IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 25 September 2024, accepted 14 November 2024, date of publication 26 November 2024,
date of current version 17 December 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3506663

RESEARCH ARTICLE

An Efficient Random Forest Classifier for Detecting Malicious Docker Images in Docker Hub Repository

MARAM ALDIABAT[1,2], QUSSAI M. YASEEN[3,4], AND QUSAI ABU EIN[2]
[1]Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849, USA
[2]Department of Computer Information Systems, Jordan University of Science and Technology, Irbid 22110, Jordan
[3]AIRC Center, Department of Information Technology, Ajman University, Ajman, United Arab Emirates
[4]Department of Software Engineering, Jordan University of Science and Technology, Irbid 22110, Jordan

Corresponding author: Qussai M. Yaseen (q.yaseen@ajman.ac.ae)

ABSTRACT The number of exploits of Docker images involving the injection of adversarial behaviors into the image's layers is increasing immensely. Docker images are a fundamental component of Docker. Therefore, developing a machine learning classifier that effectively predicts and classifies whether a Docker image contains injected malicious behaviors is crucial as a proactive approach. This paper proposes a machine learning model to assess the feasibility of employing machine learning algorithms for detecting the security status of Docker images available in the Docker Hub repository. The paper develops a machine learning model for detecting malicious Docker images by using a newly created dataset containing Docker images associated with 14 corresponding features that were specifically chosen as they are critical indicators of potential security risks in Docker images, and the dataset was published for research purposes. Moreover, the paper developed and tested several machine learning algorithms using Docker image features: Naïve Bayes, Decision Tree, Random Forest, Gradient Boosting, Extreme Gradient Boosting, and Neural Network. The results show that the Random Forest classifier demonstrates exceptional accuracy, achieving a 99% F1-score and an AUC of 100%. This performance refers to its capability to accurately classify the images and effectively distinguish between secure and insecure images, in addition to the minimal error rate of less than 1%, outperforming state-of-the-art models to identify malicious Docker images.

INDEX TERMS Docker, Docker images, Docker containers, machine learning, malware detection.

I. INTRODUCTION

One of the main challenges in application development and testing is managing various versions of the same environment and even dealing with different environments. For instance, consider a project that demands the combination of a MySQL database with NodeJS. Compatibility issues may arise between different versions of both environments or the underlying operating system, potentially causing project delays. Developers may spend valuable time resolving these problems. Therefore, Docker is a valuable solution to address these challenges by managing the OS level to present software as a container, similar to a virtual machine. It simplifies the containerization of diverse applications and provides a standardized approach for creating, deploying, and managing containers. By utilizing Docker, all the necessary project environments can be containerized, ensuring consistency among all developers working on the application.

Docker's architecture employs a client-server architecture, including a client, server, and command-line interface, as illustrated in Figure 1. The client represents the users interacting with Docker to pull, build, and run images existing in repositories. Conversely, the server is the Docker daemon within the Docker host managing image-related operations. The command-line interface is the command that the client uses to request the images like pull, scan, inspect, run, or any other commands. Docker images, the basic structure in this architecture, are self-contained packages with

The associate editor coordinating the review of this manuscript and approving it for publication was Bing Li.

essential components called ''layers''. Building an image involves constructing Dockerfile instructions that specify the dependencies required for constructing the image. Images are accessible by pulling from Docker repositories, such as Docker Hub [1]. Executing a run command with Docker images represented as Docker containers allows the creation of virtual applications that consume less memory and disk space, operating more efficiently compared to traditional installation applications [2]. This efficiency stands out as a primary benefit of using Docker.

Docker has gained popularity as a containerization solution, making container management as simple as possible [3]. In addition, Docker offers efficient virtualization technology that simplifies the process by utilizing containers, enabling the simultaneous execution of multiple containers on a single host. Due to the nature of distributing Docker images and the image creation process, Docker provides an automated deployment and building process on Docker Hub. This process utilizes trigger events from external repositories, such as GitHub, which works on notifying the host about the availability of a new image through commits made on GitHub.

The Docker host makes the pulling of new images and restart of containers available. This automated production and building process enables the most straightforward creation of new images on Docker Hub. During this process, there is a possibility of downloading dependencies from third-party sources, which can introduce vulnerabilities. Some of these dependencies may be unsecured, thereby increasing the potential for attacks on Docker Hub due to the insecure or vulnerable dependencies and files within the downloaded images from third-party sources [4]. However, it is essential to note that while containers can run concurrently, if one application within a container becomes infected, it can harm the entire system. The potential for malicious activities to inject harmful elements into image layers, intending to exploit the applications built on Docker images, poses a threat. This drawback should be considered when employing Docker for virtualization purposes [2].

Given the vulnerabilities present in downloaded Docker images, containers, and running applications, researchers have directed their attentions toward detecting these vulnerabilities either by developing systems and tools that utilize ML models or by updating and merging the current frameworks to create new strength approach works to distinguish between malicious and expected behaviors in Docker Images and Docker containers.

Various tools and researchers have addressed the vulnerabilities in Docker components. Some of these tools are used to verify whether Docker containers or packages are secure or not, such as Clair [5], Anchor [6], Micro Scanner [7], and Synk [8]. However, despite the many studies that explored techniques and frameworks for detecting malicious Docker images or containers, there is yet to be a public dataset that combines all the relevant features of containers or images. Thus, this research addresses this issue by building a machine-learning application using a novel dataset. The dataset includes the most commonly used features, and based on these features, the application is trained using classification techniques and then selects the best-fit classifier. The ultimate goal is to build a robust and precise model capable of reliably detecting the security status of Docker images, outperforming the performance of state-of-the-art models specifically designed for this purpose.

While reading the state-of-the-art work, some questions were triggered and considered as research questions for conducting this work. They include:

1) How can the scanning process be used to build a Machine Learning model?
2) How can the docker image' features help to develop an algorithm that predicts malicious images?
3) Can malicious images be identified without relying on a CVE database or static analysis to identify Docker vulnerabilities?

This paper addresses these questions and provides a comprehensive discussion and analysis of these questions. The contributions of this paper are as follows.

1) **Machine Learning Application:** This work successfully finds the applicability of applying Machine Learning techniques in identifying malicious Docker images available on the Docker Hub Repository.
2) **Creation of a Novel Dataset:** A new dataset comprising 778 images and 14 essential features has been created. This dataset is a crucial resource for determining the security status of Docker Images.
3) **Development of an Exceptional ML Algorithm:** A Machine Learning algorithm has been built, achieving a remarkable F-score of 99%, an AUC of 100%, and an error rate of less than 1%. This result exceeds the results found by state-of-the-art models.

The core strength of this idea lies in working on the images rather than focusing on containers triggered by state-of-the-art work. The core idea is detecting and preventing malware in the foundational elements of Docker images. This helps to enhance the effectiveness of the overall security framework in Docker.

The organization of the paper is as follows. Section II discusses some related work on Docker images and containers. Section III outlines the methodology employed to construct and preprocess the new dataset. The experiments and results are discussed in Section IV. Section V demonstrates a comprehensive comparison with the state-of-the-art approaches. Section VI discusses a case study. Section VII explains the direct answers to the research questions. Section VIII discusses the strengths and limitations of the proposed approach. The conclusions and the future work are discussed in Sections IX.

II. RELATED WORK

Docker is an open-source platform designed to simplify building, deploying, and running applications by utilizing containers. Containers are isolated environments that pack-

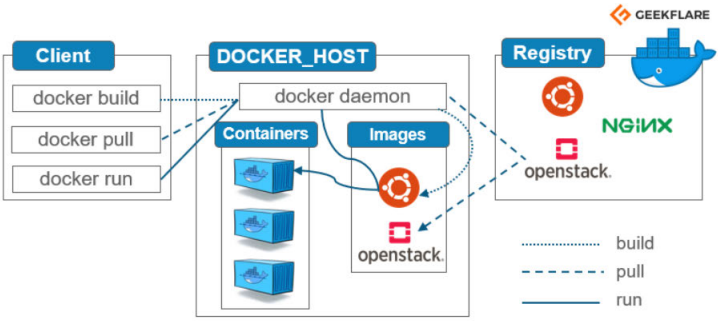


FIGURE 1. Docker architecture components: Client-Server architecture [9].

age an application along with its dependencies, libraries, and configuration files, ensuring consistent performance across different systems and environments. Unlike traditional virtual machines, Docker containers share the host operating system's kernel, making them faster, more efficient, and lightweight. The platform operates on a client-server architecture, where the Docker client interacts with the Docker daemon to manage containers, images, and other components. Docker images, which act as templates for creating containers, include the application and all its dependencies, while containers are the running instances of these images, providing isolation from the host system and other containers. Additionally, Docker Hub serves as a public repository where users can share and download pre-built images, facilitating quick and consistent application deployment. With its ability to ensure consistency, efficiency, and scalability, Docker has become a crucial tool in software development and deployment, integrating seamlessly with DevOps workflows and making it easier to build, test, and deploy applications in any environment [3].

Although Docker has been extensively studied, there is limited research on using machine-learning models to predict vulnerabilities in Docker images. Most existing approaches focus on detecting malicious behaviors in Docker containers, analyzing logs, system calls, or pods, and often rely on frameworks or systems tailored for container-level analysis. To address this gap, this section explores state-of-the-art frameworks and machine-learning models used in securing both Docker containers and Docker images. It highlights significant contributions, identifies limitations, and discusses opportunities for advancing Docker security through these methods.

This section reviews state-of-the-art advancements in these two areas, beginning with a discussion on Docker containers and their vulnerability detection mechanisms, followed by an exploration of methods applied to secure Docker images. These subsections highlight key contributions, limitations, and opportunities for further research in each domain.

A. DOCKER CONTAINERS

This subsection discusses the state-of-the-art achievements of the Docker containers and container scanning tools.

Javad and Toor [10] conducted a study to compare the containers' detection tools, including Clair [5], Anchore [6], and Microscanner. They utilized the Detection Hit Ratio (DHR) to assess the performance of each tool in detecting vulnerabilities, both in OS and non-OS packages. They evaluated 59 Docker images to detect the vulnerabilities using these three tools. The DHR is calculated using a mathematical formula that considers the number of vulnerabilities detected by the tools divided by the total number of vulnerabilities in the image. The results of this study indicate that existing container detection tools perform poorly in detecting vulnerabilities, as they miss some vulnerabilities.

Kohli [11] investigated a novel framework for effectively detecting security issues in Docker containers through a combination of two existing approaches. They utilized AppArmor security profiles for Docker, which restrict the accessibility to the operating system and its applications to prevent potential threats. They used this profile with all pre-installed Docker images from Docker Hub in official and unofficial categories. Then, they used the Clair tool to scan Docker images and identify any exploitation within the packages that may cause a threat to the Docker image. Clair successfully detected the vulnerabilities within OS and non-OS packages. The development of this framework was prompted by the identified problems in the current systems for the vulnerability detection process, such as leaks and outdated methods. Their system helps the developer to have a mechanism to improve the security issue in container applications by using AppArmor and Clair.

Khairi et al. [12] proposed an approach to detect the anomaly-based technique by monitoring the containers using system calls. Their work dealt with a graph-based model to analyze the properties of the syscall, which helps to identify the activities manifested by anomalies. Their results showed that their auto-encoder neural network model can detect the anomalies with an accuracy of 99.8% by utilizing a dataset of 11,700 secure and 1,980 insecure system calls. This accuracy exceeded the results of state-of-the-art models that also used the system calls for intrusion detection in Docker containers.

Karn et al. [13] employed ML models to identify malicious pods. The used dataset consists of system calls from pods

(running containers) to detect crypto-mining processes in Kubernetes clusters [14] by monitoring system calls in kernels. They developed a model for detecting crypto-mining processes in running pods. Based on their findings, the model was used to decide whether to remove or restart the container with a new image. Their approach successfully constructed a decision tree machine-learning model using system calls with an accuracy of 97.2% to predict malware's presence in a set of running containers.

Tien et al. [15] detected anomalies in the Docker containers using system call events as features. The authors extracted the system calls from container logs while running, and then they calculated the frequency of 14 events occurring within the container per second. The results showed that their approach achieved an accuracy of 96% using the KubAnomaly algorithm with Neural Network.

Olufogorehan et al. [16] identified the real vulnerabilities by collecting features from logs while monitoring data when containers are running. They checked for abnormal behavior in system calls compared to normal execution. They employed both static and dynamic approaches for analysis. The static analysis involved comparing package versions with Common Vulnerabilities and Exposures (CVE) databases to assess their security and identify vulnerabilities. In contrast, the dynamic approach utilized ML algorithms, specifically unsupervised learning, to detect vulnerabilities and extract system call features such as CPU utilization, memory usage, network traffic, and system calls. A vulnerability detection model was trained using an ML algorithm. Dynamic detection proved to be more effective than static using unsupervised ML. To enhance performance, they combined both approaches, using k-Nearest Neighbors, and achieved an 86% detection rate.

The achievements of each work focusing on Docker containers are summarized in Table 1.

B. DOCKER IMAGES

This section discusses the state-of-the-art achievements in detecting Docker image vulnerabilities with or without using machine learning.

Haque and Babar [17] explored the exploits in base images for Docker containers in Docker Hub by analyzing real-world applications in 4,681 containers published on GitHub. The goal was to create a dataset comprising 261 base images and their corresponding tags and list the exploitable vulnerabilities using the Anchore tool. They assessed the impact of these vulnerabilities in base images, as these serve as the foundational layer before running any container on top of them. Any derived container from these base images would inherit the vulnerabilities from its base image. Their findings include a list of base images with their tags and corresponding exploitable vulnerabilities or confirmation of the absence of vulnerabilities. This work represents a crucial first step towards developing a system that informs users about these security impacts.

Brogi et al. [18] proposed an alternative approach for applying the search command on Docker images by using other attributes beyond the typical docker search command lines. These attributes include the Docker image name, image size, image stars (rating), and number of pulls (downloads). They utilized software distributions with the ability to store image versions locally. To publish this idea, they introduced a Graphical User Interface (GUI) that helps users specify the attributes they are looking for. Then, the GUI loads all suggested images that match the user specifications. Their objective was to address challenges related to the scanning process and time-consuming in the search command line interface. Therefore, their system was built upon micro-services, emphasizing that it is not intended to replace the Docker repository but rather to reduce the time-consuming nature of the search process in public repositories.

Jain et al. [19] conducted a study to assess the static analysis of vulnerabilities in Docker images. They mentioned the fundamental concepts related to security issues in Docker images by identifying three types of threats. These include *"Caution Containers"*, which refer to the vulnerabilities within running containers; *"Vulnerable Images"*, which are associated with built-in threats and malicious behaviors during image creation that aim to harm projects built upon such images; and Finally *"Images with Attacks"* involves downloading images without knowing their source, refer to the images that are not being pushed to Docker Hub, raising security concerns. Their recommended methodology emphasizes the importance of scanning Docker images rather than containers. However, this approach will help to prevent the transfer of malicious behaviors from images to the containers that run on these vulnerable images.

Huang et al. [20] conducted research to explore a new detection mechanism for threats in Docker images' applications, files, and containers. Their framework utilized three approaches: the Common Vulnerabilities and Exposures (CVE) database, CalmAV or an existing machine learning model, and monitoring computer resources and network requests such as IP and DNS requests. Initially, the CVE database was employed to identify the threats in the applications associated with each Docker image on Docker Hub. Consequently, they attempted to eliminate malicious files in Docker images through two methods. For known malicious behaviors, they used a malicious library called (ClamAV) to compare file features. An existing machine learning model was employed for unknown malicious files, which was discovered by [21] by using random forest-gradient boosting classifiers. Finally, for malicious activities within Docker containers, they monitored the behavior of local resources such as CPU and memory to detect Denial-of-Service (DoS) attacks. Additionally, they looked for malicious connections by analyzing network activity by IP and DNS requests.

Kwon and Lee [22] proposed an approach that focused on diagnosing vulnerabilities in Docker images using a system that analyzed the image name, version, and severity levels to determine its vulnerability score. They extracted

TABLE 1. The related works in docker containers.

Author	Year	Research Aspect	Machine Learning	Achievement
Khairi et al. [12]	2022	Detection of the malicious Docker containers' system calls	Auto-encoder neural network	Build a model to detect anomalies in the system calls of Docker image containers. Achieve an F-score of 99.8% and a higher AUC of 99.9%.
Kohli et al. [11]	2022	Detection of Docker Containers' threats	None	Build a hybrid framework of AppArmor security profiles and Clair tool to detect threats in Docker containers. Their framework exceeds the detection rate achieved by state-of-the-art approaches in Docker container security.
Javad et al. [10]	2021	None	Evaluation of Docker containers' detection tools	Comparing container detection tools, including Clair [5], Anchore [6], and Microscanner, they concluded that existing tools exhibit poor performance in detecting vulnerabilities, as they misclassified some of them.
Karn et al. [13]	2021	Detection of malicious containers using system calls in pods	Decision Tree Model	Build a model that decides whether to remove or restart the container inside the pods based on the existing crypto-mining activities. Their F-score is 97%.
Olufogorehan et al. [16]	2019	Detect the vulnerabilities using containers' logs	Unsupervised Learning Algorithm	Employ two approaches, dynamic and static analysis of containers' logs, then build an ANN model. They were able to detect 85% of the vulnerabilities in containers using logs.
Tien et al. [15]	2019	Detect anomalies using containers' system calls	KubAnomaly algorithm with NN	Build a model with accuracy of 96% that could detect the anomalies within the docker containers.

metadata from Docker images using Docker CLI [2]. Their system serves as an intermediate validation component, informing users whether it is safe to download or upload images from the Docker Hub [1]. The system consists of two parts: IVD module and IVE module [23]. IVD extracts vulnerability metadata, such as the total number of vulnerabilities, the number of unapproved vulnerabilities, CVE severity and its details, package name, and version, using the Clair tool approach [5]. Subsequently, IVD creates a list of vulnerabilities and sends it to IVE for calculating the image's vulnerability score. IVE determines whether the image is vulnerable by comparing the computed score with the threshold score for vulnerabilities, which is defined based on their experience. Their limitation is that static analysis may not be able to detect abnormal behavior.

Watada et al. [25] suggested that scanning a container's or image's code can help identify many vulnerabilities that existing scanning tools cannot detect. Therefore, Pinnamaneni et al. [24] focused on analyzing the code of Docker images and identifying vulnerabilities using a user interface integrated with a machine-learning model. Specifically, their work examined the code of Docker images developed in the Python programming language. They designed a user interface that allows users to input code and detect vulnerabilities by leveraging a stored database. They constructed three models to enhance their detection capabilities, including the voting mechanisms of random forests, decision trees, and boosting algorithms. Remarkably, their model achieved an accuracy of 100% on a dataset comprising 100 code samples of Docker images.

The state-of-the-art methods focused on vulnerability detection using machine learning, tools, and systems. However, it has yet to specifically address the secure determination of a Docker image based on its features by applying machine-learning techniques. Although the Snyk tool [8] offers some capabilities, it does deem static analysis as any existing tool. There is a growing demand for an efficient and intelligent solution to detect secure Docker images. This solution can be achieved using the Docker image name and its version in the Graphical User Interface (GUI). The interface is connected with a machine learning technique capable of extracting specific features of this image version, enabling it to determine the security status of the image. Therefore, this research proposed an approach that avoids the aforementioned limitations of state-of-the-art methods.

The achievements of each work focusing on Docker containers are summarized in Table 2.

III. THE PROPOSED METHODOLOGY

This section provides an in-depth overview of detecting Docker images on Docker Hub. It describes the creation of a new dataset related to Docker images with their features and the methodology used, starting from the dataset creation and preprocessing stages and ending with splitting the data into training and testing sets to train and then evaluating the classifiers. The goal is to identify the best-performing algorithm and compare its performance with the state-of-the-art models in predicting malicious images, containers, and applications in Docker.

Figure 2 illustrates the general structure of this section. It starts with the discussion of establishing authentication for the Docker and Snyk tools. Then, features are extracted using the execution results of the Docker command line. Finally, the preprocessing steps are undertaken to train classification algorithms.

A. DOCKER AND SNYK AUTHENTICATION

To collect the features for the images available in Docker Hub [1], logging in to Docker and Snyk tool [8] using the command line is necessary. The command to log in to Docker Hub is *"docker login"* using the user's username and password as depicted in Figure 3a. Once the login is successfully established, it allows the use of any command line related to Docker, for example, *"docker pull image_name"*, *"docker search image_name"*, *"docker image ls"*, or *"docker inspect image_name: tag"*.

The approach used to collect the images' features primarily focuses on the scan command line because the scanning results list detailed information about docker images. Therefore, to reach this valuable information, the *"docker scan image_name:tag"* command is used. However, the scan command is connected to the Snyk tool and requires

TABLE 2. The related work in docker images.

Author	Year	Container's Component	Machine Learning	Achievement
Kwon et al. [22]	2022	Docker Images Vulnerabilities	None	Build a DIVDS system of two modules that analyze images statically based on the features extracted by the command line. The system works in the static version by comparing it with CVE severity levels.
Haque et al. [17]	2021	Base Images	None	Their findings include a list of base images with their tags and corresponding exploitable vulnerabilities or confirmation of the absence of vulnerabilities based on the Anchore scanning tool.
Jain et al. [19]	2021	Assessment of the static analysis of vulnerabilities in Docker images	None	They emphasize the importance of scanning Docker images rather than containers. This approach helps to prevent the transfer of malicious behaviors from images to the containers that run on these vulnerable images.
Pinnamaneni et al. [24]	2020	Docker Images Code	Using Random Forest Model	Analyze the Docker image code and use the code libraries as features in the dataset to build an ML model for a dataset consisting of 100 instances and five features (Libraries).
Huang et al. [20]	2019	Detection for threats in Docker images and containers	None	They explore a new detection mechanism for threats in Docker images and containers. They rely on utilizing the CVE database, an existing machine learning model, and monitoring computer resources and network requests such as IP and DNS requests.
Brogi et al. [18]	2017	Search command line interface for Docker images	None	Build a system that enhances the search CLI using image name, image size, image stars, and the number of pulls. Their system reduces the time-consuming nature of the search process in public repositories.

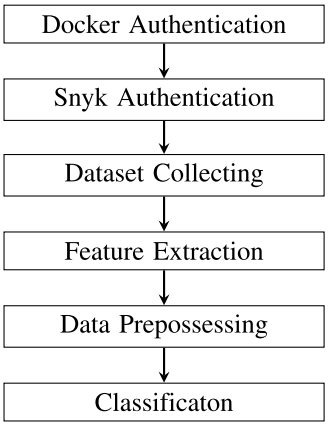
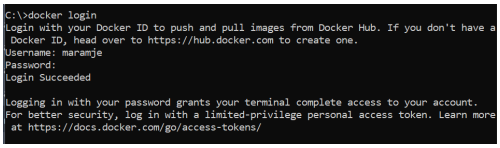
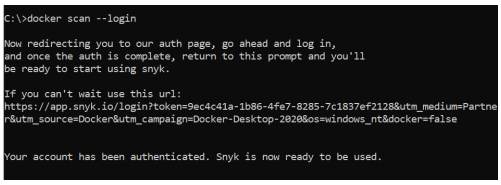


FIGURE 2. Methodology flowchart.



(a) Docker authentication using the *"docker login"* command line



(b) Snyk authentication using the *'docker scan –login'* command line

FIGURE 3. Authentication process to log in to Docker Hub and Snyk tool via the command line.

authentication using the *"docker scan –login"* command, as shown in Figure 3b. That will be redirected to a webpage to complete the authentication process. Once the authentication is completed, the docker scan command can be used with any pulled image.

B. DATASET COLLECTING

This work collected a new and recent dataset[1] that had never been utilized by any previous research. The dataset consists of 14 features and 778 instances, and it has been made available for research purposes.

C. FEATURE EXTRACTION

The new dataset was carefully constructed to ensure it contains essential features of Docker images. Figure 4 illustrates the steps used in dataset creation.

The approach for feature extraction, along with some of the dataset feature names, is based on the methodology used in DIVDS [22]. DIVDS utilized features such as the *last update time*, *Docker image vulnerabilities IDs with their severity ranking*, and *layers IDs*, identifying these as critical for predicting whether a Docker image is secure. They primarily relied on the 'Pull' and 'Inspect' commands to gather data.

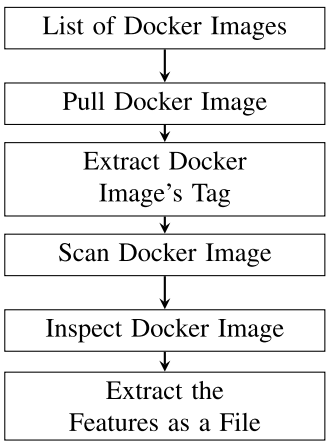


FIGURE 4. Flowchart of collecting experimental data: start by extracting features for all official Docker images, then repeat the same processes with the search list for each official image.

[1] https://github.com/MaramJehad9/DockerImagesPrediction/tree/main

In this study, we adopt a similar process but enhance it by incorporating additional commands, including 'Pull,' 'Scan,' 'Inspect,' 'Search,' and 'List.' The use of the 'Scan' command, for instance, allowed us to capture more detailed information regarding vulnerability categories and the number of vulnerabilities in each category rather than just the vulnerability IDs and severity rankings used in DIVDS. Additionally, we included other important features like *base images*, *alternative base images*, *image size*, and the *number of layers in each pulled image* (corresponding to the layer IDs in DIVDS).

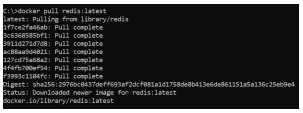
By incorporating more detailed information, we observed improved performance from the machine learning model. However, extracting certain features, particularly from the 'Inspect' command, raised concerns about potential null values for some details. To mitigate this issue and improve the robustness of the dataset, we supplemented these with the 'Scan' command, which provided a more comprehensive view of image vulnerabilities.

Algorithm 1 explains the dataset collection process starting from the *Main* Procedure. The *Main* has two parts: the first one works to list official images using *RetrieveDockerImages* and extracts their features using *ExtractDockerImagesFeatures*. The second part works to loop through these official images and extract the search list for each image.

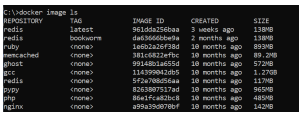
Starting from the first part, which begins by retrieving all Docker images labeled as *"Official"* on Docker Hub, as demonstrated in *RetrieveDockerImages* Function. Subsequently, the features of these official images are extracted through *ExtractDockerImagesFeatures* Function. *ExtractDockerImagesFeatures* pulls the image using another function calling *PullDockerImages*. If the pull is successful, as depicted in Figure 5a, all locally pulled images are then listed to extract information such as size, last update, and tag (version), as explained in the *RetrieveDockerImageTag* Function, with an example of the results elaborated in Figure 5b, Figure 5c and Figure 5d.

Once the tag of the pulled image is identified, the algorithm passes its name and tag to the scanning process through *ScanDockerImages* Function, with an example of the results presented in Figure 5e. The Docker scan result has many important features as listed below and as highlighted in Figure 6a:

1) Looking for the string *'Base Image:'* to extract the name of the base image.
2) Searching for the string *'Package Manager:'* for extracting the package manager value.
3) Counting the occurrences of the following statements in the scan results: *"Low severity vulnerability found," "Medium severity vulnerability found,"*, *"High severity vulnerability found,"* and *"Critical severity vulnerability found."* This provides insight into the number of vulnerabilities at each severity level.
4) Identifying the presence of either *"No vulnerable paths found."* or *"According to our scan, you are currently using the most secure version of the selected base image"* as shown in Figure 6b. These phrases indicate that the scanned image is the most secure version, and then the image is labeled as secure.
5) Detecting the strings *"Recommendations for base image upgrade:"* or*"Your base image is out of date,"* both of which suggest unsecured Docker images as shown in Figure 6c.
6) Locating *'Base Image Vulnerabilities Severity'* to determine the number of base images listed below this line and store the value as the count of alternative base images.
7) Identifying the presence of *"Tested /(d)+ dependencies for known issues"* to extract the count of tested dependencies.



(a) Example of using the "docker pull image name: tag" command



(b) Example of using the "docker image ls" command

(c) Example of using the "docker scan image name: tag" command

(d) Example of using the "docker inspect image name: tag" command

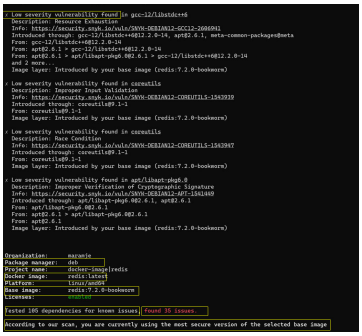
(e) Example of using the "docker search image name" command

FIGURE 5. Examples of using the Docker image Command Line Interface include pull, scan, inspect, list, and search.

Subsequently, the image name and version are utilized in the inspect command within the *InspectDockerImage* Function. The function works to extract the number of layers by looking for the keyword *"RootFS,"*, which contains a list of layers

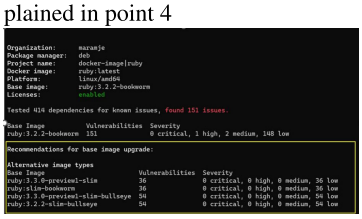
Pseudocode 1 Data Collecting Algorithm

```
procedure Main
    official_images_list ← RetrieveDockerImages()
    ExtractDockerImagesFeatures(official_images_list)
    for image_name in official_images_list do
        SearchDockerImages(image_name)
    end for
end procedure

function RetrieveDockerImages
    Establish Docker Connection by "docker login" CLI
    Establish Snyk Connection by 'docker scan –login' CLI
    official_images ← Retrieve the names of all Docker images labeled as 'Official'
    return official_images
end function

function ExtractDockerImagesFeatures(images_list)
    for image_name in images_list do
        if PullDockerImage(image_name) is not an error then
            size, tag, last_update ← RetrieveDockerImageTag(image_name)
            scan_results ← ScanDockerImages(image_name, tag)
            layers ← InspectDockerImages(image_name, tag)
            Save (image_name, size, tag, last_update, scan_results, layers) to ExcelFile
        end if
    end for
end function

function PullDockerImage(image_name)
    return the execution of 'docker pull image_name'
end function

function RetrieveDockerImageTag(image_name)
    local_images ← Execution of the command line 'docker image ls'
    for image in local_images do
        if image_name equals image['image_name'] then
            tag, last_update, size ← image['tag'], image['last_update'], image['size']
        end if
    end for
    return size, tag, last_update
end function

function ScanDockerImages(image_name, tag)
    return Execution of 'docker scan image_name:tag'
end function

function InspectDockerImages(image_name, tag)
    inspect_results ← Execution of 'docker inspect image_name:tag'
    layers ← count of layers in inspect_results
    return layers
end function

function SearchDockerImages(image_name)
    search_results ← Execution of the command line 'docker search image_name'
    ExtractDockerImagesFeatures(search_results)
end function
```



(a) Example of Docker scan image results. Showing the features' label and its corresponding values that help to extract the Docker features as outlined in points 1 to 7. E.g., the value of the package manager feature in the nginx image is deb

(b) Illustrates the process of labeling Docker images as secure. As explained in point 4



(c) Illustrates the process of labeling Docker images as insecure, as explained in point 6.

FIGURE 6. Examples of using the extracting the exact naming of Docker images' features in the results of each command line.

and counts them as presented in Figure 5d. The count of these layers is then used as the layer feature for each Docker image. Finally, the extracted features are saved in an Excel file. Using official images in the dataset is insufficient because many images are pushed without labeling. To make the dataset representative of all cases in any Docker image category, randomly selecting images from any category is necessary. To achieve this, the dataset was expanded through a search command that lists all images with similar names to those searched, regardless of any category. This is explained in the second part of the 'Main' Procedure. The search is conducted across all official images using the 'SearchDockerImages' function, and all steps of the 'ExtractDockerImageFeature' function are repeated for the newly obtained images.

D. DATA PREPROCESSING

This subsection discusses the preparation of the data for the classification process; several preprocessing steps were applied as shown below.

1) DATA CLEANSING

After merging the features obtained from the command line results, it was noticed that certain information regarding the package manager or base images was missing. As a remedy, these missing values were replaced with 'NaN.' To enhance the performance of the classifiers, the 'dropna()' function was utilized to eliminate Docker images with 'NaN' values.

2) FEATURE ELIMINATION

This process involves dropping the 'Docker Image Name' feature from the dataset because it does not provide meaningful variation due to its unique values, which can be considered noise in the analysis or modeling task context.

3) DATA UNIFORMITY AND TRANSFORMATION

Certain features are expressed in different units, such as the 'size' containing data in KB, MB, and GB. It was crucial to standardize its representation by converting all units to bytes before scaling and normalizing the data values using a scaling function to put all values between 0 and 1. Furthermore, the 'last time updated' contains seconds, minutes, hours, days, weeks, months, and years units. These units were converted to the *day* to facilitate consistent analysis, enabling effective scaling and comparison. It makes the column values comparable and suitable for the learner while preserving the patterns in the dataset. Next, the Encoder was utilized on categorical columns to convert the variables *'Package Manager'*, *'Base Image'*, and *'Tag'* into numerical values. This procedure ensures uniformity in the scale of all features within the dataset, ultimately enhancing the performance of ML algorithms that were built on top of this data.

After preprocessing, the dataset was refined to include 14 carefully selected features. Each of these features was chosen for its ability to provide meaningful insights into the security status of Docker images, capturing various characteristics that influence their vulnerability. Below is a detailed description of each feature and its significance in assessing Docker image security, as shown in table 3.

E. MACHINE LEARNING CLASSIFICATION MODELS AND EVALUATION MATRICES

Machine learning (ML) is a subset of artificial intelligence that enables systems to learn and improve from experience without being explicitly programmed. ML models use data to identify patterns and make predictions or decisions. Broadly, ML can be categorized into three types: supervised learning, unsupervised learning, and reinforcement learning. Supervised learning involves training models on labeled data to predict outputs for new inputs, while unsupervised learning identifies hidden patterns in unlabeled data. Reinforcement learning focuses on decision-making by training agents through trial and error to maximize rewards [26], [27].

In this work, two categories of algorithms were employed: supervised learning and Neural Network algorithms. The machine learning models utilized include Naïve Bayes,

TABLE 3. Description of the dataset features.

Features name	Description
Tag (Docker Image Version)	The tag identifies the version of the Docker image. This feature is essential because certain versions may contain vulnerabilities that have been patched in newer releases. Analyzing the tag helps the model identify images that might be outdated or insecure.
Last Update (Measured in Days)	This feature represents the number of days since the Docker image was last updated. Images that haven't been updated recently may lack security patches and are more likely to include outdated dependencies, increasing their vulnerability to attacks.
Package Manager	The package manager used in the image (e.g., apt, yum) provides information about how dependencies are managed within the image. Different package managers have varying levels of vulnerability, depending on how frequently they are maintained and updated.
Base Image	The base image is the foundational layer upon which other layers are built. The choice of base image is crucial because insecure base images can propagate vulnerabilities to all dependent layers. Identifying the base image allows for assessing its security status and its impact on the entire image.
Number of Alternative Base Images	This feature indicates how many more secure versions of the base image are available. The presence of alternative base images suggests opportunities to replace insecure images with more secure options, helping to mitigate risks.
Number of Tested Dependencies	This feature captures the number of software libraries or packages tested for vulnerabilities. A higher number of tested dependencies can indicate greater scrutiny and a reduced likelihood of undetected security issues.
Critical Severity Vulnerabilities	This feature represents the count of vulnerabilities categorized as critical. Critical vulnerabilities are the most severe and can have immediate, far-reaching consequences if exploited. This feature plays a pivotal role in assessing the overall risk posed by an image.
High Severity Vulnerabilities	High severity vulnerabilities are significant and can result in substantial damage to systems or applications. By identifying the number of such vulnerabilities, this feature contributes to evaluating the potential risk level.
Medium Severity Vulnerabilities	Vulnerabilities of medium severity are less critical than high or critical ones but still represent a considerable risk. These vulnerabilities often provide adversaries with opportunities to exploit a system indirectly.
Low Severity Vulnerabilities	Low severity vulnerabilities are less likely to cause immediate harm but can still be used in conjunction with other exploits to compromise a system. Including this feature ensures a comprehensive analysis of an image's security posture.
Layers	The number of layers in the Docker image represents the steps involved in building the image. A higher number of layers often correlates with increased dependencies and a larger attack surface, which could make the image more vulnerable.
Size	The size of the Docker image, including all layers and dependencies, is an important feature. Larger images may include unnecessary components or unused dependencies, which can introduce additional vulnerabilities.
Number of Vulnerable Dependencies	This feature counts the dependencies within the Docker image that have known vulnerabilities. By identifying these dependencies, the model can better assess the potential risks associated with the image.

Decision Tree, Random Forest, Gradient Boosting, Extreme Gradient Boosting, and Neural Network.

Evaluation metrics were used to evaluate the performance of ML models and provide quantitative values for comparing different algorithms. The choice of evaluation metrics depends on the specific task, such as classification, regression, or clustering. This research used classification evaluation metrics, including Precision, Recall, F1 Score, Accuracy, Receiver Operating Characteristic (ROC), Area Under the Curve (AUC), and Error Rate. These metrics help to assess the overall performance of the classification models being evaluated [28] and [29].

Precision: measures the proportion of correctly predicted positive instances out of the total number of instances predicted as positive; Equation 1 shows how to calculate the precision.

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}} \tag{1}$$

Recall: refers to the number of true positive instances out of the total number of instances that should be identified as positive. Equation 2 shows how to calculate the recall.

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}} \tag{2}$$

F1 Score: is a performance metric used in classification tasks to evaluate the balance between precision and recall. It combines precision and recall by taking their means, providing a balanced evaluation of the model's performance. Equation 3 shows how to calculate the F1 score.

$$\text{F1 Score} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3}$$

Accuracy: is a commonly used metric in classification tasks to assess the performance of the algorithms. It is calculated by considering the values in the confusion matrix, as shown in Equation 4. A higher accuracy score indicates better performance; however, the accuracy alone is insufficient for evaluating the model's performance, especially in imbalanced datasets, so it will be better to consider F1-Score.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}} \tag{4}$$

ROC: is a graphical tool used to assess the performance of binary classification algorithms and evaluates the model's ability to distinguish between positive and negative classes by plotting the True Positive Rate (TPR) on the x-axis and the False Positive Rate (FPR) on the y-axis. TPR and FPR are calculated using the following equations 5 and 6:

$$\text{True Positive Rate (TPR)} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}} \tag{5}$$

$$\text{False Positive Rate (FPR)} = \frac{\text{False Positive}}{\text{False Positive} + \text{True Negative}} \tag{6}$$

AUC: is a single numerical value that summarizes the classifier's overall performance based on the ROC curve. AUC quantifies the classifier's ability to discriminate between positive and negative examples in the dataset. An AUC value closer to 1.0 indicates a better-performing classifier [30].

IV. EXPERIMENTS AND RESULTS

This section discusses the results obtained in this research using previously mentioned classifiers and applying the machine learning evaluation matrices on each classifier.

After collecting and preprocessing the dataset, it was divided into training and testing sets, using a testing rate of 25%. This ratio was selected by following multiple trials with various partition rates; it was noticed that a testing rate of 30% resulted in insufficient training data for the model to learn effectively. On the other hand, a testing rate of 20% led to the model being well-trained on the data but struggling to generalize effectively during the testing process.

In the initial stage, the dataset started with 14 features and 313 Docker images to assess the feasibility of using ML algorithms to predict Docker image behaviors. The results affirmed the potential of ML algorithms for this classification task. Subsequently, the dataset was expanded by incorporating more Docker images and introducing a pivotal feature related to Docker image construction. This vital feature refers to the layers of Docker images, representing the instructions that the Docker image follows in the running version, as specified in the Docker file. These instructions are transformed into layers within the Docker image during image-building. Consequently, the count of layers for each Docker image was used as a feature in the dataset. Therefore, these 14 features collectively played a significant role in determining the security status of Docker images. The dataset was then expanded to include 778 images with 14 features. Among these data points, 468 were classified as insecure Docker images, while 310 were labeled as secure. The dataset underwent preprocessing to prepare the data for various algorithms to predict malicious Docker images.

A. EVALUATING RESULTS USING BINARY CLASSIFICATION MATRICES

This subsection presents the results using the Confusion Matrix, Precision, Recall, F1 Score, and Accuracy.

The confusion matrix was used to present the performance of each classifier using true predictions in the positive label (TP), false predictions in the positive label (FN), true predictions in the negative label (TN), and false predictions in the negative label (FP) as present in Figure 7. The matrix demonstrates that the Random Forest, XGBoost, and Gradient Boost classifiers share similar values in these four categories. Notably, only one error occurred in predicting both negative and positive labels. To present a different aspect of the evaluation, additional matrices such as precision, recall, F1 score, and accuracy were computed for each class label independently, as summarized in Table 4, and for the classifiers' overall performance in both class labels, depicted in Figure 8. These matrices provide comprehensive insights into the performance of various algorithms when classifying Docker images based on 14 features extracted from the images. Significantly, the Random Forest, Gradient Boost, and XGBoost classifiers boast the highest F1 scores, each achieving an impressive 99%. These high scores indicate that the three models accurately identify the security status of Docker images using the provided features.

B. EVALUATING RESULTS USING ROC AND AUC CURVES

In addition to the matrices employed before to demonstrate classifier performance, the ROC and AUC curve play a significant role in evaluating each classifier, as depicted in Figure 9. All classifiers show outstanding discrimination between the binary labels in the current dataset. Notably, the values of 1.0 for Random Forest, Gradient Boost, and XGBoost show exceptional discrimination between the binary labels, signifying the effective performance of these models on new datasets. These high values do not indicate realism or problems within the dataset but reflect the models' strong performance. Despite all the trained models performing well, considering the F1-score with the AUC value reveals that these models exceed at distinguishing between positive and negative samples and accurately predicting image labels based on image features. However, there is no cut-off to say this is the best classifier among those three.

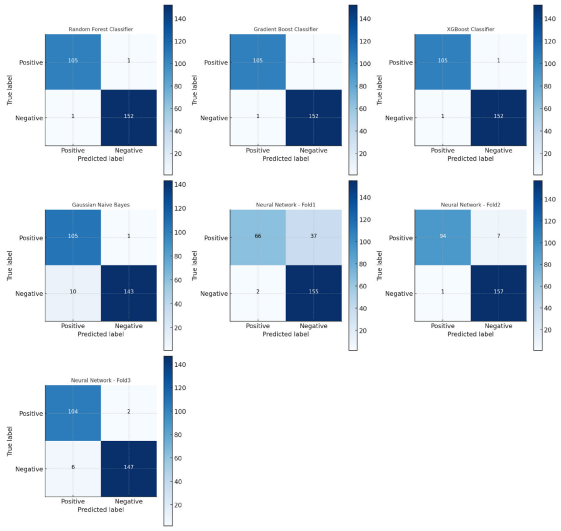


FIGURE 7. The Confusion Matrix For Each Classifier.

TABLE 4. Evaluation matrices for each class labels using machine learning classifiers.

ML Classifier	Class Label	Precision	Recall	f1-score
Random Forest Classifier	Insecure	0.99	0.99	0.99
	Secure	0.99	0.99	0.99
Gradient Boost Classifier	Insecure	0.99	0.99	0.99
	Secure	0.99	0.99	0.99
XGBoost Classifier	Insecure	0.99	0.99	0.99
	Secure	0.99	0.99	0.99
Gaussian Naive Bayes Classifier	Insecure	0.99	0.93	0.96
	Secure	0.91	0.99	0.95
Neural Network - Fold1	Insecure	0.81	0.99	0.89
	Secure	0.97	0.64	0.77
Neural Network - Fold2	Insecure	0.96	0.99	0.98
	Secure	0.99	0.93	0.96
Neural Network - Fold3	Insecure	0.99	0.96	0.97
	Secure	0.95	0.98	0.96

C. EVALUATING RESULTS USING ERROR RATE IN TRAINING AND TESTING DATA

The error rate matrices were used to decide the optimal classifier among these three high-performing models mentioned in the previous section. These error rates were plotted using 14 estimators for each classifier, and the model with the lowest testing error rate was selected. As shown in Figures 10, 11, and 12, the error rates for all three classifiers show minor prediction errors in both the training and testing phases, each falling below 0.05. However, the zoomed-in for these error values reveals that the RF has almost exact error rates on training and testing data when the number of estimators equals 6, 8, or 10. In contrast, XGBoost and Gradient Boosting show some variance greater than 0.003 between the train and test data.

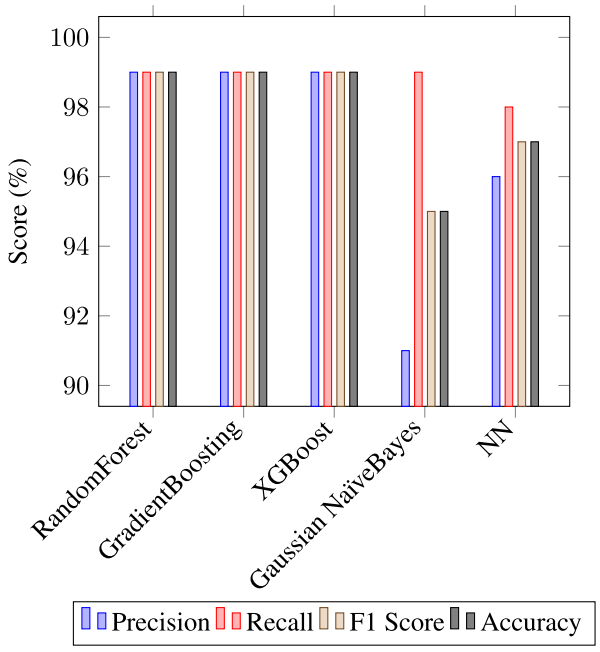


FIGURE 8. Performance comparison using Precision, Recall, F1 Score, and Accuracy for each classifier.

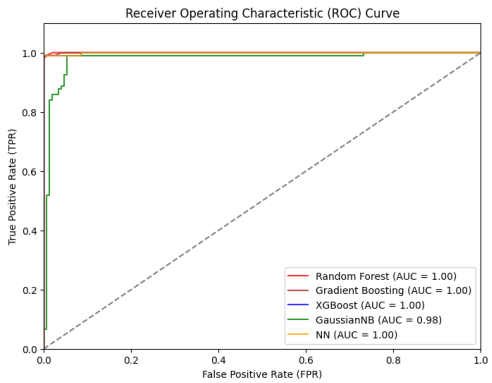


FIGURE 9. Receiver Operating Characteristic (ROC) curve and Area Under the Curve (AUC) for each classifier.

D. OVERFITTING PREVENTION AND MODEL GENERALIZATION

To prevent overfitting and ensure the model generalizes well across different data subsets, we applied k-fold cross-validation with k = 5. Cross-validation works by splitting the dataset into five equal parts (folds). The model is trained on four of these folds and tested on the remaining one. This process is repeated five times, with a different fold used as the test set each time. By averaging the performance across all five folds, cross-validation provides a more robust estimate of how the model performs on unseen data, reducing the risk of overfitting that may occur when training and testing on a single dataset split.

By applying this strategy, the model learned patterns in the data without becoming overly specific to the training set, thus allowing it to perform well on unseen data. This technique also allows us to detect cases where the model might perform well on the training data but poorly on the test data, which would indicate overfitting.

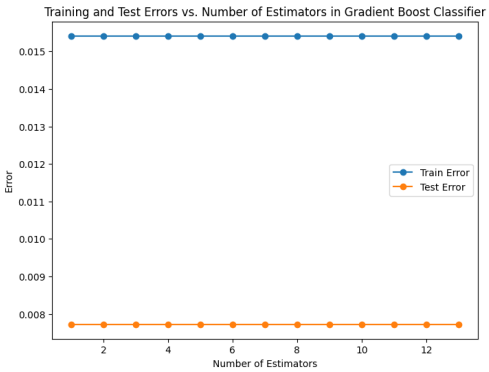


FIGURE 10. Gradient boost classifier's error rate in training and testing data.

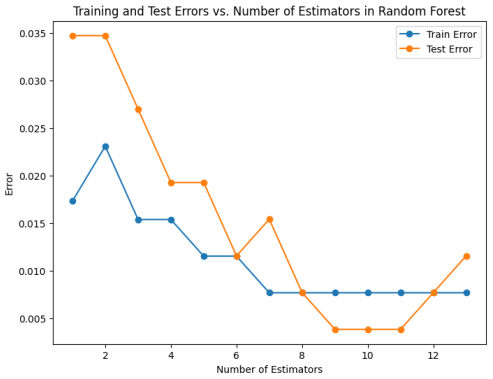


FIGURE 11. Random forest classifier's error rate in training and testing data.

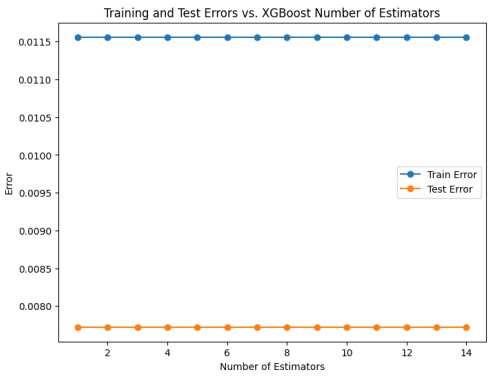


FIGURE 12. XGBoost classifier's error rate in training and testing data.

E. EVALUATING RESULTS FROM AN END-USER PERSPECTIVE

Considering the values of the F1 Score, AUC, and error rate from the previous sections, the final decision is that random forest accurately predicts Docker images' security status. So, assuming that RF is released and needs testing by end users to evaluate its performance, nine Docker images were randomly selected from Docker Hub, as shown in Figure 13. Following the same steps for extracting features

	Docker Name	Tag	Last Update	Size	Package Manager	Base Image	Number of Alternative Base Images	Number of Tested Dependencies	Number of Vulnerabilities	Critical Severity	High Severity	Medium Severity	Low Severity	Layers	secure	Prediction Using RF	Actual Labels
0	jon33333/spss:latest	latest	5 years	527MB	deb	0	0	270	673	0	12	292	369	12	0	0.0	0
1	yopero/spss-to-csv:latest	latest	5 years	670MB	deb	0	0	257	786	66	197	161	362	8	0	0.0	0
2	oraclelinux:9	9	2 hours	225MB	rpm	0	0	182	30	0	4	26	0	1	0	0.0	0
3	kasmweb/oracle-8-desktop:develop	develop	22 hours	8GB	rpm	0	0	1288	31	0	0	0	31	6	1	0.0	1
4	almalinux/9-minimal	latest	3 weeks	86MB	rpm	16	0	98	6	0	0	6	0	1	1	1.0	1
5	dokken/almalinux-8	latest	59 minutes	386MB	rpm	0	12	245	0	0	0	0	0	2	0	0.0	0
6	spack/almalinux8	latest	23 hours	906MB	rpm	0	16	289	0	0	0	0	0	8	0	0.0	0
7	spack/almalinux9	latest	24 hours	831MB	rpm	0	12	277	0	0	0	0	0	8	0	0.0	0
8	alpine	3.16	4 days	6MB	apk	16	0	15	0	0	0	0	0	1	1	1.0	1
9	alpinelinux/docker-cli	latest	5 days	94MB	apk	0	0	19	0	0	0	0	0	3	1	1.0	1

FIGURE 13. A small sample of Docker images for evaluating the random forest model after training and testing its performance.

from the Docker images, the features were input into the RF model to predict their security status. The RF model correctly identified the security status of eight out of the nine unseen images. However, an error was observed in the classification of the fourth image, *'kasmweb/oracle-8-desktop:develop'*. The main reason for the misclassification of this part is that the number of secure images in the original dataset was insufficient for the model to distinguish the secure images accurately. Overall, the evaluation of RF using those nine images shows that the model may encounter some issues with secure images but generally performs well.

F. FEATURE IMPORTANCE AND MODEL INSIGHTS

In this study, the Random Forest classifier was trained using 14 key features extracted from Docker images. These features were selected based on their relevance to the security and vulnerability assessment of the images. The model was able to identify the most influential features in detecting malicious Docker images. Figure 14 referencing the feature importance plot, shows the importance score of each feature in the model's decision-making process.

The most influential features identified by the model include:

Last Update: This feature, representing the number of days since the Docker image was last updated, was found to have the highest impact. Docker images not recently updated are more likely to contain vulnerabilities due to outdated dependencies or security patches.

Medium Severity: This feature captures the number of vulnerabilities classified as medium severity in the image. These vulnerabilities are significant enough to compromise security but are often overlooked if not critically monitored.

Alternative Base Images: The availability of more secure base images indicates that there are safer options for building a Docker image. If a potentially insecure or outdated base image is used when a more secure alternative is available, this raises a red flag. The decision to not use a more secure base image suggests that the image might be compromised or that the developers have overlooked security best practices. Therefore, images built on outdated or vulnerable base images are more likely to introduce security risks.

Critical Severity: Another highly significant feature was the presence of vulnerabilities with critical severity. Docker images with critical security issues pose immediate risks to the system and require urgent attention.

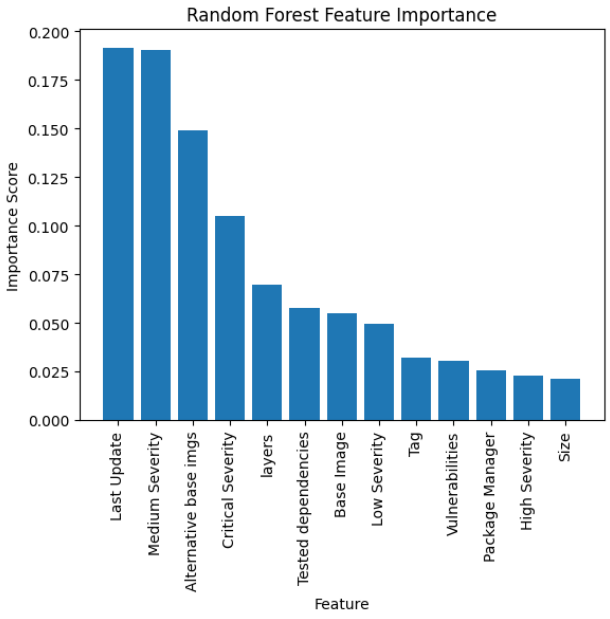


FIGURE 14. Random forest feature importance for Docker image classification.

Layers: The number of layers in the Docker image also contributed to its classification. Images with more layers often have more dependencies, which may increase their attack surface and vulnerability.

Other features such as tested dependencies, base image, and various severity levels of vulnerabilities (e.g., low and high) also contributed to the model's predictive accuracy, though to a lesser extent.

This feature importance analysis provides valuable insights into how adversaries might exploit specific vectors within Docker images. Understanding the influence of these features allows developers to prioritize which aspects of Docker images need closer examination, thus enhancing overall security practices.

V. COMPARISON WITH STATE-OF-THE-ART MODELS

The comparison with state-of-the-art models is summarized in Table 5, containing various aspects such as the components used for predicting vulnerabilities (whether images or containers), the features of the dataset (whether they are image features, system calls, logs, or others), the dataset size used for prediction, and whether machine learning is applied or a framework is developed. Additionally, their achievements.

Olufogorehan et al. [16] focused on detecting vulnerabilities in Docker containers; their accuracy reached 86% in detecting exploits using Artificial Neural Networks. However, they identified a limitation in their algorithm – the lack of image features. The main features used in the prediction process are the frequency of system calls and the execution time of system calls.

Karn et al. [13] detected crypto-mining activities in sets of running containers called pods. They utilized a dataset of 5000 system call frames and 14 syscall n-grams as features extracted from healthy pods. Then, a Decision Tree model was employed, getting an accuracy of 97.2%.

Tien et al. [15] detected anomalies in the Docker containers using system call events. Their approach extracted the system calls from container logs while it was running to calculate the frequency of 14 events occurring within the container per second. The highest accuracy achieved was 96% using the KubAnomaly algorithm with NN.

Javad and Toor [10] compared the existing container scanning tools, and they found that the existing tools have poor performance in detecting vulnerabilities, as they missed some vulnerabilities.

Kohli [11] investigated a novel framework for effectively detecting security issues in Docker containers through a combination of AppArmor security profiles and the Clair tool to identify any exploitation within the packages that may cause a threat to the Docker image.

Khairi et al. [12] detected the anomaly-based technique by monitoring the containers using system calls. Their results showed that their auto-encoder neural network model can detect the anomalies with an accuracy of 99.8% by utilizing a dataset of 11,700 secure and 1,980 insecure system calls.

Pinnamaneni et al. [24] achieved perfect accuracy using voting mechanisms of random forests with a dataset of 100 Docker images. However, achieving 100% accuracy with such a small dataset raises concerns about overfitting and generalizability.

Haque et al. [17] aimed to detect base image exploits by analyzing Docker applications on GitHub. Although they successfully identified a dataset of 261 base images and their vulnerabilities using Anchore. Their approach shares similarities with others relying on existing tools, and it only provides entirely new insights by using base images instead of containers.

Huang et al. [20] developed an effective hybrid mechanism to detect threats in Docker images and containers. Nevertheless, their approach heavily relied on the CVE database for identifying significant threats, which are still around the same initial basics.

Brogi et al. [18] developed a novel system to enhance the search command for Docker images, particularly addressing security issues related to image versions. Their system allows users to specify the tag they need to download. Therefore, they contributed to improved security practices.

Jain et al. [19] focused on assessing the static analysis of Docker image vulnerabilities, highlighting that detecting Docker images is more effective than running containers. Their approach provides valuable insights into the vulnerabilities in static images, offering a proactive approach to security.

The DIVDS system [22] relied on static analysis and CVE to detect vulnerabilities in Docker images without applying machine learning. The current work on this system considers additional image attributes and applies machine learning as what we explained in feature imporatnce.

Table 5 shows that the proposed model outperformed the state-of-the-art models by achieving a detection rate of 99% using the Random Forest classifier. It exceeds the state-of-the-art models on containers by focusing on their basic components - images. Moreover, it outperforms image studies by combining docker images' features with machine learning, resulting in a lower false positive and false negative rate in the prediction process.

When comparing the results with Pinnamaneni et al. [24], their dataset consisted of only 100 samples, significantly smaller than the dataset used in this study. Additionally, there is no evidence in their work to suggest that they implemented any techniques to prevent overfitting in their model. In contrast, Khairi [12] achieved higher metrics with an accuracy of 99.2%, an F1-score of 99.8%, precision of 99.7%, and recall of 99.8%. However, his approach differs from ours. Khairi's work focused on detecting vulnerabilities after running the images, whereas our approach performs detection before the images are executed on a local machine. This provides a significant advantage as it aligns with the recommendation made by Jain et al. [19], which emphasizes the importance of scanning Docker images before running containers on top of them. Therefore, our approach can be considered more proactive in preventing vulnerabilities before the deployment begins.

VI. CASE STUDY

To illustrate how our Random Forest model can be applied in real-world scenarios, consider a large-scale web application that relies on Docker images for various services. The application uses Nginx for web hosting, MySQL for databases, and Node.js to run backend processes. These images, which are regularly pulled from Docker Hub, are crucial for the smooth functioning of the system. However, using publicly available Docker images presents certain risks, as they may contain vulnerabilities that attackers could exploit. These vulnerabilities could potentially allow unauthorized access or lead to compromised data security.

To mitigate these risks, our Random Forest model is integrated into the system's continuous integration and deployment (CI/CD) pipeline. Assuming we are using Jenkins to manage CI/CD, the first step is to connect Jenkins to the GitHub repository or local project. Next, we set up our pipeline as illustrated in Figure 15. The pipeline first pulls Docker images and installs basic requirements such as Docker and the Docker SDK to enable scanning and inspection of the image.

In the next stage, the scan process begins. This involves running a Python script to extract image features and perform a prediction using the pre-trained Random Forest model developed in this work. If the image is deemed secure, the

TABLE 5. Comparison with state-of-the-art models.

Author	Year	ML Algorithm	Docker Component	Dataset Size	results
Khairi [12]	2022	Neural Network	Containers	Two datasets containing 11,700 normal and 1,980 system call traces	• F1_score of 99.8% • Precision of 99.7% • Recall of 99.8% • Accuracy of 99.2%
Kohli [11]	2022	Not Used	Containers	Use security plugins of WordPress website	This paper mostly evaluates the effectiveness of combining AppArmor with Clair for vulnerability detection without providing exact percentages for the metrics like precision or recall
Karn et al. [13]	2021	Decision Tree	Containers	5000 system calls frame with 35 features	• F1_score of 97% • Precision of 97% • Recall of 97% • Accuracy of 97%
Tunde et al. [16]	2019	Artificial Neural Network	Containers	100s of system calls	86% overall detection coverage for combined static and dynamic approaches
Tien et al. [15]	2019	Neural Network	Containers	26,660 of Abnormal 15,000 of Normal Behaviors	• F1_score of 97.7% • Precision of 97.7% • Recall of 97.7%
Haque et al. [17]	2022	Not Used	Images	261 Base Images	No specific F-scores, precision, or recall were calculated. Results were presented in terms of the prevalence of high exploitability vulnerabilities in base images
Kwon [22]	2020	Not Used	Images	Did not mention (using Docker Image Repository)	Evaluation was based on detecting vulnerabilities using static analysis techniques, providing qualitative detection scores
Pinnamaneni et al. [24]	2020	Voting Model	Images	100 image codes	• F1_score of 100% • Precision of 100% • Recall of 100% • Accuracy of 100%
Huang et al. [20]	2019	Not Used	Images	261 base images	No specific precision, recall, or F1-score values are provided, but the framework addresses detecting known and unknown threats using CVE, ClamAV, and pretrain machine learning model
Brogi et al. [18]	2017	Not Used	Images	None	No specific machine learning accuracy or F1-score. The results focus on improving search functionality and efficiency.
Jain et al. [19]	2021	Not Used	Images	None	Specific performance metrics such as precision, recall, or accuracy are not mentioned
This Work	2023	Random Forest	Images	778 Images	• F_score of 99% • Precision of 99% • Recall of 99% • Accuracy is not considered due to imbalance dataset.

integration process continues. If the image is flagged as insecure, the pipeline halts, and the developer is notified to either find an alternative secure image or consider a different version of the image to ensure a secure CI/CD product.

For instance, if an Nginx image pulled from Docker Hub is flagged as insecure due to a critical vulnerability in its base layer, the development team would be prompted to locate another version of Nginx, possibly with a different package manager (such as switching from apt to yum) or with a version that has received the necessary security updates. Once a secure version is found, the image is rescanned. If the model classifies the new image as

benign, the pipeline proceeds with the integration and deployment.

On the other hand, if the image is classified as secure, the CI/CD process continues without interruption. This ensures that only safe and verified images are deployed, reducing the risk of security breaches. **The key benefits of this approach are:**

- Automated Security Checks: By embedding the Random Forest model in the CI/CD pipeline, security checks are automated, reducing the manual workload on the development and operations teams.
- Proactive Vulnerability Mitigation: The model provides a proactive layer of security by detecting potential vulnerabilities before they reach production, allowing teams to address issues early.
- Seamless Integration: Once a secure image is identified, the deployment process continues without delay, ensuring security and efficiency in the development lifecycle.

VII. DISCUSSIONS

This paper is based on ideas from previous work that address three essential questions. The following subsections discuss and answer those questions.

A. HOW CAN THE SCANNING PROCESS BE USED TO BUILD A MACHINE LEARNING MODEL?

As mentioned earlier, scanning the image using the scan command line helps list various details, including image name, version, base name, last changes, vulnerabilities, image dependencies, the severity level of the vulnerabilities, and package name. The inspect and listing commands provide further details for any given image name. The features obtained from these commands, especially from the scanning process, are important, with a particular emphasis on the layers that are considered essential in building Docker images and containers.

B. HOW CAN THE DOCKER IMAGE' FEATURES HELP TO DEVELOP AN ALGORITHM THAT PREDICTS MALICIOUS IMAGES?

The crucial features utilized in the Docker Image Vulnerability Diagnostic System (DIVDS) [22], such as last update time, package manager, image ID, layers ID, vulnerabilities ID, and severity rating, were employed in its detection process.

The question triggered: What if other details are used as numeric values? e.g., number of dependencies, number of vulnerable dependencies, number of alternative secure base images, categorize the number of vulnerabilities into severity levels and image size. Machine learning algorithms were applied after creating the dataset using the suggested features of Docker images. The results reveal that all algorithms perform well, mainly when AUC values are 98% and above. This refers to all algorithms trained using this data effectively distinguishing between secure and insecure images based on Docker image features. Consequently, it indicates that the extracted image features contribute to the accurate predictive capabilities of identifying malicious Docker images.''

C. CAN MALICIOUS IMAGES BE IDENTIFIED WITHOUT RELYING ON A CVE DATABASE OR STATIC ANALYSIS TO IDENTIFY DOCKER VULNERABILITIES?

Due to the use of similar procedures in the existing Docker scanning tools like Clair, Anchor, Snyk, and Micro-scanner, which refer to the Common Vulnerabilities Database (CVE) to identify potential malicious behaviors in Docker containers or images, a question arose: Is it possible to rely on image propriety without referring to the CVE to determine image security status? Yes, it is possible. This research relies on the Snyk tool, which is considered the best tool based on its features. The best features depend on the Snyk Security Team, which actively works to ensure that the Snyk Vulnerability Database contains the most up-to-date vulnerabilities and is trying to reduce false positive rates [31]. Therefore, when the dataset was created, the vulnerabilities and dependencies values were primarily sourced from the Snyk database.

Regarding static analysis, yes, it is possible to use dynamic analysis. However, the intention is to detect the images before running them to avoid potential harm to applications built on top of the running versions of images - containers. This research is focused on static analysis as the main interest of the work. The dynamic version involves similar procedures to state-of-the-art work, applying machine learning to container workflows or pods, such as dealing with system calls or log datasets.

D. MODEL PERFORMANCE ANALYSIS

In our experiments, the Neural Network (NN) model did not perform as well as the Tree-based classifiers, achieving a lower F1 score, precision, and recall. This outcome can be attributed to several factors:

- Architecture Complexity: The shallow architecture of the NN may have been insufficient to capture the sophisticated relationships between features. Compared to tree-based models like Random Forest, which is better at handling structured data like Docker image features.
- Hyperparameter Tuning: Unlike RF, which is less sensitive to hyperparameters, the NN requires careful tuning. The initial NN model used in this study was not deeply optimized, which likely contributed to its lower performance.
- Imbalanced Data: Neural Networks struggle with class imbalances unless addressed through data augmentation, class weighting, or resampling techniques. The RF classifier's robustness to such issues made it a better fit for this dataset.

```
// Define a pipeline that pulls Docker images, scans for vulnerabilities, and deploys if secure
pipeline {
    agent any
    stages {
        stage('Pull Docker Image') {
            steps {
                bat 'docker pull nginx:latest'
                bat 'docker pull mysql:latest'
                bat 'docker pull node:latest'
            }
        }
        stage('Install Docker SDK') {
            steps {
                // Install the Docker SDK for Python to interact with Docker
                bat 'pip install docker --verbose'
            }
        }
        stage('Scan for Vulnerabilities') {
            steps {
                script {

                    def nginxScanResult = bat(script: 'python scan_docker_image.py nginx', returnStdout: true)
                        .trim()
                    def mysqlScanResult = bat(script: 'python scan_docker_image.py mysql', returnStdout: true)
                        .trim()
                    def nodeScanResult = bat(script: 'python scan_docker_image.py node', returnStdout: true).
                        trim()

                    // Output the scan results
                    echo "Nginx Scan Result: ${nginxScanResult}"
                    echo "MySQL Scan Result: ${mysqlScanResult}"
                    echo "Node.js Scan Result: ${nodeScanResult}"
                }
            }
        }
        stage('Deploy if Secure') {
            when {
                expression { nginxScanResult == 'Secure' && mysqlScanResult == 'Secure' && nodeScanResult == '
                    Secure' }
            }
            steps {

                echo "All images are secure. Proceeding with deployment..."
                bat 'docker-compose -f docker-compose.yml up -d'
                bat 'docker ps'
                echo "Services are up and running."
            }
        }

        stage('Halt if Insecure') {
            when {
                expression { nginxScanResult != 'Secure' || mysqlScanResult != 'Secure' || nodeScanResult != '
                    Secure' }
            }
            steps {

                error "One or more images are insecure. Halting the pipeline."
            }
        }
    }
}
```

FIGURE 15. Jenkins pipeline used in the deployment process.

Future work could explore more sophisticated NN architectures, hyperparameter optimization strategies, and techniques for handling class imbalances better in the NN model.

VIII. STRENGTHS AND LIMITATIONS

Recognizing that every new idea in any discipline has strengths and limitations is essential. It is the same for the prediction of malicious Docker images. The strength of this idea lies in having the first work that applies machine-learning algorithms to predict malicious images using Docker image features. It serves as a basis for all previous efforts by other researchers focused on containers. This thesis addresses the detection step before creating containers. From this perspective, this step works as an authentication before creating the upper layer (containers). However, a notable limitation is dataset collection, which happens by writing code that iterates over specific Docker images falling into three categories: official, verified, and sponsor OS images. Some images may not exist in these three types; users might create images without categorizing them (create and push to Docker Hub). Consequently, addressing all Docker images in the Docker Hub repository can take time and effort. The

dataset is currently limited to 778 images, and expansion is necessary to include more diverse Docker image features' values.

IX. CONCLUSION AND FUTURE WORK

This paper has proposed a machine model to assess the feasibility of employing machine learning algorithms for detecting the security status of Docker images available in the Docker Hub repository. The contribution of this work comes in two main ideas: the first one involves creating a new dataset containing Docker images associated with 14 corresponding features, and the second one is training the machine learning techniques using Docker image features. The results have shown that the Random Forest classifier demonstrates exceptional accuracy, achieving a 99% F score and an AUC of 100%. This performance refers to its capability to accurately classify the images and effectively distinguish between secure and insecure images, in addition to the minimal error rate of less than 1%. The main benefit of this work goes to the Docker repository users, providing them with an efficient way to assess the security of a pulled Docker image before the running process. This prevents malicious behaviors in the containers and the applications that will build on top of these images. Furthermore, the proposed model has been compared to several state-of-the-art models and the comparisons have shown that the proposed approach achieved a competitive F1-score and outperformed other models. Moreover, the new dataset has been published online for research purposes.

In future work, we plan to explore additional features that play a significant role in constructing Docker images, which may lead to improved predictive models. Moreover, we plan to increase the size of the Docker images dataset, given the growing usage of Docker repositories and the need for developers to enhance image layers and other features. Consequently, considering updated versions of Docker images is essential. Furthermore, we plan to consider containers and their associated applications for each image. This approach could yield better predictions for Docker images and provide a more extensive dataset for training deep learning algorithms.

REFERENCES

[1] Dockerhub. (2024). *Docker Hub Repositry*. [Online]. Available: https://hub.docker.com/

[2] A. Mouat, *Using Docker: Developing and Deploying Software With Containers*. Sebastopol, CA, USA: O'Reilly Media, 2015.

[3] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, p. 2, Mar. 2014.

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

[5] Quay. (2021). *Clair*. [Online]. Available: https://github.com/quay/clair

[6] V. Laurikainen, "Securing container-based environments with Anchore," Bachelor's thesis, Dept. Inf. Commun. Technol., JAMK Univ. Appl. Sci., 2022.

[7] Aqua. (2021). *Aqua Microscanner*. [Online]. Available: https://www.aquasec.com/news/microscanner-new-free-image-vulnerability-scanner-for-developers/

[8] Snyk. (2024). *Snyk Tool*. [Online]. Available: https://www.snyk.io/

[9] Avi. (Nov. 2022). *Understanding Docker for Beginners—The Container Technology*. [Online]. Available: https://geekflare.com/understanding-docker-for-beginner/

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

[11] O. Javed and S. Toor, "An efficient threat detection framework for Docker containers using AppArmor profile and clair vulnerability scanning tool," Universita della svizzera italiana, Lugano, Switzerland, Tech. Rep., 2022, p. 19.

[12] A. El Khairi, M. Caselli, C. Knierim, A. Peter, and A. Continella, "Contextualizing system calls in containers for anomaly-based intrusion detection," in *Proc. Cloud Comput. Secur. Workshop*, New York, NY, USA, Nov. 2022, pp. 9–21, doi: 10.1145/3560810.3564266.

[13] R. R. Karn, P. Kudva, H. Huang, S. Suneja, and I. M. Elfadel, "Cryptomining detection in container clouds using system calls and explainable machine learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 674–691, Mar. 2021.

[14] Kubernetes. (2024). *Kubernetes Website*. [Online]. Available: https://kubernetes.io/

[15] C. Tien, T. Huang, C. Tien, T. Huang, and S. Kuo, "KubAnomaly: Anomaly detection for the Docker orchestration platform with neural network approaches," *Eng. Rep.*, vol. 1, no. 5, Dec. 2019, Art. no. e12080.

[16] O. Tunde-Onadele, J. He, T. Dai, and X. Gu, "A study on container vulnerability exploit detection," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, Jun. 2019, pp. 121–127.

[17] M. U. Haque and M. A. Babar, "Well begun is half done: An empirical study of exploitability & impact of base-image vulnerabilities," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2022, pp. 1066–1077.

[18] A. Brogi, D. Neri, and J. Soldani, "DockerFinder: Multi-attribute search of Docker images," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, Apr. 2017, pp. 273–278.

[19] V. Jain, B. Singh, M. Khenwar, and M. Sharma, "Static vulnerability analysis of Docker images," *IOP Conf. Ser., Mater. Sci. Eng.*, vol. 1131, no. 1, Apr. 2021, Art. no. 012018.

[20] D. Huang, H. Cui, S. Wen, and C. Huang, "Security analysis and threats detection techniques on Docker container," in *Proc. IEEE 5th Int. Conf. Comput. Commun. (ICCC)*, Dec. 2019, pp. 1214–1220.

[21] H. Cui, D. Huang, Y. Fang, L. Liu, and C. Huang, "Webshell detection based on random forest–gradient boosting decision tree algorithm," in *Proc. IEEE 3rd Int. Conf. Data Sci. Cyberspace (DSC)*, Jun. 2018, pp. 153–160.

[22] S. Kwon and J.-H. Lee, "DIVDS: Docker image vulnerability diagnostic system," *IEEE Access*, vol. 8, pp. 42666–42673, 2020.

[23] Clair. (2024). *Clair Documentation*. [Online]. Available: https://github.com/quay/clair/tree/master/Documentation

[24] J. Pinnamaneni, S. Nagasundari, and P. Honnavalli, "Identifying vulnerabilities in Docker image code using ML techniques," in *Proc. 2nd Asian Conf. Innov. Technol. (ASIANCON)*, Aug. 2022, pp. 1–5. [Online]. Available: https://api.semanticscholar.org/CorpusID:252850963

[25] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, "Emerging trends, techniques and open issues of containerization: A review," *IEEE Access*, vol. 7, pp. 152443–152472, 2019.

[26] M. Soori, B. Arezoo, and R. Dastres, "Artificial intelligence, machine learning and deep learning in advanced robotics, a review," *Cognit. Robot.*, vol. 3, pp. 54–70, Jan. 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2667241323000113

[27] S. Brown, "Machine learning, explained," MIT Sloan School Manage., Cambridge, MA, USA, 2021, [Online]. Available: https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained

[28] H. Dalianis, "Evaluation metrics and evaluation," in *Clinical Text Mining*. Cham, Switzerland: Springer, 2018, pp. 45–53, doi: 10.1007/978-3-319-78503-5_6.

[29] J.-O. Palacio-Niño and F. Berzal, "Evaluation metrics for unsupervised learning algorithms," 2019, *arXiv:1905.05667*.

[30] S. Narkhede, "Understanding AUC-ROC curve," *Towards Data Sci.*, vol. 26, no. 1, pp. 220–227, 2018.

[31] SnykDatabase. (Nov. 2023). *Snyk Vulnerability Database*. [Online]. Available: https://docs.snyk.io/scan-using-snyk/snyk-open-source/manage-vulnerabilities/snyk-vulnerability-database/

MARAM ALDIABAT received the B.Sc. degree in software engineering and the M.Sc. degree in information systems from Jordan University of Science and Technology, Irbid, Jordan. She is currently pursuing the Ph.D. degree in software engineering with Auburn University, Auburn, AL, USA. Her research focuses on enhancing software practices using machine learning techniques.

QUSSAI M. YASEEN received the B.Sc. degree in computer science from Yarmouk University, in 2002, the M.Sc. degree in computer science from Jordan University of Science and Technology, in 2006, and the Ph.D. degree in computer science from the University of Arkansas, Fayetteville, AR, USA, in 2012. He developed new approaches for mitigating insider threat in relational databases with the University of Arkansas. After completed the Ph.D. degree in 2012, he was with Yarmouk University, from 2012 to 2014. From 2014 to 2021, he was with Jordan University of Science and Technology. Currently, he is an Associate Professor with Ajman University, United Arab Emirates. He has received funds from different institutions to tackle different cybersecurity issues. He has published several papers in refereed journals and conferences. He served as the chair/the track-chair/an organizer/a TPC member for cybersecurity and information technology conferences and workshops.

QUSAI ABU EIN received the master's and Ph.D. degrees from Ibaraki University, Japan. He is an Associate Professor with the Department of Computer Information Systems, Jordan University of Science and Technology. His research interests include web social analysis, web analytics, data analysis, and information retrieval.