sh`, δ-SCALPEL will locate node 1 in the CLDG and add its `nodeName` to the retain list. If node 1 has a non-null `nextNode`, δ-SCALPEL accesses the node it points to (i.e., node 3) and adds its `nodeName` to the retain list. It then proceeds to the subsequent nodes (node 4 and node 5) and adds their `nodeName` to the retain list. In other words, when the token contained in the CTL is `/bin/bash`, δ-SCALPEL visits nodes 1, 3, 4, and 5 in turn and adds their `nodeName` to the retain list. When the token is `dash`, δ-SCALPEL visits nodes 4 and 5 and adds their `nodeName` values to the retain list.

Additionally, for each path, we use the `ldd` command to obtain its dependent dynamic link libraries and add them to the retain list. Finally, we add all files from the file information list, excluding those with paths containing `/bin/`, `/sbin/`, or `/lib/`, to the retain list. The retain list is then used to construct the slim `rootfs.tar` file.

In this section, we address the issue that the tokens in the CTL, as mentioned in Sect. III-B3, do not necessarily consist entirely of command tokens by matching them with the CLDG.

D. Slim Image Building

Based on the static inspection result of the fat image from step 2, we first configure the metadata for the slim image, including the image name, exposed ports, image architecture, etc. Then we use the slim `rootfs.tar` file to set the layers of the slim image. At this point, δ-SCALPEL generates the corresponding slim image from the fat image.

IV. Evaluation

We evaluate δ-SCALPEL by answer the following three research questions (RQs):

- ***RQ1***: How much can the image size be reduced by using δ-SCALPEL?
- ***RQ2***: How long will it take to perform image slimming using δ-SCALPEL?
- ***RQ3***: How well does δ-SCALPEL reduce image attack surfaces and potential vulnerabilities?

A. Experiment Setup

1) Dataset: To evaluate the δ-SCALPEL model proposed in this paper, we select the top 30 NPM projects from the top 1000 most depended-upon packages listed on GitHub[9]. Each of these selected projects contains executable test suites triggered directly by the `npm test` command without any other configuration. For each project, we download its source code from the address specified in the NPM repository and include it in a Dockerfile to create an image. We use Docker Hub's official images, `node:current-slim` and `node:current`, to create images for each project. Finally, we create a dataset containing 120 images for model evaluation. In order to verify whether the project runs correctly in the container generated from the slimmed image, we execute the project's test suites using the `npm test` command and assess its status based on the test suites' results: If the test suites' results before and after slimming are consistent, the image slimming is considered successful; otherwise, it is deemed a failure.

There are two points that need explanation. First, the criterion for determining whether δ-SCALPEL runs successfully in this paper is whether the slimmed image can operate normally. We assume that the image built by developers is intended to support the normal operation of the project, that is, to ensure the normal functionality of the project. To this end, we run the test suite of the project within the container to determine if the container operates normally: If the test suite runs successfully, it indicates that the container functions properly; otherwise, the container fails to operate normally, and the image slimming is considered unsuccessful. Second, although we select JavaScript-based projects to build the dataset, δ-SCALPEL can be extended to other programming languages because the code analysis phase is implemented using CodeQL, which can analyze more than 11 programming languages[10], such as Java and C/C++.

2) Implementation: All experiments are conducted on a host with 16 CPU cores and 32 GB of memory. We use Go as the development language for δ-SCALPEL, perform static analysis of Node.js code with CodeQL, and utilize the go-dockerclient[11] package as the client for the Docker remote API.

B. RQ1: The Image Size Reduced by Using δ-SCALPEL

In a Dockerfile, users can use `CMD` or `ENTRYPOINT` to specify the command or script to run when starting a container [40]. These two commands are optional. As a result, when building an image with a Dockerfile, users can choose to explicitly specify the container's entry point or leave it unspecified. This section evaluates the image slimming effect

[9]https://gist.github.com/anvaka/8e8fa57c7ee1350e3491

[10]https://codeql.github.com/docs/codeql-overview/supported-languages-and-frameworks/

[11]https://github.com/fsouza/go-dockerclient

(i.e., size of the slimmed image, slimming ratio, and running status of the container generated by the slimmed image) of δ-SCALPEL. We select DockerSlim [16] and SummSlim [13] as baselines. For DockerSlim, we use its default `build` command, which analyzes, profiles, and optimizes container images to generate supported security profiles, and is its most popular option; for SummSlim, we follow the usage described on its GitHub page[12]. We compare the image slimming effects of δ-SCALPEL, SummSlim, and DockerSlim in two scenarios: specifying the entry point and not specifying the entry point.

1) Specify the Entry Point: We specify the entry point of the image for each project in its Dockerfile as `CMD ["npm", "test"]`, then build project images based on the base images `node:current-slim` and `node:current`, respectively. For the generated project images, we use δ-SCALPEL, SummSlim, and DockerSlim to perform image slimming. We then start the slimmed images using the `docker run` command and observe the container's running state.

The evaluation results are shown in Tab. I. For images based on the `node:current-slim` base image, δ-SCALPEL achieves a slimming rate ranging from 9.7% to 26.5%. The minimum rate (9.7%) is observed in the strip-ansi project image, reducing its size from 607 MB to 548 MB, while the maximum rate (26.5%) is achieved in the node-portfinder project image, shrinking from 226 MB to 166 MB. SummSlim performs slightly better, with a slimming rate between 8.2% and 30.8%. It records the minimum rate (8.2%) on the strip-ansi project image (shrinking from 607 MB to 557 MB) and the maximum (30.8%) on the nodejs-websocket project image (shrinking from 227 MB to 157 MB). Notably, DockerSlim significantly outperforms both, achieving slimming rates between 45.1% and 79.6%. Its minimum rate (45.1%) is found in the node-portfinder project image (226 MB to 124 MB), and its remarkable maximum rate of 79.6% is achieved with the strip-ansi project image, which is dramatically reduced from 607 MB to a mere 124 MB. When the base image is `node:current`, all models show substantially higher slimming rates: δ-SCALPEL's rate ranges from 46.3% to 61.4%, SummSlim's from 57.6% to 81.9%, and DockerSlim's from 89.2% to 91.9%.

Regarding the functionality of the slimmed images, all images processed by δ-SCALPEL function normally. Similarly, all images processed by SummSlim function normally, with the single exception of the lru-cache project image. In contrast, DockerSlim exhibits poor functional preservation; only the images for the nodejs-websocket, prompt, and cookie-parser projects processed by it function normally.

2) Do not Specify the Entry Point: In this subsection, we evaluate the robustness of δ-SCALPEL in image slimming under the scenario where the container entry point is not specified. We use the same method as in Sect. IV-B1 to build images for the 30 NPM projects, based on both `node:current-slim` and `node:current`. However, in the Dockerfile, we remove the entry point. Similarly, we use δ-SCALPEL, SummSlim, and DockerSlim to slim these images. Afterward, we access the containers generated from these slimmed images, manually enter `npm test` command in the project root directory to run the test suites, and observe the results.

The evaluation results are shown in Tab. II. The slimming effect of δ-SCALPEL is consistent regardless of whether an entry point is specified: All NPM projects run correctly within containers generated from the slimmed images, and the resulting image size remains the same. When using SummSlim, images built on `node:current-slim` are reduced to an average size of 297 MB, achieving a maximum slimming rate of 30.8% and a minimum of 8.2%; similarly, images based on `node:current` are reduced to an average of 388 MB, with slimming rates ranging from 57.6% to 81.9%, and all these slimmed images run normally. In contrast, while DockerSlim reduced all images to a consistent 123 MB, the resulting slimmed images were unable to run normally.

Analysis of the results shows that DockerSlim fails to slim the image without specifying the container entry point. This is because the inability to capture the code's runtime behavior forces DockerSlim to retain only the basic container runtime environment, resulting in the slimmed image failing to run reliably and lacking robustness. SummSlim, however, does not suffer from this limitation, and its slimming effect even surpasses that of δ-SCALPEL. To this end, we analyze the image slimming results from δ-SCALPEL and SummSlim. Our analysis reveals that size differences in the slimmed images are primarily observed in the following five system paths: `/usr/bin`, `/usr/include`, `/usr/lib`, `/usr/local`, and `/usr/share`. SummSlim achieves a higher slimming rate in these paths compared to δ-SCALPEL.

Further analysis shows that although we manually remove the container entry point in the Dockerfile during image building, SummSlim adopts the base images' entry point (i.e., `node:current-slim` and `node:current`) for built images. This entry point, `docker-entrypoint.sh`, dynamically prepends the `node` command based on the first argument's characteristics (whether it is an option, non-system command, or non-executable file) before executing the final command via `exec`. With no explicit startup command specified, SummSlim captures `node` as the effective entry point. Consequently, when processing images without an explicit entry point, SummSlim executes `node` command at startup and captures its system dependencies. Manual dataset analysis reveals that project test suites rarely involve complex system calls, with relevant scenarios remaining simple. Therefore, SummSlim successfully slims images using Node.js runtime dependencies even without explicit entry points.

Nonetheless, SummSlim still lacks sufficient robustness, primarily in two scenarios: (1) the container entry point is not specified, and the project code has complex dependencies on the system runtime environment; (2) a container entry point is specified, but the project code is not executed directly or not all the code paths are executed. We conduct an evaluation to compare the robustness of δ-SCALPEL and Summslim in these two scenarios, with results shown in Tab. III.

In the first scenario, containers created from images slimmed by SummSlim cannot guarantee the proper execution of the project code. For example, if the project

[12]https://github.com/prcuzz/SummSlim

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3640123

TABLE I: Comparison of the image slimming effect among δ-SCALPEL, SummSlim, and DockerSlim when the entry point is explicitly specified. ● indicates that the slimmed image runs normally, while ○ indicates that it fails to run normally.

Project	Model	Basic Image (With Entry Point)					
		node:current-slim			node:current		
		Original Size	Size After Slimming/ Slimming Ratio	Running Status	Original Size	Size After Slimming/ Slimming Ratio	Running Status
semver	δ-SCALPEL	551MB	492MB/10.7%	●	1.45GB	769MB/48.2%	●
	SummSlim		491MB/10.9%	●		585MB/60.6%	●
	DockerSlim		126MB/77.1%	○		124MB/91.6%	○
chalk	δ-SCALPEL	555MB	496MB/10.6%	●	1.45GB	772MB/48%	●
	SummSlim		509MB/8.3%	●		603MB/59.4%	●
	DockerSlim		125MB/77.5%	○		125MB/91.6%	○
nodejs-websocket	δ-SCALPEL	227MB	168MB/26%	●	1.12GB	445MB/61.2%	●
	SummSlim		157MB/30.8%	●		250MB/78.2%	●
	DockerSlim		124MB/45.4%	●		124MB/89.2%	●
lru-cache	δ-SCALPEL	372MB	321MB/13.7%	●	1.27GB	598MB/54%	●
	SummSlim		310MB/16.7%	●		235MB/81.9%	○
	DockerSlim		133MB/64.2%	○		124MB/90.4%	○
minimatch	δ-SCALPEL	332MB	272MB/18.1%	●	1.23GB	549MB/56.4%	●
	SummSlim		261MB/21.4%	●		354MB/71.9%	●
	DockerSlim		123MB/63%	○		133MB/89.4%	○
strip-ansi	δ-SCALPEL	607MB	548MB/9.7%	●	1.50GB	825MB/46.3%	●
	SummSlim		557MB/8.2%	●		651MB/57.6%	●
	DockerSlim		124MB/79.6%	○		124MB/91.9%	○
node-glob	δ-SCALPEL	335MB	275MB/17.9%	●	1.23GB	552MB/56.2%	●
	SummSlim		264MB/21.2%	●		357MB/71.7%	●
	DockerSlim		133MB/60.3%	○		133MB/89.4%	○
commander.js	δ-SCALPEL	389MB	328MB/15.7%	●	1.29GB	605MB/54.2%	●
	SummSlim		317MB/18.5%	●		411MB/68.9%	●
	DockerSlim		124MB/68.1%	○		125MB/90.5%	○
yallist	δ-SCALPEL	384MB	324MB/15.6%	●	1.28GB	601MB/54.1%	●
	SummSlim		313MB/18.5%	●		407MB/68.9%	●
	DockerSlim		123MB/68%	○		133MB/89.9%	○
estraverse	δ-SCALPEL	276MB	216MB/21.7%	●	1.17GB	493MB/58.9%	●
	SummSlim		207MB/25%	●		300MB/75%	●
	DockerSlim		123MB/55.4%	○		124MB/89.7%	○
deepmerge	δ-SCALPEL	263MB	202MB/23.2%	●	1.16GB	479MB/59.7%	●
	SummSlim		193MB/26.6%	●		286MB/75.9%	●
	DockerSlim		125MB/52.5%	○		126MB/89.4%	○
node-fs-extra	δ-SCALPEL	315MB	254MB/19.4%	●	1.21GB	531MB/57.1%	●
	SummSlim		249MB/21%	●		342MB/72.4%	●
	DockerSlim		135MB/57.1%	○		124MB/90%	○
node-jsonwebtoken	δ-SCALPEL	313MB	252MB/19.5%	●	1.21GB	529MB/57.3%	●
	SummSlim		244MB/22%	●		338MB/72.7%	●
	DockerSlim		123MB/60.7%	○		131MB/89.4%	○
node-which	δ-SCALPEL	553MB	492MB/11%	●	1.45GB	769MB/48.2%	●
	SummSlim		492MB/11%	●		584MB/60.7%	●
	DockerSlim		125MB/77.4%	○		126MB/91.5%	○
prompt	δ-SCALPEL	253MB	192MB/24.1%	●	1.15GB	470MB/60.1%	●
	SummSlim		184MB/27.3%	●		278MB/76.4%	●
	DockerSlim		124MB/51%	●		124MB/89.5%	●
shelljs	δ-SCALPEL	345MB	284MB/17.7%	●	1.29GB	613MB/53.6%	●
	SummSlim		279MB/19.1%	●		451MB/65.9%	●
	DockerSlim		127MB/63.2%	○		126MB/90.5%	○
winston	δ-SCALPEL	292MB	230MB/21.2%	●	1.19GB	507MB/58.4%	●
	SummSlim		222MB/24%	●		315MB/74.1%	●
	DockerSlim		125MB/57.2%	○		125MB/89.7%	○
ws	δ-SCALPEL	307MB	246MB/19.9%	●	1.2GB	523MB/57.4%	●
	SummSlim		240MB/21.8%	●		334MB/72.8%	●
	DockerSlim		124MB/59.6%	○		126MB/89.7%	○
minimist	δ-SCALPEL	303MB	242MB/20.1%	●	1.2GB	519MB/57.8%	●
	SummSlim		236MB/22.1%	●		356MB/71%	●
	DockerSlim		124MB/59.1%	○		124MB/89.9%	○
node-portfinder	δ-SCALPEL	226MB	166MB/26.5%	●	1.12GB	443MB/61.4%	●
	SummSlim		157MB/30.5%	●		251MB/78.1%	●
	DockerSlim		124MB/45.1%	○		124MB/89.2%	○
css-loader	δ-SCALPEL	450MB	369MB/18%	●	1.35GB	646MB/53.3%	●
	SummSlim		361MB/19.8%	●		453MB/67.2%	●
	DockerSlim		123MB/72.7%	○		125MB/91%	○
express	δ-SCALPEL	314MB	253MB/19.4%	●	1.21GB	530MB/57.2%	●
	SummSlim		245MB/22%	●		338MB/72.7%	●
	DockerSlim		126MB/59.9%	○		125MB/89.9%	○
async	δ-SCALPEL	342MB	271MB/20.8%	●	1.24GB	548MB/56.8%	●
	SummSlim		263MB/23.1%	●		356MB/72%	●
	DockerSlim		123MB/64%	○		124MB/90.2%	○
yargs	δ-SCALPEL	529MB	467MB/11.7%	●	1.43GB	745MB/49.1%	●
	SummSlim		458MB/13.4%	●		552MB/62.3%	●
	DockerSlim		123MB/76.7%	○		124MB/91.5%	○

TABLE I: (Continued) Comparison of the image slimming effect among δ-SCALPEL, SummSlim, and DockerSlim when the entry point is explicitly specified. • indicates that the slimmed image runs normally, while ○ indicates that it fails to run normally.

Project	Model	Basic Image (With Entry Point)					
		node:current-slim			node:current		
		Original Size	Size After Slimming/ Slimming Ratio	Running Status	Original Size	Size After Slimming/ Slimming Ratio	Running Status
body-parser	δ-SCALPEL	321MB	260MB/19%	•	1.22GB	537MB/57%	•
	SummSlim		251MB/21.8%	•		344MB/72.5%	•
	DockerSlim		124MB/61.4%	○		126MB/89.9%	○
gulp	δ-SCALPEL	325MB	263MB/19.1%	•	1.22GB	541MB/56.7%	•
	SummSlim		255MB/21.5%	•		349MB/72.1%	•
	DockerSlim		123MB/62.2%	○		126MB/89.9%	○
postcss-loader	δ-SCALPEL	424MB	362MB/14.6%	•	1.32GB	639MB/52.7%	•
	SummSlim		353MB/16.7%	•		447MB/66.9%	•
	DockerSlim		123MB/71%	○		134MB/90.1%	○
cookie-parser	δ-SCALPEL	316MB	255MB/19.3%	•	1.21GB	532MB/57.1%	•
	SummSlim		247MB/21.8%	•		340MB/72.6%	•
	DockerSlim		125MB/60.4%	•		126MB/89.8%	•
browserity	δ-SCALPEL	278MB	217MB/21.9%	•	1.17GB	494MB/58.8%	•
	SummSlim		209MB/24.8%	•		301MB/74.9%	•
	DockerSlim		123MB/55.8%	○		125MB/89.6%	○
log4js	δ-SCALPEL	449MB	387MB/13.8%	•	1.35GB	664MB/52%	•
	SummSlim		378MB/15.8%	•		472MB/65.9%	•
	DockerSlim		125MB/72.2%	○		127MB/90.8%	○

TABLE II: Comparison of the image slimming effect among δ-SCALPEL, SummSlim, and DockerSlim when the entry point is not explicitly specified.

Model	Basic Image (Without Entry Point)					
	node:current-slim			node:current		
	Avg. Size After Slimming	Max Slimming Ratio	Min Slimming Ratio	Avg. Size After Slimming	Max Slimming Ratio	Min Slimming Ratio
δ-SCALPEL	303MB	26.5%	9.70%	582MB	61.4%	46.3%
SummSlim	297MB	30.8%	8.2%	388MB	81.9%	57.6%
DockerSlim	123MB	79.7%	45.6%	123MB	92%	89.3%

TABLE III: Robustness comparison of δ-SCALPEL and Summslim. • means that the project code can run normally in the container generated by the slimmed image, while ○ means that it cannot run normally.

Model	Scenario	With Entry Point		Without Entry Point
	Case	Project Code Not Executed Directly	Not All Code Paths Executed	Complex Project Runtime Dependencies
δ-SCALPEL	Running Status	•	•	•
SummSlim		○	○	○

code needs to execute the system command like `ps aux --sort=-%cpu | head -10 | awk 'print "USER:"$1,"PID:"$2,"CPU:"$3,"%","MEMORY:"$4,"%","COMMAND:"$11'`, SummSlim will fail to capture the dependencies (e.g., `ps`, `head`, `awk`) on the runtime environment because the code isn't actually executed when no entry point is specified. Consequently, image slimming fails.

In the second scenario, containers created from SummSlim-slimmed images still cannot guarantee the proper execution of the project code. For instance, in certain development or debugging scenarios, developers may prefer the container to start without automatically launching the project code. Instead, they might want to first enter the container's Bash environment and manually execute commands to start the project or perform debugging steps. In such cases, the container's entry point is often set to `["/bin/bash"]`. On the other hand, even if the image's entry point directly launches the project code, SummSlim can capture runtime dependencies only for executed code paths. This incomplete coverage may lead to slimming failure. A typical example is code containing conditional branches, where some paths are never executed because the entry point does not trigger all possible execution flows.

For the aforementioned two scenarios, δ-SCALPEL demonstrates advantages because, regardless of whether the container entry point is explicitly specified or not, and regardless of whether it directly executes the project code, δ-SCALPEL performs static dependency analysis on the code independently of the container image, thereby ensuring greater integrity of the extracted dependencies.

3) The Composition of Slimming: Understanding the structural changes during the slimming process is important, as this provides deeper insights into how the model achieves its results. In this section, we conduct a case study to analyze the structural changes before and after the slimming of the express project image built from the `node:current` base image. The results are shown in Fig. 6.

During the image slimming process, the structural changes in `/usr/bin`, `/usr/sbin`, and `/usr/lib` are the most significant: The size of the `/usr/bin` directory decreases from 111 MB to 3.1 MB, with unused programs like `ls` and `addpart` removed. The `/usr/sbin` directory decreases from 6.5 MB to 195 kB, including removal of utilities like `mkfs` and `sulogin`. Similarly, the `/usr/lib` directory size decreases significantly from 570 MB to 6.4 MB, with libraries such as `apt` eliminated.

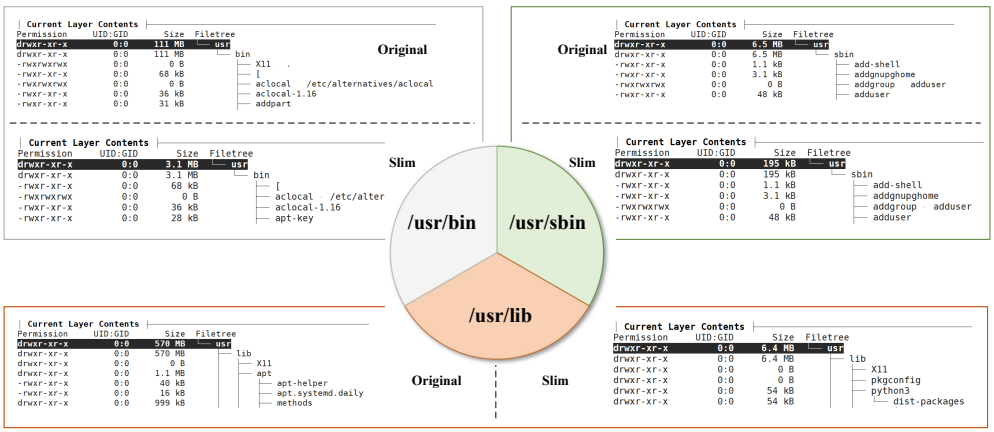


Fig. 6: An example of the structural changes before and after the slimming of the express image based on `node:current`.

`/usr/bin` stores the vast majority of user-executable binaries, containing the programs most frequently used by users for daily operations. `/usr/sbin` houses system administration binaries intended for privileged users (such as system administrators), typically utilized for system maintenance, configuration, and repair tasks. `/usr/lib` contains software libraries and auxiliary files that provide essential services to programs located in `/usr/bin` and `/usr/sbin`. These three folders contain basic components designed to meet all possible Linux application scenarios, but these components are often redundant for specific projects. When building a image, the components under these folders need to be trimmed to retain only the minimal set of dependencies required to support the target application.

Answer to RQ1: δ-SCALPEL primarily targets three directories in the container's file system: `/usr/bin`, `/usr/sbin`, and `/usr/lib`, with the goal of removing non-project dependencies within them. Crucially, δ-SCALPEL maintains robustness whether the container entry point is explicitly specified or not, guaranteeing normal execution of project code in the container launched from the slimmed image.

C. RQ2: Time Consumption of Image Slimming Using δ-SCALPEL

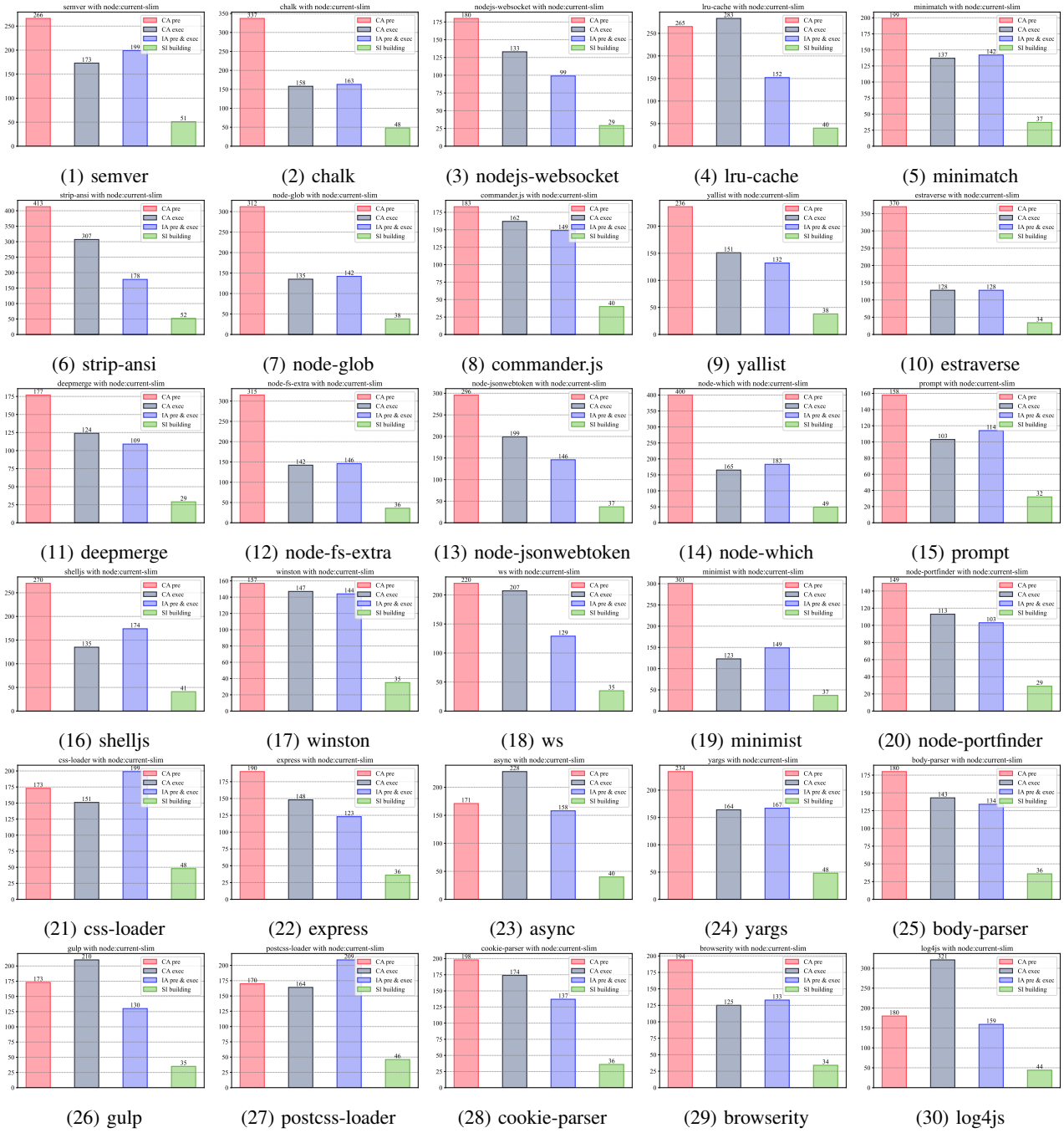
1) Details of Time Consumption: Based on the δ-SCALPEL's framework, we divide it into four parts: code analyzer preparation (CA pre, we refer to the three modules in step 1 collectively as the code analyzer), code analyzer execution (CA exec), image analyzer preparation and execution (IA pre & exec), and slim image building (SI building). In the scenario where the container entry point is specified, we evaluate the efficiency (i.e., time consumption of image slimming) of δ-SCALPEL using the base images `node:current-slim` and `node:current`, respectively.

Evaluation results are shown in Fig. 7 and Fig. 8. Generally, the code analyzer preparation part is time-consuming (127-435s). In this part, δ-SCALPEL configures the image for code analyzer, which includes setting up the base image, configuring the CodeQL environment, and downloading and installing NPM dependencies for the project. Due to network limitations, the majority of the time is spent downloading the base image and NPM dependencies, resulting in an extended preparation time for the code analyzer. In the code analyzer execution part, it takes over 100 seconds to complete the process (with the longest analysis time being 326 seconds for the log4js image based on `node:current`). This extended time is due to the need to analyze the data dependencies of both the project code and the packages it depends on. The long dependency chain of the project results in a significant amount of source code that must be analyzed.

Different base images have minimal impact on the execution efficiency of the code analyzer, as it only analyzes the project code and the packages it depends on, regardless of the base image. In contrast, the size of the base image affects the runtime of the image analyzer (99-361s). When the base image is switched from `node:current-slim` to `node:current`, the root file system content increases, resulting in more data for the image analyzer to process, which extends the runtime. This also increases the size of the `rootfs.tar` file generated by the image analyzer, which in turn affects the building time of the slim image (29-71s).

2) Influence of Image Size on Time Consumption: Additionally, we evaluate how image size impacts δ-SCALPEL's execution time consumption. The results are shown in Figs. 9 and 10. Evaluation results show that for both the `node:current-slim` and `node:current` base images, δ-SCALPEL's processing time scales with the application image size: Processing time increases proportionally with larger application images. Specifically, for application images built on the base image `node:current-slim`, the longest processing time observed for δ-SCALPEL occurrs with the strip-ansi image (950s), while the shortest is with the node-portfinder image (394s). Similarly, when processing images built on `node:current`, δ-SCALPEL's longest processing time occurrs with the node-which image (978s), while the shortest is with the node-portfinder image (498s).

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3640123



(1) semver (2) chalk (3) nodejs-websocket (4) lru-cache (5) minimatch

(6) strip-ansi (7) node-glob (8) commander.js (9) yallist (10) estraverse

(11) deepmerge (12) node-fs-extra (13) node-jsonwebtoken (14) node-which (15) prompt

(16) shelljs (17) winston (18) ws (19) minimist (20) node-portfinder

(21) css-loader (22) express (23) async (24) yargs (25) body-parser

(26) gulp (27) postcss-loader (28) cookie-parser (29) browserity (30) log4js

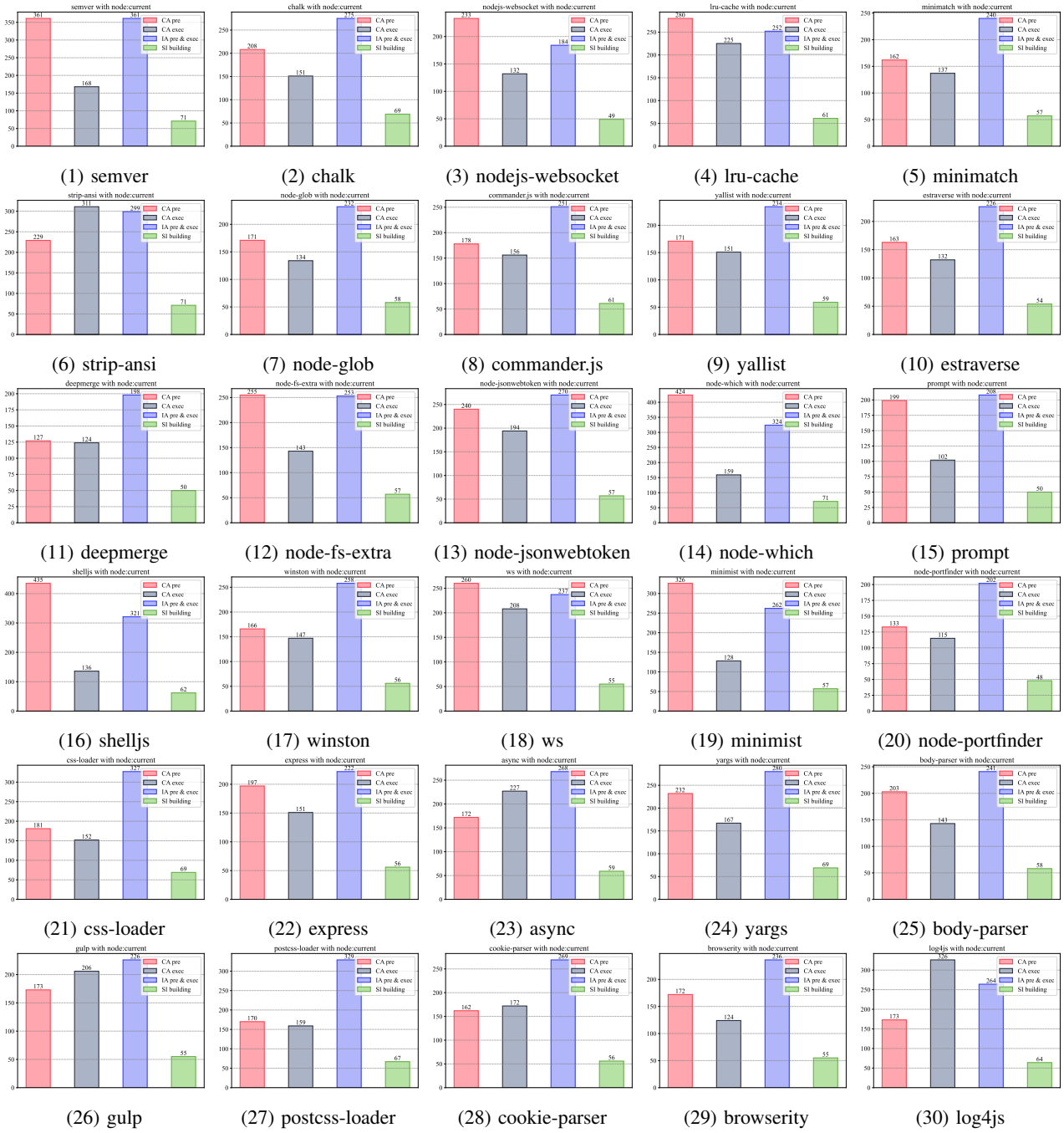
Fig. 7: Time consumption of δ-SCALPEL when specifying the entry point. The base image is `node:current-slim`. CA pre indicates code analyzer preparation, CA exec indicates code analyzer execution, IA pre & exec indicates image analyzer preparation & execution, and SI building indicates slim image building. The Y-axis indicates the runtime, expressed in seconds (s).

Answer to RQ2: The processing time of δ-SCALPEL is approximately proportional to the size of the target image. For images based on `node:current-slim`, sizes range from 226 MB (minimum) to 607 MB (maximum), with corresponding processing times between 394s and 950s. For images based on `node:current`, sizes range from 1.12 GB (minimum) to 1.5 GB (maximum), with processing times between 498s and 978s.

D. RQ3: The Image Attack Surfaces and Vulnerabilities Reduced by δ-SCALPEL

The primary goal of image slimming is to reduce the image size, improving the ease of transmission and storage. An indirect goal is to minimize the attack surface of the image. Identifying and removing unnecessary code fragments and components is a key approach to reducing the attack surface [41]–[44]. According to the Application Container Security Guide (NIST SP 800-190) [45], minimizing container attack

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3640123



(1) semver (2) chalk (3) nodejs-websocket (4) lru-cache (5) minimatch

(6) strip-ansi (7) node-glob (8) commander.js (9) yallist (10) estraverse

(11) deepmerge (12) node-fs-extra (13) node-jsonwebtoken (14) node-which (15) prompt

(16) shelljs (17) winston (18) ws (19) minimist (20) node-portfinder

(21) css-loader (22) express (23) async (24) yargs (25) body-parser

(26) gulp (27) postcss-loader (28) cookie-parser (29) browserity (30) log4js

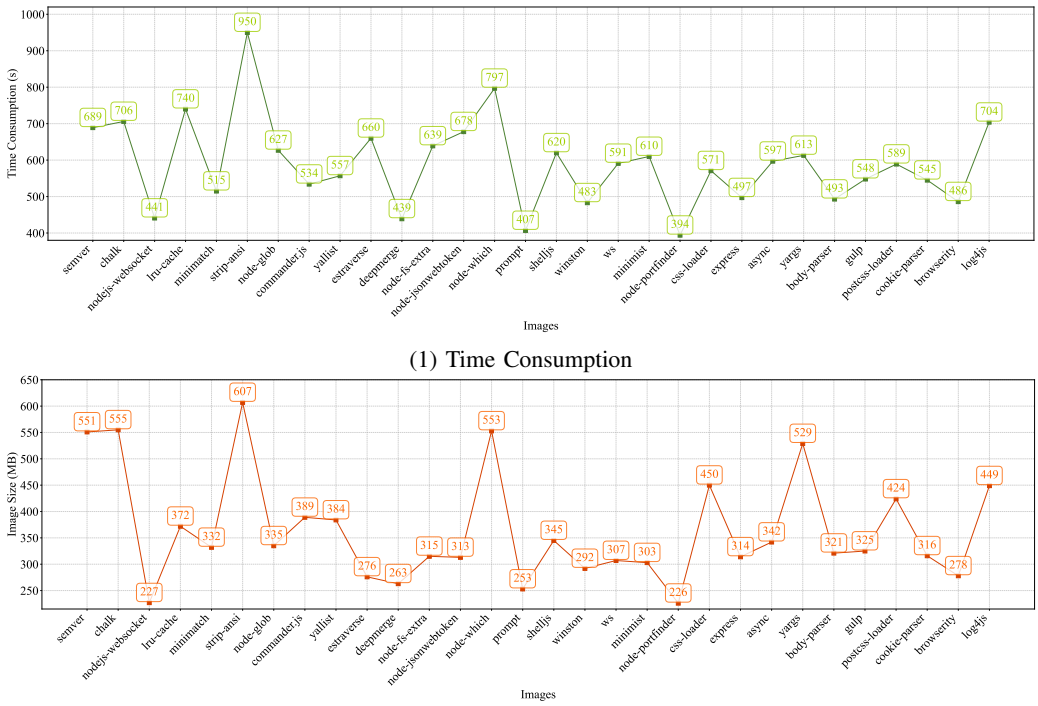
Fig. 8: Time consumption of δ-SCALPEL when specifying the entry point. The base image is `node:current`. CA pre indicates code analyzer preparation, CA exec indicates code analyzer execution, IA pre & exec indicates image analyzer preparation & execution, and SI building indicates slim image building. The Y-axis indicates the runtime, expressed in seconds (s).

surfaces can be achieved by restricting available functionalities. Aligned with this principle, we quantify image attack surface reduction through the count of executable commands before and after slimming. This metric is grounded in the CLI (Command-Line Interface)'s critical role as the primary conduit for accessing container runtime capabilities, including process execution, inspection, and orchestration, which is leveraged by both legitimate users and attackers.

1) Statical Analysis: We use the selected 30 NPM projects to build images based on `node:current-slim` and `node:current`, respectively. We then use δ-SCALPEL to slim these images and calculate the average number of commands supported by them before and after slimming.

The evaluation results are shown in Fig. 11. `node:current-slim` is a condensed version of `node:current`, containing only the minimal packages required to run Node.js [46]. As shown in Fig. 11 (1),

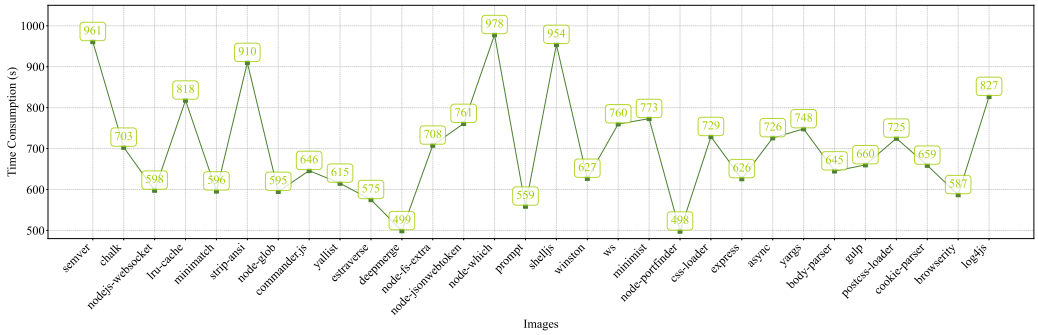
This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3640123



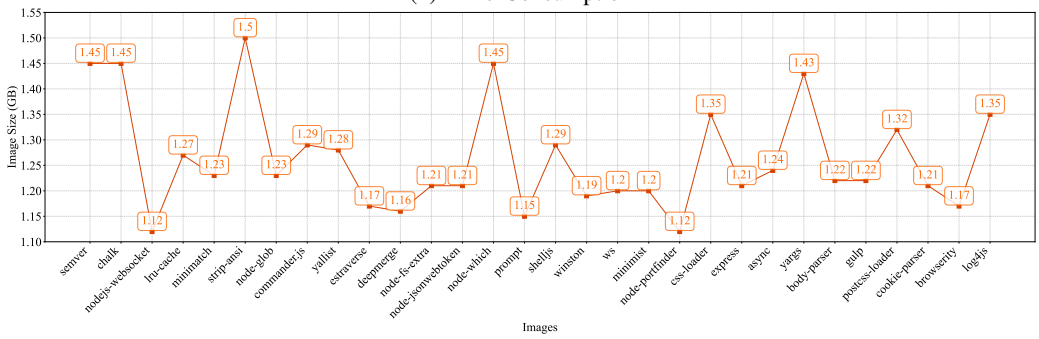
(1) Time Consumption

(2) Image Size

Fig. 9: Influence of the original image size change on the time consumption of image slimming, where the base image used is `node:current-slim`. It should be noted that time consumption refers to the time required to use δ-SCALPEL to generate a slim image, comprising code analyzer preparation, code analyzer execution, image analyzer preparation and execution, and slim image building.



(1) Time Consumption

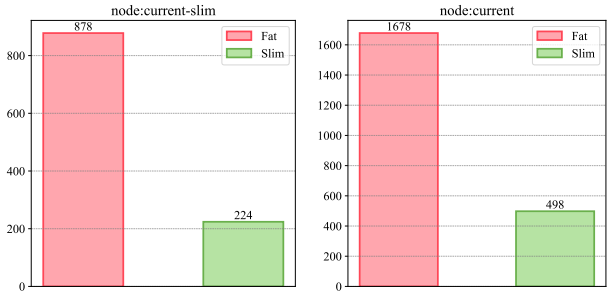


(2) Image Size

Fig. 10: Influence of the original image size change on the time consumption of image slimming, where the base image used is `node:current`. It should be noted that time consumption refers to the time required to use δ-SCALPEL to generate a slim image, comprising code analyzer preparation, code analyzer execution, image analyzer preparation and execution, and slim image building.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3640123

`node:current` supports 1,678 commands, whereas in Fig. 11 (2), `node:current-slim` supports 878 commands, which is 800 fewer than `node:current`. That is to say, without using δ-SCALPEL, the attack surface of the image based on `node:current-slim` is reduced by 47.7% compared to the image based on `node:current`. For images based on `node:current-slim`, the average number of supported commands decreases to 224 after slimming with δ-SCALPEL, leading to a 74.5% reduction in the attack surface compared to the pre-slimming state. In comparison, for images based on `node:current`, the average number of supported commands is 498, resulting in a 70.3% decrease in the attack surface relative to its original state.



(1) node:current-slim (2) node:current

Fig. 11: Comparison of the average number of commands supported by the images based on `node:current-slim` and `node:current`, both before and after the slimming process. The Y-axis indicates the number of commands supported by the image.

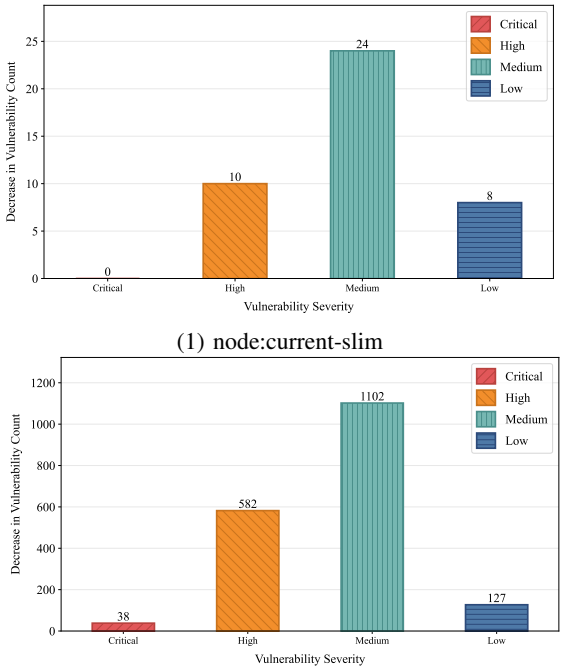
Furthermore, we use the Grype[13] tool to compare the average number of vulnerabilities by severity level (critical, high, medium, and low) in the image before and after slimming, as shown in Fig. 12.

For images based on `node:current-slim`, after processing with δ-SCALPEL, the critical vulnerabilities are reduced by an average of 0, the high vulnerabilities by an average of 10, the medium vulnerabilities by an average of 24, and the low vulnerabilities by an average of 8. Similarly, for images based on `node:current`, after processing with δ-SCALPEL, the critical vulnerabilities are reduced by an average of 38, the high vulnerabilities by an average of 582, the medium vulnerabilities by an average of 1102, and the low vulnerabilities by an average of 127.

The evaluation results further demonstrate that reducing non-essential components (e.g., unused commands and libraries) in images directly lowers the number of vulnerabilities, thereby shrinking the attack surface.

2) Case Study: In this section, we conduct a case study to illustrate the influence of image slimming on typical users and attackers. In the application scenario shown in Fig. 13, we create an express-based API service that receives POST requests at the `/calculate` endpoint, parses the `expression` field (containing mathematical expressions) from the request

[13]https://github.com/anchore/grype



(1) node:current-slim

(2) node:current

Fig. 12: Comparison of average decrease of vulnerability count by severity level (critical, high, medium, and low) in `node:current-slim`-based and `node:current`-based images before and after slimming. It should be noted that the severity level is automatically classified by the Grype tool.

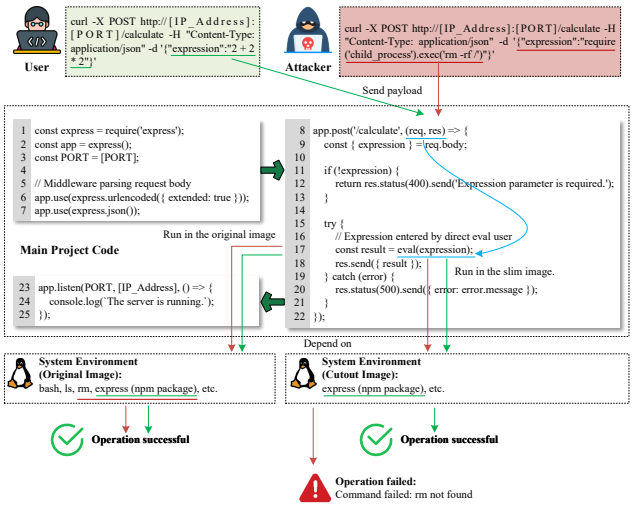


Fig. 13: An example of the influence on typical users and attackers after using δ-SCALPEL to slim the image.

body, directly executes the expression using JavaScript's `eval` function, and returns the calculation result.

For a typical user, a POST request is sent, and the value of the `expression` field is set to `2+2*2`. Upon receiving the request, the container parses and executes the value of the `expression` field using the `eval` function. This code execution process requires the

utilization of the npm express library. For an attacker, changing the value of the request's `expression` field to `require('child_process').exec('rm -rf /')` can facilitate a RCE (Remote Code Execution) attack, leading to the deletion of all files in the container's root directory. The execution of this attack relies on Node.js components (specifically, the `child_process` module and the express library) as well as the Linux system command `rm`.

The original image contains redundant components, enabling successful execution of inputs from both typical users and attackers. The slimmed image, however, includes only minimal components necessary for application runtime. Consequently, whereas typical users' requests are executed, attackers' requests fail due to missing dependencies (i.e., `rm`) required for malicious payload execution.

Answer to RQ3: Image slimming effectively optimizes storage resource consumption for images. Simultaneously, by removing redundant components, it reduces potential sources of vulnerabilities and significantly shrinks the attack surface of the image.

E. Threats to Validity

In this paper, we propose the use of static data dependency analysis to extract the environment dependencies of the project code, aiming to reduce the image size while ensuring the normal operation of the project code. The evaluation results demonstrate that the proposed δ-SCALPEL model is effective and robust. However, δ-SCALPEL still faces the following limitations:

Threats to effectiveness. From the evaluation results shown in Tab. I, Tab. II, and Tab. III, it is evident that δ-SCALPEL can achieve precise and effective image slimming, regardless of whether the container entry point is explicitly specified. In this paper, we focus on removing executable files in the `/bin` and `/sbin` folders and dynamic link library files in the `/lib` folder, as these files are closely related to the execution of the project code. The base runtime environment of the container, including files in the `/var` and `/opt` directories, is fully retained. While this ensures the smooth operation of the container and project code, it inevitably reduces the slimming rate. In the evaluation of RQ1, we find that SummSlim demonstrates advantages in specific scenarios. Thus, in future work, we will investigate integrating SummSlim's idea into δ-SCALPEL to further enhance its slimming capabilities.

Threats to efficiency. Besides effectiveness, efficiency is a crucial evaluation metric to determine whether a model can be applied in the real production environment. As shown in the evaluation results in Fig. 9 (1) and Fig. 10 (1), the processing time of δ-SCALPEL for `node:current-slim`-based images ranges between 394s and 950s, while the processing time for `node:current`-based images ranges between 498s and 978s. Both exceed 5 minutes, with the maximum reaching 16 minutes. The most time-consuming parts being the preparation of the image for the code analyzer and the execution of the code analyzer itself. The running time for the image preparation part of the code analyzer is constrained by the local network environment, as it takes considerable time to download the base image and NPM packages. Despite optimizing the static data dependency analysis algorithm for the code analyzer, we still encounter challenges of efficiency. Therefore, in our future research, we will enhance the efficiency of the code analyzer by incorporating more advanced analysis algorithms and designing more effective package filtering mechanisms.

Moreover, δ-SCALPEL currently only supports slimming built images. When image contents require modification (e.g., adding new dependencies), users must manually adjust the image and re-run δ-SCALPEL. Consequently, its ability to handle slimming of modified images requires future enhancement.

V. Conclusion

Image slimming is a valuable area of research that can help reduce the image size and minimize the attack surface of containers. Existing methods determine a project's environment dependencies by observing the operational behavior of the container. However, these approaches have significant limitations, including the incomplete extraction of the environment dependencies and failure to slim the image, especially when the image's entry point is not specified. In this paper, we propose δ-SCALPEL, a robust model that utilizes static code analysis technology to extract the environment dependencies of a project, thereby addressing the limitations of existing dynamic-based methods. Our evaluation results demonstrate that δ-SCALPEL effectively reduces the image size while ensuring the normal operation of the project.

Data Availability Statement

The replication package of δ-SCALPEL is available at https://zenodo.org/records/13982576.

Acknowledgments

This work was supported by National Key Research and Development Program of China (No.2023YFB3106600).

References

[1] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys european conference on computer systems 2007*, 2007, pp. 275–287.

[2] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2013, pp. 233–240.

[3] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *2015 IEEE International Conference on cloud engineering*. IEEE, 2015, pp. 386–393.

[4] D. Merkel *et al.*, "Docker: lightweight linux containers for consistent development and deployment," *Linux j*, vol. 239, no. 2, p. 2, 2014.

[5] Z. Zou, Y. Xie, K. Huang, G. Xu, D. Feng, and D. Long, "A docker container anomaly monitoring system based on optimized isolation forest," *IEEE Transactions on Cloud Computing*, vol. 10, no. 1, pp. 134–145, 2019.

[6] T. Combe, A. Martin, and R. Di Pietro, "To docker or not to docker: A security perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, 2016.

[7] N. Muhtaroglu, B. Kolcu, and İ. Arı, "Testing performance of application containers in the cloud with hpc loads," in *Proceedings Of The Fifth International Conference On Parallel, Distributed, Grid And Cloud Computing For Engineering*. Civil-Comp, 2017.

[8] P. Liu, S. Ji, L. Fu, K. Lu, X. Zhang, W.-H. Lee, T. Lu, W. Chen, and R. Beyah, "Understanding the security risks of docker hub," in *Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part I 25*. Springer, 2020, pp. 257–276.

[9] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt, "Large-scale analysis of the docker hub dataset," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–10.

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

[11] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps, "A dataset of dockerfiles," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 528–532.

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

[13] Z. Zhang, H. Huang, S. Xu, Q. Zhou, T. Zhang, X. Jia, and W. Zhang, "Summslim: A universal and automated approach for debloating container images," in *2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2024, pp. 132–141.

[14] H.-C. Kuo, "Attack surface reduction in contemporary operating systems via practical kernel debloating," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2022.

[15] M. Souppaya, J. Morello, and K. Scarfone, "Application container security guide," National Institute of Standards and Technology (NIST), Special Publication (NIST SP) 800-190, Sep. 2017. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf

[16] slimtoolkit, "Optimize your experience with containers. make your containers better, smaller, more secure and do less to get there (free and open source!)," 2022, accessed: 2024-09-09. [Online]. Available: https://github.com/slimtoolkit/slim

[17] J. Liu, J. Han, and C. Huang, "Vulnerability detection in source code using statice analysis," *Journal of Cyber Security*, vol. 7, no. 4, pp. 100–113, 2022.

[18] A. Dawoud and S. Bugiel, "Bringing balance to the force: Dynamic analysis of the android application framework," *Bringing balance to the force: dynamic analysis of the android application framework*, 2021.

[19] M. Ahmad, V. Costamagna, B. Crispo, F. Bergadano, and Y. Zhauniarovich, "Stadart: addressing the problem of dynamic code updates in the security analysis of android applications," *Journal of Systems and Software*, vol. 159, p. 110386, 2020.

[20] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 443–458.

[21] J. Han, C. Huang, and J. Liu, "bjcnet: A contrastive learning-based framework for software defect prediction," *Computers & Security*, vol. 145, p. 104024, 2024.

[22] ——, "bjenet: a fast and accurate software bug localization method in natural language semantic space," *Software Quality Journal*, pp. 1–24, 2024.

[23] J. Han, C. Huang, S. Sun, Z. Liu, and J. Liu, "bjxnet: an improved bug localization model based on code property graph and attention mechanism," *Automated Software Engineering*, vol. 30, no. 1, p. 12, 2023.

[24] L. Chen, Y. Chen, S. Xiao, Y. Song, L. Sun, Y. Zhen, T. Zhou, and Y. Chang, "Egfe: End-to-end grouping of fragmented elements in ui designs with multimodal learning," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[25] Y. Ding, S. Chakraborty, L. Buratti, S. Pujar, A. Morari, G. Kaiser, and B. Ray, "Concord: clone-aware contrastive learning for source code," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 26–38.

[26] C. Huang, N. Wang, Z. Wang, S. Sun, L. Li, J. Chen, Q. Zhao, J. Han, Z. Yang, and L. Shi, "Donapi: Malicious npm packages detector using behavior sequence knowledge mapping," *arXiv preprint arXiv:2403.08334*, 2024.

[27] B. Steenhoek, H. Gao, and W. Le, "Dataflow analysis-inspired deep learning for efficient vulnerability detection," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[28] C. Wang, R. Ko, Y. Zhang, Y. Yang, and Z. Lin, "Taintmini: Detecting flow of sensitive data in mini-programs with static taint analysis," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 932–944.

[29] R. Opdebeeck, J. Lesy, A. Zerouali, and C. De Roover, "The docker hub image inheritance network: Construction and empirical insights," in *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2023, pp. 198–208.

[30] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, "Docker ecosystem–vulnerability analysis," *Computer Communications*, vol. 122, pp. 30–43, 2018.

[31] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE access*, vol. 7, pp. 52 976–52 996, 2019.

[32] Docker, Inc., "Set up automated builds," https://docs.docker.com/docker-hub/builds/, 2024, accessed: 2024-09-30.

[33] B. Tak, H. Kim, S. Suneja, C. Isci, and P. Kudva, "Security analysis of container images using cloud analytics framework," in *Web Services–ICWS 2018: 25th International Conference, Held as Part of the Services Conference Federation, SCF 2018, Seattle, WA, USA, June 25-30, 2018, Proceedings 16*. Springer, 2018, pp. 116–133.

[34] M. Souppaya, J. Morello, and K. Scarfone, "Application container security guide," National Institute of Standards and Technology, Tech. Rep., 2017.

[35] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A measurement study on linux container security: Attacks and countermeasures," in *Proceedings of the 34th annual computer security applications conference*, 2018, pp. 418–429.

[36] O. Javed and S. Toor, "Understanding the quality of container security vulnerability detection tools," *arXiv preprint arXiv:2101.03844*, 2021.

[37] B. Security, "Over 30% of official images in docker hub contain high-priority security vulnerabilities," https://www.banyansecurity.io/blog/over-30-of-official-images-in-docker-hub-contain-high-priority-security-vulnerabilities/, 2024, accessed: 2024-09-30.

[38] Threatpost, "Malicious docker containers earn crypto-miners $90,000," https://threatpost.com/malicious-docker-containers-earn-crypto-miners-90000/132816/, 2018, accessed: 2024-09-30.

[39] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Cimplifier: automatically debloating containers," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 476–486.

[40] Docker, Inc., "Dockerfile overview," https://docs.docker.com/build/concepts/dockerfile/, 2024, accessed: 2024-09-30.

[41] S. Mishra and M. Polychronakis, "Shredder: Breaking exploits through api specialization," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 1–16.

[42] C. Mulliner and M. Neugschwandtner, "Breaking payloads with runtime code stripping and image freezing," *Black Hat USA*, 2015.

[43] A. Quach, A. Prakash, and L. Yan, "Debloating software through {Piece-Wise} compilation and loading," in *27th USENIX security symposium (USENIX Security 18)*, 2018, pp. 869–886.

[44] H. Koo, S. Ghavamnia, and M. Polychronakis, "Configuration-driven software debloating," in *Proceedings of the 12th European Workshop on Systems Security*, 2019, pp. 1–6.

[45] National Institute of Standards and Technology, "Application container security guide," https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf, National Institute of Standards and Technology, Tech. Rep. NIST SP 800-190, 2017, accessed: 2024-09-30.

[46] Node.js, "Node.js docker official images," https://github.com/nodejs/docker-node/tree/58c3b39e5948f82c594395857193cd97d01c690e, 2023, accessed: 2024-09-30.