Layered Security Analysis for Container Images: Expanding Lightweight Pre-Deployment Scanning

Shafayat Hossain Majumder*, Sourov Jajodia*, Suryadipta Majumdar†, and Md. Shohrab Hossain*
*Department of CSE, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh
Emails: {monsieurmajumder, sourov.jajodia72}@gmail.com, mshohrabhossain@cse.buet.ac.bd
†Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Canada
Email: suryadipta.majumdar@concordia.ca

***Abstract*—Containerization using tools such as Docker has transformed the way applications are deployed and managed in various organizations. However, the use of containers also presents new security challenges due to the potential vulnerabilities that may be present in the container images. To address this, various vulnerability detection tools that use static analysis have been developed that focus more on OS packages, and fail to detect known package vulnerabilities. In this paper, we propose a pre-deployment methodology that detects vulnerabilities in container images targeting both the OS and application packages using a layered approach, where static tools generally fail to detect vulnerabilities in those images. Our solution offers a high degree of customizability and control over its performance in detecting vulnerabilities in images. Users can choose a specific scan profile that is tailored to their particular needs, enabling the detection of vulnerabilities that are either more common, more unique, or a balance of the two. This adaptability makes our solution more flexible and better suited to meet the specific needs of our users. We evaluate our proposed framework against 111 both official and community images collected from Docker Hub to demonstrate its effectiveness compared to other popular static analysis tools for containers.**

***Index Terms*—Container Security, Static Analysis, Layered Analysis, Pre-deployment Methodology, Docker**

I. Introduction

Containers have gained significant popularity as a viable alternative to traditional virtualization methods since the advent of Docker [1] in 2013. They provide numerous benefits such as the ability to bundle an application with its dependencies into a single unit, and efficient utilization of system resources by sharing the host operating system kernel. This has led to over 70% of organizations deploying containers in production, often utilizing shared container orchestration platforms, such as Kubernetes [2], Docker Swarm [3], Mesos [4], and Nomad [5]. This widespread adoption has transformed the industry, enabling organizations to easily deploy and manage their applications with increased efficiency.

However, the adoption of containers also presents new security challenges. Container images may contain from outdated and/or vulnerable packages to misconfigured settings, and since containers are run with root privileges, they may be susceptible to attacks [6] [7]. Studies have shown that on average, each Docker image contains at least one vulnerability, with some images having up to 50 vulnerabilities. Wist et al. reported in 2021 that 46% of the official images, and 68% of the community images had at least one high rated vulnerability [8]. Such vulnerable containers hosted on a shared cloud platform have the potential to affect other containers or hosts. For instance, in July 2017, it was reported that a hacker uploaded several malicious Docker images on Docker Hub [9]. These images were downloaded more than five million times before they were removed, resulting in the mining of 545 Monero digital coins worth approximately $900,000 [10]. Similar cases have been reported in the past [11], highlighting the importance of implementing a pre-upload vulnerability scanning methodology that can detect such vulnerable containers early on and speedy enough for customer satisfaction.

To detect and address the security risks associated with container images fast, various vulnerability detection tools that use static analysis are could be used. Static analysis is a method used to detect vulnerabilities by analyzing the image without running it, therefore identifying potential security risks really fast. Popular tools for static analysis of container images include Grype [12], Snyk [13], Trivy [14], Clair [15] etc. These tools list the packages installed in the container and match them against a vulnerability database, allowing for the identification of vulnerabilities in the packages and enabling administrators to take the appropriate action to mitigate the risks. However, while static analysis can be an effective method for identifying vulnerabilities in container images, it may fail to detect known vulnerabilities. This is primarily due to two reasons: (i) the results of static analysis scans can be influenced by the quality of the analysis tool and the accuracy of its vulnerability database; and (ii) these tools mainly focus on base OS packages, ignoring many application packages.

Most existing static analyzers for container security have limited accuracy since they mainly focus on identifying vulnerabilities in OS packages, but not application packages. As a result, even the best performing static analysis tool missed ≈34% of the vulnerabilities in their evaluation [16]. While dynamic analysis is a more effective method for identifying vulnerabilities in container images, it is slower than static analysis [17]. Therefore, it is not ideal for scenarios where a faster scan is required, such as prior to deploying images to shared cloud platforms like Docker Hub or Kubernetes. In such scenarios, long waiting for an image to be scanned for vulnerabilities is not practical. To effectively tackle this issue, we propose a novel solution that utilizes the power of

static analysis, targets both the OS and application packages, and provides superior detection and coverage compared to conventional static analysis tools for containers.

The primary contributions of this paper are as follows:

- Our tool is the first to perform lightweight pre-deployment scanning of container images in a layered fashion, while increasing vulnerability landscape coverage without the performance overhead associated with alternative methods like dynamic analysis.
- Our approach achieves higher vulnerability detection rates by combining different state-of-the-art tools specialized in detecting both base OS package and application package vulnerabilities, without actually requiring improvements to the vulnerability scanning techniques themselves, surpassing stand-alone static analyzers focused solely on base OS packages.
- Our tool empowers users with customizability and control of the tool's scan behavior and performance by incorporating tunable performance parameters and scan profiles e.g. light, adaptive and thorough scans, enabling them to meet safety requirements tailored to their specific scenarios.
- We implement our tool for Docker, the leading containerization platform, and evaluate it with 111 images (both official and community) from Docker Hub, showcasing its superior effectiveness compared to popular static analysis tools with an average 112% increase in CVE detection, surpassing tools like Grype, Snyk, Trivy.

The rest of the paper is structured as follows, Section II presents related works, followed by required preliminaries to understand our work in Section III. Section IV explains our methodology used in this paper, Section V provides the technical approach to our architecture. Section VI finally provides the details of our experimental data, and Section VIII concludes the paper.

II. Related Works

In this section, we explore prior research in the realm of cloud container security and vulnerability detection, with a particular focus on Docker containers and associated security practices. We delve into existing studies that address Docker container security, examine methodologies for identifying common vulnerabilities and exposures (CVEs) within container images, and explore the landscape of cutting-edge vulnerability detection tools.

In the DIVA (Docker Image Vulnerability Analysis) study [18], a comprehensive analysis of over 356,000 community and official images revealed significant vulnerability disparities. On average, community images exhibited 180 vulnerabilities, while official images displayed 75 vulnerabilities. The study emphasized the persistence of outdated images, which prolonged vulnerability propagation. Vulnerability detection was performed using Clair. Similarly, another investigation [8] found that certified images were unexpectedly more vulnerable than official images, with community images averaging 150 vulnerabilities and official images averaging 70 vulnerabilities. The study identified the most vulnerable CVEs and generated a vulnerability frequency based on CVSS scores. Interestingly, no correlation was observed between image features and vulnerability occurrences. In a separate study [16], three static analysis tools, namely Clair, Anchore, and Microscanner, were evaluated by analyzing 59 public images specifically targeting Java applications. Although these studies provided valuable vulnerability statistics through static analysis of container images, their detection coverage and analysis were limited due to their focus on base-level or OS-level analysis. In contrast, our proposed system offers broader vulnerability coverage by examining both the base and package layers of an image, encompassing a more comprehensive assessment of vulnerabilities.

In studies conducted by researchers [19] and [20], the effectiveness of static and dynamic analysis in detecting vulnerabilities in Docker images was investigated. The use of Clair for static analysis was highlighted in [19], while [20] compared different container implementations with Docker and explored multiple security tools for both dynamic and static analysis. While dynamic analysis tools were found to outperform static analysis in these studies, our paper demonstrates that the detection rate of static analysis can be further improved. Furthermore, in resource-constrained scenarios such as pre-deployment scanning for shared cloud platforms, our solution's adoption of static analysis proves to be the more favorable choice due to its lower resource consumption.

In their work [21], the authors introduce DAVS, a novel tool for Docker image analysis that utilizes Dockerfile information to perform targeted scanning of specific image layers using CVEBinTool. A comparison between Clair and CVEBinTool is presented, revealing a low overlap between the two tools. However, the paper lacks a proposed methodology for effectively combining these tools to enhance overall vulnerability detection in an image. In contrast, our proposed solution not only addresses this gap by incorporating both base and package layer analysis, but also offers different analysis modes, including identification of unique vulnerabilities, maximum vulnerability detection, and balanced performance across these metrics. This comprehensive approach strengthens the overall vulnerability detection capabilities of our solution.

In their work [22], the authors explore Docker's internal and external security, highlighting security systems like SELinux and AppArmor. They outline different aspects of internal security in Docker, including process isolation, file system isolation, IPC isolation, device isolation, and network isolation. Papers such as Docker-sec [23] and Lic-Sec [24] propose container protection techniques. Cavas [25] tracks containers using security containers and employs OWASP Zap for dynamic analysis. The use of security tools and static image analysis is discussed in [17]. Alyas et al. [26] propose container profiling for detecting suspicious activities. Securing Docker containers is explored in [27]. In [28], the authors present RSDS, a system that combines dynamic and static analysis to extract necessary system calls for a specific container, reducing the risk of triggering security

vulnerabilities in the host kernel. While these studies touch on container vulnerability protection measures, they offer limited insights into vulnerability detection specifically within container images, whereas our proposed solution focuses on comprehensive vulnerability detection in container images.

III. Background

In this section, we provide backgrounds on the layered structure of Docker images, the process of static analysis and how various static analysis tools work.

A. Docker Image Layers and Contents

Docker images are mainly composed of two layers: base OS layer and non OS layer. Base OS layer are the first few layers of a Docker image and typically contains the operating system components. On the other hand, the non OS layers, also known as application package layers, contain the user's application code and other dependencies required by the application. These layers are writable, and users can add, remove or modify their contents. Each layer has a specific structure of contents, with the base OS layer consisting of files such as system libraries and kernel modules. In contrast, the non OS layers contain application-specific files such as binaries, libraries, and configurations. These files make up the application stack that is built on top of the base OS layer. Overall, understanding the layout of a Docker image and its layer contents is crucial for creating and managing containers effectively. By knowing what types of files each layer contains and how they are structured, users can optimize the build process and ensure their containerized applications are secure and efficient.

B. Container Image Scanning with Static Analysis Tools

Container image scanning using static analysis is a process of identifying vulnerabilities within container images. The scanning process involves analyzing the names and versions of the packages in the container image against a vulnerability database that contains publicly known security vulnerabilities, also known as Common Exposures and Vulnerabilities (CVEs). Some popular open-source container scanning tools include Grype [12], Snyk [13], Trivy [14] etc. Grype is an open-source tool that can analyze container images and report vulnerabilities based on OS and application package metadata and contents. Snyk is a cloud-based tool that provides comprehensive security testing for container images and can detect vulnerabilities in both OS packages and application packages. Trivy is another open-source tool that can scan container images for vulnerabilities in OS packages, application dependencies, and libraries.

Although these tools excel at identifying vulnerabilities in OS packages, they are less effective at detecting vulnerabilities in application packages. OS packages typically have a standardized structure, with well-documented and easily identifiable vulnerabilities. In contrast, application packages exhibit greater complexity and diversity, posing challenges for static analysis tools in accurately pinpointing application package vulnerabilities. Nevertheless, leveraging container scanning tools remains a crucial aspect of container security. Their efficacy can be enhanced by integrating them with complementary tools, as demonstrated in Section VI, to achieve broader coverage and improved vulnerability detection.

IV. Methodology

In this section, we present the methodology of our tool that utilizes static analysis for pre-deployment scanning of container images to enhance vulnerability detection coverage with the usage of existing tools.

Fig. 1 shows an overview of our proposed approach that covers two major steps:

- **Base analysis**, which primarily detects and identifies known vulnerabilities in the image's operating system base images (such as Ubuntu, Alpine, and Debian), which serve as the canvas for the image to build upon.
- **Package analysis**, which attempts to detect known vulnerabilities introduced in the image by application layers containing user packages.

By combining these two components, we achieve comprehensive coverage of both base OS and application-level packages, surpassing the capabilities of standard commercial static analysis tools. Our approach integrates these components based on scan profiles and threshold scores, merging the detected vulnerabilities to generate a comprehensive report. Furthermore, our system calculates a vulnerability score (v-score) to assess the image's vulnerability status and determine its eligibility for uploading to the shared cloud platform. The subsequent subsections provide a detailed description of these components and the functioning of our architecture.

A. Approach Overview

The main architecture is the driver of the components which binds all the modules and holds the program logic. The main architecture is tunable, that is, it has various parameters that can be modified by the user according to his/her scanning needs. Also, our architecture has various types of scans, which will also be discussed here.

*1) **User Choices**:* Our tool is dynamic in a sense that it gives the users couple of choices to configure and tune the tool's performance. The users can configure settings, such as the *scan type, scan profile, and t-score*, as explained here and shown in Fig. 2.

1) Scan Types: Users have the choice to pick from three scan types, as shown in Fig. 2, namely: light, thorough and adaptive, to allow the users choose how deep the tool would scan. This is discussed in more details below.
2) Scan Profiles: The scan profiles in our system detect vulnerabilities using different tools and vary in terms of the number and types of vulnerabilities they catch. From Fig. 2 it is seen that we have defined three scan profiles: maximum, balanced, and unique vulnerabilities. The unique vulnerability option utilizes Grype to find the most distinct vulnerabilities, while the maximum

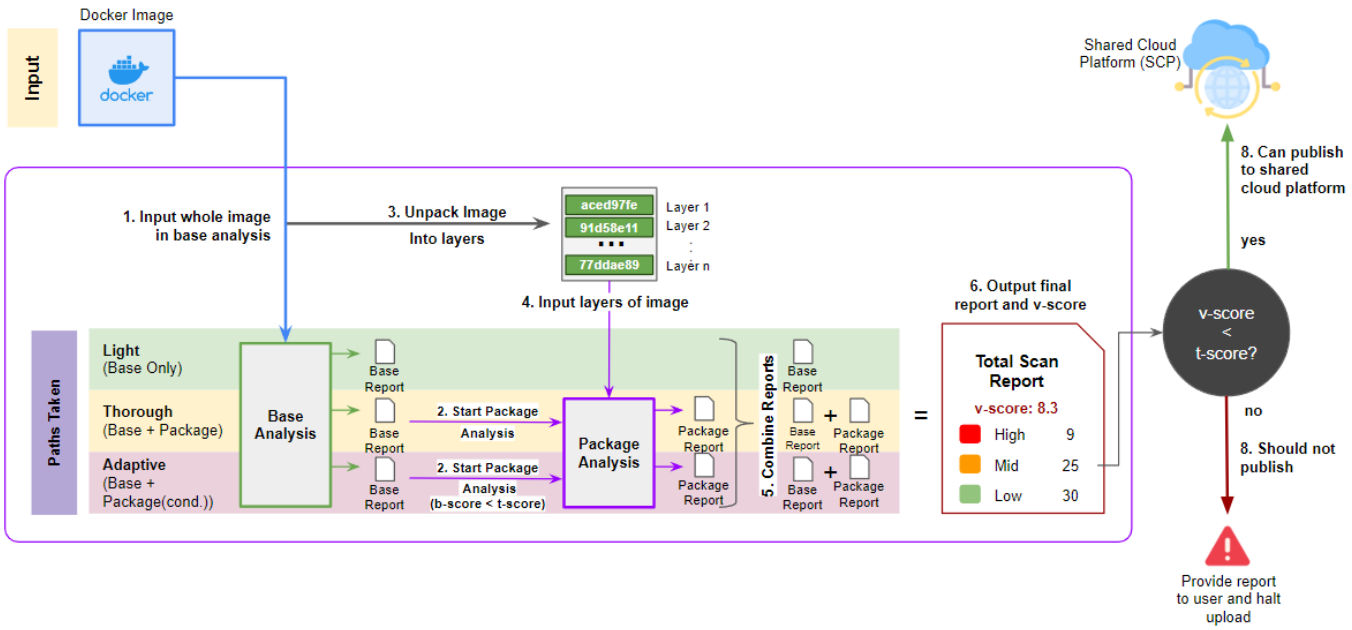


Fig. 1: Proposed approach.

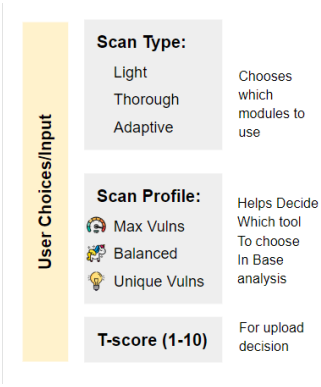


Fig. 2: Categories of user choices.

vulnerability option employs Snyk to detect the highest number of vulnerabilities.

3) T-score: The threshold score (t-score) enables the user to define the acceptable level of vulnerability for the image. Following the complete scan, a vulnerability score (v-score) is calculated. If the v-score is below the t-score, the image is eligible for uploading to the cloud platform. The t-score determines the user's tolerance for vulnerability based on the severity of the image.

While the tool comes with default values for these parameters, offering users multiple choices empowers them with greater control. Considering that the acceptable vulnerability landscape varies across different use cases, this flexibility proves invaluable in assisting users to align the tool with their specific requirements.

*2) **Scan Types**:* There are three scan types that can be chosen, that is: light, thorough and moderate. These are discussed below:

1) Light Scan: Light scan is the fastest scan among the three, where only the base analysis is performed. Therefore, this scan delivers the performance and detectability equivalent to that achieved by running static analysis tools exclusively.
2) Thorough Scan: By sequentially triggering both the base analysis and package analysis modules, this scan surpasses the detection capabilities of the Light Scan. With comprehensive coverage of both base and application package layers, it achieves the highest vulnerability detection rate, but at the cost of highest processing time.
3) Adaptive Scan: The Adaptive scan is ideal for resource-constrained scenarios where a thorough scan is desired without excessive resource consumption, such as in shared cloud platforms where much resources cannot be allocated for scanning the vulnerability an image. It starts with the base analysis module and only invokes the package analysis if the detection rate in the base analysis falls below the acceptable vulnerability threshold (b-score $<$ t-score). If the base analysis already exceeds the threshold (b-score $\geq$ t-score), the scan concludes with a vulnerability report, skipping the package analysis. Fig. 3 shows the workflow for this scan.

B. Base Analysis

According to the Section III-A, container images are typically comprised of multiple layers, each corresponding to a line in the configuration file. Among these layers, the base OS layer contains commands to include the operating system on

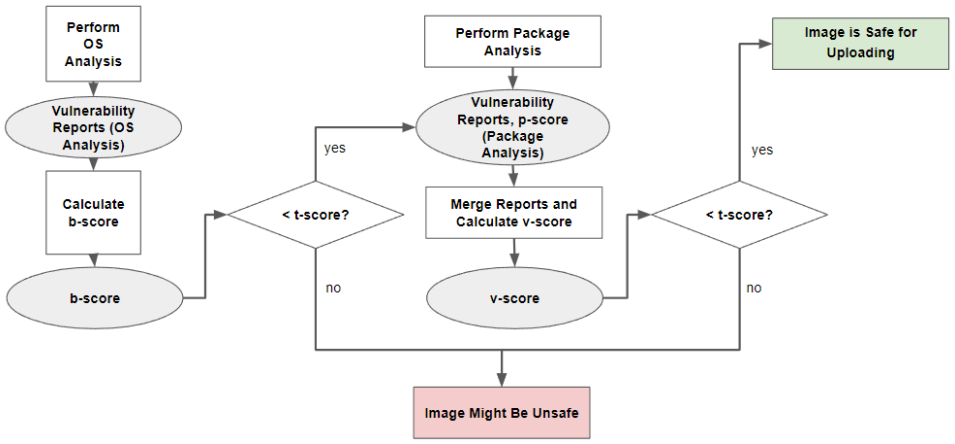


Fig. 3: Adaptive scanning workflow.

which the container will run. These layers, which we refer to as base layers, contain a variety of configuration files, binary files, and OS packages that may be vulnerable [29]. Additionally, OS packages may be installed later through image creation commands, which also have the potential to be vulnerable [21]. The goal of base analysis is to detect vulnerabilities introduced by these cases.

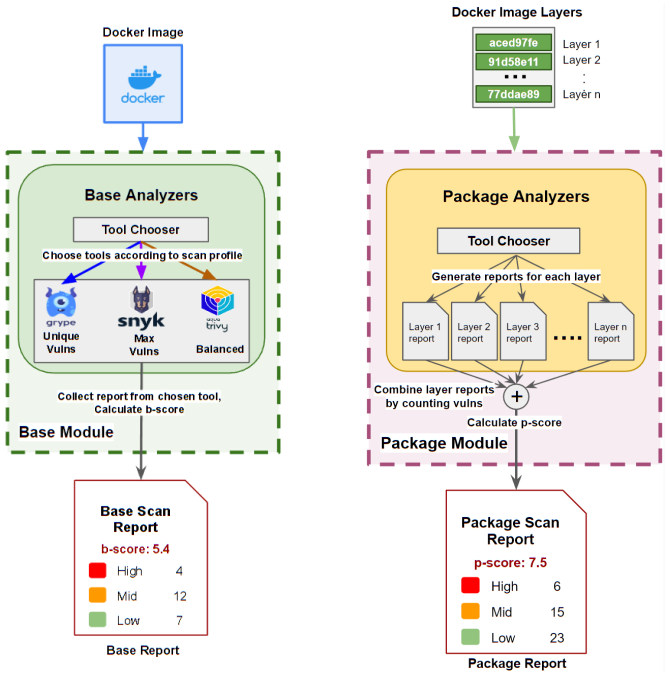
We utilize established commercial tools, namely Grype, Trivy, and Snyk, for base analysis. These tools extract OS information, limited package details, and match them against the CVE database, reporting vulnerabilities along with relevant data such as CVE numbers and CVSS severity scores.

Fig. 4a illustrates the steps involved in base analysis. A Docker image is provided as input and passed through static analysis tools (Grype, Snyk, and Trivy) based on the chosen scan profile. Each tool generates a base report, including CVE entries, detection layers, and frequencies. The base score (b-score) is calculated as the mean of the CVSS scores of the reported vulnerabilities.

C. Package Analysis

Package analysis is essential for detecting vulnerabilities in application packages within container images, typically located in non OS layers above the base OS. These layers consist of application packages, binaries, codes, and database files, all prone to vulnerabilities. Unlike commercial static analysis tools that are inadequate in detecting non OS application package vulnerabilities and other introduced files, package analysis plays a vital role in addressing these limitations.

Nonetheless, package analysis presents challenges such as images having a higher number of application packages compared to base OSs, with users having the ability to introduce vulnerabilities through modifications. Indexing approaches used by commercial tools are less effective for application package vulnerabilities. Instead, a more effective approach involves hash-matching binary and program files with vulnerable signatures, albeit with slightly longer processing times. Intel's CVE Binary Tool (binTool) adopts this strategy, making it the tool of choice for package analysis.



(a) Base analysis overview. (b) Package analysis overview.

Fig. 4: Overview of the two most important approaches: base and package analysis.

The steps for package analysis are depicted in Fig. 4b. Initially, the packages used in constructing the image are extracted from different layers of the Docker images, each corresponding to a Dockerfile command. These layers are then analyzed to identify potentially vulnerable files (PVFs) within the image. To investigate these PVFs for vulnerabilities, the powerful BinTool is employed. By comparing binary files to a database of Common Vulnerabilities and Exposures (CVEs), BinTool ensures comprehensive detection of application vulnerabilities throughout the image, layer by layer. Finally, the layer reports are combined to prepare the package report,

and a package score (p-score) is calculated from the CVEs using their CVSS scores, similarly to the base layer b-score calculation.

Package vulnerability detection tools don't perform well in detecting base OS vulnerabilities in regards of base analysis tools, because these tools: 1) follow different methods to detect vulnerabilities, 2) while base analysis tools are specialized for base OS vulnerability detection, the package analysis tools are not. Therefore, both of these tools are necessary for a better coverage of the vulnerability landscape of container images.

V. Framework Design

This section provides the design of our framework with its implementation details.

A. The Main Architecture

The main architecture combines all the modules discussed above, as well as a workflow to bind them and use their outputs to finally find a pass or fail result. We discuss the inputs, workflow and output of the main architecture in detail.

1) Input: The main architecture takes a target image as input, specifically Docker images from Docker Hub or local Docker images. The image is directly supplied to the base analysis component, while for the package analysis module, some pre-processing is required. In this step, we utilize binTool to scan binaries and application packages for vulnerabilities. However, binTool cannot directly scan container images, so we locally save the Docker image and extract its layers, which then serve as the input for the package analysis module.

2) Vulnerability Scanning: Our vulnerability detection process begins after the input is prepared. With three scanning modes available, the base analysis module is triggered initially. Upon completion, the base report and associated b-score are obtained. Light scanning compares the b-score to the t-score to determine the upload decision, while thorough scanning activates the package analysis module to generate a package vulnerability report and p-score. The merging of these reports and scores (b-score and p-score) yields the final v-score, which is compared to the t-score for the upload decision. Adaptive scanning adjusts between light and thorough scanning based on the image's vulnerability level and user preferences, skipping the package scan if the b-score exceeds the t-score, and following a similar workflow otherwise.

3) Combination of Reports: Upon completion of the scanning process, depending on the chosen scan type, we may obtain both the base and package scan reports, each accompanied by a corresponding score: the base score (b-score) and package score (p-score), respectively. These scores are computed by aggregating the CVSS scores of the detected vulnerabilities using methods such as average, harmonic mean, or geometric mean. In our implementation, we employ the average method as it is easily interpretable and provides a comprehensive measure of the vulnerability landscape. Once we have both reports, the next step involves merging them to generate the final vulnerability score (v-score) for the Docker image. Considering that the reports may contain duplicate entries for the same vulnerabilities, we follow the subsequent steps:

1) For vulnerabilities with matching CVE IDs and layer numbers in both reports, we consider them as the same vulnerability. We determine the frequency of these CVEs by selecting the highest count from their appearances in the reports.
2) Vulnerabilities with differing CVE IDs or layer numbers, or both, are treated as distinct vulnerabilities and are directly included in the final report without any modifications.

These steps basically does a union operation of the two reports. This can be easily done in O(n) using techniques like sliding window, given the vulnerabilities are sorted by CVE scores. After which, he v-score is calculated from the merged report, and its comparison to the user-defined t-score determines if the container image is safe for uploading or not. In both cases, the user can view the final vulnerability report to take future steps if he requires.

B. Base Analysis

The Base Analysis component is the initial stage in our architecture, focused on scanning the base OS of the Docker image. This includes examining OS packages and dependencies, some of which may also be introduced in subsequent layers of the container. To generate vulnerability reports, we utilize three well-known and effective open-source container image static analysis tools: Grype, Snyk, and Trivy. These tools are specifically chosen for their expertise in identifying vulnerabilities related to Docker base images. Table I provides additional information about these tools.

Scanning Tool	**Grype**	**Snyk**	**Trivy**
Version	0.52.0	1.1088.0	0.34.0
Created By	Anchore	Snyk IO	Aqua Security
OS Packages Supported	9	9	16
Language Packages Supported	8	-	10
Extra Support	-	Unmanaged softwares (2)	-

TABLE I: Container image static analysis tools.

Another important part is how these tools behave differently to the same base images. Upon our manual checking of the vulnerabilities caught in 111 images using these tools, we have observed the following trend:

1) *Grype* finds unique vulnerabilities, that are not normally detected by the other tools.
2) *Snyk* detects the most amount of vulnerabilities.
3) *Trivy* provides a more balanced vulnerability detection.

The different behaviors of these tools, stemming from differences in their vulnerability databases and implementation, have inspired the creation of various scan profiles in our tool. These customizable profiles provide greater control over the tool's behavior, allowing it to excel in different

situations. While we currently utilize three specific tools in our base analysis, our architecture is not restricted to them, demonstrating the potential for interchangeability with other tools to achieve a range of results.

Grype and Trivy are Linux packages installed for scanning container images, while Snyk serves as the official Docker container vulnerability scanner, accessed through dockerd's (Docker daemon) APIs. These tools generate a package and OS information list from the image, which is then queried against their respective vulnerability databases. They report vulnerabilities related to the specific OS and package versions. However, these tools may not detect back-ported packages or fixed vulnerabilities, limiting their coverage. Nonetheless, their speed and efficiency make them suitable for the initial layer of our architecture.

C. Package Analysis

To enhance our understanding of the image's vulnerability landscape, we conduct package analysis using the CVE Binary Tool. This analysis is triggered under specific conditions and allows us to delve deeper into the image's layers. The tool performs static analysis by examining binary file strings for matches with known vulnerabilities in popular open source libraries. Although BinTool currently supports 254 packages, it cannot be directly applied to images. To overcome this limitation, we follow a specific workflow to make it compatible with container analysis.

1) The image is first saved locally. This can be done by `docker save` *`imageName`* `>` *`imageName`*`.tar` command, built into the Docker command line. This saves the provided image to a TAR archive file.
2) Upon extraction, the archive file reveals the structure below.
   a) *repositories* file that contains the image information and SHA256 hash of the content of the image.
   b) *manifest.json* file containing configuration file names and the SHA256 hashes of the layers.
   c) *hash named config.json* file, provides additional image details such as architecture, environment variables, creation time, layer history, and the commands used to create each layer.
   d) *hash named layer folders*, as many as there are commands needed to create the image. Each layer folder again contains 3 files inside it:
      i) *JSON* file containing layer related information.
      ii) *VERSION* file having layer version.
      iii) *layer.tar* file which contains the actual contents of that layer, including binary files, packages, info and configuration files etc.
   e) We extract the `layer.tar` file to access its contents and relabel each layer for clarity. This results in n layers, n representing total layers in the image.
3) After extraction, we run binTool on each layer, receiving individual layer reports.
4) Vulnerability reports of each layer are merged as a final report for the package analysis layer.

The package analysis level enhances our understanding of package vulnerabilities in the image, offering a more comprehensive view. Fig. 5 presents the workflow for this layer, showcasing its effectiveness in detection.

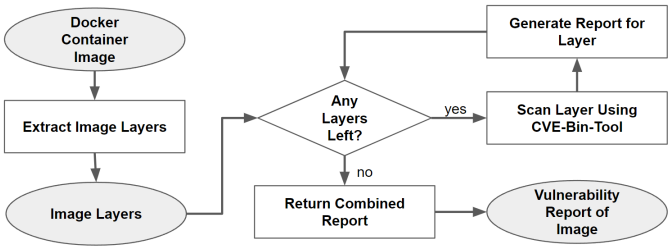


Fig. 5: Workflow for package layer scanning.

VI. Experiment

We evaluate our architecture through practical experiments on both official and community Docker images collected from Docker Hub. In the following sections, we present the experimental results.

A. Image Discovery

We analyze vulnerabilities in various Docker images, which are constructed using Dockerfiles containing diverse Docker commands. To evaluate the effectiveness of our architecture, we focus on two image categories: official and community images sourced from Docker Hub.

Docker Hub acts as an open-source repository hosting official images maintained by Docker and its partners, as well as community images from the Docker community. Official images undergo rigorous testing, earning them widespread trust and adoption for popular applications. To assess our architecture, we focused on the top 50 latest-tagged official images, representing most recently patched versions. From this selection, 33 images were chosen at random for analysis, examples include Alpine, Nginx, Busybox, Ubuntu, Python, Redis, among others. These images serve as parent images or as the basis for creating other images. As vulnerabilities can propagate from parent images to child images [18], we select these parent images as prime candidates for our investigation. Pulling these images from Docker Hub, we conducted our evaluation in parallel with renowned tools, promptly deleting each image after analysis to optimize storage.

Collecting community images, which comprise almost 99% of Docker Hub repositories, presented a challenge without a predefined list of commonly used images. Nonetheless, we successfully obtained 78 Dockerfiles from two random public GitHub repositories[1][2], to create and analyze corresponding images. Adhering to our methodology, we deleted each community image post-analysis to ensure storage efficiency. In

[1]https://github.com/jessfraz/dockerfiles
[2]https://github.com/komljen/dockerfile-examples

Metrics		Official Images: 33						
		Grype	Snyk	Trivy	CVE-Bin-Tool	Our Architecture		
						Combination of Grype & BinTool	Combination of Snyk & BinTool	Combination of Trivy & BinTool
Total CVEs	Average	187.09	305.24	186.24	241.21	427.84	546.12	427.12
	Median	78	108	78	264	345	423	346
	Mode	0	0	0	7	8	8	8
	Standard Deviation	348.64	559.43	347.26	186.47	475.85	674.82	474.0
	Max	1112	1796	1113	614	1561	2246	1563
	Min	0	0	0	2	2	2	2
Unique CVEs	Average	77.91	61.18	77.72	156.66	230.93	214.45	230.72
	Median	16	21	32	151	195	194	197
	Mode	0	0	0	7	8	8	8
	Standard Deviation	125.65	92.09	124.07	125.83	223.68	194.53	222.07
	Max	389	297	390	430	705	654	706
	Min	0	0	0	2	2	2	2
Unique Packages	Average	40.48	16.96	40.33	9.66	49.78	26.45	49.63
	Median	37	11	37	8	43	24	43
	Mode	0	0	0	2	3	3	3
	Standard Deviation	48.029	19.65	47.2	7.36	52.42	24.52	51.62
	Max	154	80	153	26	170	13	170
	Min	0	0	0	1	1	1	1

TABLE II: Number of vulnerable CVEs and packages found per official image.

Metrics		Community Images: 78						
		Grype	Snyk	Trivy	CVE-Bin-Tool	Our Architecture		
						Combination of Grype & BinTool	Combination of Snyk & BinTool	Combination of Trivy & BinTool
Total CVEs	Average	71.24	182.69	65.09	183.15	254.04	365.53	247.93
	Median	3.5	125.5	0	60.5	111	199	70
	Mode	0	0	0	3	3	3	3
	Standard Deviation	125.4	268.92	124.95	214.5	288.48	455.93	288.66
	Max	767	1244	768	934	1375	2007	1376
	Min	0	0	0	2	2	2	2
Unique CVEs	Average	40.012	48.98	36.25	128.82	165.31	174.32	162.33
	Median	2.5	20.5	0	52	86.5	93	58.5
	Mode	0	0	0	3	3	3	3
	Standard Deviation	67.07	67.18	66.47	148.078	179.8	196.48	179.6
	Max	336	263	367	695	737	910	738
	Min	0	0	0	2	2	2	2
Unique Packages	Average	19.91	15.39	19.28	7.53	27.28	22.78	26.7
	Median	2	0	8.5	6	9	18	7
	Mode	0	0	0	2	2	2	2
	Standard Deviation	29.91	18.19	30.14	6.27	33.3	22.8	33.46
	Max	127	67	127	32	145	91	145
	Min	0	0	0	1	1	1	1

TABLE III: Number of vulnerable CVEs and packages found per community image.

total, our analysis encompassed 111 images, offering a comprehensive assessment of vulnerabilities within both official and community images.

B. Comparison of Our Proposed Framework and Different Tools to Detect Vulnerabilities

In this series of experiments, we conduct a comparative analysis between our solution and major existing tools such as Grype, Snyk, Trivy, and CVEBinTool, focusing on the average number of detected vulnerabilities. After gathering the images, we independently run each of the four tools and subsequently execute our pipeline, activating both layers of our architecture for comprehensive analysis.

Tables II and III present diverse vulnerability metrics for official and community images, respectively. Our analysis encompasses a total of 111 images, considering three key factors: total CVE count, unique CVE count, and unique package count.

Both the official and community images exhibit a consistent pattern where the individual performance of the four tools varies depending on the specific scenario, such as identifying unique or maximum vulnerabilities. However, our pipeline consistently outperforms the standalone usage of these tools. This superiority is attributed to the significant differences in the domains covered by the three OS package scanner tools

and binTool. Simultaneously utilizing these tools leads to a higher detection rate of vulnerabilities compared to using them individually. Notably, binTool demonstrates a higher average detection of vulnerabilities, including some unique ones. On the other hand, the other three tools excel in detecting unique packages. By combining the outputs of these two tool sets, we achieve enhanced detection and coverage, as clearly demonstrated in the table results. Based on the data, it can be inferred that:

1) *To detect the most amount of CVEs*: The combination of Snyk and binTool detects the highest number of vulnerabilities in images compared to other tool combinations or individual tools, with Snyk detecting the maximum number of CVEs among the base analysis tools (Grype and Trivy), while utilizing it with CVE-Bin-Tool improves vulnerability detection estimation.
2) *For identifying unique CVEs*: For official images, the combinations of Grype with binTool and Trivy with binTool have shown superior performance. However, for community images, the combination of Snyk with binTool emerges as the top performer. Therefore, the choice between the two can be adjusted based on the type of image being analyzed.
3) *For detecting unique packages*: Grype consistently outperforms Trivy and remains the preferred option. Considering the prevalence of community images in real-world scenarios, utilizing Grype in combination with binTool is the recommended choice for improved vulnerability detection.

Another observation is that in community images, the average number of vulnerabilities is lower compared to official images. This is attributed to the utilization of base layer images such as Alpine and Ubuntu, which inherently have lower vulnerabilities at the OS level.

C. Reported Images with atleast one vulnerability

We also compare the number of images with atleast one reported vulnerability among the tools, which shows the strength of our pipeline's detection. This analysis provides an estimation of the pipeline's superior coverage and detection capability compared to other tools. In Table IV, we observe that static analysis tools achieved approximately 80% coverage for official images, while our pipeline identified at least two vulnerabilities even in the best-performing images (Table II), therefore detecting vulnerability for all official images tested. For community images, detection rates varied, with Trivy performing the lowest at 44.87%, while other tools achieved closer to 60-70% detection. However, our architecture achieved 100% vulnerability coverage by leveraging binTool's detection of application package vulnerabilities. Thus, our proposed architecture excels in detecting vulnerabilities that other tools may miss, showcasing the strength of combining base OS and application package vulnerability analysis.

Image Class		Official Image	Community Image
Number of Images		33 (100%)	78 (100%)
Grype		26 (78.78%)	46 (58.97%)
Snyk		27 (81.81%)	47 (69.25%)
Trivy		27 (81.81%)	35 (44.87%)
CVE-Bin-Tool		33 (100%)	78 (100%)
Our architecture	**Grype and BinTool**	33 (100%)	78 (100%)
	Snyk and BinTool	33 (100%)	78 (100%)
	Trivy and BinTool	33 (100%)	78 (100%)

TABLE IV: Number (percentage) of vulnerable images having at least one vulnerability detected by our framework and other existing tools.

VII. Discussion

This study introduces a comprehensive framework for detecting and assessing vulnerabilities in container images. By calculating a vulnerability score (v-score) based on CVSS scores obtained from identified CVEs and comparing it to a threshold score (t-score), users can make informed decisions on whether to proceed with the image or discard it. Images with v-scores lower than the t-score are considered suitable for creating containers, while those with higher v-scores pose a threat and should not be deployed. These more vulnerable images can be further analyzed in a controlled environment to determine the specific vulnerabilities contributing to their high scores.

Our framework comprises of two essential steps: base analysis and package analysis. These two steps are specifically designed to address vulnerabilities related to OS packages and non-OS packages, respectively, thus providing superior coverage over a target image's vulnerability landscape. These steps are seamlessly integrated within a pipeline, allowing for easy incorporation of other scanning tools without requiring any architectural changes. By running the testing image through additional tools and comparing their results with our framework, a comprehensive analysis of the image can be obtained.

While our focus in this study is on Docker containers, it is important to note that the process of creating and working with container images may vary slightly across different container platforms such as Kubernetes or Apache Mesos. However, the underlying concept of container image creation and analysis remains consistent. Our framework is adaptable to these platform-specific differences, requiring only minor adjustments in the image extraction process. Regardless of the container platform, our framework provides a flexible and portable solution for assessing the security of containerized applications, leveraging the benefits of containerization in software packaging and distribution.

VIII. Conclusion

Containerization has revolutionized software packaging and deployment, but it brings security challenges due to the presence of vulnerable packages and inclusions in container images. In this study, we presented a two-layer architecture to detect vulnerabilities in container images before deploying them in production environments. The first layer focused on OS-level package scanning by analyzing base OS layers, while the second layer employed an application package analysis tool to examine non-OS layers. The novelty lies in finding the gap of tools and employing them in our architecture to achieve better vulnerability detection. Our experiments with 111 Docker images, including both official and unofficial images from Docker Hub, revealed that our architecture consistently outperformed individual commercial static analysis tools like Grype, Snyk, and Trivy in detecting vulnerabilities. We achieved higher coverage and detection rates, particularly for vulnerabilities that other tools failed to report.

Despite these achievements, our work has limitations. Currently, our tool focuses on pre-deployment scanning and lacks post-deployment analysis, which we plan to address in future work by incorporating dynamic analysis and learning-based techniques. Additionally, our evaluation is specific to Docker images and does not encompass other container technologies. Future research can expand the scope of our tool and provide recommendations for addressing specific vulnerabilities. We will also investigate the time requirement of our framework at different stages. Furthermore, we aim to enhance our tool with monitoring capabilities for individual or bulk images. Overall, our proposed architecture offers developers and organizations a valuable means to better understand the gap of existing static analysis tools and ensure the security of their container images.

Acknowledgments

The authors thank the anonymous reviewers for their valuable comments. This material is based upon work supported by the Natural Sciences and Engineering Research Council of Canada and Department of National Defence Canada under the Discovery Grants RGPIN-2021-04106 and DGDND-2021-04106.

References

[1] "Docker," https://www.docker.com/, 2023, accessed: May 19, 2023.
[2] "Kubernetes," https://kubernetes.io/, 2023, accessed: May 19, 2023.
[3] "Docker Swarm," https://docs.docker.com/engine/swarm/, 2023, accessed: May 19, 2023.
[4] "Apache Mesos," https://mesos.apache.org/, 2023, accessed: May 19, 2023.
[5] HashiCorp, "Nomad," https://www.nomadproject.io/, 2023, accessed: May 19, 2023.
[6] "Docker Security," https://www.docker.com/resources/security, accessed: May 19, 2023.
[7] A. Duarte and N. Antunes, "An empirical study of Docker vulnerabilities and of static code analysis applicability," in *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*. IEEE, 2018, pp. 27–36.
[8] K. Wist, M. Helsem, and D. Gligoroski, "Vulnerability analysis of 2500 Docker Hub images," in *Advances in Security, Networks, and Internet of Things: Proceedings from SAM'20, ICWN'20, ICOMP'20, and ESCS'20*. Springer, 2021, pp. 307–327.
[9] "Docker Hub," https://hub.docker.com/, accessed: 19 May, 2023.
[10] J. Jackson, "Untrusted Docker Hub images found with monero cryptojacking malware," https://bit.ly/DockerHubMoneroMalware/, The New Stack, 2019, accessed: May 19, 2023.
[11] A. Y. Wong, E. G. Chekole, M. Ochoa, and J. Zhou, "Threat modeling and security analysis of containers: A survey," *arXiv preprint arXiv:2111.11475*, 2021.
[12] I. Anchore, "Grype: a simple and efficient package vulnerability scanner," https://github.com/anchore/grype, 2023, accessed: May 19, 2023.
[13] Snyk, "Snyk container," https://snyk.io/product/container-vulnerability-management/, 2023, accessed: May 19, 2023.
[14] A. Security, "Trivy: A simple and comprehensive vulnerability scanner for containers," https://github.com/aquasecurity/trivy, 2019, accessed: May 19, 2023.
[15] CoreOS, "Clair," https://coreos.com/clair/, 2016, accessed: May 19, 2023.
[16] O. Javed and S. Toor, "Understanding the quality of container security vulnerability detection tools," *arXiv preprint arXiv:2101.03844*, 2021.
[17] V. Jain, B. Singh, M. Khenwar, and M. Sharma, "Static vulnerability analysis of Docker images," in *IOP Conference Series: Materials Science and Engineering*, vol. 1131, no. 1. IOP Publishing, 2021, p. 012018.
[18] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on Docker hub," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 269–280.
[19] O. Tunde-Onadele, J. He, T. Dai, and X. Gu, "A study on container vulnerability exploit detection," in *2019 ieee international conference on cloud engineering (IC2E)*. IEEE, 2019, pp. 121–127.
[20] S. P. Mullinix, E. Konomi, R. D. Townsend, and R. M. Parizi, "On security measures for containerized applications imaged with Docker," *arXiv preprint arXiv:2008.04814*, 2020.
[21] T.-P. Doan and S. Jung, "DAVS: Dockerfile analysis for container image vulnerability scanning," *CMC-COMPUTERS MATERIALS & CONTINUA*, vol. 72, no. 1, pp. 1699–1711, 2022.
[22] T. Bui, "Analysis of docker security," *arXiv preprint arXiv:1501.02967*, 2015.
[23] F. Loukidis-Andreou, I. Giannakopoulos, K. Doka, and N. Koziris, "Docker-Sec: A fully automated container security enhancement mechanism," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1561–1564.
[24] H. Zhu and C. Gehrmann, "Lic-Sec: an enhanced AppArmor Docker security profile generator," *Journal of Information Security and Applications*, vol. 61, p. 102924, 2021.
[25] K. A. Torkura, M. I. Sukmana, F. Cheng, and C. Meinel, "Cavas: Neutralizing application and container security vulnerabilities in the cloud native era," in *Security and Privacy in Communication Networks: 14th International Conference, SecureComm 2018, Singapore, Singapore, August 8-10, 2018, Proceedings, Part I*. Springer, 2018, pp. 471–490.
[26] T. Alyas, S. Ali, H. U. Khan, A. Samad, K. Alissa, and M. A. Saleem, "Container performance and vulnerability management for container security using docker engine," *Security and Communication Networks*, vol. 2022, 2022.
[27] J. Wenhao and L. Zheng, "Vulnerability analysis and security research of Docker container," in *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)*. IEEE, 2020, pp. 354–357.
[28] X. Wang, Q. Shen, W. Luo, and P. Wu, "RSDS: Getting system call whitelist for container through dynamic and static analysis," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 2020, pp. 600–608.
[29] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, "Docker ecosystem–vulnerability analysis," *Computer Communications*, vol. 122, pp. 30–43, 2018.