ISSN: 2182-2069 / E-ISSN: 2182-2077

Understanding the Effectiveness of SBOM Generation Tools for Manually Installed Packages in Docker Containers

Nobutaka Kawaguchi[1*], Charles Hart[2], and Hiroki Uchiyama[3]

[1*]Security & Trust Research Department, Hitachi, ltd., - Tokyo, Japan.
nobutaka.kawaguchi.ue@hitachi.com, https://orcid.org/0000-0003-0213-4785

[2]Security Innovation Lab., Hitachi America, ltd., - CA, USA.
charlie.hart@hal.hitachi.com, https://orcid.org/0009-0007-5250-5624

[3]Security Innovation Lab., Hitachi America, ltd., - CA, USA.
hiroki.uchiyama@hal.hitachi.com, https://orcid.org/0009-0000-6044-3135

Received: March 11, 2024; Revised: May 13, 2024; Accepted: June 15, 2024; Published: August 30, 2024

Abstract

Software Bill of Materials (SBOM), which is a standardized format for the machine-readable list of components included in software, is a key technology for addressing software supply chain attacks. Since Docker containers, now prevalent for software distribution and deployment, typically consists of hundreds of packages, the use of automation tools to generate their SBOMs is recommended. Currently, several OSS-based SBOM generation tools are available, playing indispensable roles in automating SBOM utilization. Generally, the tools make use of information from several package managers and databases of popular software to create SBOMs from the container images. On the other hand, some Docker containers include packages that were manually downloaded and installed by the authors without the package managers. Despite this, few studies have been conducted on how pervasive manually installed packages are and how accurate SBOM generation tools are in identifying them. To investigate the issue, we collected 3500+ popular Docker container images from the Docker Hub and assessed the accuracy of the SBOMs generated by two prominent OSS tools. The result showed 3000+ manual installations of 800+ packages that are either downloaded with Linux commands or copied directly from host systems. 51% of the containers included one or more manually installed packages. We found that SBOM tools can overlook 30-70% of the installations, which include both recent and outdated versions of major software and many niche or specialized tools. In addition, at least 27.7% of the manually installed packages were executed or read using the default settings of the Docker containers, and neither of the tool identified 22.7% of them, including those with known CVE vulnerabilities. Finally, the results revealed that at least 1.1% of the installations are overlooked by the generators, although actively used and associated with known vulnerabilities.

Keywords: SBOM, Vulnerability Management, Docker.

1 Introduction

The recent increase in cyberattacks targeting software supply chains, such as SolarWinds (US Government Accountability Office, 2021), (Kasya - CISA, 2021a), and (log4j - CISA, 2021b) demonstrates the need for software transparency because these attacks exploit vulnerable components

Journal of Internet Services and Information Security (JISIS), volume: 14, number: 3 (August), pp. 191-212.
DOI: *10.58346/JISIS.2024.I3.011*

*Corresponding author: Security & Trust Research Department, Hitachi, ltd., - Tokyo, Japan.

unknowingly included in the software used in the victim organizations. The software Bill of Materials (SBOM) (United States Department of Commerce, 2021), which is a kind of machine-readable ingredient list of software, is the key technology for accurately checking whether any known vulnerable components have been mixed in the software through the software supply chain. With strong governmental initiatives, SBOM has become prevalent in several industries.

Because software may contain hundreds of components (e.g., packages) and the installed software does not always come with its SBOM, the automated creation of SBOM is essential for software users as wells as developers (Sundara Bala Murugan et al., 2024). Accordingly, several SBOM generators that inspect the target software and create their SBOM have been developed and widely utilized (Arora et al., 2022). However, the accuracy of the SBOM by the tools has not been extensively investigated or discussed. The generators usually rely on package managers (both OS package managers such as apt-get for Debian OS and application managers such as pip for python) to collect software information in a system (Nadgowada & Luan, 2021). Accordingly, it is uncertain whether they can also identify packages installed manually without the package managers.

The use of SBOM tools is essential, especially when handling intricate software composed of myriad layered components (Viticchié et al., 2018). Software containers are among the bulkiest software categories, as they typically include several applications, their dependent libraries, and OS-related files. Given the prevalence of containers today, it is a feasible research question to ask how effectively SBOM tools automate the generation of SBOMs from containers.

Thus, this study focuses on the accuracy of the SBOM of Docker containers created by two prominent open-source SBOM generators. Dockers are among the most popular platforms for software distribution, and they suffer from several vulnerability issues (Liu et al., 2020). We collected more than 3500 popular Docker containers from a prominent repository and analyzed how accurately the SBOM generators identified packages that were installed manually using download commands (e.g., wget and curl) or the COPY operation during the Docker image-build process (Surendar et al., 2024).

As a result, we found that more than 50% of the containers include one or more manually installed packages. Our analysis reveals that even a generator equipped with a package signature database overlooks 30% of the installation of these packages, some of which have known vulnerabilities, while the other generator, which seems to depend on the package managers solely, works worse, overlooking more than 70% of the installation. We also found that at least 27.7% of the manually installed packages were executed or read when the containers were run without any arguments or with dummy arguments forged by generative AI, and 22.7% of their existences were overlooked by the generators. Because actively used packages (i.e., packages read or executed when the container runs) are more susceptible to exploits than dormant ones, overlooking of these packages could impose critical risks to the container users. Moreover, the experiments revealed that at least 1.1% of all installations are overlooked, although actively used and associated with one or more CVEs.

The major contributions of this paper are as follows:

1. This is the first study regarding the performance of SBOM generators to create SBOMs of manually installed packages in Docker containers using a large-scale dataset.
2. The analysis results revealed the accuracy of the SBOM generators with the five viewpoints; (i) overlooked rate for manually installation packages, (ii) overlooked rate for containers with different number of the packages, (iii) overlooked rate for the actively used packages, (iv) high risk installations of software associated with one or more CVEs, (v) the difference of performance in the two prominent generators.

3. This study demonstrates the usefulness of generative AI in supporting the SBOM analysis; extraction of manually installed packages, and forging of dummy arguments.

To the best of our knowledge, this is the first study to evaluate the accuracy of prominent SBOM generators for manually installed packages in Docker containers with a large-scale dataset, and to uncover the hidden risks. This research outcome will help both software suppliers and consumers understand the limitations of the current tools and their wise usage.

The remainder of this paper is organized as follows. Section 2 introduces the background of the study. Section 3 explains the analysis pipeline, and Section 4 presents the results. Sections 5 and 6 discuss the recommendations for Docker container developers and users and the limitations of our work, respectively. Section 7 discusses the related works, and Section 8 concludes the paper.

2 Background

In contemporary digital landscapes, the software supply chain is increasingly vulnerable to several threats, particularly through continuous integration and deployment (CI/CD) pipelines (Ladisa et al., 2023). Accordingly, various frameworks and guidelines have been published to render CI/CD pipelines more robust to threats (NIST, 2015; NIST, 2021; Linux Foundation, 2022a). SBOM is the core technology in these efforts. SBOM is a machine-readable format that specifies packages and their attributes (e.g., version, supplier name, dependency, license) included in software (United States Department of Commerce, 2021). SBOM allows both software suppliers and consumers to understand what consists of their applications and systems in a more transparent manner than ever before, which will, in turn, mitigate the fatigue of security operators in identifying and prioritizing vulnerabilities in their environments (Smale et al., 2023; Hamed et al., 2023). While it is not a mandatory option, SBOM usually includes CPEs (NIST, n.d.a) and Purls (Package-url, n.d.): identifiers for packages and versions, which makes it easy to discover vulnerable packages in software through matching with vulnerability databases, such as NVD (NIST, n.d.b).

The CISA has identified six types of SBOM to cover the software lifecycle spanning both suppliers and consumers (e.g., Build SBOM on the supplier side, Runtime SBOM on the consumer side) (CISA, 2023a). The use of SBOM has been strongly supported by some governments. Currently, Executive Order 14028 (US Government, 2021) in the U.S. and the Cyber Resilience Act (CRA) (EU, n.d.) in the EU request software suppliers to provide consumers with their SBOMs. The adaptation of SBOM is also underway in several sectors such as healthcare, automotive, energy, IT, and finance (CISA, 2023b; Linux Foundation, 2022b).

There are several issues with the effective use of SBOM, including its management and privacy (Xia, 2023). One of the primary issues is how to create SBOM in the first place. Although there is no restrictive format specification for SBOM, some SBOM formats, such as SPDX (Linux Foundation, 2023), CycloneDX (OWASP, 2023), and SWID (NIST, 2018), have been widely used. Several open-source and commercialized SBOM generators have been developed to support these formats (Arora et al., 2022; Anchor, n.d.a; Aquasecurity n.d.; Snyk, n.d.). On the other hand, however, it is still uncertain how precise and accurate the SBOMs created by the generators are. While previous studies have revealed that some generators may overlook a substantial fraction of software components (Balliu et al., 2023; Heiderup & Plate, 2023), their research coverage is limited to a small number of software projects.

This study focuses on the accuracy of SBOM for a wide range of Docker containers. Docker is a containerization technology for building, distributing, and running software applications (Kim et al.,

2023). While Docker containers are becoming increasingly prevalent in the enterprise systems (Haque & Babar 2022), they occasionally include vulnerable packages that have been exploited by cyber-attacks (Liu et al., 2020). Currently, several SBOM generators can generate SBOMs for packages installed in the container images. However, they often rely on package managers to collect the packages inside the images (Nadgowada & Luan, 2021). Thus, these tools may overlook packages installed with file download commands instead of package managers. For example, package managers for C/C++ are less prevalent compared to more modern languages (Tang et al., 2022), which might result in the manual installation of binary codes written in C/C++. Despite this, there have been few works regarding this research area. For example, although the work (Torres-Arias et al., 2023) compares the primary statistics of SBOMs of Docker containers generated by different OSS tools, it lacks the deep analysis on the overlooked packages by those tools.

3 Analysis Methodology

Manually Installed Packages

This section describes the analysis methodology used to evaluate the accuracy of open-source SBOM generators in creating SBOMs for packages manually installed in Docker containers.

As mentioned before, manually installed packages are referred to as those that are installed in the container environment without using OS package managers (e.g., *apt-get install, apk add, yum install*) or application managers (e.g., *pip install, npm install*) when their images are built. The installation can be performed by either downloading archives from specific URLs using Linux commands (e.g., wget, curl, git) or the COPY operation specified in the Docker files.

As mentioned in (Nadgowada & Luan, 2021), most SBOM generators are primarily dependent on package managers; therefore, it may seem challenging for them to identify manually installed packages accurately. However, even when a package is copied without using package managers, the archive files often leave artifacts specific to the managers (e.g., package.json for npm and requirement.txt for pip), which may help generators identify the package names and versions. In addition, copied or downloaded packages can be registered with managers (e.g., *dpkg -i xxx.dpkg*). Moreover, some generators are equipped with signature databases for major software applications (Anchor, 2022). Thus, it is not obvious whether a manually installed package can be identified using a given SBOM generator.

We also put special focus on manually installed packages that are either executed or read when the containers are launched. As the introduction of Runtime SBOM (CISA, 2021a) indicates, such ‘active’ packages require more attention than dormant packages because they can impose more risks if attackers try to exploit their vulnerabilities. One issue for monitoring the containers activities is that some containers require specific parameters to be set to work as originally intended. In this experiment we monitor the behavior of the containers in their ‘default’ state with no parameters. If the execution terminates with an error, we feed dummy parameters forged by the Generative AI to expand the analysis coverage. We consider that this approach provides a primary overview of the risks of manually installed and actively used packages.

Analysis Pipeline

Figure 1 shows the pipeline used in the analysis. First, it downloads container images from the Docker Hub (Docker Hub, n.d.), the most popular Docker repository. The downloaded container images

undergo the extraction of their manually installed packages, SBOM generation by two SBOM generators, and monitoring of the executed/read packages. The pipeline then identifies manually installed packages that either of the generators (i.e., SG-X and SG-Y) overlooked by checking their absence in the SBOMs. Additionally, the pipeline runs the containers and monitors packages that are executed or read during the run. Then, the overlooked packages are matched with the monitored logs and vulnerability databases to identify high-risk packages: those that are overlooked, actively used, and vulnerable.

We designed the pipeline for handling thousands of containers in an automated manner. However, owing to the lack of global and comprehensive identifiers for software packages (CISA, 2023c), this process is not error-free. Thus, we manually verified the results and corrected mismatches after the pipeline completes. The verification process includes searches on GitHub and freeware hosting sites and the analysis of Docker images and containers. In particular, we made a lot of effort to avoid underestimating the performance of the SBOM generators. This labor-intensive task required more than 100 hours of working time.

Note that the aim of this research is not to evaluate exactly how the strings of package names from generators conform to CPE or Purl. Instead, we focus on measuring how accurately the generators identify the existence of each manually installed package inside the containers while allowing some inconsistencies in the naming convention.

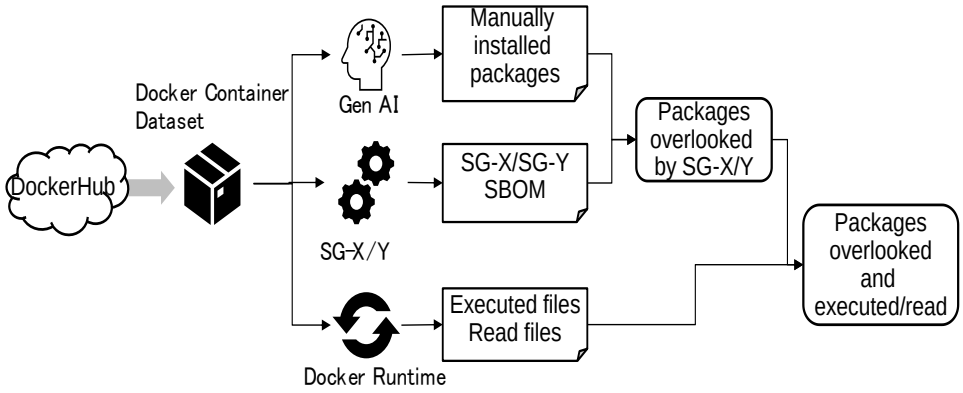


Figure 1: Analysis Pipeline of Manually Installed Packages from Docker Hub

Dataset

We first crawled the Docker Hub using the 16 keywords shown in List 1 to create a list of 5,000 top-pulled container images in Nov., 2023. The keywords were chosen to cover popular web applications, software development environments, and security tools. The pulling count for an image is the sum of all the tags. Then, the pipeline downloaded container images that had ‘:latest’ tags in Nov. and Dec. 2023. The dataset included 3,514 images. The most pulled images were Python, Redis, Postgres, Node.js, and Httpd. According to their manifests, the images have been created between 2014 and 2023; and 34% of them are in 2023. Further details are provided in the following sections.

List 1: Keywords for Docker Hub Search

cache, language, messaging, runtime, cd, load_balancer, monitoring, security, ci, logging, network, storage, content_management, orchestration, web_server, database, message_broker, proxy

Extraction of Manually Installed Packages

The pipeline extracts manually installed packages from the output of the ‘docker history’ command, which shows the configuration history of an image based on the metadata of the image layers. Figure 2 shows an example of this command. In this case, the wget manually downloads the Redis package from a URL specified by environmental parameters.

Although there are several clues to identifying the names and the versions of the packages, it is not easy to design one-size-fits-all rules to extract them from various container images. Thus, the pipeline uses generative AI to extract manually installed packages. More specifically, it feeds OpenAI (OpenAI, n.d.) the history data and environmental parameters from Docker image metadata and then obtains a list of names, versions, and directory paths of the manually installed packages. List 2 shows the query prompt and List 3 is an example of response about the Redis container. Directory paths were used for the human verification process, as previously mentioned. As shown in List 3, a container can include more than one package installed manually.

When package names and versions are embedded in the download URLs or environmental values, OpenAI generally works well. But it can fail to identify package versions when the URLs only include ambiguous words (e.g., ‘latest,’ ‘stable’) to express the version information.

```
<missing>                                                                 6 week
s ago   RUN /bin/sh -c set -eux;   savedAptMark="$(apt-mark showmanual)";  apt-g
et update;  apt-get install -y --no-install-recommends   ca-certificates   wget
    dpkg-dev   gcc   libc6-dev   libssl-dev   make   ;  rm -rf /var/lib/apt/lists
/*;   wget -O redis.tar.gz "$REDIS_DOWNLOAD_URL";  echo "$REDIS_DOWNLOAD_SHA *re
dis.tar.gz" | sha256sum -c -;  mkdir -p /usr/src/redis;  tar -xzf redis.tar.gz -
C /usr/src/redis --strip-components=1;  rm redis.tar.gz;   grep -E '^ *createBoo
lConfig[()]"protected-mode",.*, *1 *,.*[)],$' /usr/src/redis/src/config.c;  sed -
ri 's!^( *createBoolConfig[()]"protected-mode",.*, *)1( *,.*[)],)$!\10\2!' /usr/s
rc/redis/src/config.c;  grep -E '^ *createBoolConfig[()]"protected-mode",.*, *0 *
```

Figure 2: A Sample of Manually Installed Packages Installations

List 2: Query to Extract Manually Installed Packages

```
Below are (i) environmental values from ‘docker inspect' and (ii) history from 'docker history' command.
Could you please answer the packages that were installed without using package managers?
  ----------
(i) Environmental values from 'docker inspect': {env}
(ii) image history from 'docker history': {history}
  ----------
Return the answer as a JSON Object. If there is no package installed without using package managers, just return []
The answer can include more than one element.
Each element in the answer has the following attributes.
 (1)'PackageName’, name of a package installed without using package managers,
 (2)'PackageVersion', the version of the package,
 (3)'DirectoryPath', the directory where the package is finally copied.

Examples of expected answers:
[
      {{"PackageName":"abc","PackageVersion":2.1,"DirectoryPath":"/usr/lib/abc/"}},
      {{"PackageName":"Dix","PackageVersion":4.1,"DirectoryPath":"/usr/local/bin/"}}
]
Note that, such packages are usually downloaded from the Internet using 'wget', 'curl' or other commands.
```

List 3: Reply from Generative AI for Query about Manually Installed Packages in Redis

```
[
  {"PackageName":"gosu","PackageVersion":"1.17",
   "DirectoryPath":"/usr/local/bin/"},
  {"PackageName":"redis","PackageVersion":"7.2.4",
   "DirectoryPath":"/usr/local/bin/"}
]
```

SBOM Generation

In this study, we used two open-source SBOM generators to create SBOMs for the downloaded containers, denoted as SG-X and SG-Y, respectively. Both are prominent tools that receive more than 5k stars on GitHub. To avoid unexpected consequences, we will not disclose their names in this paper. We use their latest versions as of Nov. 2023. Both the tools support several SBOM formats, and we used SPDX for this evaluation. In SPDX, a package name and the version are described in 'PackageName' and 'PackageVersion' attributes. We use the tools with basic options, without customizing their configuration files. It is known that SG-X is equipped with a signature database of popular packages. The signature database is maintained by SG-X developers, and the users get the update by installing the latest version.

Execution of Docker Containers

In this phase, the pipeline executes the downloaded containers with 'docker run' command, and their executed or read files are monitored by the pipeline using eBPF (eBPF, n.d.). Some containers immediately stop running if the required parameters are missing, resulting in low coverage of actively used packages. However, it will require a prohibitive amount of time to manually check and craft parameters when handling thousands of containers.

To streamline the analysis while increasing the monitoring coverage, we make use of the observation that the error message from a container often includes instructions to run it correctly. The monitoring process uses a two-step approach. First, a container is run without any parameters. If the container crashes with an error message, the pipeline feeds the message to Generative AI (we again use OpenAI) to forge dummy parameter values based on the message. The container was then run again with the forged parameters. Details of this approach are described in (Kawaguchi & Hart, 2024).

Identifying Overlooked Packages

When a manually installed package does not match any components included in the corresponding SBOM, the package is considered overlooked by the generator. This process involves both package name matching and package version matching as follows.

(1) Package Name Matching

The pipeline performs matching between two package names *n1* and *n2*, one from the manually installed packages and the other from SBOM. First, it divides *n1* and *n2* into tokens according to certain delimiters ('/', '-, ' '_') as they are sometimes used to concatenate software names and other information such as developer names. It then determines that *n1* and *n2* possibly point to the same package if the case-insensitive normalized Levenshtein distance (Yujian & Bo, 2007) between any tokens and the original tokens from the two strings is shorter a threshold (we set it to 0.2).

(2) Package Version Matching

In contrast to package names, the pipeline uses exact numerical matching for the package versions. If version strings of both an extracted manually installed package and an entry in the SBOM follow the semantic versioning (Preston-Werner T., n.d.), the pipeline only checks the version core (i.e., major, minor, and patch versions) for comparison, because responses from the generative AI often do not include further information. If either of the version strings does not follow the semantic versioning, the pipeline performs string matchings while ignoring substrings after '-' or '_' in the strings.

Given a manually installed package *P*, the pipeline concludes that *P* is a 'overlooked package' by an SBOM generator if the corresponding SBOM does not include an entry that matches its name and version. If the SBOM includes one or more entries matched with *P*, *P* is regarded as a 'matched package'.

(3) Manual Verification and Correction

As mentioned before, we validated both 'overlooked packages' and 'matched packages' by hand and corrected the results if needed. As to the package name, we dealt with synonyms such as 'java' and 'jdk', and several unexpected inconsistencies such as 'aliyun-python-sdk-iot' and 'aliyunsdkiot_py2.'

Regarding the package version, the outcomes of manually installed package extraction sometimes lack valid version strings as mentioned above. In this case, we first try to identify the exact version by searching the internet. If this attempt fails, we presume that the versions from the generators were correct, and the decisions are made based only on the package name. In addition, if the 'PackageVersion' of a package in SBOMs is expressed with the git source control version (CVE Project, n.d.) instead of the semantic versioning, we check the GitHub repository to address the corresponding semantic versioning if applicable.

Identifying Executed/Read Packages

The pipeline employs a heuristic approach that uses clues from file paths and container image metadata to identify the name and version of a package for a read or executed file. A detailed discussion is provided in (Kawaguchi & Hart, 2024). The matching approach between the executed/read packages and overlooked packages is the same as that in the previous section.

4 Analysis Results

This section elaborates on the analysis results for the 3,514 container images from the Docker Hub.

Manually Installed Packages

Out of all downloaded containers, the pipeline identified that 1,799 (51.2%) containers included one or more manually installed packages. The total number of extracted installations was 3,327 over 876 unique package names with different versions.

Our validation by hand found that the responses from the generative AI for the 779 installations lacked valid version strings, and we then resorted to manual inspection, as mentioned before. Moreover, we found 45 installations inappropriate for evaluation use; we were not able to identify exact package names for some of them, and the others are those that were uninstalled during the

container build process so that SBOM generators did not identify them. These 45 entries are excluded from the further evaluation.

Figure 3 shows the distribution of the number of packages manually installed in the container. Approximately 85% of the containers had only one or two packages. By contrast, the four containers included more than 15 packages. This distribution has long-tailed properties.

Figure 4 shows the most frequently installed package names across the different versions. The top four packages (Node.js, Yarn, Python, and OpenJDK) accounted for 29.6% of all the installations. Six out of the top 10 packages are relevant to program runtime environments, and they earn more than 19k starts on GitHub. This suggests that while most OS package managers support such fundamental software, container producers still tend to install them manually from source codes. On the other hand, 88.5% of the packages were installed only three or fewer times for more than 3000 containers. This long-tail property indicates the difficulty of identifying manually installed packages. In addition, we found that manual installation was prevalent even with major packages. Regarding Node.js, our dataset included at least 576 installations in total, and 320 of which are manually installed ones, whereas the others (=256) were through OS package managers, such as apt.

The installed versions varied among the same package names. Figure 5 shows the major version distributions of Node.js, OpenJDK and Redis. The different versions of Node.js were widely distributed between the ver.0 and 21.0. On the other hand, OpenJDK installations were concentrated in specific versions: 8u, 11, and 17. Regarding Redis, newer versions had more installations than earlier ones.

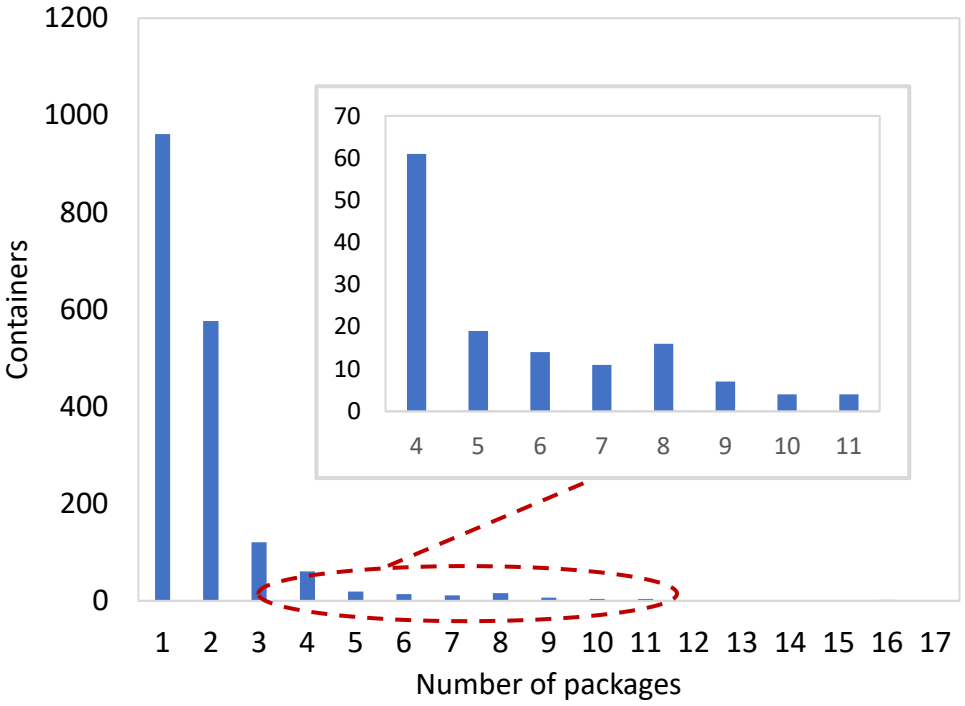


Figure 3: Distribution of the Number of Manually Installed Packages

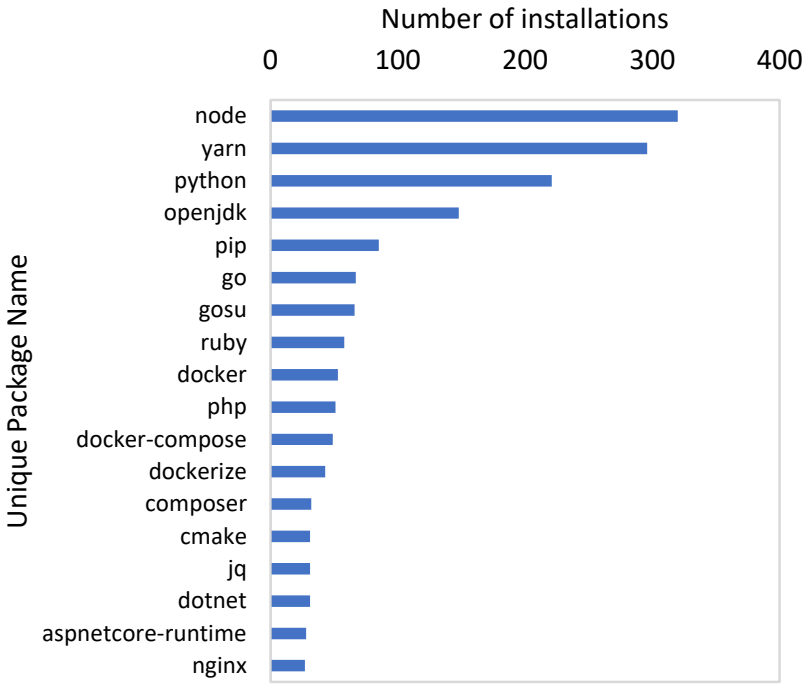


Figure 4: Installation Ranking of Manually Installed Package Names

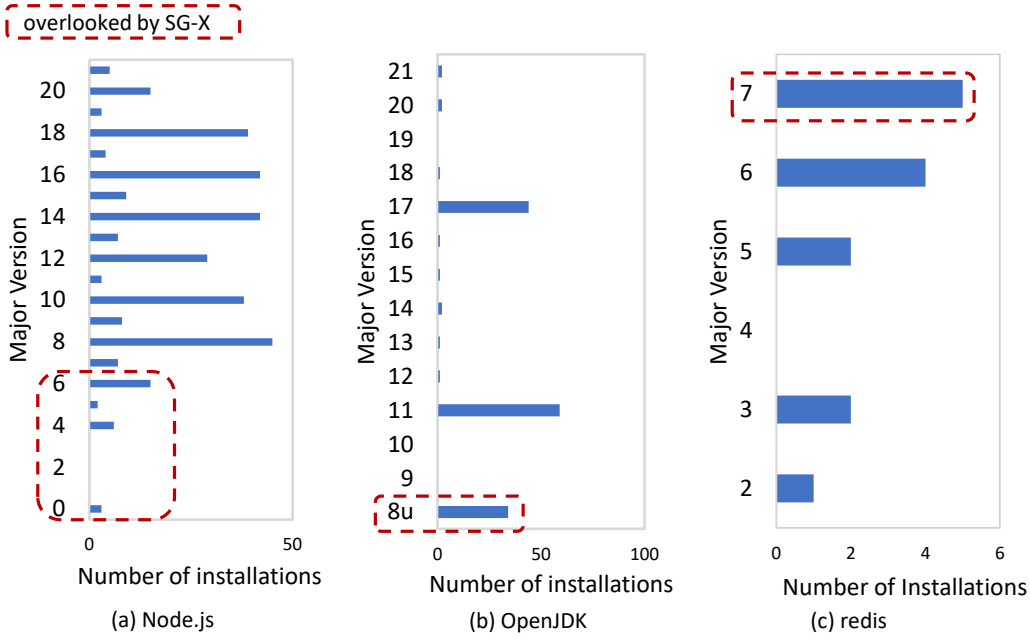


Figure 5: Version Variations of Manually Installed Package Names

Overlooked Packages

Table 1 lists the identification results of the two SBOM generators. Since the generators encountered technical issues while analyzing some container images, the total numbers here are different from each other. SG-X identified 60% of the installations of manually installed packages, whereas SG-Y's identification rate was 28%. The two generators overlooked 37% and 72% of the manual installations

respectively. The results included those whose package versions were incorrectly identified. For some packages, the generators successfully identified their names but returned ‘unknown’ for their versions. The following statistics in this paper classified these cases into ‘overlooked’ categories. These results show that the identification performance can vary between the generators. SG-X has signature databases for major software, which contributes to expanding the coverage, whereas SG-Y seems to depend on OS/application package managers. When paying attention to the number of unique package names, they overlooked more than 50% of the unique packages, and their gap was not as wide as that of overlooked installations. This indicates the signature database in SG-X focuses on a small number of major software that account for many installations.

Figure 6 shows the unique packages that were overlooked the most by the two generators. SG-X failed to identify gosu, the Go language implementation of the sudo command. It also overlooked certain docker-related packages. Notably, as Figure 5 shows, some early versions of Node.js ($\leq$ 6.0) and OpenJDK (8u) were not identified. SG-X relies on signature databases, and it seems that signature-based discovery for the early versions is not as thorough as that for the latest versions, although some containers keep using them. Also, SG-X failed to identify the latest versions of some prominent packages. For example, it overlooked ver.7.2.3 of Redis, which was the latest version at the time of this experiment, while it identified versions earlier than 7.0.0. In addition, the plot of the cumulative ratio shows the long-tailed property. Notably, 438 overlooked unique package names, which accounts for 88%, were installed only three or less times for the 3500+ containers. This unreliable identification results demonstrate the challenges signature-based approaches face. Additionally, SG-X failed to identify the dotnet runtime and aspnetcore packages for Linux.

On the other hand, SG-Y, which relies more strongly on package managers, was significantly less accurate than SG-X. It failed to identify all the versions of major software (e.g., Node.js, python3 runtime, OpenJDK, go runtime, and ruby runtime). Although the results seem to discourage the use of SG-Y, it still worked better than SG-X for certain packages. For instance, SG-Y successfully identified the dot-net runtime and aspnetcore overlooked by SG-X. We discuss the details later. These results indicate that neither of the generators completely outperforms the other, and it is important to wisely use both of them depending on the situation to deal with manually installed packages.

The comparison between Figure 4 and Figure 6 shows that while SG-X identified eight out of the top 10 manually installed packages, SG-Y identified only two of them. These results corroborate that signature databases are still effective in covering the major software.

We also found that most overlooked packages came from C/C++ binaries, Erlang (e.g., otp), and PHP scripts (e.g., php extensions of Memcached and Redis). On the other hand, packages written in modern languages such as Java and Python were identified even when they are installed manually. For example, SBOM generators successfully identified package names and versions of Java libraries by analyzing their metadata in .jar files. In addition, we found that most Go-lang binaries other than gosu were successfully identified because the package information is embedded in the binaries.

Some packages change their main implementation languages at major version updates, which affects identification accuracy. For example, Docker-compose changed the main language from Python to Go when it transitioned from v1 and v2. While SG-X overlooks their v1 versions, it successfully identifies their v2 versions by extracting package information from Go-lang binaries.

Figure 7 shows the ratio of containers where all the manually installed packages were identified by the generators. As expected, this ratio generally decreases as the containers have more packages. On the other hand, in SG-X, the ratio graph reached its peak of 0.67 when the number of packages is two.

We found that this is attributed to frequently appearing combinations of major packages such as Nodejs and Yarn. Both packages are relevant to JavaScript environments and are frequently installed concurrently, and SG-X is able to identify both. On the other hand, the identification ratio of SG-Y for containers with two packages is only 0.10. These results indicate that the gap in the performance of SBOM completeness can widen when handling containers with multiple manually installed packages.

Figure 8 shows the changes in the ratio of containers that include one or more manually installed packages and the ratio of those that include any package(s) overlooked by both generators according to the containers' creation dates. Each bin corresponds to a period of six months. More than one-third of the pulled containers come from the year 2023, while the dataset also includes those of 10 years old. As this figure shows, none of the two ratios shows significant changes over time. The results are contrary to our initial expectation that as cyber security awareness increases, software packages must be increasingly maintained through package managers to improve their transparency. One plausible reason for the counter intuitive result is that OS package managers, which should offer adequate application/library packages for a specific OS, do not cover many relatively niche applications. As mentioned before, 88% of all manually installed packages only appear three or fewer times for 3500 containers, and it would be difficult for the OS package managers to support all of them. This will also hold true for the signature database of SBOM generators.

Table 2 shows a comparison between SG-X and SG-Y for containers that both generators successfully create the SBOMs. Combining the two SBOM generators can increase the coverage for manually installed packages by 3.4 % compared to the sole use of SG-X. While the improvement could be considered marginal, we found that SG-Y can identify aspnetcore and dotnet packages better than SG-X. For example, for aspnetcore runtime ver. 3.1.20, SG-X outputs 'PackageName: Microsoft.AspNetCore' and 'PackageVersion: 3.100.2021.474242', while SG-Y outputs 'PackageName: Microsoft.AspNetCore.App.Runtime.linux-x64', and 'PackageVersion: 3.1.20'. This package name from SG-X is a deprecated one and only applicable for versions less than 2.28. In addition, the version representation from SG-X resembles a build number, and it does not match the common representation for aspnetcore runtime versions (GitHub Advisory Database, 2022). Thus, although the output from SG-X could be an approximate representation of the package, it would not work well for vulnerability matching.

Table 1: Identification Performance of SBOM Generators. The Numbers in Parentheses show the Ratio to the Total Number of Installations for each SBOM Generator

	SG-X	**SG-Y**
Installation whose package name and version are correctly identified	1880(60%)	908(28%)
Installation whose package version is 'unknown'	103(3%)	6(0%)
Overlooked Installations (including those whose package versions are incorrect while the package names are correct)	1175(37%)	2286(72%)
Unique package names for overlooked installations	497	591

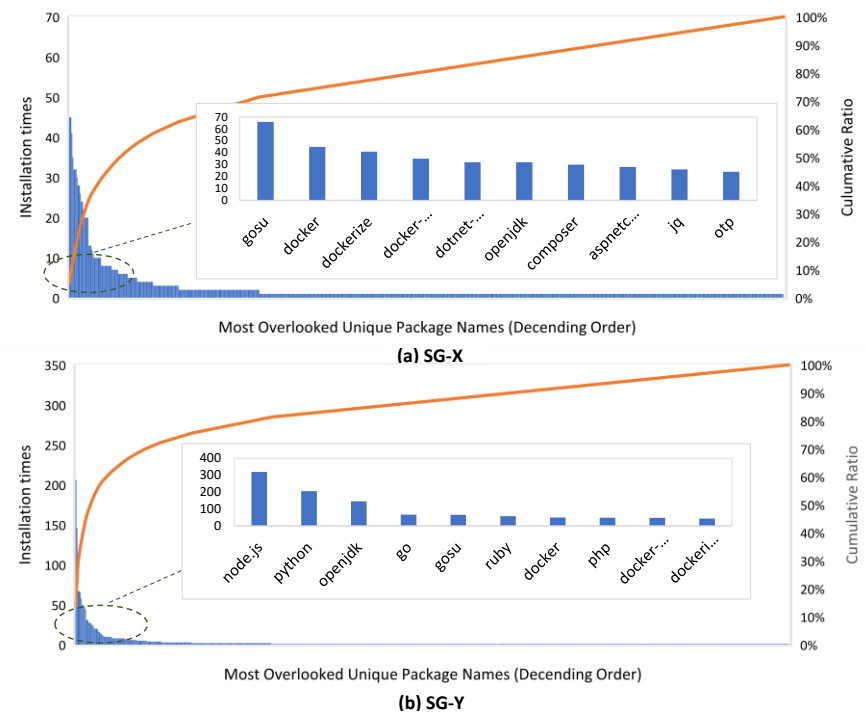


Figure 6: Most Overlooked Unique Package Names

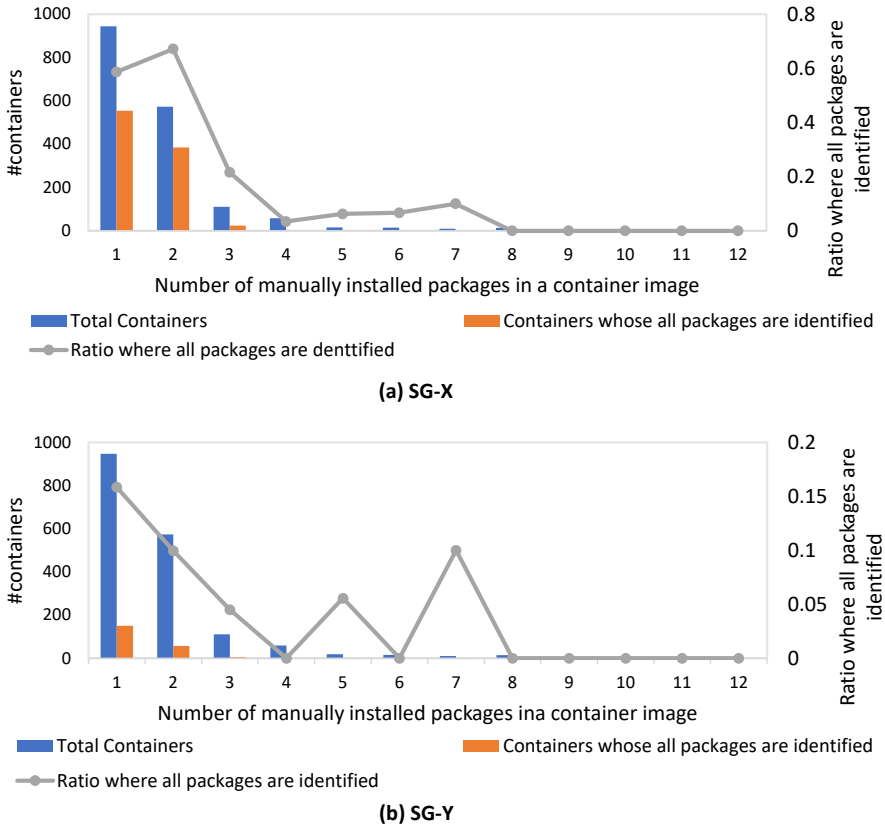


Figure 7: Ratio of Containers whose All Manually Installed Packages are Identified

Table 2: Comparison between two SBOM Generators

	SG-Y identified	SG-Y overlooked
SG-X identified	839	1040
SG-X overlooked	64	1213

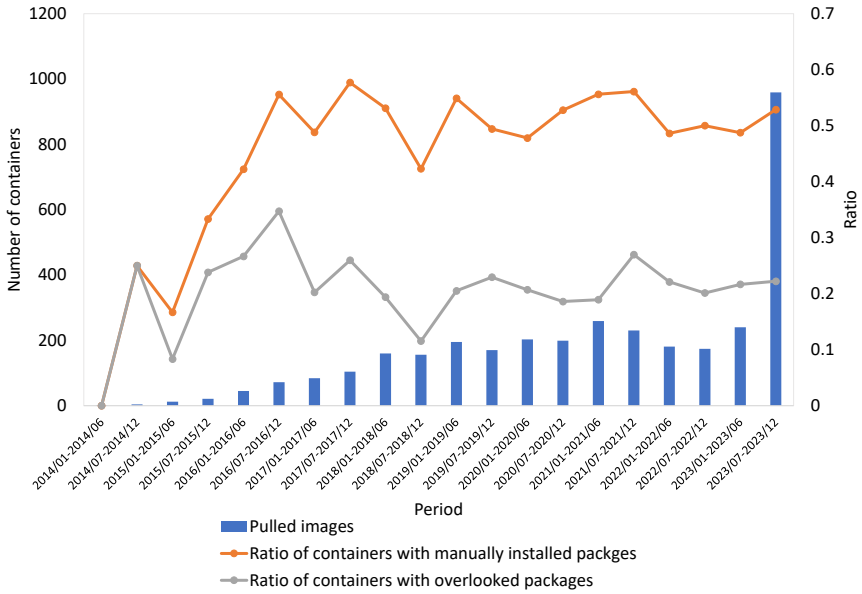


Figure 8: The Ratio of Containers with Manually Installed Packages and Overlooked Packages by both Generators Overtime between 2014 and 2023

Overlooked and Executed / Read Packages

The monitoring process in the pipeline reveals that 27.7% (i.e. 923 out of the 3327 cases) of manually installed packages are either read or executed during the experiment described in Sec. 3., which accounts for 227 out of 867 unique package names. Then, 22.7% of the installations of such active packages were overlooked by both generators.

Figure 9 (a) shows the ranking of manually installed packages based on their executed or read times. In addition, Figure 9 (b) shows the packages in (a) that are also overlooked by both generators. Note that even when a package was executed or read more than once in a container, it was just counted once. From the viewpoint of software transparency, the packages listed in Figure 9 (b) will impose high risks because they become active when containers are launched while they do not appear in the SBOMs. While if a package becomes active or not can depend on specific parameter values, those that are read or executed with default or forged parameters will be considered active with most configuration settings. Table 3 lists some known vulnerabilities in the packages shown in Figure 9(b). This includes both major and relatively niche software in the dataset, and the total number of the installations is 37, accounting for 1.1% of the manually installed packages, or the 4.0% of the actively used packages.

With the two-step approach mentioned in Sec. 3., 238 containers were run twice where the second run was with parameters forged by generative AI based on error messages from the first run. We found that for the 23 containers, the second run yielded more execution and read operations than the first run. Eleven of the 23 containers are relevant to database systems (e.g., Postgres, MariaDB, and CouchDB),

and setting dummy passwords (e.g., ‘mypassword’) and the authentication methods makes them run longer, revealing more packages than the first run.

We admit that there could be more vulnerable, active, and overlooked packages in the dataset than we were able to recognize in this experiment. While we did not successfully run all containers with adequate parameters and Table 3 could include more vulnerable packages, the results still demonstrate that a substantial number of containers with active and overlooked vulnerable packages are downloadable on the Docker Hub.

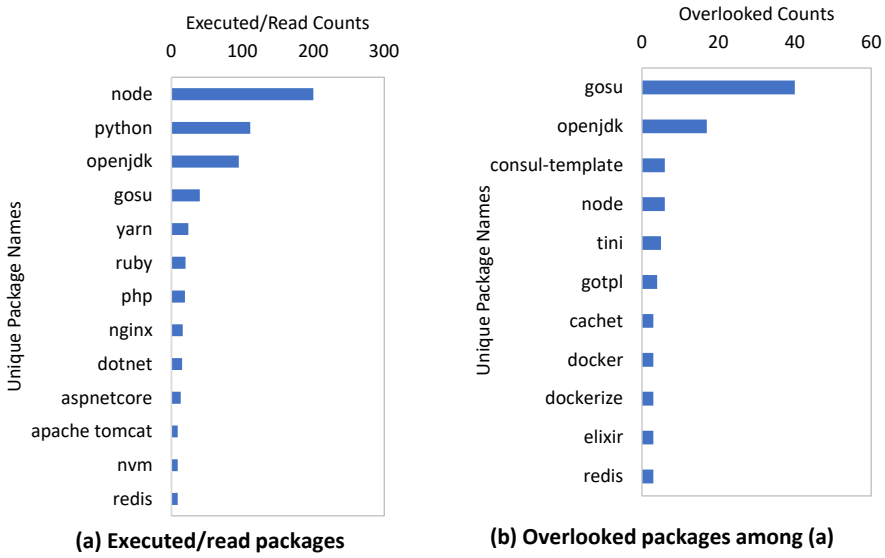


Figure 9: Active and Overlooked Packages

Table 3: Vulnerable Packages that are Overlooked by SBOM Generators and Become Active While Containers were Run with Default or Dummy Parameters

#	Package name	version	# of installations in the dataset	Exemplar vulnerabilities.
1	cachet	2.3.13	3	CVE-2023-43661
2	consul-template	0.12.2	6	CVE-2022-38149
3	fluent-bit	0.12.1	1	CVE-2020-35963
4	grafana	7.2.1	1	CVE-2023-3128
5	logstash-oss	7.10.2	1	CVE-2021-22138
6	nats-server	2.2.6	2	CVE-2022-24450
7	node.js	4.9.1	1	CVE-2018-7158
8	openjdk	8u312	17	CVE-2023-21967
9	owncloud	10.0.10	1	CVE-2021-35846
10	prometheus	1.8.2	2	CVE-2019-3826
11	rabbitmq	3.10.6	1	CVE-2023-46118
12	wordpress	4.9.1	1	CVE-2021-44223

Accuracy of Extraction of the Manually Installed Packages Using Generative AI

Of the 3327 installations extracted by the approach in List 2, their 779 occurrence lack version information, and 45 correspond to either of them; (1) the packages were deleted during the container build process (2) the installed files were actually not packages (e.g., a short bash script written by the

container author, a configuration file). Thus, the precision of this approach was 98.6% for package names, whereas 23.4% of the results required human intervention for the versions.

Regarding recall, it is difficult to accurately evaluate the score owing to the lack of ground truth in our dataset. However, it can be said that our approach covers more packages than conventional ones (Majumder et al., 2023; Doan & Jung et al., 2022) which depend on CVEBinTool (intel, n.d.), a signature-based package extractor. For example, CVEB in Tool deals with only two of the 12 packages listed in Table 3. As of March 2024, CVEBinTool covers 355 unique packages with known vulnerabilities, and requires a dedicated setting for each package. In this regard, our approach of using generative AI outperformed this tool in terms of scalability and coverage. The combination of our approach and the tool could improve both precision and recall, which will be the focus of our future work.

5 Recommendation for Container Producers and Users

While the objective of SBOM is to cover all packages in the software, the results demonstrate the challenge of obtaining a precise SBOM for Docker containers. SBOMs can be generated by container image producers and users, and there are recommendations for both sides.

For Container Producers

It would be a good practice for container image producers to use package managers as much as possible to make the SBOM more accurate. As mentioned in (Tang et al., 2022), package managers have not been frequently used for developments with C/C++; however, some modern C/C++ managers such as conan (JFrog, n.d.) and vcpkg (Microsoft, n.d.) can work with SBOM generators. Although Figure 8 shows that the use of package managers has not become as prevalent as expected, we have also observed some progresses recently. For example, while a series of containers for software development from CircleCI (e.g., circleci/redis from Docker Hub) manually installed gosu using a wget, their successor series (e.g., cimg/redis from Docker Hub (CircleCI, n.d.)) have switched to using apt-get for gosu installation, which makes their SBOMs more accurate.

In addition, when producers generate SBOMs by themselves, they should accurately check packages that must be installed according to the dockerfile included in them before delivery to the users. Some tools are available for support the verification (Tap8stry, n.d.).

For Container Users

Compared with producers, container users have fewer options for their actions. Nevertheless, it is a good practice to check whether the corresponding SBOM for a container includes information on packages that should be the core function. For example, if the container is used for using Redis server but the SBOM does not include the Redis package, which could be a warning sign.

6 Limitation of this Work

The containers in our dataset are all from a single repository, the Docker Hub. While the Docker Hub is the largest container repository, the trends in manually installed packages could differ in other repositories, such as Quay.io (Quay.io, n.d.a) depending on their hosting policies.

Most SBOM generators are still in their premature stages and are evolving gradually. While we use their snapshots as of November 2023, the result can be changed by updates thereafter. In particular, package signature updates could have a substantial impact on the analysis results.

As mentioned before, we spent a lot of time confirming and correcting the results from our pipelines, and the analysis is still not fully automated. Owing to the inconsistencies in package name conventions and the lack of universal package identifiers (CISA, 2023c), matching between SBOM and vulnerability databases remains a challenging problem.

Finally, our analysis does not mention the false positives of the SBOM generators (Mayers, 2023). The precision and accuracy of SBOMs can be a tradeoff, which will be a part of our future work.

7 Related Work

There have been few studies on the evaluation of SBOM generators. Some researchers evaluated the precision and accuracy of SBOMs for a few Java-based projects by comparing the ground truths from Maven (Balliu et al., 2023; Heiderup & Plate et al., 2023). The results show both the accuracy and precision can be lower than 50%. Santiago Torres-Arias compared SBOMs generated by two SBOM generators using a dataset of approximately 700 popular containers (Torres-Arias et al., 2023). The results show substantial differences in the number of packages between SBOMs from different generators. These studies are focused on specific software projects or primary statistics of SBOMs, and did not study the identification accuracy of manually installed packages for a large-scale dataset. Note that the two SBOM generators in our study also appeared in some of these studies.

Arushi Arora et al. conducted qualitative studies on several open-source SBOM tools based on taxonomies and functionalities (Arora et al., 2022). The SBOM Benchmark (Interlynk, n.d.) offers quality scores of SBOM from different angles, including compliance with guidelines from NTIA (NTIA, 2021) and completeness of license descriptions. Although our research does not focus on license issues, license management is another major objective of SBOM, which we will address in a future work. Some studies have investigated the prevalence and adaptation issues in creating SBOMs in open-source projects and enterprise software (Kanemoto et al., 2023; BI et al., 2023; Stalnaker et al., 2024). They have revealed that many software developers struggle to ensure the transparency and completeness of SBOMs, and the qualities can differ between projects and employed languages. In addition, understanding the contents of a large SBOM is difficult for human developers. Jones et al. proposed a visualization technique to identify changes and differences in SBOMs using a series of software (Jones & Tate, 2023). Additionally, for human operators, the usability of a tool can effectively handle vulnerable packages (Kim et al., 2023).

As many Docker containers suffer from vulnerabilities (SlimAI, 2022), several Docker Image Vulnerability Scanners (DIVSes) have been developed (Anchor, n.d.b; Aqua security, n.d., Quay, n.d.b) to inspect containers and detect vulnerable components. These tools are relevant to SBOM generators, and some can detect vulnerabilities and generate SBOMs. Some studies (Haque & Babar 2022; Malhotra et al., 2023) combined DIVSes to detect vulnerabilities in popular Docker images, such as base, official, and verified images. However, similar to SBOM generators, DIVSes are not free from overlooked vulnerabilities owing to the lack of complete vulnerability databases and the inability to list all components in images. According to the study (Javed & Toor, 2021), even the best DIVS can miss 35% of known vulnerabilities in an image. Some studies combined several DIVSes and CVEBinTool (Intel, n.d.) to increase the detection coverage of vulnerabilities (Majumder et al., 2023; Doan & Jung 2022). In particular, (Doan & Jung 2022) evaluated the effectiveness of CVEB in Tool in

identifying known vulnerable packages installed manually using a wget and curl during the container build process. In this regard, the work may look similar to ours, but our work covers a wider range of manually installed packages regardless of the existence of known vulnerabilities by using generative AI, including the analysis of active packages.

Table 4 shows the comparison of our work with related existing research introduced above. As this table indicates our work handles more Docker container datasets than existing works, and especially focuses on how accurately the latest SBOM tools handle the manually installed packages, some of which include vulnerabilities. In this regard, our work outperforms existing literature.

Table 4: Comparison of our Work with Existing Research

Work	**Analysis Target**	**Number of Datasets**	**Accuracy Analysis of SBOMs from different tools**	**Focus on manually installed packages**	**Analysis of vulnerabilities in Docker containers based on scanning tools (i.e., DIVSes, SBOM tools)**
(Balliu et al., 2023)	Java projects	<10	Yes	No	No
(Torres-Arias et al., 2023)	Docker Container	700	Limited (only the number of components is compared)	No	No
(Javed & Toor, 2021)	Docker Container	59	No	No	Yes
(Doan & Jung 2022)	Docker Container	10 for thorough analysis	No	Yes	Yes
Our work	Docker Container	3514	Yes	Yes	Yes

8 Conclusion

This study evaluated two prominent SBOM generators with respect to the accuracy of their SBOMs, with a particular focus on manually installed packages in more than 3500 Docker containers. The results indicate that several issues need to be addressed, including the handling of the latest and early versions of a few prominent tools, myriad niche tools, and certain languages. Most significantly, the results suggest the generators could overlook 30-70% of all the manual installations. On the other hand, the combination of multiple SBOM generators will be beneficial to expand the coverage. We also demonstrated that generative AI can support the analysis of manually installed packages better than existing tools. Understanding the current status quo is the first step towards realizing a secure software supply chain.

The discussion in this paper is a snapshot with a dataset from a certain Docker repository and SBOM generators that were popular at the time of the experiments. Since the generators have been evolving as authorities recommend the use of SBOMs and more Docker producers are aware of the security issues, this trend can change over time. On the other hand, our package analysis in this research depends on the capability of Generative AI. The recent advances in AI technologies will enables us to analyze packages in containers more accurately and extensively over time, which may result in revealing more potential risks than we are currently aware of.

Thus, we will continue this research with more SBOM generators and more intricate execution settings of Docker containers to further clarify and prioritize the risks from overlooking manually installed packages and other SBOM-relevant issues.

References

[1] Anchor (2002). Detecting binary artifacts with Syft, https://anchore.com/blog/detecting-binary-artifacts-with-syft/
[2] Anchor (n.d.a). Syft. https://github.com/anchore/syft
[3] Anchor (n.d.b). Grype. https://github.com/anchore/grype
[4] Aquasecurity (n.d.). Trivy. https://github.com/aquasecurity/trivy
[5] Arora, A., Wright, V. L., & Garman, C. (2022). *SoK: A Framework for and Analysis of Software Bill of Materials Tools* (No. INL/JOU-22-68388-Rev000). Idaho National Laboratory (INL), Idaho Falls, ID (United States).
[6] Balliu, M., Baudry, B., Bobadilla, S., Ekstedt, M., Monperrus, M., Ron, J., & Wittlinger, M. (2023). Challenges of producing software bill of materials for java. *IEEE Security & Privacy*. 12-23.
[7] Bi, T., Xia, B., Xing, Z., Lu, Q., & Zhu, L. (2024). On the way to sboms: Investigating design issues and solutions in practice. *ACM Transactions on Software Engineering and Methodology*, *33*(6), 1-25.
[8] CircleCI (n.d.), cimg/redis. https://hub.docker.com/r/cimg/redis/tags.
[9] CISA (2021a). Kaseya Ransomware Attack: Guidance for Affected MSPs and their Customers. https://www.cisa.gov/uscert/kaseya-ransomware-attack
[10] CISA (2021b). CISA Issues Emergency Directive Requiring Federal agencies to mitigate apache log4j vulnerabilities. https://www.cisa.gov/news/2021/12/17/cisa-issues-emergency-directive-requiring-federal-agencies-mitigate-apache-log4j
[11] CISA (2023a). Types of Software Bill of Materials (SBOM). https://www.cisa.gov/sites/default/files/2023-04/sbom-types-document-508c.pdf
[12] CISA (2023b). SBOM-a-rama. https://www.cisa.gov/news-events/events/sbom-rama
[13] CISA (2023c). Software Identification Ecosystem Option Analysis, 2023. https://www.cisa.gov/resources-tools/resources/software-identification-ecosystem-option-analysis
[14] CVE Project (n.d.). CVE5.0 Product and Version Encodings. https://github.com/CVEProject/cve-schema/blob/master
[15] Doan, T. P., & Jung, S. (2022). DAVS: Docker file Analysis for Container Image Vulnerability Scanning. *Computers, Materials & Continua*, *72*(1), 1699-1711.
[16] Docker Hub (n.d.). Docker Hub Web Page. https://hub.docker.com/
[17] eBPF (n.d.). eBPF Homepage. https://ebpf.io/
[18] EU (n.d.). The European Cyber Resilience Act. https://www.european-cyber-resilience-act.com/
[19] GitHub Advisory Database (2022). GHSA-vh55-786g-wjwj .NET Information Disclosure Vulnerability. https://github.com/advisories/GHSA-vh55-786g-wjwj
[20] Hamed, A. S. E., Elbakry, H. M., Riad, A. E., & Moawad, R. A (2023). Proposed Technical Debt Management Approach Applied on Software Projects in Egypt. *Journal of Internet Services and Information Security*, *13*(3), 156-177.
[21] Haque, M. U., & Babar, M. A. (2022). Well begun is half done: An empirical study of exploitability & impact of base-image vulnerabilities. *In IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 1066-1077.
[22] Heiderup, J., & Plate, H. (2023). *In SBOMs We Trust: How Accurate, Complete, and Actionable Are They?*. FOSDEM2023.
[23] Intel (n.d.). cve-bin-tool. https://github.com/intel/cve-bin-tool

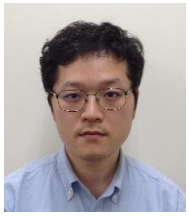
[24] Interlynk (n.d.). SBOM Benchmark. https://sbombenchmark.dev
[25] Javed, O., & Toor, S. (2021). *Understanding the Quality of Container Security Vulnerability Detection Tools*. arXiv.2021.03844v1[cs.CR].
[26] JFrog (n.d.). Conan C/C++ Package Manager. https://conan.io
[27] Jones R., Tate L. (2023). *Visualizing Comparisons of Bills of Materials*. arXiv:2309.11620v1[cs.HC].
[28] Kanemoto Y., Arakawa R., Akiyama M. A Study of Software Transparency Assessment Based on a Large-Scale Survey of SBOM. *In Proceedings of IPSJ Computer Security Symposium* 2023.
[29] Kawaguchi, N., & Hart, C. (2024). On the Deployment Control and Runtime Monitoring of Containers Based on Consumer Side SBOMs. *In IEEE 21st Consumer Communications & Networking Conference (CCNC)*, 1022–1025. https://doi.org/10.1109/CCNC51664.2024.10454654
[30] Kim, T., Park, S., & Kim, H. (2023). Why Johnny Can't Use Secure Docker Images: Investigating the Usability Challenges in Using Docker Image Vulnerability Scanners through Heuristic Evaluation. *In Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23)*. Association for Computing Machinery, New York, NY, USA, 669–685. https://doi.org/10.1145/3607199.3607244
[31] Ladisa, P., Plate, H., Martinez, M., & Barais, O. (2023). SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. *In IEEE Symposium on Security and Privacy (SP)*, 1509-1526. https://doi.org/10.1109/SP46215.2023.10179304.
[32] Linux Foundation (2022a). Supply-chain Levels for Software Artifacts (SLSA). https://slsa.dev/
[33] Linux Foundation (2022b). The State of Software Bill of Materials (SBOM) and Cyber security Readiness. https://www.linuxfoundation.org/research/the-state-of-software-bill-of-materials-sbom-and-cybersecurity-readines
[34] Linux Foundation (2023). System Package Data Exchange (SPDX), https://spdx.dev/
[35] Liu, P., Ji, S., Fu, L., Lu, K., Zhang, X., Lee, W. H., & Beyah, R. (2020). Understanding the security risks of docker hub. *In Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020*, 257-276. https://doi.org/10.1007/978-3-030-58951-6_13
[36] Majumder, S. H., Jajodia, S., Majumdar, S., & Hossain, M. S. (2023). Layered Security Analysis for Container Images: Expanding Lightweight Pre-Deployment Scanning. *In IEEE 20th Annual International Conference on Privacy, Security and Trust (PST)*. https://doi.org/10.1109/PST58708.2023.10320152
[37] Malhotra, R., Bansal, A., and Kesseniti, M. (2023). Vulnerability Analysis of Docker Hub Official Images and Verified Images. *In IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 150-155. https://doi.org/10.1109/SOSE58276.2023.00025.
[38] Mayers J.S. (2023). A purl of wisdom on SBOMs and vulnerabilities. https://www.chainguard.dev/unchained/a-purl-of-wisdom-on-sboms-and-vulnerabilities.
[39] Microsoft (n.d.). Vcpkg. https://vcpkg.io/en/
[40] Nadgowada, S., & Luan, L. (2021). Orion: Go Beyond Package Manager Discovery for Your SBOM. https://thenewstack.io/orion-go-beyond-package-manager-discovery-for-your-sbom/
[41] NIST (2018). Software Identification (SWID) Tagging. https://csrc.nist.gov/projects/software-identification-swid/guidelines, 2018.
[42] NIST (2022). Secure Software Development Framework (SSDF). https://csrc.nist.gov/Projects/ssdf, 2022.
[43] NIST (n.d.a). Official Platform Enumeration (CPE) Dictionary. https://nvd.nist.gov/products/cpe
[44] NIST (n.d.b). National Vulnerability Database. https://nvd.nist.gov/vuln/search

[45] NIST, U. (2015). Special Publication 800-161 Supply Chain Risk Management Practices for Federal Information Systems and Organizations. https://csrc.nist.gov/publications/detail/sp/800-161/rev-1/final
[46] NTIA (2021). Framing Software Component Transparency: Establishing a Common Software Bill of Materials. https://www.ntia.gov/sites/default/files/publications/ntia_sbom_framing_2nd_edition_20211021_0.pdf.
[47] OpenAI (n.d.). OpenAI Web Page. https://openai.com/
[48] OWASP (2023). CycloneDX. https://cyclonedx.org/
[49] Package-url (n.d.). purl-spec. https://github.com/package-url/purl-spec
[50] Preston-Werner T. (n.d.). Semantic Versioning 2.0.0. https://semver.org
[51] Quay.io (n.d.a). Quay.io web page. https://quay.io
[52] Quay.io (n.d.b). clair. https://github.com/quay/clair
[53] SlimAI (2022). Public Container Report. https://www.slim.ai/blog/container-report-2022.
[54] Smale, S., Dijk, R., Bouwman, X., Ham, J., & Eeten, M. (2023). No One Drinks from the Firehose: How Organizations Filter and Prioritize Vulnerability Information. *In IEEE Security and Privacy*, 1980-1996. https://doi.org/10.1109/SP46215.2023.10179447.
[55] Snyk (n.d.). Snyk-CLI. https://docs.snyk.io/snyk-cli/commands/sbom
[56] Stalnaker, T., Wintersgill, N., Chaparro, O., Penta, M. D., German, D.M., Poshyvanyk, D. (2024). BOMs Away! Inside the Minds of Stakeholders: A Comprehensive Study of Bills of Materials for Software Systems. *In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)* 1–13. https://doi.org/10.1145/3597503.3623347
[57] Sundara Bala Murugan, P., Arasuraja, G., Prabakaran, P., & Aruna, V. (2024). Feasibility Design and Analysis of Process-aware Accounting Information System for Business Management. *Indian Journal of Information Sources and Services, 14*(2), 56–62. https://doi.org/10.51983/ijiss-2024.14.2.09
[58] Surendar, A., Saravanakumar, V., Sindhu, S., & Arvinth, N. (2024). A Bibliometric Study of Publication- Citations in a Range of Journal Articles. *Indian Journal of Information Sources and Services, 14*(2), 97–103. https://doi.org/10.51983/ijiss-2024.14.2.14
[59] Tang, W., Xu, Z., Liu, C., Yang, S., Li, Y., Luo, P., & Liu, Y. (2022). Towards Understanding Third-party Library Dependency in C/C++ Ecosystem. arXiv:2209.02575v2[cs.SE], 2022.
[60] Tap8stry (n.d.). orion. https://github.com/tap8stry/orion?utm_source=thenewstack&utm_medium=website&utm_content=inline-mention&utm_campaign=platform
[61] Torres-Arias, S., Geer, D., & Meyers, J. (2023). A Viewpoint on Knowing Software: Bill of Materials Quality When You See It. *IEEE Security & Privacy, 21*(6), 50-54. https://doi.org/10.1109/MSEC.2023.3315887
[62] United States Department of Commerce (2021). The Minimum Elements for a Software Bill of Materials (SBOM). https://www.ntia.doc.gov/files/ntia/publications/sbom_minimum_elements_report.pdf
[63] US Government (2021). Executive Order (EO) 14028, Improving the Nations Cyber Security. https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity, 2021.
[64] US Government Accountability Office (2021). SolarWinds Cyberattack Demands Significant Federal and Private-Sector Response (infographic). https://www.gao.gov/blog/solarwinds-cyberattack-demands-significant-federal-and-private-sector-response-infographic
[65] Viticchié, A., Basile, C., Valenza, F., & Lioy, A. (2018). On the impossibility of effectively using likely-invariants for software attestation purposes. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, 9*(2), 1-25.

[66] Xia, B., Bi, T., Xing, Z., Lu, Q., & Zhu, L. (2023). An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead. *In Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. https://doi.org/10.1109/ICSE48619.2023.00219

[67] Yujian, L., & Bo, L. (2007). A normalized Levenshtein distance metric. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *29*(6), 1091-1095.

Authors Biography



Nobutaka Kawaguchi was born in Kanagawa, Japan in 1980. He received the B.S. and M.S degrees in information technology from Keio University in 2003, 2005, respectively, and received the Ph. D degree from the graduate school of Keio University in 2008.
He joined Hitachi ltd. in 2008, and he has been engaged in the research and development of cyber security and information security technologies. His current research topics include malware analysis, advanced persistent threats, software supply chain security.
He is a member of the IEEE, IPSJ. CISSP.

Charles Hart is a researcher at Hitachi America Ltd. specializing in industrial and product cybersecurity technology policy.
He formerly served in executive positions at Hitachi Data Systems, Savvis, Symantec, Veritas, Storage Networks, and MFS Investment Management, including several senior cybersecurity and software management roles. He was an early proponent of software bill of materials, software supply chain integrity, and product security generally, and has served on several related bodies including the NTIA Software Transparency Project, CISA working groups, Automotive ISAC, and the Industrial Internet Consortium. He has authored or co-authored many publications and he recently completed technical editing for the book “Software Supply Chain Security” by Cassie Crossley (O’Reilly Media, 2024).
He is also a frequent contributor to standards bodies including ISO/ANSI, IETF, OASIS, and SAE as a software supply chain and product security expert. Mr. Hart is a graduate of Boston College.

Hiroki Uchiyama was born in Osaka, Japan in 1978. He received the B.S. and M.S. degree in Informatics from the Kyoto University, Japan, in 2003 and the Ph.D. degree in Informatics from the Kyoto University, Japan, in 2012.
He joined Hitachi, Ltd. in 2003. From 2023 to present, he was seconded to Hitachi America, Ltd. and assigned to lab manager with Security Innovation Laboratory. He has been engaged in security operation technology especially for vulnerability management.
He is a member of IPSJ and IEEJ. CISSP.