Received July 20, 2020, accepted August 22, 2020, date of publication September 2, 2020, date of current version September 16, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3021044

Container Image Access Control Architecture to Protect Applications

SUNG-HWA HAN[1], (Member, IEEE), HOO-KI LEE[2], (Member, IEEE), SUNG-TAEK LEE[3], (Member, IEEE), SUNG-JIN KIM[4], (Member, IEEE), AND WON-JUNG JANG[5], (Member, IEEE)

[1]Department of ITPM, Soongsil University, Seoul 06978, South Korea
[2]Department of Cyber Security Engineering, Konyang University, Nonsan, South Korea
[3]Department of Computer Science, Yongin University, Kyeonggi, South Korea
[4]Department of Intelligent Systems Engineering, Cheju Halla University, Cheju 63092, South Korea
[5]Department of Intellectual Property for Startups, Catholic Kwandong University, Gangneung, South Korea

Corresponding author: Won-Jung Jang (wjjang@cku.ac.kr)

ABSTRACT A container platform allows various applications to be deployed or run after installation. A user can download or execute a container image with the required application. To apply the configuration management system, a container image uses a union filesystem composed of multiple layers. To provide stability, important application files must be protected from unauthorized access. However, the container image used for distributing an application does not have its own protection function, and it is not protected by the container platform. The access control function provided by the operating system cannot protect the applications because the container environment is not considered. In this study, a container image access control architecture is proposed that can ensure a safe application operating environment by denying unauthorized direct access to container images. The proposed architecture enforces the access control function after the container image is downloaded, denying unauthorized access to the container image layer directory. Because the access control function is provided at the kernel level, there is a security advantage that users cannot bypass. To verify this approach, the functions and performance were determined empirically according to the proposed architecture. Functional verification confirmed that the proposed architecture denies unauthorized access to the container base image and allows access only to authorized users. It was also confirmed that the proposed architecture ensures the performance of the container platform in the same way as before, and that the proposed container image access control architecture is sufficiently effective.

INDEX TERMS Access control, container, image protection, layered image, container image.

I. INTRODUCTION

A container platform provides various functions necessary for information service building and operation, and its usage rate is increasing because it can reduce system costs. If an application required for the provisioning of an information service is created and deployed as a container image, the user of the container can execute the application simply by running the container image without a complicated installation process [1]. Containers share the kernel of the host system, and thus an application running on a container shares the resources of the host system [2]. In this way, the information service based on a container platform increases the resource efficiency of the system [3].

A container platform provides an extremely limited interface. Users can download, create, upload, execute, and monitor container images through this container platform interface [4]. A container platform used for application distribution and execution is responsible for the security of the container image that contains the application. However, the current container platform provides protection against application execution [5], but does not provide protection against container images stored in the host system. Many operating systems (OSs) provide SecureOS, allowing a denial of unauthorized access to protect important files or directories. However because the current version of SecureOS does not consider the container platform, it cannot protect dynamically executed applications in a container image.

When the container operation environment is summarized, the downloaded container image is stored in the container image repository, but the platform provides no protection function for the container image stored therein. Therefore, it can be said that the security of the container image within the container platform, which primarily/mainly serves

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

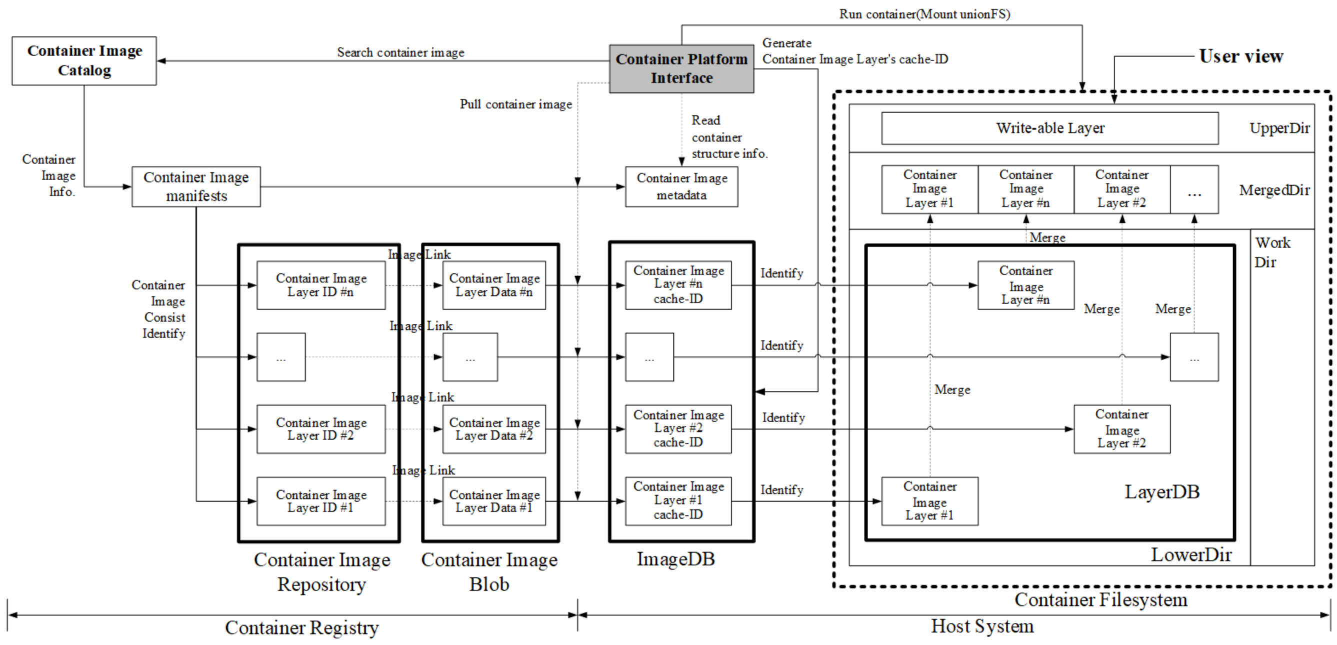

Container Image Catalog
Search container image
Container Platform Interface
Run container(Mount unionFS)
Generate Container Image Layer's cache-ID
User view
Pull container image
Read container structure info.
Container Image Info.
Container Image manifests
Container Image metadata
Write-able Layer
UpperDir
Container Image Layer #1
Container Image Layer #n
Container Image Layer #2
MergedDir
Merge
Work Dir
Container Image Consist Identify
Container Image Layer ID #n
Image Link
Container Image Layer Data #n
Container Image Layer #n cache-ID
Identify
Container Image Layer ID #2
Container Image Layer Data #2
Container Image Layer #2 cache-ID
Container Image Layer ID #1
Container Image Layer Data #1
Container Image Layer #1 cache-ID
LayerDB
LowerDir
Container Image Repository
Container Image Blob
ImageDB
Container Filesystem
Container Registry
Host System

FIGURE 1. **Container image distribution structure.**

the purposes of simple distribution and execution of the application, is weak. If a container is executed in such a secure environment, the mount and source directories for the container can be accessed by all users with access to the container platform. Moreover, the source directory constituting the container image can be accessed even while the container is running.

For example, if a web service is provided as a container, an unauthorized person can access the source directory and upload malware. Furthermore, when repeatedly executed, the web server acts as a malware distribution system; consequently, all users who access this web service are attacked by the malware. To overcome this security limitation, in this study, a container image access control architecture is proposed that can deny unauthorized access from the time the container image is downloaded. To verify the effectiveness of the proposed architecture, we compare the functionality and performance test results with those of legacy access control functions.

Improving the container platform according to the container image access control architecture proposed in this study provides the following benefits:

1) While maintaining the functionality of the container platform, direct access by unauthorized persons to the source directory of the container image can be denied. Therefore, maintaining the integrity of the container image while the container is running becomes possible.

2) If the container image is unmounted, the source directory can be accessed to change, extend, or upgrade the application in the container image.

3) Because the security function is applied only to the container platform, it consumes minimal necessary resources and its interference with other applications is minimized.

The remainder of this paper is organized as follows. Section 1 describes the background and objectives of this study. Section 2 describes the container image distribution environment, application security requirements, and current file/directory access control technology. In Section 3, the security environment of the container platform after the threats to the container image are identified is analyzed. In Section 4, a container-based image access control architecture is proposed for container image protection, and a method for providing its functions is described. In Section 5, the effectiveness of the proposed architecture is analyzed based on functional and performance verification. Finally, in Section 6, some concluding remarks are provided.

II. RELATED STUDIES

A. CONTAINER IMAGE DISTRIBUTION & EXECUTION

A container image uses a union filesystem to apply the application configuration management [6]. The types of union file systems supported by the current container platform include an advanced multi-layered unification filesystem (AUFS) and overlay and overlay2 file systems. AUFS and overlay file systems were previously used, whereas stable overlay and overlay2 filesystems are currently applied [7].

Fig. 1 shows a structure in which a user searches for, downloads, stores, and executes container images in Docker, a typical container platform using the overlay2 filesystem.

When a user downloads a container image, *container image metadata* and *container image layer data* constituting the container image are downloaded [8].

The container registry consists of a *container image catalog*, *container image manifests*, *container image repository*, and *container image binary large object (BLOB)* [9]. The *container image repository* is registered with a *container* *image layer ID* that can identify *container image layer data* constituting the container image [10].

The container user searches the container image in the *container image catalog* of the container registry. When the container user requests a container image download, the container registry identifies the *container image layer IDs* for the *container image layer data* constituting the container image by checking the *container image manifests* for the container image requested by the user. Thereafter, the container registry delivers the *container image layer data* identified using the identified *container image layer ID* to the user along with *container image manifests* [11].

The *container platform interface* generates a *container image layer cache-ID* for a *container image layer* for a union filesystem mount when storing the *container image layer*, and registers it in *ImageDB*. Thereafter, the *container platform interface* stores the *container image layer*, which uncompresses the *container image layer data* in the *LayerDB* of the host system. Because of this process, even if the same container image is downloaded, the storage path of the *container image layer* is different for each host system. The *container platform interface extracts* only the necessary information regarding the container image from the received *container image manifests* and stores it in the *container image metadata* with the generated *container image layer cache-ID* [12].

To mount the overlay2 filesystem, a typical filesystem supported by the container platform and the mount structures of *Upperdir*, *MergedDir*, *Lowerdir*, and *Workdir*, must be satisfied. For this, the *container platform interface* uses the *container image layer cache-ID* directory of *ImageDB* as the mount directory of the overlay2 filesystem. All downloaded *container image layer cache-ID*s were assigned a *LowerDir*. *WorkDir* is a directory used to merge image layers into *MergedDir*. *MergedDir* is a directory that shows the result of merging the *container image layer* identified by the *container image layer cache-ID*, and provides an integrated view to users. *UpperDir* is responsible for user I/O processing by adding a write-able layer at the time of the mounting [13].

The *container image layer cache-ID* directory required for the container image execution is registered in the *container image metadata* [14]. When the container user executes the container image, the *container platform interface* mounts the *container image layer cache-ID* directory of the *container image metadata* as a union filesystem [15].

B. APPLICATION SECURITY REQUIREMENT

Basically, an application used to provide information services must be provided in a secure environment and protected from threats such as external attacks or service abuse even during operation [16]. To this end, security requirements for information services have been proposed by many national public and international standard organizations. Among the typical security requirements, items related to this study are shown in Table 1 [17].

According to these security requirements, the information service must also be secure based on the container platform [18]. It should be possible to deny unauthorized access by identifying a subject wanting to access the information service. The change in the configuration of the information service must be made in an authorized state, and all unauthorized changes must be denied.

TABLE 1. General security requirements.

Requirement	Description
Access control	Unauthorized inquiry, modification, and transmission of important information should be denied.
Identification and authentication	All users accessing information systems and services must be identified and verified.
Maintenance	All security functions must be provided in a cost effective manner.
Systems and services acquisition	Only reliable information services should be used. When using external information services, only information services operated in an independent environment should be applied.
System and communications protection	All information transmissions must be monitored and protected.
System and information integrity	All information must be protected from unauthorized modification such as malware.

C. CURRENT FILE ACCESS CONTROL TECHNOLOGIES

To satisfy the requirements of Table 1, the legacy system uses SecureOS to protect the application and important information. SecureOS typically includes security-enhanced Linux (SELinux) and Application Armor (AppArmor) of Linux/Unix-type operating systems as well as a group policy object (GPO) in Windows operating systems. All three provide policy-based file/directory access control [19].

SELinux accepts the mandatory access control model and operates by enforcing an object-oriented security policy [20]. The security policy of SELinux consists of a security ID (SID) and permission, which are identifiers for an access control policy. The file/directory list, which is the actual protected object for the SID, is mapped to the SELinux of each system [21]. When the user sets the security policy, SELinux enforces the access control policy for the file/directory for the SID [22], [23].

AppArmor accepts a role-based access control security model. The access control policy is defined as a resource group, and it denies unauthorized access to the file/directory belonging to this group [24].

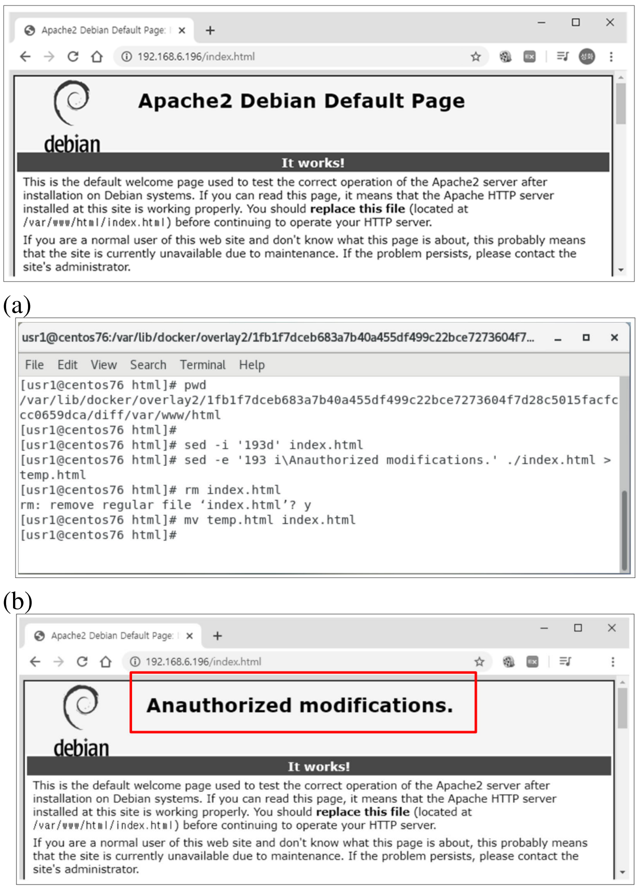
GPO applies the discretionary access control (DAC) model. Although there is a disadvantage in that the policy must be registered for all access control targets, it is easy to check the policy-enforced status and the DAC is easy to use [25].

All three SecureOS types are kernel-based access control technologies, which deny user bypass access. However because it is a policy-based access control technology, the access control policy can be enforced only if the file or directory to be protected is known in advance.

III. SECURITY ENVIRONMENT ANALYSIS OF CONTAINER PLATFORM

A. SECURITY THREAT TO CONTAINER IMAGE

As is well known, the container engine protects the execution environment of the container. A container engine uses



Apache2 Debian Default Page

It works!

This is the default welcome page used to test the correct operation of the Apache2 server after installation on Debian systems. If you can read this page, it means that the Apache HTTP server installed at this site is working properly. You should **replace this file** (located at /var/www/html/index.html) before continuing to operate your HTTP server.

If you are a normal user of this web site and don't know what this page is about, this probably means that the site is currently unavailable due to maintenance. If the problem persists, please contact the site's administrator.

(a)

```
usr1@centos76:/var/lib/docker/overlay2/1fb1f7dceb683a7b40a455df499c22bce7273604f7...
File  Edit  View  Search  Terminal  Help
[usr1@centos76 html]# pwd
/var/lib/docker/overlay2/1fb1f7dceb683a7b40a455df499c22bce7273604f7d28c5015facfc
cc0659dca/diff/var/www/html
[usr1@centos76 html]#
[usr1@centos76 html]# sed -i '193d' index.html
[usr1@centos76 html]# sed -e '193 i\Anauthorized modifications.' ./index.html >
temp.html
[usr1@centos76 html]# rm index.html
rm: remove regular file ‘index.html’? y
[usr1@centos76 html]# mv temp.html index.html
[usr1@centos76 html]#
```

(b)

Anauthorized modifications.

It works!

This is the default welcome page used to test the correct operation of the Apache2 server after installation on Debian systems. If you can read this page, it means that the Apache HTTP server installed at this site is working properly. You should **replace this file** (located at /var/www/html/index.html) before continuing to operate your HTTP server.

If you are a normal user of this web site and don't know what this page is about, this probably means that the site is currently unavailable due to maintenance. If the problem persists, please contact the site's administrator.

(c)

FIGURE 2. Unauthorized modification through direct access to container image layer: (a) Apache web server's default web page in container, (b) direct access to container image layer and unauthorized modification default page, (c) after default unauthorized modification.

namespaces and cgroups to individually apply environmental variables, resources, and process management for application during a container operation to ensure an independent operating environment [26].

However, the container engine does not protect the container image. Fig. 2 shows an example of an unauthorized person directly accessing the container image layer and modifying an important file while the container is running. Fig. 2(a) shows that the user accessed the default page of the Apache Web Server in the container, which represents a state of stability. Fig. 2(b) shows the process of an unauthorized user-modifying index.html through direct access to the container image layer. Consequently, the title of the default page was changed, as shown in Fig. 2(c).

As described above, the container engine only protects the application included in the container image but does not protect the container image required to run the application. In this environment, unauthorized users can directly access the container image layer to change files or leak important information.

In particular, in a container platform environment where multiple container images can be executed, the unauthorized modification effect of the container image layer is reflected in all container applications.

B. SECURITY ANALYSIS OF CONTAINER PLATFORM

Applications that run on the container platform must run in a secure environment according to the security requirements. Applications running through a container image execution ensure process isolation using a namespace. Independence is also guaranteed in the resource allocation for the application in a container using cgroups [27].

The unauthorized modification in Fig. 2 is an example of using the security vulnerability of the union file system supported by the container platform. Container images composed of multiple layers are treated as read-only for file systems [28] but are write-able for users. In this environment, unauthorized persons can directly access the container image layer to modify files or leak important information.

In the Legacy system, SecureOS was used to improve security vulnerabilities. However because the current version of SecureOS does not consider the container platform, it cannot improve the security vulnerability of the container platform.

To be provided with an access control function for important information by SecureOS, the object to be protected must be identified. To protect the container image, it is necessary to identify the container image layer directory to be protected. However, the container image layer directory is a container image layer cache-ID directory that is dynamically created when downloading a container image [29], and it is actually impossible for the user to identify the container image layer directly because it is a sha-256 digest type.

Because a container image is composed of multiple layers, it can be stated that it is a safe environment only when the container is executed while each container image layer is protected. However, the current container platform does not provide protection for container images. In addition, there is no separate external security function to protect the container image.

Therefore, the current container platform is not a safe environment for protecting container images.

IV. CONTAINER-BASED IMAGE PROTECTION ARCHITECTURE

The container platform, the main goal of which is to distribute, execute, and operate applications required to provide information services, is responsible for protecting container images according to the security principles [30], [31].

For this security environment, in this study, a container image access control architecture is proposed that blocks unauthorized access to container images.

A. ARCHITECTURE ELEMENT

The container image access control architecture proposed in this study was isolated from the container engine to ensure the provisioning of the container platform function. The DAC model is accepted, and after the container image

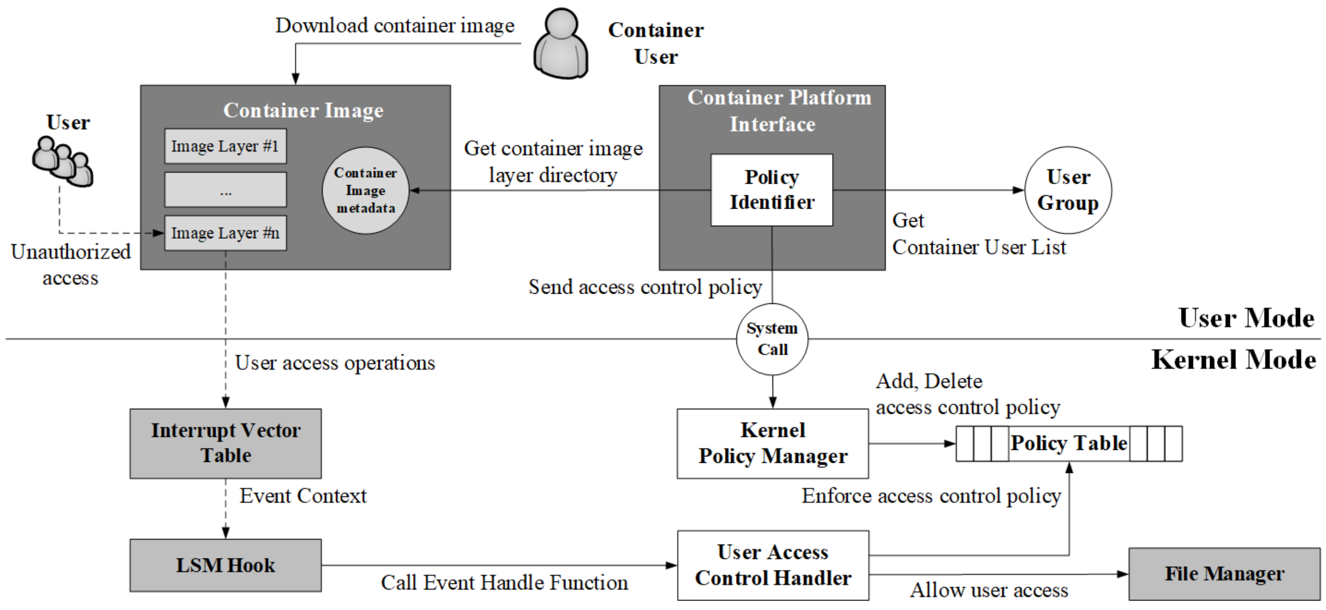


FIGURE 3. **Container-based image protection architecture.**

layer directory to be protected is dynamically identified, only an authorized container user using the container platform interface can access the container image layer directory.

The proposed architecture is composed of four modules, as shown in Fig. 3. The *policy identifier* manages the access control policies required when users download the *container images*. The *policy identifier* collects user information that can access the container image from the *user group* information of the host system. Thereafter, the container image layer directory (*container image layer cache-ID* in Fig. 2) is collected from the *container image metadata* of the container image downloaded to the host system, and the access control policy is generated and transmitted to the *kernel policy manager*.

The *kernel policy manager* adds or deletes the access control policy delivered by the *policy identifier* to the kernel-level policy table.

A *policy table* is a database in which access control policies delivered to the kernel are registered. For the convenience of access control policy management and high-speed policy application, the access control policy consists of a container ID, user ID, and container image layer directory.

The *user access control handler* denies unauthorized access to the container image layer directly by monitoring the user's file I/O. If the container user is an event accessing the container image layer using the *container platform interface*, this is allowed, and other access is denied.

B. SEQUENCE OF SECURITY FUNCTIONS

The sequence diagram of the entire process of denying unauthorized access to container images and allowing authorized access is shown in Fig. 4.

1) MANAGING AN ACCESS CONTROL POLICY

The container image access control architecture proposed in this study should allow only container user access to the container image layer. To this end, for the container image layer, the policy to deny all users is enforced by default, and the container user access allowance policy is added and deleted to protect the container image.

The management of the access control permission policy operates as a post action immediately after the user interface execution of the container platform. The commands of the *container platform interface* used to add access control policy are download, build, and commit, and the delete command is used to remove it.

When a user downloads, builds, or commits a container image, the *policy identifier* checks and collects the user group account information. Next, the *policy identifier* accesses the *container image metadata* to obtain the container image layer directory for access control. After generating the 2D table by combining the identified user ID and container image layer directory, the system call is used and delivered to the *kernel policy manager* along with the container ID.

The *kernel policy manager* adds or deletes the access control policy delivered by the *policy identifier*. When adding an access control policy, an access control policy consisting of the container ID, user ID, and container image layer directory is registered in the *policy table*.

2) ENFORCEMENT ACCESS CONTROL POLICY

The *user access control handler* monitors the file I/O of the operating system and enforces the access control policy for events accessing the container image layer directory.

All file I/Os occurring in the operating system are registered and processed as events of the interrupt vector table [32]. The Linux Security Module (LSM) is a trigger interface of the event handle function for handling user-defined file I/O access control when a file I/O occurs [33], and a *user access control handler* is applied whenever the file I/O executes a handle function to apply

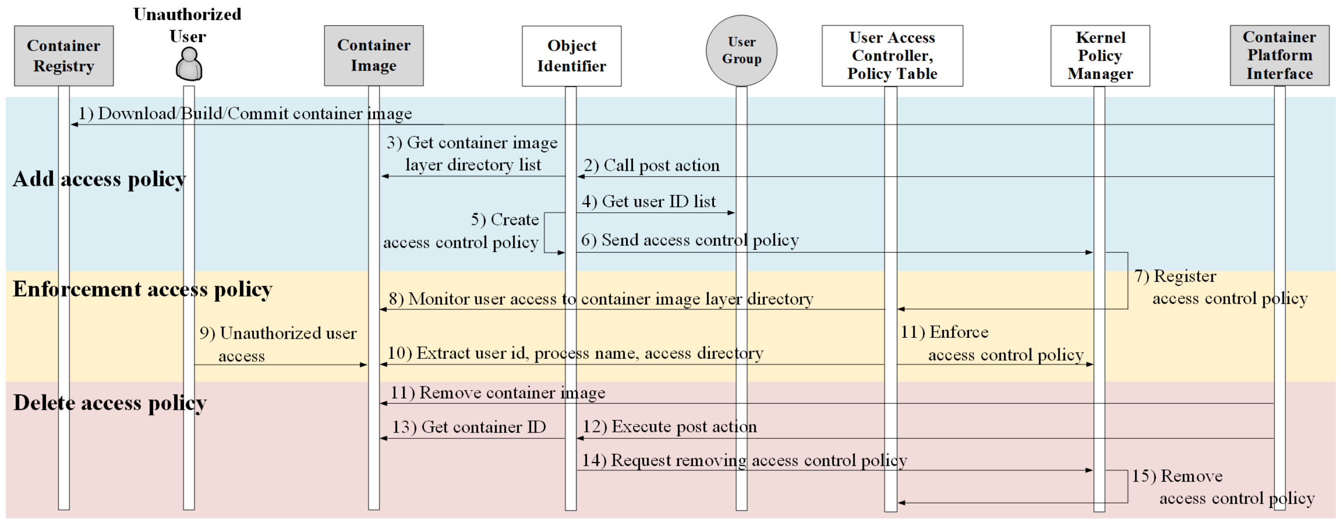

Container Registry
Unauthorized User
Container Image
Object Identifier
User Group
User Access Controller, Policy Table
Kernel Policy Manager
Container Platform Interface
Add access policy
1) Download/Build/Commit container image
3) Get container image layer directory list
2) Call post action
4) Get user ID list
5) Create access control policy
6) Send access control policy
7) Register access control policy
Enforcement access policy
8) Monitor user access to container image layer directory
9) Unauthorized user access
10) Extract user id, process name, access directory
11) Enforce access control policy
Delete access policy
11) Remove container image
13) Get container ID
12) Execute post action
14) Request removing access control policy
15) Remove access control policy

FIGURE 4. **Sequence diagram for container image protect.**

an access control function. When executing the event handle function in the LSM, the *event context* is delivered [34].

The *user access control handler* extracts the user ID, process name, and access object container image layer directory, which are the access subjects, from this *event context* and compares it with the access control policy to determine whether access is allowed. If an event for a file I/O is accessed by a container user applying the container platform interface, the container image layer directory is allowed, and container users who do not use the container platform interface or unauthorized users who attempt to use the container platform interface are denied access to the file I/O event.

V. IMPLEMENTATION

For the architecture proposed in this study to be effective, the access control function to be provided should operate normally, and the function and performance of the container platform should not be affected. Therefore, a functional test for the access control function, a regression test for the container platform function, and a performance test are conducted to verify the effectiveness of the proposed container image access control architecture.

A. FUNCTION TEST

1) FUNCTION TEST SCENARIO

The access control function proposed in this study only provides an access container using a container platform interface, with the default access denial policy applied to the container image layer directory. Therefore, the access control policy uses a container user (Uid) and a *container platform interface* (CPi) as the parameters. To describe the access control mechanism for the container image, we use the following definitions.

$$U = \{x|x \textit{ is all access control mechanism}\}$$

$$P = \{x \in U|\textit{Access control policy enforced to container platform}\}$$

$$AR = \{x|x \textit{ is set of all access request}\}$$

Under the above conditions, the access control policy enforced result (PER) by the access control function (ACD) for r of the access request (AR) is defined as follows:

$$PER = ACD(r), \textit{ but } r \in AR$$

Here,

$$CPi_p = \{x|x \textit{ is access request } r, \textit{ a member of } CPi \textit{ in policy } P\}$$

$$Uid_p = \{x|x \textit{ is access request } r, \textit{ a member of } Uid \textit{ in policy } P\}$$

When all values of CPi_P and Uid_P are true, access to the container image layer directory is allowed by the access control policy P. In this case, PER_A is expressed as

$$PER_A = ACD(r_{CPI_P \cdot Uid_P}), \textit{ but } r \in AR. \quad (1)$$

When the access control policy is enforced for an event, but is blocked by PER_D, the De Morgan law is applied to (1) and is defined as (2)

$$PER_D = ACD(r_{\overline{CPI_P}} + r_{\overline{Uid_P}}), \quad \textit{but } r \in AR. \quad (2)$$

Equation (1) indicates that the user's container image access event is allowed (all CPi and Uid are matched), and (2) indicates that it is denied (one or more CPi and Uid are unmatched). Therefore, if the functions of (1) and (2) are verified, the access control suggested in this study can be said to be valid.

2) FUNCTION TEST ITEMS

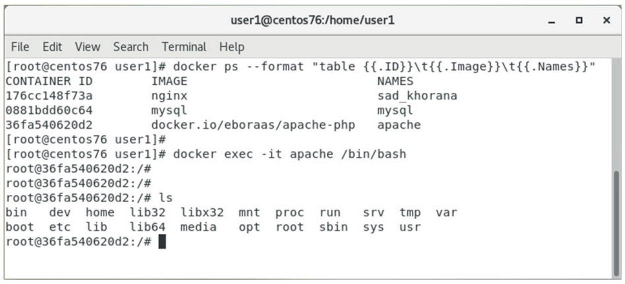
For the functional test items of the container base imageless control architecture suggested in this study, the functional verification items in Table 2 were calculated according to (1) and (2).

3) FUNCTION TEST RESULTS

As a result of testing Func_P1, when the container user accesses the container image using the *container platform*

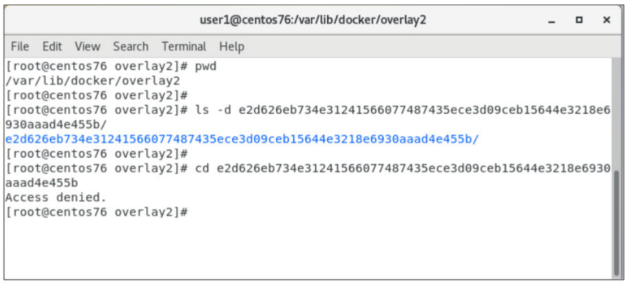
TABLE 2. Functional test items.

Test ID	Description
Func_P1	Verify (1) After downloading the container image, verify whether the container user (Uid), the subject of the access control policy, is allowed when accessing the container image layer directory using *Container Platform Interface* (CPi).
Func_N1	Verify $PER_D = ACD(r_{\overline{CPi_P}})$ of (2). After downloading the container image, verify whether the container user (Uid), the subject of the access control policy, is denied direct access to the container image layer directory without using the *Container Platform Interface* (CPi).
Func_N2	Verify $PER_D = ACD(r_{\overline{Uid_P}})$ of (2). After downloading the container image, verify whether the container user (Uid), not the subject of the access control policy, is denied when accessing the container image layer directory using the *Container Platform Interface* (CPi).



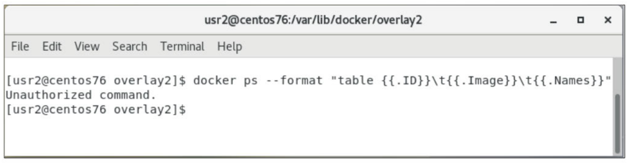
```
[root@centos76 user1]# docker ps --format "table {{.ID}}\t{{.Image}}\t{{.Names}}"
CONTAINER ID        IMAGE                          NAMES
176cc148f73a        nginx                          sad_khorana
0881bdd60c64        mysql                          mysql
36fa540620d2        docker.io/eboraas/apache-php   apache
[root@centos76 user1]#
[root@centos76 user1]# docker exec -it apache /bin/bash
root@36fa540620d2:/#
root@36fa540620d2:/#
root@36fa540620d2:/# ls
bin   dev  home  lib32  libx32  mnt  proc  run   srv  tmp  var
boot  etc  lib   lib64  media   opt  root  sbin  sys  usr
root@36fa540620d2:/# █
```

FIGURE 5. Verification results of allowed user access.



```
[root@centos76 overlay2]# pwd
/var/lib/docker/overlay2
[root@centos76 overlay2]#
[root@centos76 overlay2]# ls -d e2d626eb734e312415660774874 35ece3d09ceb15644e3218e6
930aaad4e455b/
e2d626eb734e312415660774874 35ece3d09ceb15644e3218e6930aaad4e455b/
[root@centos76 overlay2]#
[root@centos76 overlay2]# cd e2d626eb734e312415660774874 35ece3d09ceb15644e3218e6930
aaad4e455b
Access denied.
[root@centos76 overlay2]#
```

FIGURE 6. Verification result of denied user access using unpermitted interface.



```
[usr2@centos76 overlay2]$ docker ps --format "table {{.ID}}\t{{.Image}}\t{{.Names}}"
Unauthorized command.
[usr2@centos76 overlay2]$
```

FIGURE 7. Verification result of denied unauthorized user access.

interface to the container image, it is confirmed that the container user's access is allowed, as shown in Fig. 5.

As a result of testing Fun_N1, when the container platform user accesses the container image without using the *container platform interface*, it is confirmed that the user's access is denied, as shown in Fig. 6.

As a result of testing Func_N2, it was confirmed that unauthorized access to the container image using the *container platform interface* is denied, as shown in Fig. 7.

B. REGRESSION TEST

Even if the access control function by the architecture suggested in this study operates normally, the original function of the container platform cannot be compromised. To verify that the access control function proposed in this study does not affect the container platform, a regression test, as shown in Table 3, was applied.

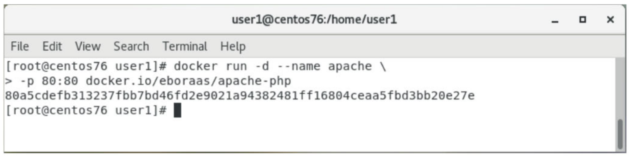
As a result of the Reg_T1 test, it was confirmed that the container is normally executed and the application of the container image is normally executed, as shown in Fig. 8.

As a result of the Reg_T2 test, it was confirmed that file copying to the container can be normally applied, as shown in (a) in Fig. 9, and is normally applied as shown in (b).

As a result of the Reg_T3 test, it was confirmed that the changed container, as shown in Fig. 10, was successfully committed and a new container image was generated.

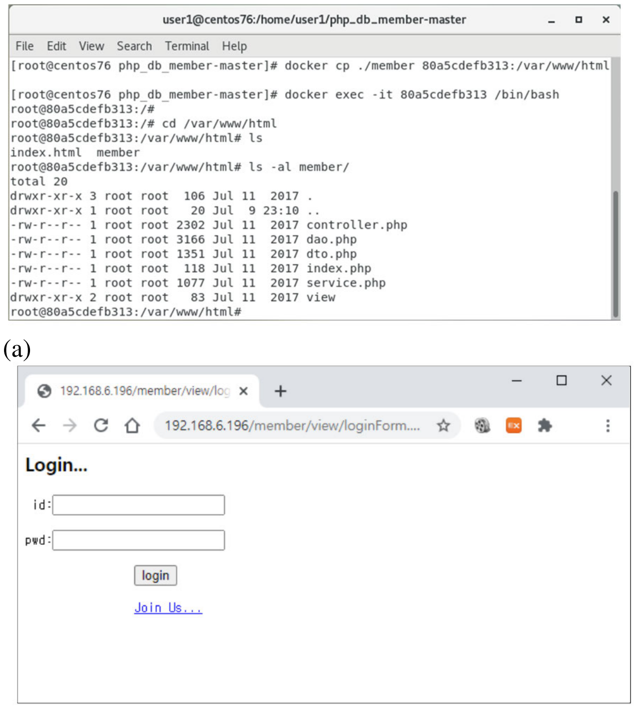
TABLE 3. Regression test items.

Test ID	Description
Reg_T1	Run container image to verify the execution of the application loaded in the container image
Reg_T2	Verify that it is applied normally by copying the file to the running container
Reg_T3	Verify whether a new container image can be generated by adding a file to the running container and committing it



```
[root@centos76 user1]# docker run -d --name apache \
> -p 80:80 docker.io/eboraas/apache-php
80a5cdefb313237fbb7bd46fd2e9021a94382481ff16804ceaa5fbd3bb20e27e
[root@centos76 user1]# █
```

FIGURE 8. Regression test result of container image execution.



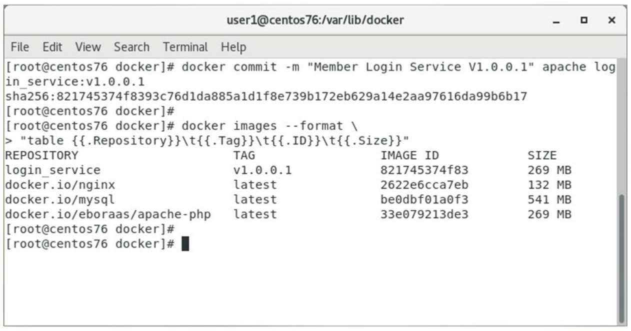
```
[root@centos76 php_db_member-master]# docker cp ./member 80a5cdefb313:/var/www/html

[root@centos76 php_db_member-master]# docker exec -it 80a5cdefb313 /bin/bash
root@80a5cdefb313:/#
root@80a5cdefb313:/# cd /var/www/html
root@80a5cdefb313:/var/www/html# ls
index.html  member
root@80a5cdefb313:/var/www/html# ls -al member/
total 20
drwxr-xr-x 3 root root  106 Jul 11  2017 .
drwxr-xr-x 1 root root   20 Jul  9 23:10 ..
-rw-r--r-- 1 root root 2302 Jul 11  2017 controller.php
-rw-r--r-- 1 root root 3166 Jul 11  2017 dao.php
-rw-r--r-- 1 root root 1351 Jul 11  2017 dto.php
-rw-r--r-- 1 root root  118 Jul 11  2017 index.php
-rw-r--r-- 1 root root 1077 Jul 11  2017 service.php
drwxr-xr-x 2 root root   83 Jul 11  2017 view
root@80a5cdefb313:/var/www/html#
```

(a)

192.168.6.196/member/view/log
192.168.6.196/member/view/loginForm....
Login...
id:
pwd:
login
Join Us...

(b)

FIGURE 9. Regression test result of file copy to container: (a) file copy from host to container and (b) check file copy result.


user1@centos76:/var/lib/docker
File Edit View Search Terminal Help

```
[root@centos76 docker]# docker commit -m "Member Login Service V1.0.0.1" apache log
in_service:v1.0.0.1
sha256:821745374f8393c76d1da885a1d1f8e739b172eb629a14e2aa97616da99b6b17
[root@centos76 docker]#
[root@centos76 docker]# docker images --format \
> "table {{.Repository}}\t{{.Tag}}\t{{.ID}}\t{{.Size}}"
REPOSITORY                    TAG                 IMAGE ID            SIZE
login_service                 v1.0.0.1            821745374f83        269 MB
docker.io/nginx               latest              2622e6cca7eb        132 MB
docker.io/mysql               latest              be0dbf01a0f3        541 MB
docker.io/eboraas/apache-php  latest              33e079213de3        269 MB
[root@centos76 docker]#
[root@centos76 docker]#
```

FIGURE 10. Regression test result of container commit.

TABLE 4. CPU rate measurement result.

Mechanism	Measurement result						
SELinux	Count	1	2	3	4	5	Avg.
	Result(ms)	3.7	3.4	3.1	3.3	3.4	3.48
	count	6	7	8	9	10	
	Result(ms)	3.4	3.6	4.0	3.5	3.4	
Proposed Architecture	Count	1	2	3	4	5	Avg.
	Result(ms)	3.6	3.5	3.5	3.4	3.6	3.50
	count	6	7	8	9	10	
	Result(ms)	3.2	3.3	3.4	3.7	3.8	

C. PERFORMANCE VERIFICATION

The container platform is intended to download multiple container images and run multiple containers. Therefore, the delay of a container operation by the access control function suggested in this study should be minimized.

To this end, we measured the CPU rate (%) required for a policy registration and the enforcement time of the access control policy on an Intel i7-8700 CPU, 32 GB of RAM, and a 10 TB HDD, and compared the results using SELinux.

1) CPU RATE MEASUREMENTS

After measuring the CPU share required to register the access control policy, we checked the effect. To this end, the CPU occupancy is measured when 5000 policies are registered using a shell script for access control policy registration, and the same method is applied to SELinux to compare the measurement results and verify the performance.

Table 4 shows the results of ten measurements of the CPU share when registering 5000 access control policies in the container image access control architecture in both this study and SELinux.

As a result of checking the average value of the performance measurement, it was confirmed that the CPU share of the proposed architecture is mostly the same as that of SELinux.

2) MEASUREMENT OF POLICY ENFORCEMENT TIME

The downloading of multiple container images means that more container image layers are downloaded, in which case, many access control policies can be registered, although the enforcement time of the access control policy for the user should be minimized.

To measure the access control policy enforcement time for users, 500, 1000, 2000, 3000, 4000, 5000, and 6000 dummy policies are registered in the policy table using a shell script, and the last policy enforcement time is then measured. To check the validity of the proposed architecture, the measurement results were compared with the SELinux policy enforcement time.

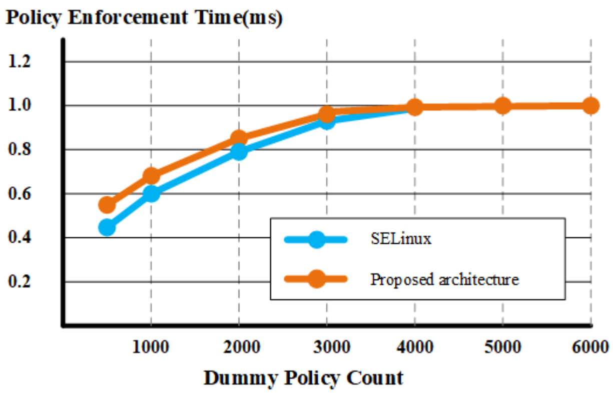


FIGURE 11. Regression test result of file copy to container.

To measure the last policy enforcement time, "time (ms) = the matching access control policy application time minus the policy search start time" is applied.

Fig. 11 shows the average time taken to apply the last access control policy 10 times when multiple dummy access control policies are registered. When the number of dummy policies was less than 3,000, the policy application time of the proposed architecture was longer than that of SELinux. However, starting from the 4,000 dummy policies, the proposed architecture and the SELinux policy application time were the same at 1 ms.

D. ANALYSIS RESULTS

As a result of the functional verification, it was confirmed that the container image access control architecture allows only container image access by container users using the container platform interface and denies access to the container image or container users accessing the container image layer directly.

During the regression test, it was confirmed that the access control function of this study did not affect the original function provided by the container platform.

The performance when providing the access control function was also confirmed to be almost the same as that of the existing SELinux.

Therefore, it can be determined that the container image access control architecture proposed in this study is sufficiently effective in terms of function and performance.

VI. CONCLUSION

The usage rate of container platforms is increasing owing to both their features and a reduction in the information service construction cost. In particular, with the support of the Kubernetes orchestration tool, the rate has significantly increased.

In this study, the threat of unauthorized modification attacks of container images was confirmed by directly accessing the container image layer downloaded to the host system.

To improve the security vulnerability of the container platform, in this study, a container image access control architecture was proposed. The proposed architecture provides a dynamic access control function for the container image layer directory, and it is enforced at a single point at the kernel level; thus, the user cannot bypass the access control function.

In particular because the container user operates as a post action when using the container platform interface, there is an advantage in that the container image can be protected without additional user manipulation. In addition because the access control function operates independently of the container engine and consumes very few resources, there is an advantage of not impairing the original function provided by the container platform.

As a result, it was confirmed that the container image access control architecture proposed in this study can increase the container platform security by protecting the container image of the information service based on the container platform.

However, during the regression test coverage applied in this study, the application execution through the container image is run, the file is copied to the container, and the committed changes to the container are executed. Because an unidentified side effect may occur when using the access control function proposed in this study, it is necessary to expand and verify the regression test coverage. In addition, only the overlay2 file system, which is the container platform with the most verification environments, was selected. If the same performance measurement is performed in AUFS or an overlay file system, the reliability of the performance result is expected to be higher.

Monitoring and tracking functions for the access control function proposed in this study are also needed according to the general security requirements, and thus further research is necessary.

REFERENCES

[1] C. Kaewkasi and K. Chuenmuneewong, "Improvement of container scheduling for docker using ant colony optimization," in *Proc. 9th Int. Conf. Knowl. Smart Technol. (KST)*, Feb. 2017, pp. 254–259.

[2] S. Wu, C. Niu, J. Rao, H. Jin, and X. Dai, "Container-based cloud platform for mobile computation offloading," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2017, pp. 123–132.

[3] C. C. Spoiala, A. Calinciuc, C. O. Turcu, and C. Filote, "Performance comparison of a WebRTC server on docker versus virtual machine," in *Proc. Int. Conf. Develop. Appl. Syst. (DAS)*, May 2016, pp. 295–298.

[4] D. Lucia and J. Michael, "A survey on security isolation of virtualization, containers, and unikernels," US Army Res. Lab. Aberdeen Proving Ground United States, Ground, MD, USA, Tech. Rep. ARL-TR-8029, May 2017.

[5] S. H. Han, H. K. Lee, G. Y. Gim, and S. J. Kim, "Empirical study on anti-virus architecture for container platforms," *IEEE Access*, vol. 8, pp. 134940–134949, 2020.

[6] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "Cntr: Lightweight OS Containers," in *Proc. USENIX Annu. Tech. Conf.*, Jul. 2018, pp. 199–212.

[7] A. Dewald, M. Luft, and J. Suleder, "Incident Analysis and Forensics in Docker Environments," ERNW White Paper, ERNW, Feb. 2018.

[8] A. Ahmed and G. Pierre, "Docker container deployment in fog computing infrastructures," in *Proc. IEEE Int. Conf. Edge Comput. (EDGE)*, Jul. 2018, pp. 1–8.

[9] B. Noel, D. Michelino, M. Velten, R. Rocha, and S. Trigazis, "Integrating containers in the CERN private cloud," *J. Phys., Conf. Ser.*, vol. 898, Oct. 2017, Art. no. 092045.

[10] T. Xu and D. Marinov, "Mining container image repositories for software configuration and beyond," in *Proc. 40th Int. Conf. Softw. Eng. New Ideas Emerg. Results*, 2018, pp. 49–52.

[11] J. Blomer, P. Buncic, G. Ganis, N. Hardi, R. Meusel, and R. Popescu, "New directions in the CernVM file system," *J. Phys., Conf. Ser.*, vol. 898, Oct. 2017, Art. no. 062031.

[12] L. Ma, S. Yi, and Q. Li, "Efficient service handoff across edge servers via docker container migration," in *Proc. 2nd ACM/IEEE Symp. Edge Comput.*, Oct. 2017, pp. 1–13.

[13] Z. Lu, J. Xu, Y. Wu, T. Wang, and T. Huang, "An empirical case study on the temporary file smell in dockerfiles," *IEEE Access*, vol. 7, pp. 63650–63659, 2019.

[14] W. Gerlach, W. Tang, K. Keegan, T. Harrison, A. Wilke, J. Bischof, M. D'Souza, S. Devoid, D. Murphy-Olson, N. Desai, and F. Meyer, "Skyport-container-based execution environment management for multi-cloud scientific workflows," in *Proc. 5th Int. Workshop Data-Intensive Comput. Clouds*, Nov. 2014, pp. 25–32.

[15] L. Ma, S. Yi, N. Carter, and Q. Li, "Efficient live migration of edge services leveraging container layered storage," *IEEE Trans. Mobile Comput.*, vol. 18, no. 9, pp. 2020–2033, Sep. 2019.

[16] F. A. Aloul, "The need for effective information security awareness," *J. Adv. Inf. Technol.*, vol. 3, no. 3, pp. 176–183, Aug. 2012.

[17] R. S. Ross, S. W. Katzke, and L. A. Johnson, "Minimum security requirements for federal information and information systems," in *Proc. NIST*, 2006, p. 200.

[18] R. Chandramouli and R. Chandramouli, "Security assurance requirements for linux application container deployments," US Dept. Commerce, Nat. Inst. Standards Technol., Gaithersburg, MS, USA, Tech. Rep. NISTIR 8176, Oct. 2017.

[19] K. Salah, J. M. Alcaraz Calero, J. B. Bernabé, J. M. Marín Perez, and S. Zeadally, "Analyzing the security of windows 7 and linux for cloud computing," *Comput. Secur.*, vol. 34, pp. 113–122, May 2013.

[20] S. Smalley, C. Vance, and W. Salamon, "Implementing SELinux as a Linux security module," *NAI Labs Rep.*, vol. 1, no. 43, p. 139, 2001.

[21] D. Kilpatrick, W. Salamon, and C. Vance, "Securing the X Window system with SELinux," *Tech. Rep.*, vol. 3, no. 6, pp. 1–33, 2003.

[22] Hanson, C., "SELinux and MLS: Putting the pieces together," in *Proc. 2nd Annu. SELinux Symp.*, Feb. 2006, pp. 1–8.

[23] L. Papachristodoulou and S. Antakis, "Security-enhanced Linux in a health information system," *Eindhoven Univ. Technol.*, Dec. 2008.

[24] H. Chen, N. Li, and M. Z. , "Analyzing and comparing the protection quality of security enhanced operating systems," in *Proc. NDSS*, Feb. 2009, pp. 11–16.

[25] K. Arya, *Windows Group Policy Troubleshooting: A Best Practice Guide for Managing Users and PCs Through Group Policy*. New York, NY, USA: Apress, 2016.

[26] S. Kehrer, F. Riebandt, and W. Blochinger, "Container-based module isolation for cloud services," in *Proc. IEEE Int. Conf. Service-Oriented Syst. Eng. (SOSE)*, Apr. 2019, pp. 17704–17709.

[27] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "KVM, xen and docker: A performance analysis for ARM based NFV and cloud computing," in *Proc. IEEE 3rd Workshop Adv. Inf., Electron. Electr. Eng. (AIEEE)*, Nov. 2015, pp. 1–8.

[28] Z. Anwar, S. Potter, C. Narayanaswami, W. Yurcik, C. A. Gunter, and R. H. Campbell, "Detecting and mitigating denial-of-service attacks on voice over IP networks," *IBM Watson Res., Columbia Univ.*, Sep. 2008.

[29] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, A. Skourtis, H. Ludwig, and D. Hildebrand, "Improving docker registry design based on production workload analysis," in *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 265–278.

[30] A. Krull, "GASSP generally-accepted system security principles: A trip to abilene," *Comput. Secur.*, vol. 5, no. 15, p. 417, 1996.

[31] A. Conklin, G. White, C. Cothren, D. Williams, and R. L. Davis, *Principles of Computer Security: Security+ and Beyond*. New York, NY, USA: McGraw-Hill, 2004.

[32] P. Krzyzanowski, "Operating system concepts what is an operating system, what does it do, and how do you talk to it?" *Rutgers Univ.*, Jan. 2014.

[33] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux security module framework," in *Proc. Ottawa Linux Symp.*, vol. 8032, Jun. 2002, pp. 6–16.

[34] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in *Proc. Found. Intrusion Tolerant Syst.*, 2002, pp. 17–31.

SUNG-HWA HAN (Member, IEEE) received the Ph.D. degree from Soongsil University, South Korea. He works as the Senior Manager of SGA Solutions Company Ltd., South Korea. He has published many articles in journals. He is a consultant of security management and has a certificate in that. His work is mainly in the areas of development, quality assurance, and common criteria for security solutions. His research interests include information security, security management, and IT security conversions.

HOO-KI LEE (Member, IEEE) is a Faculty Member of the Department of Cyber Security Engineering, Konyang University. He worked at the Ministry of Culture, Sports, and Tourism Cyber Security Center, from 2009 to 2019. He has been interested in research fields such as security engineering for high-assurance systems, security threat analysis and security evaluation, and digital forensics.

SUNG-TAEK LEE (Member, IEEE) is an Assistant Professor with the Department of Computer Science, Yongin University, South Korea. He has published various articles in journals such as the *International Journal of Technology Management*, the *Journal of Management Information Systems*, and the *Journal of Business Models*. He is interested in research fields such as the fourth industrial revolution, IT service business, startup support, and intellectual property rights.

SUNG-JIN KIM (Member, IEEE) received the B.S. degree in computer science from The Ohio State University, Columbus, USA, the M.S. degree in computer science from Sogang University, Seoul, South Korea, and the Ph.D. degree from the KAIST, Daejeon, South Korea, in 2019. He is currently an Assistant Professor with Cheju Halla University. He is interested in various security issues related to personal computers, social networks, deep Web, and IP networks. His current research interests include machine learning-based malware detection, big data analytics, social network analysis, risk analysis, Web security, network security, and cyber kill chain detection.

WON-JUNG JANG (Member, IEEE) received the Ph.D. degree. He works as a Professor with the Department of Intellectual Property for Startups, Catholic Kwandong University. He has published a number of articles in journals such as *Applied Sciences*. His books include fourth industrial revolution, how to start and start industrial revolution, and the era of new manufacturing. He is interested in research such as big data, machine learning, artificial intelligence, software engineering, and the fourth industry revolution.