Article

Kernel-Based Container File Access Control Architecture to Protect Important Application Information

Hoo-Ki Lee [1], Sung-Hwa Han [2] and Daesung Lee [3,*]

[1] Department of Cyber Security, Konyang University, Nonsan 35365, Republic of Korea
[2] Department of Information Security, Tongmyong University, Busan 48520, Republic of Korea
[3] Department of Computer Engineering, Catholic University of Pusan, Busan 46252, Republic of Korea
* Correspondence: dslee@cup.ac.kr

Abstract: Container platforms ease the deployment of applications and respond to failures. The advantages of container platforms have promoted their use in information services. However, the use of container platforms is accompanied by associated security risks. For instance, malware uploaded by users can leak important information, and malicious operators can cause unauthorized modifications to important files to create service errors. These security threats degrade the quality of information services and reduce their reliability. To overcome these issues, important container files should be protected by file-access control functions. However, legacy file-access control techniques, such as umask and SecureOS, do not support container platforms. To address this problem, we propose a novel kernel-based architecture in this study to control access to container files. The proposed container file-access control architecture comprises three components. The functionality and performance of the proposed architecture were assessed by implementing it on a Linux platform. Our analysis confirmed that the proposed architecture adequately controls users' access to container files and performs on par with legacy file-access control techniques.

Keywords: Linux container; para-virtualization; file-access control; SELinux

check for updates

Citation: Lee, H.-K.; Han, S.-H.; Lee, D. Kernel-Based Container File Access Control Architecture to Protect Important Application Information. *Electronics* **2023**, *12*, 52. https://doi.org/10.3390/electronics12010052

Academic Editor: Zbigniew Kotulski

Received: 17 October 2022
Revised: 18 December 2022
Accepted: 20 December 2022
Published: 23 December 2022

1. Introduction

Cloud platforms can be classified into virtual machines (VMs), which include operating systems (OSs), and containers, which do not include OSs, in terms of instance type. VMs implement complete virtualization and run their instances on Hypervisor. In contrast, container platforms are implementations of Linux containers (LXC) [1]. An LXC is a runtime environment comprising tools, templates, and libraries [2]. A container is a unit that composes files and directories into a single package. Technical implementations of LXC are referred to as container platforms. A container platform includes a container execution environment and related supporting systems [3]. An information service manager creates a container image, including an application file/directory, registry, and other application environments, whenever an application or information is required to be distributed via a container platform [4]. The information service manager can create a container by executing the container image [5]. Because the container is executed based on the file/directory and other environments included in the container image, the application execution environment is independent of the user environment. Moreover, containers are protected from other containers and system applications because they operate in isolated environments [6]. Therefore, multiple applications can be simultaneously executed on container platforms. Container platforms exhibit high resource efficiency because they support dynamic resource management [7]. For instance, container platforms provide high-availability (HA) and load-balancing (LB) functions [8]. Because of these characteristics, a significant number of recent information services are being constructed using container platforms [9,10].

Container-based information services may include important files depending on the service object and structure during the creation of a container image or during operation, impacting the provision of application services significantly. If the configuration file included in the container is deleted or modified because of incorrect management, the application service may malfunction [11]. In other cases, users may upload malware to a server, which can access the application configuration file and leak important information to external agents. Therefore, important files, such as configuration files, that may be included in containers must be protected from unauthorized access [12].

In legacy information services, either umask or SecureOS is used to protect important files from unauthorized access [13]. Although the system manager can easily set up the umask policy, detailed configurations cannot be established. Additionally, the umask function cannot prevent superuser access. SecureOS addresses a few of the aforementioned limitations of umask. It is a policy-based access control technique that supports multiple access control models such as mandatory access control (MAC), discretionary access control (DAC), and role-based access control (RBAC) [14]. Because SecureOS can effectively protect important application files and directories, it is extensively utilized in real-world information services. However, SecureOS was developed before container platforms were introduced. Therefore, although SecureOS focuses on the access control function for files in the OS's file system, it does not provide access control for files located inside containers. Therefore, it cannot be used to prevent unauthorized access to container files.

To address this limitation, we propose a kernel-based container file-access control architecture to prevent unauthorized access to container files. The proposed architecture provides a policy-based access control function that monitors the access of container files, allowing authorized processes to proceed and denying permission to unauthorized processes. The proposed architecture is implemented at the kernel level, preventing users from bypassing the access control function. Furthermore, the security manager can verify the enforcement of the access control function based on log files.

Ideally, the function and performance of the proposed architecture should be effective in terms of denying unauthorized access. Therefore, both the positive and negative functions and the performance of the proposed structure were verified in this study.

The primary contributions of the proposed architecture can be summarized as follows:

- All container file-access events of users can be monitored in real time.
- The security environment of application services is strengthened by denying unauthorized access to container files.
- The proposed architecture has minimal effect or other services because only a few system resources are used to provide the access control function.
- The rapid enforcement of the access control function minimizes the performance degradation of the application service.
- The proposed architecture can serve as a security tool because it supports the identification of potentially malicious attackers and processes.

The remainder of this paper is organized as follows. The operational methodology and limitations of legacy file access control techniques are explained in Section 2. In Section 3, the proposed container file access control architecture and the role of each component are discussed. In Section 4, the implementation of the kernel-based architecture is presented, and the unit function of each component with respect to container file access control functions is described. The verification of the functions and the performance of the proposed architecture is presented in Section 5. Finally, the conclusions of the study are summarized in Section 6.

2. Related Works

2.1. LXC and Container Platforms

LXC provides an environment for the independent and simultaneous operation of multiple containers within a single system [15]. The application service manager can create multiple containers by repeatedly running a single image on a single system. In other words, containers can be repeatedly executed as long as the system resources allow it [16].

The image of a container may include software (SW) that performs a specific function. The application service manager can use the SW by spawning a container and executing the container image without actually installing the SW [17], and even modify the container image if required [18]. Multiple containers that are simultaneously executed share the kernel of the OS. Typically, containers being executed share all resources of the OS. Although the OS dynamically allocates resources to each container upon its execution, specific resources can be pre-emptively selected, if required [19].

The container platform is an implementation of LXC because it satisfies the technical requirements of LXC and supports various operating environments. In addition to Docker, examples of currently used container platforms include AWS Fargate, Google Kubernetes, and Apache Mesos [20,21]. It is evident that container platforms are widely utilized as operating environments in multiple information services because of their numerous advantages.

As indicated in Figure 1, a container platform comprises a container engine, container image, container, container registry, container management interface, storage, and scheduler [22].

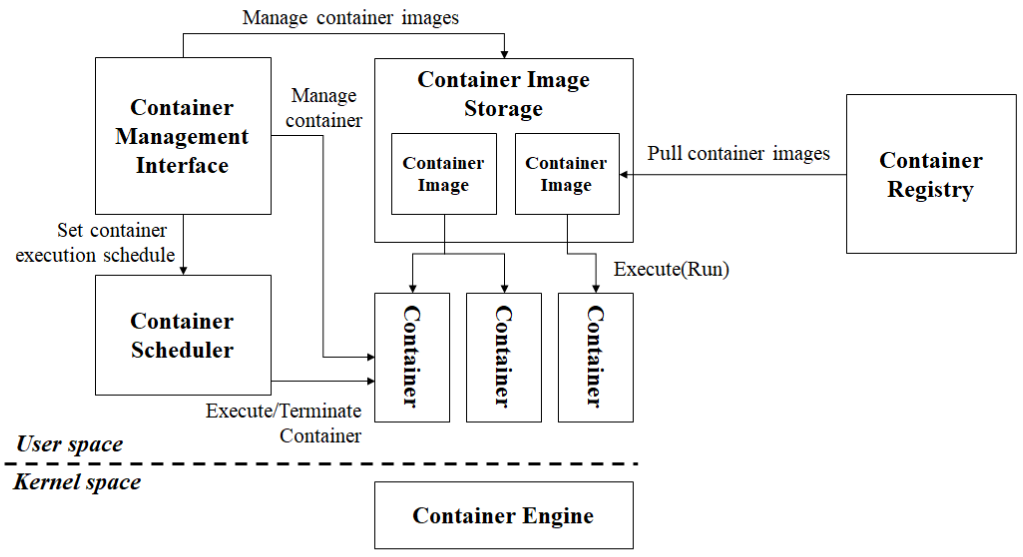

Figure 1. Container platform structure.

The container engine provides an isolated environment for the execution of containers [23]. A container image is a deployment unit that remains static prior to container execution [24]. A container registry is a server on which container images are registered, and the container management interface serves as a user management interface for the execution and monitoring of containers [25]. The container storage is a repository for container images [26]. Finally, the scheduler handles periodic executions of the container [27].

The system manager can configure network configurations, such as the maximum session and throughput, of each container [28]. The HA or LB function of the information service can be implemented by executing a container and applying the network configurations of the container platform. These advantages have promoted the use of container platforms [29].

2.2. Security Threats and Requirements

Container platform-based information services are susceptible to security threats, similar to any other service. Any container created by executing a container image includes an application environment such as a configuration file for application execution, a system account, or a library path. Therefore, a malicious attacker accessing the system environment can delete or modify the application environment, inducing the malfunction of container-based applications [30].

In general, important files may be included in the container during its execution or may be input into the container when an application service is provided. Additionally, the service manager or users can share or upload important files to the container, respectively. Thus, a service user may upload malware, which, if executed, may delete or modify user files stored in the container or leak important information included in the file [31].

Therefore, both important and user files should be protected in this environment. A secure environment [32] must guarantee confidentiality, availability, integrity, authentication, and access control. In particular, the access control function monitors user access to sensitive files and denies unauthorized access or modification by verifying whether the requesting subject is allowed to access the requested object [33,34].

All security technologies are required to satisfy specific security principles, such as identifiability, self-awareness, appropriate boundaries, convenience, locatability, and visibility. Among these, this study focuses on locatability and convenience. Locatability represents the degree of choice of the security manager, and convenience represents the ease of security-function enforcement [35]. Therefore, a protection mechanism should be developed to protect important container files, and security managers can choose to enforce it on container platforms at their own discretion [36].

2.3. Legacy File Access control Techniques

Umask and SecureOS are the two most widely used file access control techniques. Umask is an OS-level access control technique for users and user groups that is utilized in most OSs. Umask operates based on access privileges defined by the owner for each role (owner, owner group, and others) with respect to the file/directory. Furthermore, the OS monitors and identifies users (subjects) accessing files (objects). Subsequently, the roles of users are verified to determine whether access should be provided. Therefore, umask simply uses the access control function as a policy configured by the file/directory owner [37].

SecureOS is a policy-based access control technique that monitors access to resources from a single point in the kernel and denies access when the policy is violated. It provides access control functions for files/directories, application processes, and devices. Access control models, such as MAC, DAC, and RBAC, are supported in terms of subject identifiers, including security groups, user roles, users, and user groups [38].

SELinux and AppArmor are well-established SecureOS architectures provided by the Linux OS, and the group policy object (GPO) is available on Windows [39,40].

Figure 2 depicts the four-tier structure of SELinux, which is a representative SecureOS architecture. SELinux is based on an access control policy in which the user sets the policy identification number (ID) (object), including the user ID (subject), via the policy management interface, which is then transferred to the *Security Server* and converted into an object file/directory path by searching the *SELinux Filesystem*. If the *Security Server* determines the object file/directory to be complete, the user ID and file/directory path (object) are registered in the *Access Vector Cache* (AVC). Whenever a file access event occurs, the corresponding event information is transferred to AVC, which extracts the user ID and file/directory path accessing the object. If a user's file access event matches the access control policy, user access is denied, and the event is logged. Both AppArmor and GPO provide kernel-based file access control, and their structures are similar to that of SELinux [41].

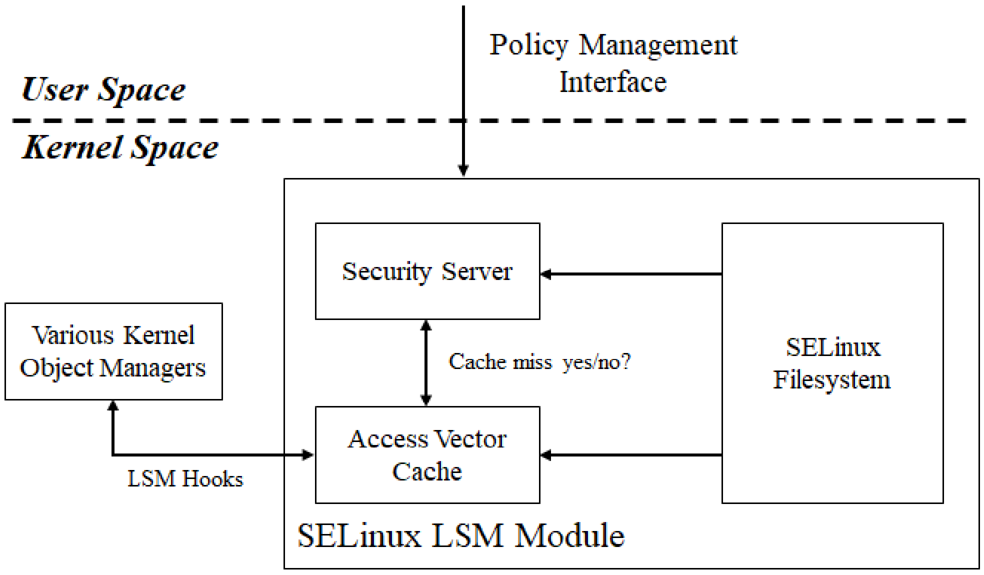

Figure 2. File access control structure of SELinux. SELinux registers the access control policy at the security server in the kernel and enforces it using the access vector cache.

2.4. Limitations of the Legacy File Access control Techniques

Umask is primarily an access control function for OSs that is also applied to the container platforms [42]. Despite its easy application, it suffers from specific limitations. For instance, umask can identify only three roles—owner, owner group, and others. Therefore, umask cannot control access provided to specific users or user groups. Additionally, umask always provides access to superuser accounts, such as root or administrator accounts [43]. This compromises the owner's understanding of the users and user-group organizations accessing the files/directories [44]. Most importantly, umask is application-specific within containers. When a user file is uploaded to a container, configuring umask for each uploaded file is inconvenient. Additionally, as diverse applications can be uploaded to containers, the application service manager is required to understand all the application characteristics to configure umask correctly, which is practically impossible.

SecureOS mitigates a few of the aforementioned limitations of umask. For instance, SecureOS uses the access control policy to control access to the files or directories of specific subjects such as users and user groups. Moreover, the SecureOS policy can deny access to the container. However, no SecureOS system, including SELinux, AppArmor, and GPO, considers the container environment [45–47]. As a result, access to the file/directory located inside containers cannot be denied using SecureOS.

Thus, the development of a dedicated mechanism for container file access control and its incorporation within system security frameworks, such as SecureOS, is essential to protect container files. Unfortunately, no such mechanism has been proposed yet.

3. Kernel-Based Container File Access control Architecture

3.1. Design Considerations

Typically, the protected object and accessing subject must be uniquely identifiable to ensure access control.

The accessing subject is the user account (user ID) or the user group account (group ID) of the host OS. Therefore, legacy access control techniques adopt the user account as the subject of the access control policy. However, this practice cannot be adopted in a container platform system because of its isolated nature. Figure 3 indicates that the same user account may exist both in the host OS as well as the container on a container platform [48]. The container platform internally uses the namespace technique, which ensures the independence of user accounts and files/directories in the container. Although the user accounts in the host OS and container have identical user IDs, the container platform is processed as a different account by namespace. Namespace separates user IDs into session IDs (*SIDs*) and independently processes each *SID*. This isolation feature is the most significant advantage of container platforms. However, access control suffers from specific disadvantages in this case. As mentioned above, an identifier is required to be used as the subject of the file in the container's access control policy because of the non-uniqueness of the user account.

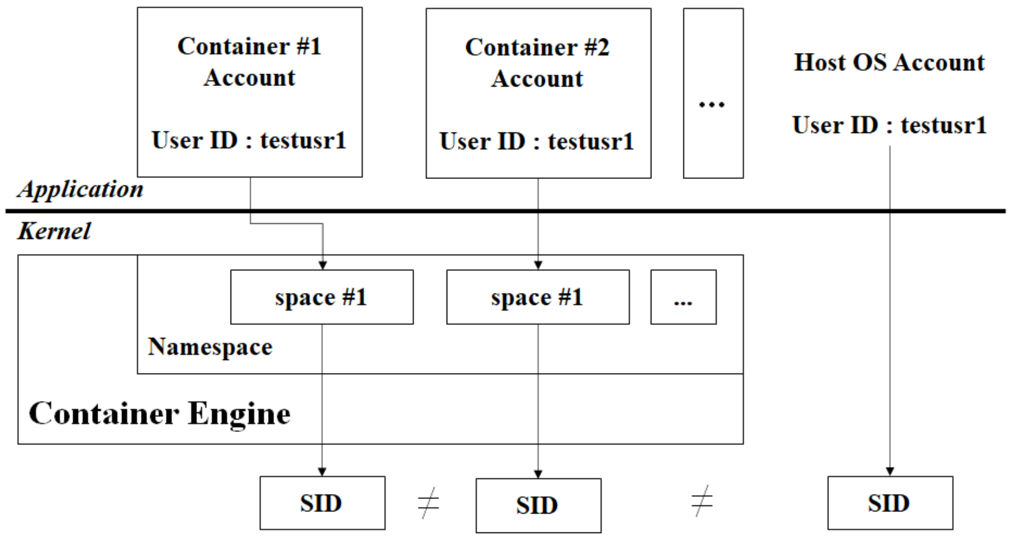

Figure 3. Scenario in which the same account may exist at multiple positions in the container platform.

Legacy file access control techniques, such as SELinux, AppArmor, and GPO, use the absolute path to the requested file as the object. However, it cannot be used as an object in container platforms. Figure 4 illustrates the difference between a file path (virtual file path, *Vf*) known to the accessing user and a real path (absolute file path, *Af*) [49]. This difference can be attributed to the namespace of the container platform. Therefore, the object of the access control policy must be a real path for the access control policy to be valid with respect to container files [50].

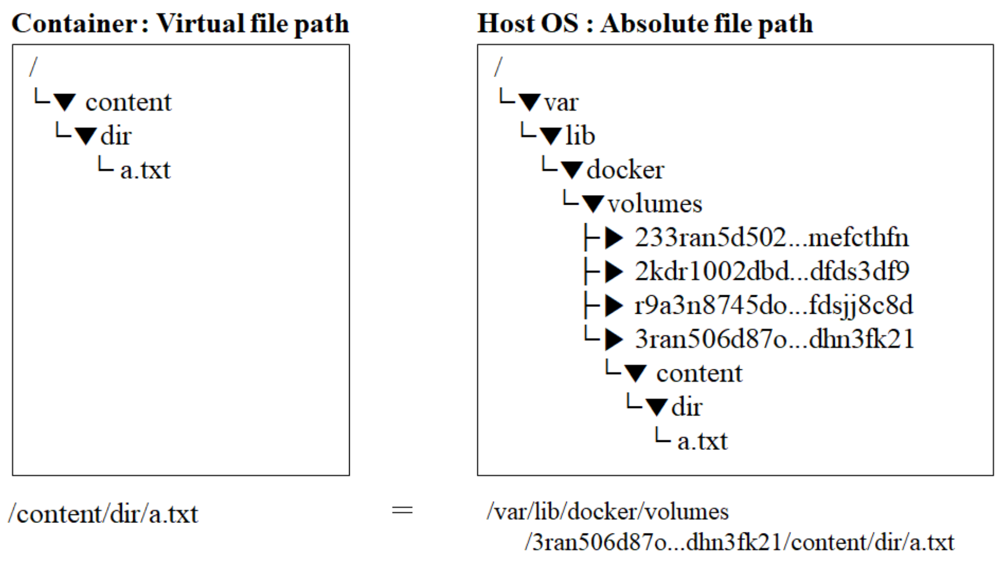
Container : Virtual file path
/
└▼ content
└▼dir
└ a.txt
Host OS : Absolute file path
/
└▼var
└▼lib
└▼docker
└▼volumes
├▶ 233ran5d502...mefcthfn
├▶ 2kdr1002dbd...dfds3df9
├▶ r9a3n8745do...fdsjj8c8d
└▶ 3ran506d87o...dhn3fk21
└▼ content
└▼dir
└ a.txt
/content/dir/a.txt
=
/var/lib/docker/volumes
/3ran506d87o...dhn3fk21/content/dir/a.txt

Figure 4. Difference between virtual and absolute file paths.

However, because typical real paths are considerably long and contain random strings, it is inconvenient to use them during the configuration of the access control policy. Moreover, it violates certain security principles. Therefore, the file path known to the user is used to register the access control policy for the container files. However, policy registration requires the conversion of a file path known to the user (*Vf*) to a real path (*Af*).

3.2. Access Control Policy Structure

The enforcement of access control must satisfy the security principles of non-bypassability and single-point enforcement. To achieve these, kernel-level implementation is essential, which facilitates the acquisition of information to identify a user's file-access event. At this level, the user's file-access event is processed by an interrupt function, and detailed information is registered in the interrupt request queue (IRQ). The *SID* of the user account and the *Af* to be accessed are uniquely identified in the IRQ; this information can be used as a component of the access control policy.

In general, access control policies should be easy to remember and implement. However, remembering *SID*s may be difficult for users. Because the *SID* of an account in a container cannot be obtained from the host OS, it cannot be used as an access control policy. Similarly, *Af* is unknown to the user; thus, it cannot be included in an access control policy because it depends on container initialization.

In this study, the container ID (*Ct*) was used as an additional identifier to facilitate the identification of files in the user container. Figure 5 illustrates the structure of the access control policy used to identify the access events associated with container files.

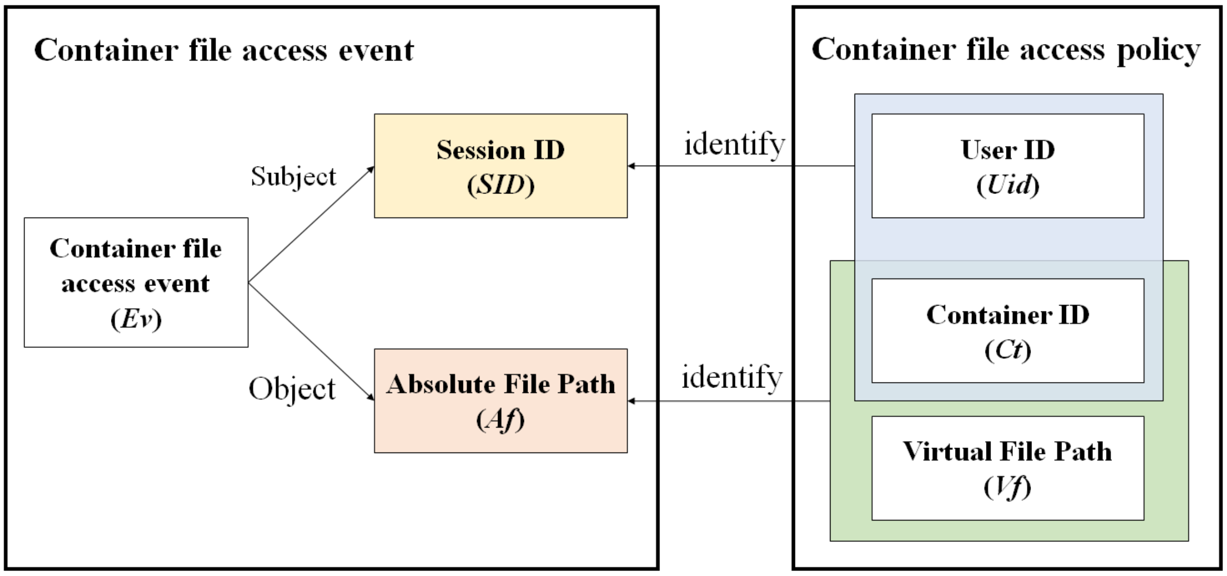

Figure 5. Structure of the container file access control policy.

The container file-access event (*Ev)* includes *SID* and *Af*. *SID* is generated when the user logs in, but it changes with every login and, thus, is not suitable to be used as an element in the container file-access policy. Therefore, the user ID (*Uid*) of the container and the container ID (*Ct*) are used to identify the container *SID*. Because of its high length, *Af* violates the security principle regarding convenience. Therefore, *Ct* and *Vf* are used to identify the *Af* of a container file.

To define the container access control mechanism, the container file access control policy and container file access event are defined as follows:

The container file access control policy is denoted by P
Container file access event is denoted by Ev
Access control policy enforcement is denoted by ACE
Access control policy enforcement result is denoted by PER
The e in which the elements of e and P match is denoted by Ev_P.

PER is defined as follows:

$$PER = ACE(Ev) = \begin{cases} 0, \; Ev \notin \mathrm{Ev}_P \\ 1, \; Ev \in \mathrm{Ev}_P \end{cases}$$

PER can have one of two outcomes—allow or deny. The following notations are used for the two cases:

The 'deny' outcome of PER is denoted by PER_1
The 'allow' outcome of $PER$is denoted by PER_0

P comprises three elements—*Ct*, *Vf*, and *Uid.* The following notations are used for them:

Ctdefined in access $-$ control policy,P, is denoted byCt_p
Vf defined in access control policy,P, is denoted by Vf_p
Uid defined in access control policy, P, is denoted by Uid_p

PER_1 can be expressed as follows:

$$PER_1 = ACE(Ev_P) = ACE(Ev_{Ct_p \bullet Vf_p \bullet Uid_p}) \tag{1}$$

Additionally, when the elements do not match those of the access control policy, the access request is allowed, corresponding to the outcome, PER_0, as follows:

$$PER_0 = ACE(Ev_{\overline{P}}) = ACE(Ev_{\overline{Ct_p}} + Ev_{\overline{Vf_p}} + Ev_{\overline{Uid_p}}) \tag{2}$$

This is obtained by applying De Morgan's law to Equation (1).

Equations (1) and (2) indicate that the user is denied access if all the three elements (i.e., *Vf*, *Ct*, and *Uid*) match with the parameters of the access control policy. Conversely, access is provided if any one of them does not match.

Therefore, if Equations (1) and (2) hold, the proposed intra-container file access control architecture can be considered to be valid.

3.3. Access Control Architecture for Container Files

The proposed access control architecture for container files is designed to satisfy the following conditions:

1. Preserve container features.
2. Minimize changes in the container platform configuration.
3. Use information known to users that can be easily used.
4. Minimize resources required for providing the access control function.

Considering the specific characteristics of the container platform, we propose a container file access control architecture, whose schematic is shown in Figure 6.

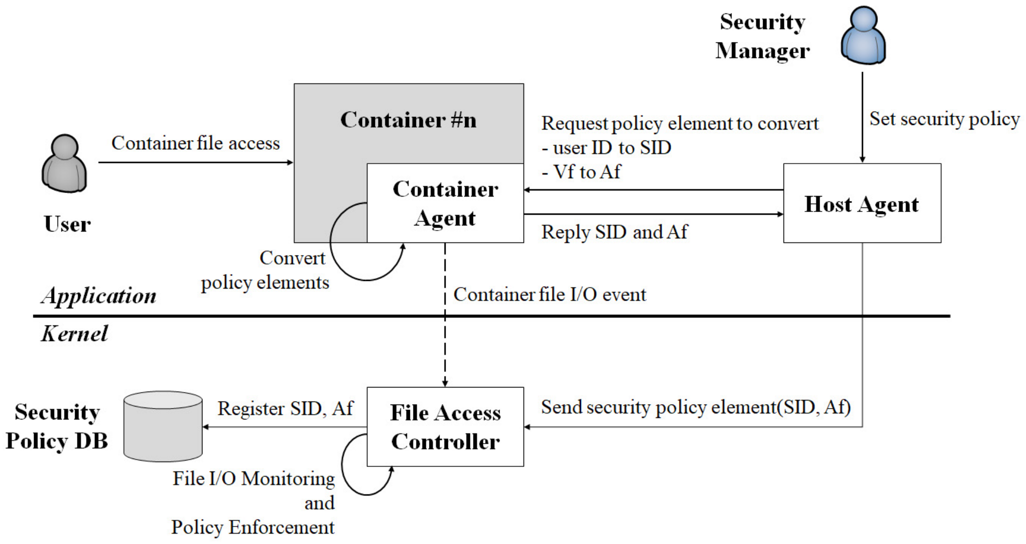

Figure 6. Schematic of the container file access control architecture.

The proposed access control architecture for container files comprises the following three components. The *Host Agent* serves as the interface of the *Security Manager* that selects the access control policy. The *Container Agent* denotes a daemon that converts the access control policy elements to actual information. The *File Access Controller* monitors user access to the container files and enforces access control policies.

The *Security Manager* selects the access control policy based on *Uid*, *Ct*, and *Vf* via the *Host Agent*. The access control policy elements (*Uid*, *Ct*, and *Vf*) are transmitted to the *Container Agent*, which converts the *Uid* and *Vf* into *SID* and *Af*, respectively. Subsequently, the *Container Agent* transmits the *SID* and *Af* to the *Host Agent*, which then transmits them to the *File Access Controller* for the registration of the access control policy. The *File Access Controller* receives the access control policy from the *Host Agent* and registers it as a kernel-based access control policy.

When a user accesses a container file, the kernel creates an IRQ for the user file in the container access event. The *File Access Controller* monitors the kernel IRQ. If the container file access event node is registered to the kernel IRQ, the *File Access Controller* obtains the *SID* and *Af* corresponding to the event. Subsequently, the *File Access Controller* enforces the access control policy by comparing the *SID* and *Af* with the selected values in the *security policy database*. If a matching set of *SID* and *Af* is identified, the file-access-event node is deleted from the kernel and the STOLEN code is returned. The accessing user is displayed a denial message. Figure 7 depicts a sequential diagram of this process.

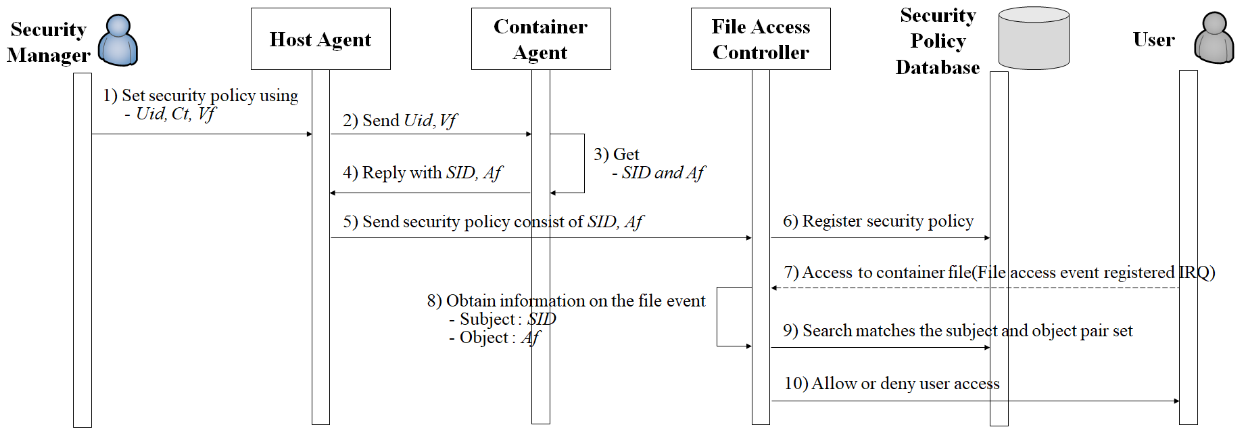
Security Manager
Host Agent
Container Agent
File Access Controller
Security Policy Database
User
1) Set security policy using
- Uid, Ct, Vf
2) Send Uid, Vf
3) Get
- SID and Af
4) Reply with SID, Af
5) Send security policy consist of SID, Af
6) Register security policy
7) Access to container file(File access event registered IRQ)
8) Obtain information on the file event
- Subject : SID
- Object : Af
9) Search matches the subject and object pair set
10) Allow or deny user access

Figure 7. Sequence diagram of providing the container file access control function.

Other file-protection techniques, such as umask or SecureOS, cannot obtain the *Af* corresponding to the container file and, thus, cannot protect container files. By contrast, the architecture proposed in this study obtains the *Af* corresponding to the container file's *Vf*.

4. Implementation on the Linux Platform

To verify the effectiveness of the proposed architecture, we implemented the access control logic in an environment based on CentOS 8.3 (kernel version 4.18.0-394) and the Docker container platform (version 4.2.0). Figure 8 depicts the implemented access control logic.

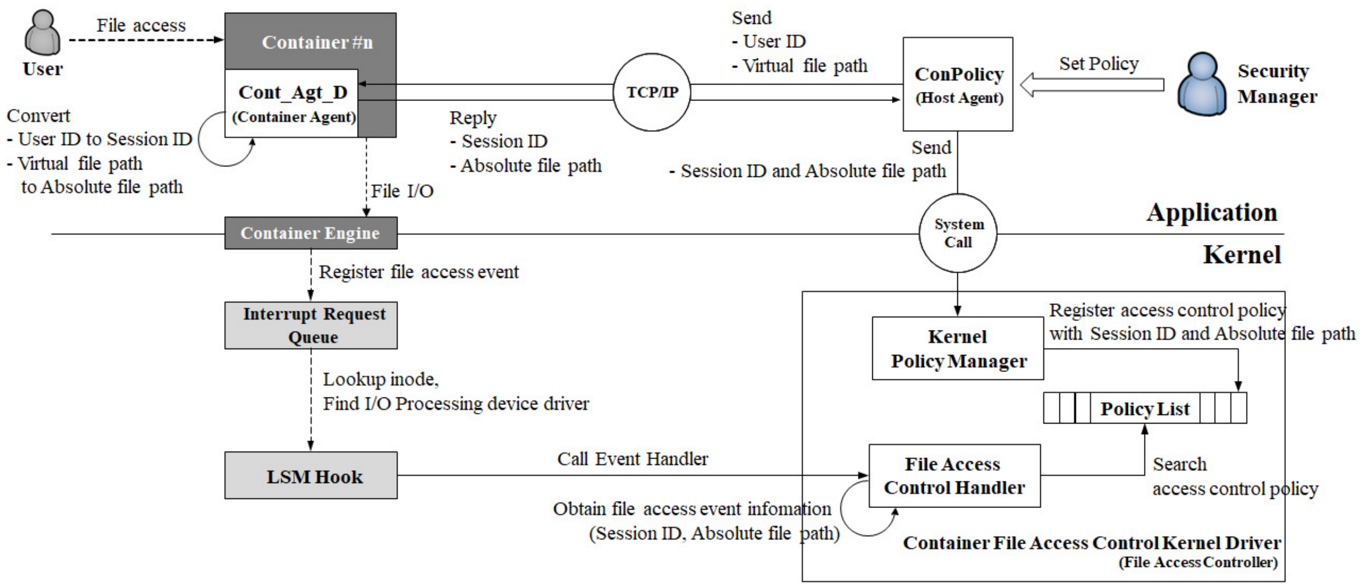

Figure 8. Structure of the container file access control function implemented on Linux.

In the proposed architecture, the *Host Agent* was implemented as *ConPolicy* executed in the Linux shell, whereas the *Container Agent* was implemented as *Cont_Agt_D*, which acted as a server daemon. The *File Access Controller* was implemented as the *Container File Access Control Kernel Driver*, which was divided into policy management (*Kernel Policy Manager*) and enforcement (*File Access Control Handler*) components. *ConPolicy* and *Cont_Agt_D* communicated using the TCP/IP socket protocol, whereas *ConPolicy* and *Container File Access Control Kernel Driver* communicated using system calls. The *Policy List* was implemented as the *Security Policy DB*. *Kernel Policy Manager* served as the system call handler for system calls initiated by the *ConPolicy*. It registered *SID* and *Af* to the *Policy List*. Finally, the *File Access Control Handler* was a kernel module used to enforce the access control policies.

4.1. ConPolicy

ConPolicy was an executable command executed in the Linux shell. Here, *Ct* represents the number used for identifying *Cont_Agt_D*. *ConPolicy* verified *Uid*, *Ct*, and *Vf*. If all the arguments were determined to be accurate, *ConPolicy* transmitted them to *Cont_Agt_D* and waited for a response from *Cont_Agt_D*. Once *ConPolicy* received the *SID* and *Af* from *Cont_Agt_D*, it transmitted them to the *Kernel Policy Manager* via a system call.

4.2. Cont_Agt_D

Cont_Agt_D was a daemon program that served as a *Container Agent*. *Cont_Agt_D* received the *Uid* and *Vf* from *ConPolicy* and obtained the *SID* on this basis. Similarly, *Cont_Agt_D* obtained the *Af* based on the *Ct* and *Vf*. Subsequently, *Cont_Agt_D* transmitted them back to *ConPolicy*.

4.3. Container File Access Control Kernel Driver

4.3.1. Kernel Policy Manager

The *Kernel Policy Manager* in the *Container File Access Control Kernel Driver* received the *SID* and *Af* from *ConPolicy* and registered them onto the *Policy List*. Figure 9 depicts the structure of the *Policy List*. To ensure rapid search in the access control policy, the *Policy List* was considered to be a two-level hashtable based on *Af* and *SID* like SELinux's policy structure.

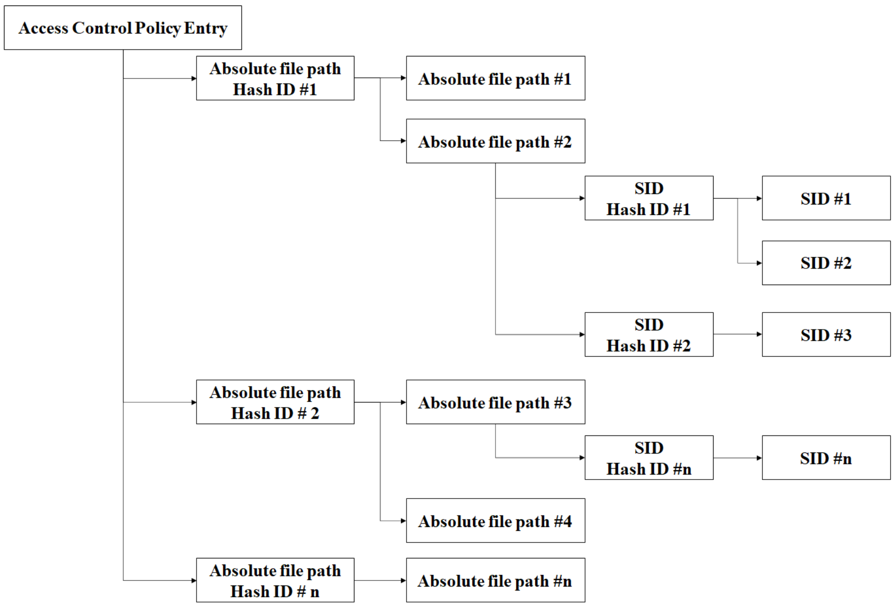

Figure 9. Structure of the *Policy List*.

4.3.2. File Access Control Handler

When a user accessed a container file, the container engine converted *Vf* into *Af* and transmitted it to the kernel. The kernel then generated a file-access-event node and registered it in the IRQ.

The *File Access Control Handler* is a kernel function used to enforce the access control policy. Typically, it registers a file access event hooking function onto the Linux security module [51]. When a file access event occurs, the file access event hooking function obtains the corresponding object and subject information [52]. Subsequently, it calls the access control policy to enforce the function registered in the *File Access Control Handler*.

In this study, the *File Access Control Handler* search-policy node in the *Policy List* corresponded to the SID and *Af* of the file access event. If a matching set of *Af* and SID were identified in the *Policy List*, the file access event was marked as STOLEN. If the interrupt function was marked as STOLEN in the kernel, its process was halted and the user was denied access to the file in the container.

5. Verification of the Container File Access Control Architecture

5.1. Function Verification Items

To ensure the operational accuracy of the container file access control function of the proposed architecture, we verified the positive and negative items, as listed in Table 1.

Table 1. Functionalities used for verification.

Class	Verification Item	Description
Positive	PEnforce_P1	• Verification of Equation (1) • After selecting the container file access control policy, verify whether file access is denied when the subject and object of the file accessed are identical to those in the registered access control policy.
Negative	PEnforce_N1	• Verification of $PER_0 = ACE(e_{\overline{Ct_p}})$, which is a part of Equation (2) • Verify whether the container ID of the file accessed is different from that in the registered access control policy, even if the *Uid* and *Vf* are identical.
	PEnforce_N2	• Verification of $PER_0 = ACE(e_{\overline{Vf_p}})$, which is a part of Equation (2) • Verify whether the container ID and *Uid* of the file accessed are identical to those in the registered access control policy, even if the *Vf* is different.
	PEnforce_N3	• Verification of $PER_0 = ACE(e_{\overline{Uid_p}})$, which is a part of Equation (2) • Verify whether the container ID and *Vf* of the file accessed are identical to those in the registered access control policy, even if the *Uid* is different.

5.2. Performance Verification

Ideally, the container file access control architecture should not interfere with other services, and the cost of access control functions should be minimal. To evaluate the performance of the proposed access control architecture from this perspective, we measured the central processing unit (CPU) share (%) required for the policy registration and the enforcement time of the access control policy. The assessment was performed using an Intel i7-8700 CPU with a memory of 32 GB and a hard disk of 10 TB. The performance of the proposed architecture was compared with that of SELinux.

5.2.1. CPU Usage

To measure the CPU usage rate required to register the access control policy, we created a shell script to register multiple access control policies. The policy registration shell script repeatedly applied *ConPolicy* a specified number of times. During the execution of the shell script and the registration of the access control policy, the CPU usage rate was recorded using the Linux top command.

To validate the performance of the proposed architecture, the same test was conducted on SELinux. The performances of the two approaches were compared with respect to the CPU usage rates.

Table 2 summarizes the mean CPU usage rates and the corresponding standard deviations over 10 iterations, amounting to the registration of 500 access control policies in SELinux and the proposed architecture for each.

Table 2. Central processing unit usage rates of SELinux and the proposed architecture.

	1	**2**	**3**	**4**	**5**	**Average**	**Standard deviation**
SELinux policy registration CPU usage rate (%)	3.8	3.6	2.9	3.6	3.4	3.60	0.319
	6	**7**	**8**	**9**	**10**		
	3.3	3.6	4.0	4.0	3.8		
Proposed architecturepolicy registration CPU usage rate (%)	**1**	**2**	**3**	**4**	**5**	**Average**	**Standard deviation**
	4.8	5.1	4.4	3.9	4.4	4.57	0.445
	6	**7**	**8**	**9**	**10**		
	4.2	4.6	4.0	5.1	5.2		

The results indicate that the CPU usage rate required for the policy registration was slightly higher (4.57%) in the case of the proposed architecture than that for SELinux (3.60%). This increment was attributed to the different tasks performed during the policy registration process. SELinux directly sets the *Af* while registering an access control policy. In contrast, a *Vf* is used while registering the access control policy by the proposed architecture. This heightens the CPU usage because the *Host Agent* is required to convert the *Vf* into an *Af*.

5.2.2. Policy Enforcement Time

To minimize the enforcement time of the container file-access control policy, we measured the final policy enforcement times when multiple access control policies were registered in the *Policy List* simultaneously. A dummy policy is a policy that is not actually used and only serves to delay the final policy enforcement time. In this experiment, it was generated by a script created for performance measurement by combining a random *Vf*, a random container ID, and a random *Uid*. The measured policy enforcement time of the proposed architecture was compared with that of SELinux to evaluate the performance of the former.

The numbers of dummy container file access control policies used to measure the final policy enforcement time were considered to be 500, 1000, 2000, 4000, 5000, and 6000. The final policy enforcement time was measured considering the starting time of the matching access control policy application.

Figure 10 illustrates the average times required to register the final access control policy during 10 iterations of registering multiple dummy access control policies.

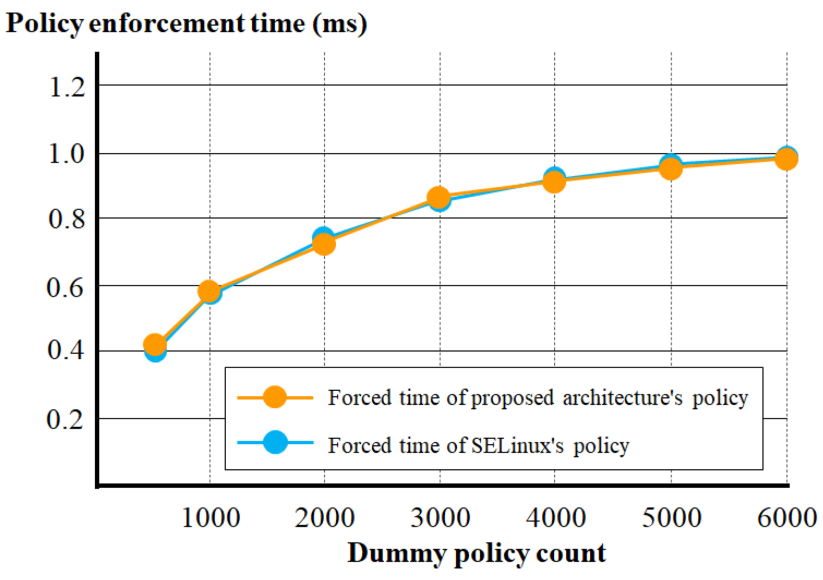

Figure 10. Security policy enforcement times of SELinux and the proposed architecture.

As shown in Figure 10, the policy enforcement time of the proposed container file access architecture is almost the same as that of SELinux.

5.3. Verification Analysis

In the functional verification step, we confirmed that the proposed access control architecture denied only those events that matched the container file access control policy and allowed the rest. Thus, based on the access control policy set by the security manager, authorized users can access sensitive files, but unauthorized users cannot.

Additionally, we determined that the CPU usage rate required to register the access control policy and the enforcement time of the proposed architecture were slightly higher than those of SELinux. This is attributed to the methodological differences between the two architectures. SELinux uses Af as the object element of the security policy. In contrast, the proposed architecture uses Vf and the container ID. These are combined to obtain the Af of the target file, which is then added to the policy database. Thus, the more complicated policy registration task of the proposed system naturally requires more time and a higher CPU usage rate than SELinux.

Depending on the number of dummy policies, there are cases where the policy enforcement times of SELinux or the proposed container file access control architecture are earlier or later than each other. However, the policy enforcement times of the two security techniques were confirmed to be virtually the same. This is because the policy-list structure of the proposed container file-access control architecture and the policy structure of SELinux are all hashtable structures. Although the process of calculating hash ID one more than SELinux's policy structure was added, it was confirmed that this process did not significantly affect the policy enforcement time.

Therefore, the proposed access control architecture can be considered to be sufficiently effective in terms of functionality and performance.

6. Conclusions

Container platform-based information services ensure easy distribution and installation of applications. The numerous advantages of container platforms, such as dynamic resource-sharing functions and fault tolerance functions, have promoted the development of information services based on container platforms as well as research on expanding their scope.

To ensure stable operation of container platform-based information services, important files of container-based applications must be protected. However, legacy file access control techniques do not support container platforms.

Therefore, in this study, a container file access control architecture comprising three components and a feature implemented at the kernel level was proposed. The proposed architecture was implemented in a Linux environment to verify the effectiveness of its functions and performance. The analysis confirmed that the proposed access control function was normally provided, and the performance of the proposed system was satisfactory.

The key contribution of this study is the expansion of the limited access control coverage of legacy security environments to container platforms, which is the first step toward comprehensive container platform security. In particular, the container file access control architecture proposed in this study can monitor user access of container files in real time and prevent unauthorized access. In addition, because the resources used for access control were confirmed to be insignificant, it does not affect other application services. Therefore, the container file access control architecture proposed in this study satisfies hitherto unfulfilled security principles for access control on container files.

Container-based application services can use various resources such as registries, processes, devices, and files/directories. The access control policy is registered in memory. As the policy count increases, the memory usage also increases proportionally. If the number of container files to be protected increases, the kernel memory-usage rate for registering the container-file access control policy increases. This can degrade the system performance. Therefore, an appropriate number of container-file access control policies must be enforced. Although all resources must be protected to provide stable service, this study proposed a protective architecture only for container files. Therefore, future research is necessary to consider the protection of other resources.

Author Contributions: Conceptualization, H.-K.L.; investigation, S.-H.H.; writing—original draft preparation, S.-H.H.; supervision, D.L.; writing—review and editing, D.L.; project administration, D.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by Tongmyong University Research, grant number 2021A017.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ferreira, A.P.; Sinnott, R. A performance evaluation of containers running on managed Kubernetes services. In Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Bangkok, Thailand, 13–16 December 2019; pp. 199–208. [CrossRef]
2. Casalicchio, E.; Iannucci, S. The state-of-the-art in container technologies: Application, orchestration and security. *Concurr. Comput. Pract. Exper.* **2020**, *32*, e5668. [CrossRef]
3. Sabharwal, N.; Pandey, P. Container image management using Google container registry. In *Pro Google Kubernetes Engine*; Apress: Berkeley, CA, USA, 2020; pp. 65–96. [CrossRef]
4. Koschmieder, L.; Hojda, S.; Apel, M.; Altenfeld, R.; Bami, Y.; Haase, C.; Lin, M.; Vuppala, A.; Hirt, G.; Schmitz, G.J. AixViPMaP®—An operational platform for microstructure modeling workflows. *Integr. Mater. Manuf. Innov.* **2019**, *8*, 122–143. [CrossRef]
5. Becker, S.; Schmidt, F.; Kao, O. EdgePier: P2P-based container image distribution in edge computing environments. In Proceedings of the IEEE International Performance, Computing, and Communications Conference (IPCCC), Computing, Austin, TX, USA, 29–31 October 2021; pp. 1–8. [CrossRef]
6. Ma, S.; Jiang, J.; Li, B.; Li, B. Maximizing container-based network isolation in parallel computing clusters. In Proceedings of the 24th International Conference on Network Protocols (ICNP), Singapore, 8–11 November 2016; pp. 1–10. [CrossRef]
7. Mampage, A.; Karunasekera, S.; Buyya, R. Deadline-aware dynamic resource management in serverless computing environments. In Proceedings of the 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Melbourne, Australia, 10–13 May 2021; pp. 483–492. [CrossRef]
8. Salhab, N.; Rahim, R.; Langar, R. NFV orchestration platform for 5G over on-the-fly provisioned infrastructure. In Proceedings of the IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Paris, France, 29 April 2019; pp. 971–972. [CrossRef]
9. Kim, B.S.; Lee, S.H.; Lee, Y.R.; Park, Y.H.; Jeong, J. Design and implementation of cloud docker application architecture based on machine learning in container management for smart manufacturing. *Appl. Sci.* **2022**, *12*, 6737. [CrossRef]
10. Ngo, M.V.; Luo, T.; Hoang, H.T.; Ouek, T.Q.S. Coordinated container migration and base station handover in mobile edge computing. In Proceedings of the GLOBECOM IEEE Global Commun. Conference, Taipei, Taiwan, 7–11 December 2020; Volume 2020, pp. 1–6. [CrossRef]
11. Huh, J.H. Implementation of lightweight intrusion detection model for security of smart green house and vertical farm. *Int. J. Distrib. Sens. Netw.* **2018**, *14*, 1550147718767630. [CrossRef]
12. Wong, A.Y.; Chekole, E.G.; Ochoa, M.; Zhou, J. Threat Modeling and Security Analysis of Containers: A Survey. *arXiv* **2021**, arXiv:2111.11475.
13. Westfall, J. Basics of Linux security. In *Set Up and Management Your Virtual Private Server*; Apress: Berkeley, CA, USA, 2021; pp. 111–131. [CrossRef]
14. Kim, D.K.; Ming, H.; Lu, L. Reflection on building hybrid access control by configuring RBAC and MAC features. In Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 18–21 February 2020; pp. 522–526. [CrossRef]
15. Mullinix, S.P.; Konomi, E.; Townsend, R.D.; Parizi, R.M. On Security Measures for Containerized Applications Imaged with Docker. *arXiv* **2020**, arXiv:2008.04814.
16. Kaiser, S.; Haq, M.S.; Tosun, A.S.; Korkmaz, T. Container technologies for ARM architecture: A comprehensive survey of the state-of-the-art. *IEEE Access.* **2022**, *10*, 84853–84881. [CrossRef]
17. Han, S.H.; Lee, H.K.; Lee, S.T.; Kim, S.J.; Jang, W.J. Container image access control architecture to protect applications. *IEEE Access* **2020**, *8*, 162012–162021. [CrossRef]

18. Setiadi, D.R.I.M. PSNR vs SSIM: Imperceptibility quality assessment for image steganography. *Multimed. Tools Appl.* **2021**, *80*, 8423–8444. [CrossRef]
19. Ge, Y.; Ding, Z.; Tang, M.; Tian, Y.C. Resource provisioning for mapreduce computation in cloud container environment. In Proceedings of the 18th International Symposium on Network Computing and Applications (NCA), Cambridge, MA, USA, 26–28 September 2019; pp. 1–4. [CrossRef]
20. Kelley, R.; Antu, A.D.; Kumar, A.; Xie, B. Choosing the right compute resources in the cloud: An analysis of the compute services offered by Amazon, Microsoft and Google. In Proceedings of the International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), Chongqing, China, 29–30 October 2020; pp. 214–223. [CrossRef]
21. Sokolowski, D.; Weisenburger, P.; Salvaneschi, G. Automating serverless deployments for DevOps organizations. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on The Foundations of Software Engineering, Athens, Greece, 23–28 August 2021; pp. 57–69. [CrossRef]
22. Buchanan, S.; Rangama, J.; Bellavance, N. Container registries. In *Introducing Azure Kubernetes Service*; Apress: Berkeley, CA, USA, 2020; pp. 17–34. [CrossRef]
23. Aziz Shah, A.; Piro, G.; Alfredo Grieco, L.; Boggia, G. A quantitative cross-comparison of container networking technologies for virtualized service infrastructures in local computing environments. *Trans. Emerg. Tel. Technol.* **2021**, *32*, e4234. [CrossRef]
24. Karn, R.R.; Kudva, P.; Huang, H.; Suneja, S.; Elfadel, I.M. Cryptomining detection in container clouds using system calls and explainable machine learning. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *32*, 674–691. [CrossRef]
25. Duan, J. Design and implementation of vulnerability attack and utilization platform based on container virtualization. In Proceedings of the 3rd International Acad. Exch. Conference on Science and Technology Innovation (IAECST), Guangzhou, China, 10–12 December 2021; pp. 368–372. [CrossRef]
26. Sun, Y.; Lei, J.; Shin, S.; Lu, H. Baoverlay: A block-accessible overlay file system for fast and efficient container storage. In Proceedings of the 11th ACM Symposium on Cloud Computing, Virtual Event, 19–21 October 2020; pp. 90–104. [CrossRef]
27. Hussein, M.K.; Mousa, M.H.; Alqarni, M.A. A placement architecture for a container as a service (CaaS) in a cloud environment. *J. Cloud Comp.* **2019**, *8*, 1–15. [CrossRef]
28. Zhang, Y.; Fu, Y.; Li, G. Research on container throughput forecast based on Arima-BP neural network. *J. Phys. Conf. Ser.* **2020**, *1634*, 012024. [CrossRef]
29. Kulkarni, S.G.; Liu, G.; Ramakrishnan, K.K.; Arumaithurai, M.; Wood, T.; Fu, X. Reinforce: Achieving efficient failure resiliency for network function virtualization based services. In Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies, Heraklion, Greece, 4–7 December 2018; pp. 41–53. [CrossRef]
30. Huh, J.H.; Seo, K. A case study of the base technology for the smart grid security: Focusing on a performance improvement of the basic algorithm for the DDoS attacks detection using Cuda. *J. Korea Multimed. Soc.* **2016**, *19*, 411–417. [CrossRef]
31. Kim, S.K.; Kim, U.M.; Huh, J.H. A study on improvement of blockchain application to overcome vulnerability of IoT multiplatform security. *Energies* **2019**, *12*, 402. [CrossRef]
32. Javed, O.; Toor, S. Understanding the Quality of Container Security Vulnerability Detection Tools. *arXiv* **2021**, arXiv:2101.03844.
33. Putra, G.D.; Dedeoglu, V.; Kanhere, S.S.; Jurdak, R. Trust management in decentralized iot access control system. In Proceedings of the IEEE International Conference on Blockchain and Cryptocurrency (ICBC), Toronto, ON, Canada, 4–7 May 2020; pp. 1–9. [CrossRef]
34. Reeves, M.; Tian, D.J.; Bianchi, A.; Celik, Z.B. Towards improving container security by preventing runtime escapes. In Proceedings of the IEEE Sec. Dev. Conference (SecDev), Atlanta, GA, USA, 18–20 October 2021; pp. 38–46. [CrossRef]
35. Lo Iacono, L.; Smith, M.; Zezschwitz, E.; Gorski, P.L.; Nehren, P. Consolidating principles and patterns for human-centred usable security research and development. In Proceedings of the European Workshop on Usable Security, London, UK, 24–26 April 2018. [CrossRef]
36. Park, N.K.; An, Y. A study of rent fee assessment on the port railway station: The litigation case study of a Korean container terminal. *J. Mar. Sci. Eng.* **2022**, *10*, 1090. [CrossRef]
37. Nam, S.M. A fuzzy rule-based system for automatically generating customized training scenarios in cyber security. *J. Korea Soc. Comput. Inf.* **2020**, *25*, 39–45. [CrossRef]
38. Vyas, P.; Shyamasundar, R.K.; Patil, B.; Borse, S.; Sen, S. SP*: An information flow secure Linux. In Proceedings of the IEEE International Conference on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), New York, NY, USA, 30 September 2021; pp. 1603–1612. [CrossRef]
39. Han, S.H.; Lee, D. Kernel-based real-time file access monitoring structure for detecting malware activity. *Electronics* **2022**, *11*, 1871. [CrossRef]
40. Cinque, M.; Cotroneo, D.; De Simone, L.; Rosiello, S. Virtualizing mixed-criticality systems: A survey on industrial trends and issues. *Future Gener. Comput. Syst.* **2022**, *129*, 315–330. [CrossRef]
41. Rossi, M.; Facchinetti, D.; Bacis, E.; Rosa, M.; Paraboschi, S. {SEApp}: Bringing mandatory access control to Android apps. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), Virtual Event, 11–13 August 2021; pp. 3613–3630.
42. Rothwell, W. Filesystem and process control. In *Beginning Perl Programming*; Apress: Berkeley, CA, USA, 2019; pp. 165–174. [CrossRef]

43. Mathas, C.M.; Vassilakis, C.; Kolokotronis, N.; Zarakovitis, C.C.; Kourtis, M.A. On the design of IoT security: Analysis of software vulnerabilities for smart grids. *Energies* **2021**, *14*, 2818. [CrossRef]
44. Kim, H.; Hahn, C.; Hur, J. Real-time detection of cache side-channel attack using non-cache hardware events. In Proceedings of the International Conference on Information Networking (ICOIN), Jeju Island, Republic of Korea, 13–16 January 2021; pp. 28–31. [CrossRef]
45. Ko, J.Y.; Lee, S.G.; Lee, C.H. Real-time mandatory access control on SELinux for Internet of Things. In Proceedings of the IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, 11–13 January 2019; pp. 1–6. [CrossRef]
46. Zhu, H.; Gehrmann, C. Kub-Sec, an automatic Kubernetes cluster AppArmor profile generation engine. In Proceedings of the 14th International Conference on Communication Systems & Networks (COMSNETS), Bangalore, India, 4–8 January 2022; pp. 129–137. [CrossRef]
47. Babu, M.V.; Suman, K.N.; Srinivasa Rao, P. Drafting software as a practicing tool for engineering drawing-based courses: Content planning to its evaluation in client–server environment. *Int. J. Mech. Eng. Educ.* **2019**, *47*, 118–134. [CrossRef]
48. Sparks, J. Enabling docker for HPC. *Concurr. Computat. Pract. Exper.* **2019**, *31*, e5018. [CrossRef]
49. Wofford, Q.; Bridges, P.G.; Widener, P. A layered approach for modular container construction and orchestration in HPC environments. In Proceedings of the 11th Workshop on Scientific Cloud Computing, Renton, WA, USA, 21 June 2020; pp. 1–8. [CrossRef]
50. Lyu, T.; Atmojo, U.D.; Vyatkin, V. Towards cloud-based virtual commissioning of distributed automation applications with IEC 61499 and containerization technology. In Proceedings of the IECON, 2021–47th Annual Conference of the IEEE Industrial Electronics Society, Toronto, ON, Canada, 13–16 October 2021; pp. 1–7. [CrossRef]
51. Ecarot, T.; Dussault, S.; Souid, A.; Lavoie, L.; Ethier, J.F. AppArmor for health data access control: Assessing risks and benefits. In Proceedings of the 7th International Conference on Internet of Things: Systems, Management and Security (IOTSMS), Paris, France, 14–16 December 2020; pp. 1–7. [CrossRef]
52. Kang, H.; Kim, J.; Shin, S. Minicon: Automatic enforcement of a minimal capability set for security-enhanced containers. In Proceedings of the IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS), Toronto, ON, Canada, 21–24 April 2021; pp. 1–5. [CrossRef]