Front. Comput. Sci., 2025, 19(5): 195106
https://doi.org/10.1007/s11704-024-2568-8

RESEARCH ARTICLE

System log isolation for containers

Kun WANG[1,2], Song WU (✉)[1], Yanxiang CUI[1], Zhuo HUANG[1], Hao FAN[1], Hai JIN[1]

1 National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

2 State Key Laboratory of Complex & Critical Software Environment, College of Information and Communication, National University of Defense Technology, Wuhan 430019, China

Abstract Container-based virtualization is increasingly popular in cloud computing due to its efficiency and flexibility. Isolation is a fundamental property of containers and weak isolation could cause significant performance degradation and security vulnerability. However, existing works have almost not discussed the isolation problems of system log which is critical for monitoring and maintenance of containerized applications. In this paper, we present a detailed isolation analysis of system log in current container environment. First, we find several system log isolation problems which can cause significant impacts on system usability, security, and efficiency. For example, system log accidentally exposes information of host and co-resident containers to one container, causing information leakage. Second, we reveal that the root cause of these isolation problems is that containers share the global log configuration, the same log storage, and the global log view. To address these problems, we design and implement a system named *private logs* (POGs). POGs provides each container with its own log configuration and stores logs individually for each container, avoiding log configuration and storage sharing, respectively. In addition, POGs enables private log view to help distinguish which container the logs belong to. The experimental results show that POGs can effectively enhance system log isolation for containers with negligible performance overhead.

Keywords container isolation, system log, cgroup, namespace, cloud computing

1 Introduction

Containers are increasingly popular in cloud computing due to the simplicity of design and natural support for micro-service and serverless computing [1–4]. Different from the traditional virtual machines (VMs), containers share the underlying host kernel without Virtual Machine Monitor (VMM) and Guest OS. This difference helps containers attain high performance and low footprint but sacrifice isolation. Containers are isolated from each other through kernel isolation primitives, such as *cgroup* (Control group) [5,6] and *namespace* [7,8] in Linux kernel. *Cgroup* helps containers realize resource isolation by accounting for and limiting resource consumption. *Namespace* allows each container to have its own isolated view of PID (Process Identification), IPC (Inter-Process Communication), network, file system, etc.

Recent studies [9–15] have shown isolation is a fundamental property of containers and indicated weak isolation could cause significant performance degradation and security vulnerability. To enhance container isolation, researchers proposed to optimize the kernel isolation primitives for the specific isolation problems [16–21]. For example, new namespaces, Sys-namespace [16] and Security namespace [17], are proposed to enhance resource view isolation and security isolation, respectively. Khalid et al. [19] optimize the CPU cgroup to enforce hardened CPU isolation. However, existing works have not isolated system log that is critical for monitoring and maintaining containerized applications. System log records detailed information about system activities and helps track down problems of OS kernel and applications [22]. System log is shared among containers, potentially introducing isolation problems. For example, one container can read the information belonging to another container, thus causing information leakage.

Current OS kernel leverage a privilege-based mechanism to enable access control on the shared system log [23], which cannot provide strict system log isolation for containers. Generally, the OS kernel only allows privileged users to perform operations on the shared system log. This prevents non-privileged containers from accessing the shared system log, diminishing the likelihood that isolation problems occur. However, any user is able to launch a privileged container easily. For example, the most popular container system, Docker [24], can run a privileged container by setting the option `--privileged=true`. The typical cloud service, Elastic Kubernetes Service (EKS) in Amazon AWS, uses `eks.privileged` as the default pod security policy to deploy containers, allowing containers to run as a privileged user [9]. As a result, containers can break the privilege-based access

Received September 6, 2022; accepted January 15, 2024

E-mail: wusong@hust.edu.cn

control by obtaining the root privilege, causing the system log isolation is not firm and strict. This motivates us to analyze the reasons and potential impacts of weak system log isolation and enhance system log isolation for containers.

This paper performs a detailed analysis of isolation problems that happen in log generation, access, and analysis. In log generation phase, the log configuration set by one container may conflict with that set by another container, leading to containers fail to use the log configuration. Similarly, log operations performed by one container can conflict with operations from another container, making system log useless in container environment. In log access phase, one container can view information that does not belong to its respective namespace, breaking view isolation. And system log might accidentally expose information of host and co-resident containers to one container, causing additional security concerns. In log analysis phase, because the log storage of one container can be occupied by another container, containers suffer from unexpected log loss, reducing the accuracy of log analysis. And one container cannot distinguish its own logs and must analyze redundant logs, reducing the efficiency and accuracy of log analysis. We further reveal that the root cause of these log isolation problems is sharing system log service among containers, including shared log configuration, shared log storage, and shared log view.

Based on our analysis, we propose a system named *private logs* (POGs) to address the isolation problems. The key idea is to provide each container with an individual system log service and avoid sharing log configuration, log storage, and log view among different containers. Specifically, each container is equipped with an individual POG where one container can create and set its own log configuration and log storage. And each POG can provide a private log view for its respective container. In addition, we design two key components: *pogctl* and *pogshim* to manage the POGs. One container can leverage *pogctl* to control and configure its own POG. *Pogshim* identifies the belonger of POGs and manages the life cycle of POGs.

In summary, our major contributions are as follows:

- We make a detailed analysis of system log isolation problems in current container environment. We reveal the root causes and the possible impacts of such isolation problems.
- We propose POGs to enhance system log isolation for containers. Our system provides each container with an individual POG that consists of private log configuration, storage, and view.
- We implement and evaluate POGs on Linux kernel 5.11.6 and Docker 20.10.12. Experimental results demonstrate that POGs can effectively enhance system log isolation for containers with negligible performance overhead.

2 Background and motivation

In this section, we first give an overview of container system and briefly describe the techniques that are used to realize isolation for containers. Then we introduce the background knowledge of system log and how current containers work with system logs. As Docker [24] is the most popular container system, we describe these backgrounds in the context of Docker and Linux. Other container systems and OSes employ similar techniques.

2.1 Containers

Containers, such as Docker [24], Rkt [25], LXC [26], have received much attention from academia and industry and are quickly becoming commodities. A container is essentially a group of processes that share the same host OS kernel, thus eliminating most of the performance overhead that VMs suffer from. Despite their efficiency and flexibility, containers suffer from a major problem: weak isolation. Here we introduce two key techniques in Linux kernel, *cgroup* and *namespace*, that help containers realize isolation.

Cgroup helps containers attain the ability of resource isolation, which is a fundamental property of containers. *Cgroup* can account for and limit resource consumption of one container, preventing a single container from exhausting host resources. As a result, guarantees and fair shares are preserved for other containers. Such host resources include CPU, memory, disk, network, PID, etc. More specifically, *cgroup* controls the resources usage for containers in two ways: *share* and *limit* [16]. *Share* refers to the priority of containers when competing for shared resources. For example, the CPU subsystem uses *share* to control the time slice that is assigned to one container for CPU execution (e.g., the container with *share* set to 2 can use twice as much CPU time as the container with *share* set to 1). And *Limit* sets the maximum amount of resources that one container can consume. For example, the PID subsystem is used to prevent PID (a logic resource in OS kernel) exhaustion by setting the maximum number of processes one container can create.

Namespace allows each container to have its own isolated view of the host system. A namespace wraps a system view in an abstraction that makes it appear to processes within the corresponding container. Changes to the system view are visible to processes that are members of the namespace, but are invisible to processes in other namespaces. There are seven namespaces in the current Linux kernel, including mount namespace, network namespace, PID namespace, user namespace, IPC namespace, UTS namespace, and cgroup namespace.

The mount namespace can be used to isolate filesystem: multiple containers can have their local root filesystem, backed by different physical directories in the host. The network namespace allows containers to use separate virtual network devices, IP addresses, ports, and IP routing tables. The PID namespace provides processes in a container with virtual PIDs that are mapped to the real PIDs in the host OS. The user namespace maps a root user inside a container to a non-privileged user on the host. The IPC namespace allows containers to have their views of IPC resources, such as signals, pipes, and shared memory. The UTS namespace allows containers to have their hostname and domain name. The cgroup namespace isolates the view of the virtual cgroup filesystem for containers.

2.2 System log

System log plays an important role in software development and maintenance. It records detailed information about system (kernel and applications) activities. As an example shown in Fig. 1, a typical log message includes occurrence time, importance level, user, and event description. Containers normally use such information to monitor behaviors and track down problems for containerized applications. Unfortunately, containers may suffer from isolation problems due to the shared system log that includes shared configuration, shared storage, and shared view.

Shared configuration In order to use system log well, users can modify the configuration (*/etc/syslog.conf*) to customize the log service. For example, users can control which logs should be recorded by setting the log level, which indicates the importance of log messages, including *info, notice, warn, err, crit, alert, emerg,* etc. If we set the level to *err*, the logs whose level is higher than *err* would be printed out. Unfortunately, the log configuration set by one container may conflict with that set by another container because it is global and shared among containers.

Shared storage The OS kernel provides a global system log storage, to which all containers and host print logs. As the volume of system logs grows rapidly, more and more space is used to store them. To reduce the occupation of storage space, the log rotation mechanism in Linux kernel uses ring buffer and a tool named *logrotate* to delete the old system logs. If one container produces a large set of logs, occupying so much storage space, the logs of co-resident containers and host would become old and be deleted. This will introduce unexpected log loss for containers, even affecting the maintenance of containerized applications.

Shared view Similarly, the view of system log is also global and shared so that additional isolation issues may be introduced. One container cannot precisely distinguish the system log messages belonging to itself. All containers have the global view of system log that includes the information of the host and all containers. As a consequence, the shared view of system log might accidentally expose system-wide information to containerized applications. This information leakage channel may expand the attack surface exposed to container users, causing security and privacy concerns for running multiple containers on the same host.

In order to address these isolation problems, current container runtime software normally leverages access control policies to limit the ability of containers to manipulate system log. The Linux kernel only allows privileged users to perform operations on the shared system log and prevents non-privileged containers from accessing the shared system log, reducing the possibility of isolation problems [23]. However, such access control cannot provide strict and firm system log isolation for containers because containers can break the privilege-based access control by obtaining the root privilege. Users are able to launch a privileged container through various methods. First, using the most popular container system, Docker [24], users can run a privileged container by setting the option `--privileged=true`. Second, the typical cloud platform, Amazon AWS, provides Elastic Kubernetes Service (EKS) for users to deploy containers. And the EKS uses `eks.privileged` as the default pod security policy, allowing containers to run as a privileged user [9]. Last, users can make non-privileged containers become privileged through existing privilege escalation attacks [18,27,28].

Fig. 1 An example of system log message

3 Analysis of system log isolation

In this section, we further make a detailed analysis of system log isolation problems in current container environment. To precisely understand the isolation problems, we split the system log workflow into three phases: log generation, log access, and log analysis (shown in Fig. 2). For each phase, we discuss the isolation problems, the significant impacts, and the root causes (summarized in Table 1).

3.1 Isolation problems in log generation

The first phase of system log workflow is log generation. In this phase, containers print their logs to the shared log storage through two important actions: setting log configuration and performing log operations. Here we analyze two main isolation problems happening with the two actions: configuration conflict and operation conflict, which are caused by the shared configuration and shared storage, respectively.

Configuration conflict: the shared log configuration set by one container conflicts with that set by another container. Containers can modify the log configuration to control what logs they want to print out. This is important for accessing logs conveniently and analyzing logs efficiently. Because the log configuration is shared among containers and any container can modify the shared log configuration, the configuration of different containers may be conflicted. For

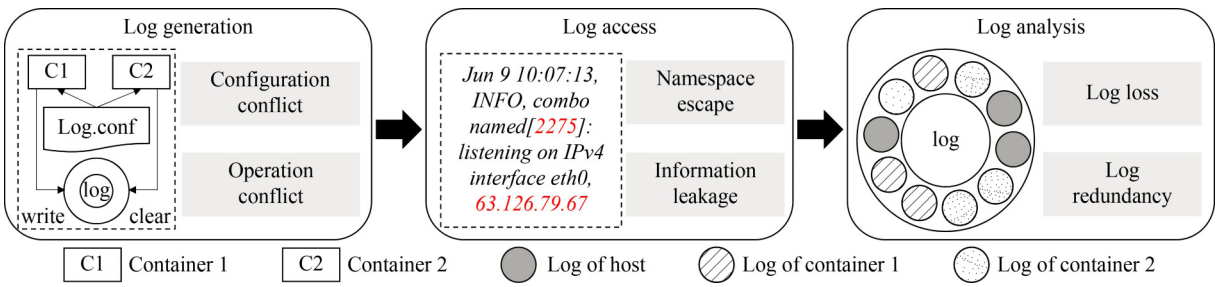


Fig. 2 Isolation problems during system log workflow in current container environment

Table 1 Isolation problems, the impacts, and the root causes

Phases	Isolation problems	Impacts	Root causes
Log generation	Log configuration set by one container conflicts with that set by another container.	Containers fail to use log configuration to customize log service.	Shared configuration
	Log operations performed by one container conflict with operations from another container.	System log fails to serve containers.	Shared storage
Log access	One container can view information that does not belong to its respective namespace.	Effectively breaking the view isolation realized by current namespace technique.	Shared view
	System log accidentally exposes information of host and co-resident containers to one container.	This information leakage causes additional security concerns (e.g. covert channel attack).	Shared view
Log analysis	The log storage of one container can be occupied by another container, causing unexpected log loss.	Reducing the accuracy of log analysis.	Shared storage
	One container cannot distinguish its own logs and must analyze redundant logs.	Reducing the efficiency and accuracy of log analysis.	Shared view

example, one container wants to print logs whose level is higher than *info* and sets the corresponding configuration to obtain comprehensive information. But another container wants to print logs whose level is higher than *err* for efficient error identification. These two requirements are in conflict, and the OS kernel cannot meet both requirements simultaneously. As a result, such conflict could lead to containers fail to modify the log configuration. In other words, the log configuration proposed to customize the log service for users becomes useless in the container-based environment.

Operation conflict: log operation performed by one container conflict with operations from another container. In log generation phase, containers can update and search the system logs with operations, such as *write*, *clear*, and *read*. As shown in Fig. 2, these operations (even if from different containers) are performed on the same shared log storage. When one container performs operations on the logs, it is important that no other containers operate on them. These operations from different containers may be in conflict and demand mutual exclusion. For instance, when one container is reading its logs and another container writes to the logs at the same time, the results returned are inaccurate. Wrong monitoring information and decision can potentially be produced due to the wrong logs. Even worse, if another container cleans the logs, nothing can be returned. This conflict would prevent containers from generating logs accurately and cause system log fails to serve containers.

3.2 Isolation problems in log access

The second phase of system log workflow is log access. In this phase, due to the shared log view, containers can read all logs, including logs of host and co-resident containers, introducing isolation problems. In this section, we demonstrate how system logs can cause namespace escape and information leakage when containers read logs through the shared view and we analyze their significant impacts.

Namespace escape: one container can view the information that does not belong to its respective namespaces through system log. As mentioned before, existing namespace mechanism can provide view isolation for containers. Each container has its own namespace to realize isolated system view and the information belonging to other namespaces is invisible to the container. However, containers can obtain the global view of the host from system log, effectively breaking the view isolation of namespace mechanism. For example, the network and PID namespace can virtualize the IP addresses and PIDs, respectively. However, containers can get the real IP addresses and PIDs from the system logs (shown as red words in the second phase of Fig. 2), making the network namespace and PID namespace ineffective. As a consequence, the shared view of system log can significantly affect the view isolation which is a fundamental property of containers.

Information leakage: system log might accidentally expose information of host and co-resident containers to one container. Due to the shared view of system log, any container is able to read all log messages, including host state information (e.g., kernel events and global kernel data) and container runtime information (e.g., application running status and resource status). This emerging information leakage channel could help optimize attack strategies and maximize attack effects, causing additional security and privacy concerns. For example, similar to the research [21] about information leakage of *procfs*, attackers can use system log to build covert channels [29] to transfer information stealthily.

3.3 Isolation problems in log analysis

The last phase of system log workflow is log analysis. In this phase, the integrity and conciseness of logs are crucial for the accuracy and efficiency of log analysis. However, containers would suffer from two main problems: log loss and log redundancy because they share the log storage and view, ultimately affecting the accuracy and efficiency of log analysis. The accuracy of log analysis is calculated by the number of events happened and the number of events analyzed. For example, if the number of events happened and analyzed are 100 and 90, respectively. The accuracy of log analysis is 90%. Obviously, if the accuracy is high, the log analysis is effective. The efficiency of log analysis refers to the completion time of log analysis. In what follows, we will discuss them in detail.

Log loss: the log storage of one container can be occupied by another container, causing unexpected log loss. In order to to reduce the occupation of log storage, the Linux kernel uses a log rotating mechanism to delete the old system logs. In other words, log storage is a kind of consumed resource whose amount is limited, and when generating new system logs, the operating system kernel may trigger the log rotating mechanism, causing the old system logs to be deleted. Because containers share the log storage, the logs of one container might be accidentally deleted to make space for the new logs produced by other containers. Even worse, an adversarial container is able to preempt the shared log storage

by producing logs continuously. Such operations will not cause CPU competition between containers because the CPU cgroup can provide CPU resource isolation for containers. However, these operations can delete the logs of other containers on the same host due to the shared log storage, introducing the log loss problem for other containers.

The high accuracy of log analysis depends on the integrity of logs, so such unexpected log loss could significantly reduce the accuracy of log analysis, affecting the maintenance of containerized applications. For example, suppose an application needs to analyze the system logs generated half an hour ago to find the cause of the problem. In a non-container environment, the competition pressure for log storage space is relatively low, and the system has sufficient storage space to store the logs required by the application, thus we can obtain the integrated log information and produce accurate analysis results. However, in a container environment, due to shared log storage between containers, the system logs generated by the application half an hour ago may have been deleted by the operating system kernel, making it impossible to accurately identify the cause of the problem through log analysis.

Log redundancy: one container cannot precisely distinguish its own and must analyze redundant logs, including that produced by the host and other containers. As shown in the example in Fig. 2, the log view of container 1 is a global log view that includes the system logs of host, container 1, and container 2. When container 1 performs log analysis, the system logs of host and container 2 become redundant logs. In addition, as the density of container deployment on the host continues to increase, the amount of redundant logs will continue to increase. For any container, the amount of redundant logs will far exceed the amount of logs generated by itself, so the problem of system log redundancy in container environments is significant.

System log redundancy can seriously affect the accuracy and efficiency of log analysis for containers. Since software systems evolve to large-scale and complex-structure, the amount of system logs grows rapidly. The logs analysis becomes labor-intensive and time-consuming [30,31]. Therefore, efficient log analysis is important. However, due to the shared log view, containers cannot only analyze their own logs. On the one hand, a large set of redundant logs are analyzed at the same time, incurring much additional performance overhead. On the other hand, the logs of other containers could interfere the analysis results, reducing the accuracy. As a result, there is an urgent need to reduce log redundancy and improve the efficiency and accuracy of log analysis for containers.

3.4 Summary of findings

The system log isolation problems, possible impacts and root causes are summarized in Table 1. And we make the following findings:

Finding 1 Using system log in current container environment can introduce several isolation problems and cause significant impacts on system usability, security, and efficiency. In log generation phase, configuration conflict and operation conflict between containers can make system log useless. Log access from containers could cause namespace escape and information leakage, significantly affecting the isolation and security of containers. In log analysis phase, log loss and redundancy would occur, reducing the accuracy and efficiency of log analysis. This finding clearly suggests the necessity of enhancing system log isolation.

Finding 2 There are three root causes of system log isolation problems: shared configuration, shared storage, and shared view. Shared configuration causes configuration conflict. Shared storage causes operation conflict and log loss. Shared view causes namespace escape, information leakage, and log redundancy. This finding sheds light on our solution to address the isolation problems.

4 POGs: private logs

To enhance system log isolation for containers, we propose *private logs* (POGs) to provide each container with individual system log instead of sharing system logs among containers. In this section, we first provide a system overview and then detail the design of POGs.

4.1 Overview of POGs

Based on the Finding 2, the root causes of system log isolation problems are shared configuration, shared storage, and shared view. Consequently, POGs is designed with the following goals:

- **Private log configuration** Shared log configuration may cause configuration conflict between containers, affecting the usability of system log in container environment. POGs aims to provide each container with its own system log configuration, such that containers can customize the log service without conflicts.
- **Private log storage** Due to shared log storage, operations from different containers may conflict with each other, and logs of one container can be deleted unexpectedly. To reduce the impacts on usability of system log and accuracy of log analysis which are caused by the above problems, POGs tries to store logs individually for each container.
- **Private log view** Shared log view introduces problems, such as namespace escape, information leakage, and log redundancy, affecting container security and the efficiency as well as the accuracy of log analysis. POGs aims to enable a private log view for each container to help containers distinguish their own logs and prevent one container from accessing others' logs.

Figure 3 shows the overview of POGs. To achieve the above goals, we provide each container with a POG that is made up of individual log configuration, log storage, and log view. First, when one container is started, we create a new log configuration file in its POG. Each container can set its own log configuration and print out logs based on the configuration. The configuration file is so small (KB level) that the overhead is negligible. Second, we also create a new ring buffer for the newly-launched containers to store their own logs. One container can set the size of its log ring buffer, based on its needs. To avoid too much memory footprint of

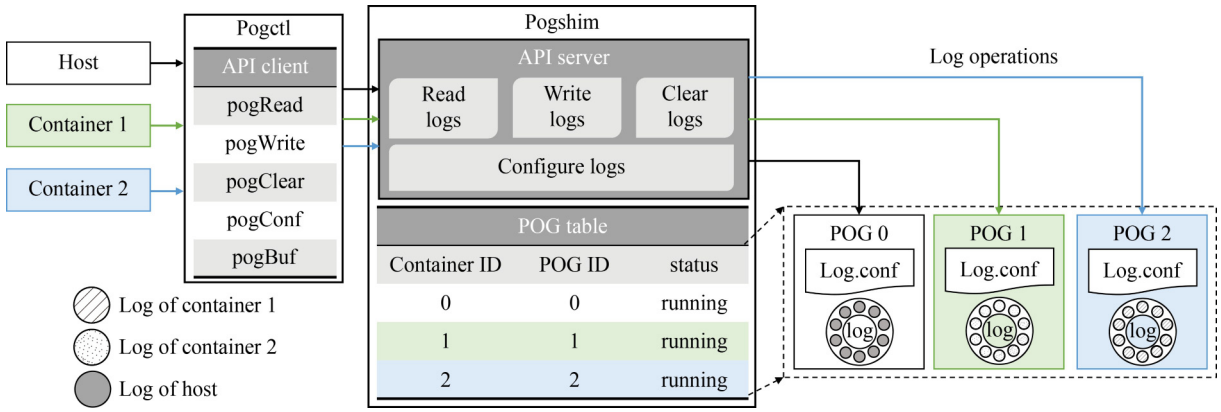


Fig. 3 Architecture of POGs

such private log storage, we account for the ring buffer into containers' respective memory cgroup. The reasons are as follows. The memory cgroup is normally used to limit the memory usage for containers. Once one container uses too much memory to conduct its log ring buffer, its available memory will be reduced due to the memory cgroup limitation. To ensure there is enough memory space for the container, the user has to reduce the size of log ring buffer. As a result, the memory footprint of per-container log ring buffer can be limited by memory cgroup. Last, we provide each container a private log view by maintaining the information about which container the POG belongs to. The log view of one POG is visible to its belonger but is invisible to other containers. Hence, one container is not able to manipulate the logs which is owned by another container.

We design two key components: *pogctl* and *pogshim* to manage the POGs. *Pogctl* provides several APIs for containers to control and configure POGs precisely. And *pogshim* realizes the management of POGs and identifies the belongers of POGs. Our system works as follows: At the startup of one container, *pogshim* creates a new POG (including a new log configuration file and ring buffer) and identifies the new container as the belonger of this new POG. Then the container can invoke APIs to configure its POG (modifying its configuration file and setting the size of its ring buffer). The logs of this container will print into its ring buffer based on its configuration. If the container calls an API to read logs, *pogshim* collects and returns the logs from the corresponding log storage and view. When the container exits, its POG (configuration file and ring buffer) will be deleted and freed by the *pogshim*.

4.2 Key components: *pogctl* and *pogshim*

1) Pogctl In order to manage POGs conveniently and efficiently, we design *pogctl*, a component that consists of several APIs. One container can call APIs to control and configure the logs in its POG. These APIs are implemented as system call style interfaces such that containerized applications can invoke them easily and flexibly. POGs APIs are shown in Table 2. One container can read its log messages using the `pogRead` operation and receive the content of its logs. Similarly, a message may be written to a POG by one container using `pogWrite` operation. Except that OS kernel can delete logs periodically, containers can delete the logs manually by invoking the `pogClear` API. If one container needs to customize the log service, it can use `pogConf` operation (needing the path of the configuration file (conf-path) as a parameter) to initialize or modify the configuration of its logs. The last API `pogBuf` can be used to set the size of log ring buffer. As a consequence, these APIs give containers considerable flexibility in managing system logs.

Table 2 POGs APIs used by containers to control and configure their logs

Function	Return value		APIs
Read logs	log messages	⇐	`pogRead()`
Write logs	bool	⇐	`pogWrite(message)`
Clear logs	bool	⇐	`pogClear()`
Set configuration	bool	⇐	`pogConf(conf-path)`[1]
Set ring buffer size	bool	⇐	`pogBuf(size)`

[1] Conf-path denotes the path of configuration file (log.conf).

2) Pogshim *Pogshim* is designed for identifying the belonger of POGs, managing the entire life cycle of POGs, and realizing the function of POGs APIs. As shown in Fig. 3, first, *pogshim* identifies each container with an ID number, similar to process ID (PID). The ID of host is zero, and containers will be assigned with a non-zero ID when they are created. Second, *pogshim* maintains a system-wide POG table, which has multiple table entries where each table entry stores a container ID, POG ID, and POG status. Hence, each container is mapped with a POG ID, identifying the belonger of POGs. And POG table provides a global view of POGs, helping OS kernel monitor and manage POGs. If a POG is created, a new table entry would be inserted into the POG table. Similarly, a table entry will be deleted when releasing a POG. Last, we propose the API server in *pogshim* to realize the function of POGs APIs and help containers control their logs. The POG APIs invoked by one container can be forwarded to the API server. After receiving API requests, the API server looks up the POG table to get the POG belonging to the container and executes the corresponding function to control logs in its POG.

4.3 POGs usage patterns

In this section, we illustrate POGs use patterns for log generation, access, and analysis.

Log generation The first use pattern is log generation, where configuration conflict and operation conflict might happen. Using POGs, both problems can be addressed. First,

one container can invoke the API `pogConf` to set its own log configurations, avoiding configuration sharing. Thus, the log configuration of this container will never conflict with others. Then, the container can call `pogBuf` to create its own log storage (a ring buffer). All operations from this container will be performed on the private log storage. Such that the operation conflict can be addressed. Last, containers can invoke `pogWrite` to generate logs to their own log storage based on their own log configuration, without any conflicts.

Log access In log access phase, original system log design introduces problems, such as namespace escape and information leakage. Here we demonstrate POGs can address such problems. With POGs, one container can call `pogRead` API to access its logs. *pogshim* provides a limited log view for the container based on the POG table. So, the `pogRead` will only return the container's log messages, instead of all log messages. As a result, the container cannot obtain information of host and other containers, preventing namespace escape. And, the information of this container is invisible to other containers, effectively closing the information leakage channel.

Log analysis In log analysis phase, containers suffer from log loss and log redundancy with the original system log design. The root causes of log loss and log redundancy are shared log storage and shared log view, and these problems can significantly reduce the accuracy and efficiency of log analysis for containers. Our system can effectively address these problems, improving the accuracy and efficiency of log analysis. First, one container can create its own log storage through `pogBuf` API. This avoids sharing log storage and reduces the log storage competition between containers, so the logs of this container cannot be deleted by other containers, avoiding log loss and improving the accuracy of log analysis. Second, with private log views, containers no longer share the global log view and can distinguish their own logs. So one container can only read its own logs for analysis, reducing log redundancy. As a consequence, POGs can help containers improve the accuracy and efficiency of log analysis.

5 Evaluation

In this section, we mainly evaluated the effectiveness of POGs for enhancing system log isolation and its performance overhead.

5.1 Experimental settings

We implemented POGs based on the Linux kernel 5.11.6. We used Docker 20.10.12 and Ubuntu 18.04 to run containers. The hardware includes a 20-core Intel Xeon E5-2650 CPU, 128 GB memory and 2 TB HDD storage. Similar to the cloud production environment, we enabled cgroups and namespaces to isolate containers. Each container was assigned with limited hardware resources (e.g., memory) and pinned to specific cores. The experimental settings and benchmark details of the individual evaluation were slightly different and further discussed in the respective sections.

5.2 Isolation improvement

The primary goal is to evaluate whether our system can thoroughly address the isolation issues discussed in 3. We used two typical workloads: httpd and MySQL, to produce system logs. We ran same workloads with various numbers of containers for the two workloads respectively. We used the difference in results collected from the one-container case and the multiple-containers case to indicate isolation. The baseline of this experiment is Docker running on native kernel. We ran same experiments on POGs and native kernel to make conditions for POGs and the comparison methods equivalent. Thus, we can demonstrate the advantages of POGs compared to Docker itself.

To demonstrate the effectiveness of POGs in solving log configuration conflict, we started two containers on a single host, represented by container 1 and container 2, respectively. Firstly, we modified the system log configuration file in container 1 and set the output log level to *info*. At this time, it is observed that the output log level is higher than *info*, indicating that the log configuration set in container 1 is effective. Then, we modified the system log configuration file in container 2 and set the output log level to *error*. The experimental results are shown in Table 3.

We observed that the logs output by the native kernel were all higher than *erro*, and there were no more *info* level log outputs, suggesting that the log configuration set by container 1 was invalid. For the results of POGs, the logs output by container 1 were not affected by the log configuration of container 2, and there were still *info* level logs output. The experimental results clearly suggest that POGs can help different containers set different log configurations without conflicts, which demonstrates the effectiveness of container private log configuration.

In order to demonstrate the ability of POGs to solve log operation conflict, we used a similar experimental method as the above. We read system logs in container 1, and clear system logs in container 2 at the same time. As shown in Table 3, we observed that the log read operation of container 1 in the native kernel did not return any log information, because there was a conflict between the system log operations of container 1 and container 2. The system logs were all cleared by container 2, and container 1 was unable to return any logs. For the results of POGs, container 1 can read its own log information, which indicates that container 2

Table 3 Isolation improvement in log generation and log access

Isolation problems	Experimental operation	Results of native kernel	Results of POGs
Configuration conflict	Set the log level of container 1 and container 2 as *info* and *erro*, respectively.	The output log level is higher than *erro*.	The output log level is higher than *info*.
Operation conflict	Read and clear system log in container 1 and container 2, respectively.	No system logs are returned.	System logs of container 1 are returned.
Namespace escape, information leakage	Read system logs in container 1 and container 2.	All system logs are returned.	Respective system logs are returned.

cannot clear the system logs of container 1, thus suggesting that POGs can solve the log operation conflict problem.

To demonstrate the ability of POGs to address namespace escape and information leakage problems, we still ran two containers (container 1 and container 2) on a single host and read system logs in the two containers. As shown in Table 3, For native kernel, container 1 and container 2 returned the same global system log information, because both containers have global system log view, which may cause namespace escape and information leakage. For POGs, container 1 and container 2 returned their respective system log information, indicating that POGs can help one container get its own view of logs, and its view is invisible to other containers, preventing namespace escape and information leakage effectively.

In order to evaluate the efficiency and accuracy improvement for log analysis, we used `logparser` benchmark to analyze logs for containers. Logparser is usually used for automated log parsing by converting raw log messages into a sequence of structured events. By applying logparser, users can correctly learn events from unstructured logs. We measured the completion time of log analysis as efficiency and the ratio of the accurate events returned from logparser over real events that happened in containers as accuracy.

Figure 4 shows the results of efficiency evaluation. For httpd (Fig. 4), When we ran one container, POGs can improve the efficiency of log analysis by 32%. This is because the logs of host in native kernel delay log analysis. For POGs, the container can distinguish its own logs accurately without analyzing the host logs additionally. As the number of containers increases, POGs can improve more efficiency. Because more containers produced more logs, delaying logs analysis more. The results of MySQl are much similar. In addition, the results of accuracy evaluation in Fig. 5 show a similar improvement trend. The reason is that POGs can help containers identify their logs and store complete logs for them. As a result, these results suggest that POGs can improve the efficiency and accuracy of log analysis effectively.

In summary, the above evaluation results demonstrate that our system can address configuration and operation conflict, prevent isolation breaking and information leakage, and improve the efficiency and accuracy of log analysis, enhancing system log isolation.

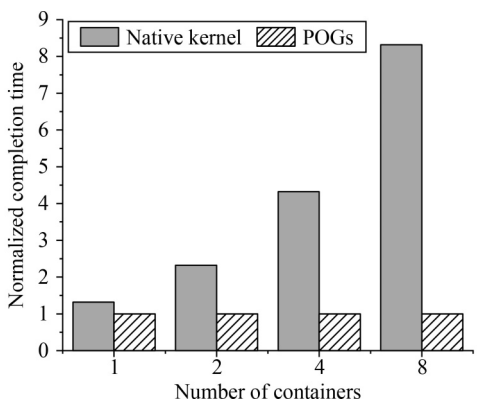


Fig. 4 The log analysis efficiency under various numbers of containers (POGs versus native Linux kernel)

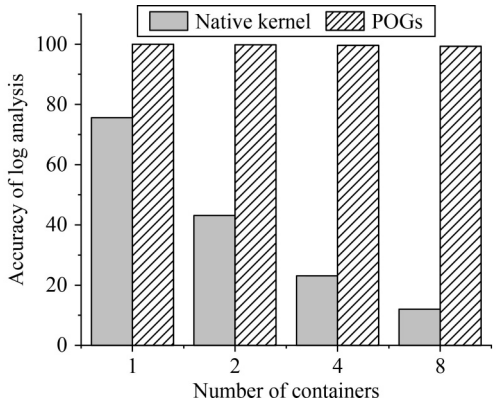


Fig. 5 The log analysis accuracy under various numbers of containers (POGs versus native Linux kernel)

5.3 Performance overhead

We evaluated the performance overhead of POGs on launching time, memory footprint, overall performance, and system scalability, compared to Docker itself which ran on the native kernel. As Kata container is the most popular container isolation system, we also compared POGs with it to show the advantages of our approach. We performed the same operations on the above systems.

To evaluate the launch time and memory footprint, we started a Ubuntu 16.04 container. We leveraged the *time* command to measure the launch time and used the *docker stats* command to print out the memory usage of this container. Figure 6(a) illustrates the results. POGs introduce 1.1% overhead of launching time, which is spent initializing per-container POG. And POGs consumes 0.04 MB (1.28%) more than the container running on native kernel. Compared to Kata container, POGs can decrease launch time and memory footprint by 6.92× and 4.72×, respectively. Because Kata container boots a per-container kernel, causing more overhead. Consequently, the additional launch time and memory introduced by our system are negligible.

To evaluate the performance overhead of POGs on applications, we ran httpd and MySQL in containers and measured their performance. As shown in Fig. 6(b) the throughput overhead of httpd and MySQL is 0.56% and 0.74%, respectively. Compared to Kata container, the throughput of httpd and MySQL is 6.79× and 6.04×, respectively. These results show that POGs does not contribute much overhead to the applications' overall performance and introduces much less overhead than existing popular container isolation methods.

Because our system creates a POG for each container, it is important POGs introduces little performance overhead when the container deployment density is large. In order to evaluate the scalability of POGs, we ran up to 1024 httpd containers on our server. We used *apache benchmark* to measure the total throughput of all containers. Results are shown in Fig. 7. Our system has a almost the same trend as the native kernel when the number of containers increases and achieves similar performance as the native kernel for any number of containers. The performance of Kata container is much worse than POGS and native kernel. Because Kata container creates a kernel for each container to enhance isolation. These results clearly

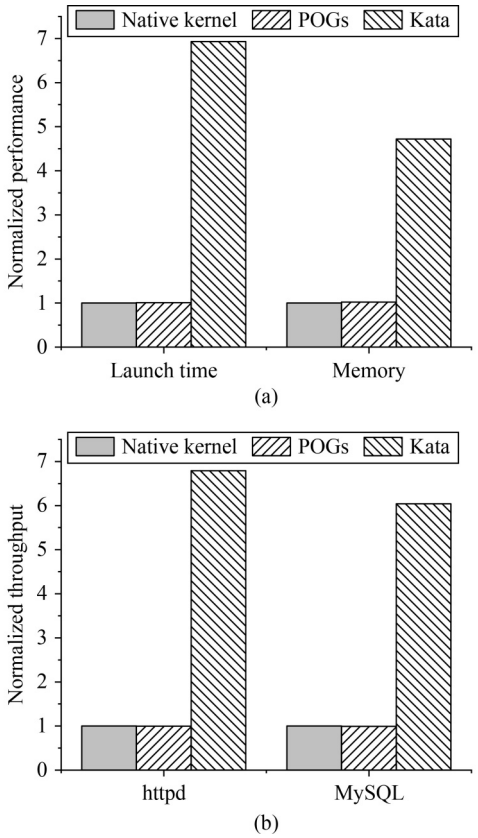


Fig. 6 The performance overhead of POGs on launch time, memory footprint, and overall performance. (a) Launch time and Memory; (b) overall performance

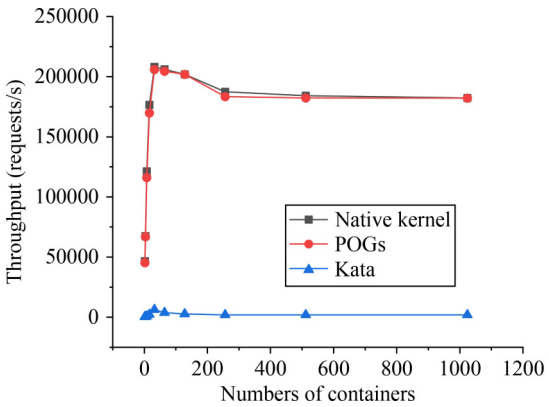


Fig. 7 Throughput scalability of POGs

suggest that providing each container with a POG will not affect the container scalability.

6 Related work

In this section, we review the realted work that inspires our research and describe the differences between our research and previous work. We mainly discuss the related work about container isolation and system log.

Container isolation Several researches demonstrated weak container isolation and its significance [9–15]. For example, Yang et al. [9] explored the isolation issues of abstract resources in OS kernel and discussed the possible security impacts. To our best knowledgement, this is the first work to analyze the isolation issues of system log.

There are two ways to enhance container isolation. One is optimizing namespace and cgroup [16–21,32,33]. For example, Huang et al. [16] enhanced the isolation of resource view by designing a new namespace to export the available resources (e.g., CPU and memory) to containers. Unlike these studies, this work proposes POGs to enhance system log isolation, which is orthogonal and complementary to these studies.

The other one trends to run containers in an individual OS kernel. X-Containers [34], LightVM [35], and Kata containers [36] used hypervisor to run containers in a light virtual machine to provide full isolation for containers. Similarly, gVisor [37] provided each container with a user-space kernel to enhance isolation. However, the per-container OS kernel and hypervisor incur significant performance overhead for containerized applications. Instead, POGs is implemented in Docker and introduces negligible performance overhead.

System log The studies about system log focus on log analysis. Developers can leverage log analysis to verify programs [38,39], identify duplicate issues [40,41], and detect anomalies [22,42]. Because the size of system log was becoming bigger and the manual log analysis become time-consuming, several studies [39–41,43] aimed to use artificial intelligence techniques to design automated log analysis. Different from the above studies, our research focuses on the significant system log isolation problems for containers. Based on the existing work, such as automated log analysis, our work can further improve the accuracy and efficiency of log analysis. So, the above studies are orthogonal and complementary to our work.

7 Conclusion

In this paper, we first perform a detailed analysis of system log isolation problems in current container environment, such as configuration conflict, operation conflict, namespace escape, information leakage, log loss, and log redundancy. We reveal that the root causes are that log configuration and storage are shared among containers, and containers have a global log view. Then we propose POGs to address these problems by providing each container with an individual POG that consists of private log configuration, storage, and view. And we design *pogctl* to help containers control and configure their POGs. Another component named *pogshim* is proposed to identify the belonger of POGs and manage the entire life cycle of POGs. Last, we implement our design on Linux kernel 5.11.6. Experimental results show that POGs can address the problems and enhance system log isolation with negligible overhead.

Acknowledgements We thank the anonymous reviewers for their helpful feedback. This work was supported by the National Key R&D Program (2022YFB4500704), the National Natural Science Foundation of China (Grant No. 62032008).

Competing interests The authors declare that they have no competing interests or financial conflicts to disclose.

References

1. Gu L, Zeng D, Hu J, Jin H, Guo S, Zomaya A Y. Exploring layered container structure for cost efficient microservice deployment. In: Proceedings of IEEE Conference on Computer Communications. 2021, 1−9
2. Li Z, Cheng J, Chen Q, Guan E, Bian Z, Tao Y, Zha B, Wang Q, Han W, Guo M. RunD: a lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In: Proceedings of 2022 USENIX Annual Technical Conference. 2022, 53−68
3. Suo K, Zhao Y, Chen W, Rao J. An analysis and empirical study of container networks. In: Proceedings of IEEE Conference on Computer Communications. 2018, 189−197
4. Zeng R, Hou X, Zhang L, Li C, Zheng W, Guo M. Performance optimization for cloud computing systems in the microservice era: state-of-the-art and research opportunities. Frontiers of Computer Science, 2022, 16(6): 166106
5. Soltesz S, Pötzl H, Fiuczynski M E, Bavier A, Peterson L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems. 2007, 275−287
6. Zhuang Z, Tran C, Weng J, Ramachandra H, Sridharan B. Taming memory related performance pitfalls in Linux Cgroups. In: Proceedings of 2017 International Conference on Computing, Networking and Communications. 2017, 531−535
7. Laadan O, Nieh J. Operating system virtualization: practice and experience. In: Proceedings of the 3rd Annual Haifa Experimental Systems Conference. 2010, 17
8. Huang Z, Wu S, Jiang S, Jin H. FastBuild: accelerating docker image building for efficient development and deployment of container. In: Proceedings of the 35th Symposium on Mass Storage Systems and Technologies. 2019, 28−37
9. Yang N, Shen W, Li J, Yang Y, Lu K, Xiao J, Zhou T, Qin C, Yu W, Ma J, Ren K. Demons in the shared kernel: abstract resource attacks against OS-level virtualization. In: Proceedings of 2021 ACM SIGSAC Conference on Computer and Communications Security. 2021, 764−778
10. Hua Z, Yu Y, Gu J, Xia Y, Chen H, Zang B. TZ-container: protecting container from untrusted OS with ARM TrustZone. Science China Information Sciences, 2021, 64(9): 192101
11. Plauth M, Feinbube L, Polze A. A performance survey of lightweight virtualization techniques. In: Proceedings of the 6th IFIP WG 2.14 European Conference on Service-Oriented and Cloud Computing. 2017, 34−48
12. Matthews J N, Hu W, Hapuarachchi M, Deshane T, Dimatos D, Hamilton G, McCabe M, Owens J. Quantifying the performance isolation properties of virtualization systems. In: Proceedings of 2007 Workshop on Experimental Computer Science. 2007, 6−es
13. Felter W, Ferreira A, Rajamony R, Rubio J. An updated performance comparison of virtual machines and Linux containers. In: Proceedings of 2015 IEEE International Symposium on Performance Analysis of Systems and Software. 2015, 171−172
14. Sharma P, Chaufournier L, Shenoy P, Tay Y C. Containers and virtual machines at scale: a comparative study. In: Proceedings of the 17th International Middleware Conference. 2016, 1
15. Xavier M G, De Oliveira I C, Rossi F D, Dos Passos R D, Matteussi K J, De Rose C A F. A performance isolation analysis of disk-intensive workloads on container-based clouds. In: Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. 2015, 253−260
16. Huang H, Rao J, Wu S, Jin H, Suo K, Wu X. Adaptive resource views for containers. In: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing. 2019, 243−254
17. Sun Y, Safford D, Zohar M, Pendarakis D, Gu Z, Jaeger T. Security namespace: making Linux security frameworks available to containers. In: Proceedings of the 27th USENIX Security Symposium. 2018, 1423−1439
18. Gao X, Gu Z, Li Z, Jamjoom H, Wang C. Houdini's escape: breaking the resource rein of Linux control groups. In: Proceedings of 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019, 1073−1086
19. Khalid J, Rozner E, Felter W, Xu C, Rajamani K, Ferreira A, Akella A. Iron: isolating network-based CPU in container environments. In: Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation. 2018, 313−328
20. Li Y, Zhang J, Jiang C, Wan J, Ren Z. PINE: optimizing performance isolation in container environments. IEEE Access, 2019, 7: 30410–30422
21. Gao X, Gu Z, Kayaalp M, Pendarakis D, Wang H. ContainerLeaks: emerging security threats of information leakages in container clouds. In: Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2017, 237−248
22. Du M, Li F, Zheng G, Srikumar V. DeepLog: anomaly detection and diagnosis from system logs through deep learning. In: Proceedings of 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017, 1285−1298
23. Love R. Linux Kernel Development. 3rd ed. New York: Pearson Education, 2010
24. Merkel D. Docker: lightweight Linux containers for consistent development and deployment. Linux Journal, 2014, 2014(239): 2
25. Xie X L, Wang P, Wang Q. The performance analysis of Docker and rkt based on Kubernetes. In: Proceedings of the 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery. 2017, 2137−2141
26. Senthil K S. Practical LXC and LXD: Linux Containers for Virtualization and Orchestration. Berkeley: Apress, 2017
27. Yang Y, Shen W, Ruan B, Liu W, Ren K. Security challenges in the container cloud. In: Proceedings of the 3rd IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications. 2021, 137−145
28. Lin X, Lei L, Wang Y, Jing J, Sun K, Zhou Q. A measurement study on Linux container security: attacks and countermeasures. In: Proceedings of the 34th Annual Computer Security Applications Conference. 2018, 418−429
29. Masti R J, Rai D, Ranganathan A, Müller C, Thiele L, Capkun S, Zürich E. Thermal covert channels on multi-core platforms. In: Proceedings of the 24th USENIX Security Symposium. 2015, 865−880
30. He S, Lin Q, Lou J G, Zhang H, Lyu M R, Zhang D. Identifying impactful service system problems via log analysis. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2018, 60−70
31. Lin Q, Zhang H, Lou J G, Zhang Y, Chen X. Log clustering based

problem identification for online service systems. In: Proceedings of the 38th IEEE/ACM International Conference on Software Engineering Companion. 2016, 102−111

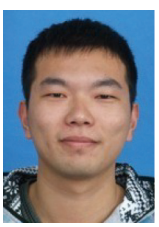
32. Wu S, Huang Z, Chen P, Fan H, Ibrahim S, Jin H. Container-aware I/O stack: bridging the gap between container storage drivers and solid state devices. In: Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. 2022, 18−30
33. Gu L, Guan J, Wu S, Jin H, Rao J, Suo K, Zeng D. CNTC: a container aware network traffic control framework. In: Proceedings of the 14th International Conference of Green, Pervasive, and Cloud Computing. 2019, 208−222
34. Shen Z, Sun Z, Sela G E, Bagdasaryan E, Delimitrou C, Van Renesse R, Weatherspoon H. X-containers: breaking down barriers to improve performance and isolation of cloud-native containers. In: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems. 2019, 121−135
35. Manco F, Lupu C, Schmidt F, Mendes J, Kuenzer S, Sati S, Yasukata K, Raiciu C, Huici F. My VM is lighter (and safer) than your container. In: Proceedings of the 26th Symposium on Operating Systems Principles. 2017, 218−233
36. Randazzo A, Tinnirello I. Kata containers: an emerging architecture for enabling MEC services in fast and secure way. In: Proceedings of the 6th International Conference on Internet of Things: Systems, Management and Security. 2019, 209−214
37. Anjali, Caraza-Harter T, Swift M M. Blending containers and virtual machines: a study of firecracker and gVisor. In: Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. 2020, 101−113
38. Beschastnikh I, Brun Y, Schneider S, Sloan M, Ernst M D. Leveraging existing instrumentation to automatically infer invariant-constrained models. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. 2011, 267−277
39. Shang W, Jiang Z M, Hemmati H, Adams B, Hassan A E, Martin P. Assisting developers of big data analytics applications when deploying on hadoop clouds. In: Proceedings of the 35th International Conference on Software Engineering. 2013, 402−411
40. Ding R, Fu Q, Lou J G, Lin Q, Zhang D, Xie T. Mining historical issue repositories to heal large-scale online service systems. In: Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2014, 311−322
41. Rakha M S, Bezemer C P, Hassan A E. Revisiting the performance evaluation of automated approaches for the retrieval of duplicate issue reports. IEEE Transactions on Software Engineering, 2018, 44(12): 1245–1268
42. He S, Zhu J, He P, Lyu M R. Experience report: system log analysis for anomaly detection. In: Proceedings of the 27th IEEE International Symposium on Software Reliability Engineering. 2016, 207−218
43. Lim M H, Lou J G, Zhang H, Fu Q, Teoh A B J, Lin Q, Ding R, Zhang D. Identifying recurrent and unknown performance issues. In: Proceedings of 2014 IEEE International Conference on Data Mining. 2014, 320−329

Kun Wang received the PhD degree from Huazhong University of Science and Technology (HUST), China in 2023. Currently he is an assistant research fellow at the College of Information and Communication in National University of Defense Technology, China. His current research interests include cloud computing, container virtualization, kernel resource isolation and intelligent operating system.

Song Wu received the PhD degree from Huazhong University of Science and Technology (HUST), China in 2003. He is a professor of computer science at HUST. He currently serves as the vice dean of the School of Computer Science and Technology and the vice head of Service Computing Technology and System Lab (SCTS) and the Cluster and Grid Computing Lab (CGCL) in HUST. His current research interests include cloud resource scheduling and system virtualization.

Yanxiang Cui received his BS degree from Huazhong University of Science and Technology (HUST), China in 2021 and is currently pursuing his MS degree in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Lab(CGCL) in HUST. His research interests include operating systems, performance evaluation, and lightweight virtualization technologies.

Zhuo Huang received the BS from Huazhong Agricultural University (HZAU), China in 2014. Currently he is a PhD candidate student in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Lab (CGCL), Huazhong University of Science and Technology (HUST), China. His current research interests include container virtualization, serverless computing optimization, and storage system.



Hao Fan received the PhD degree from Huazhong University of Science and Technology (HUST), China in 2021. Currently he is working as a post-doctor in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Lab (CGCL) in HUST. His current research interests include container technology and storage system.

Hai Jin is a Chair Professor of computer science at Huazhong University of Science and Technology (HUST), China. Jin received his PhD degree in computer engineering from HUST, China in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz. Jin worked at The University of Hong Kong, China between 1998 and 2000. He was awarded Excellent Youth Award from the National Natural Science Foundation of China in 2001. Jin is a Fellow of IEEE, Fellow of CCF, and a life member of the ACM. He has co-authored more than 20 books and published over 900 research papers. His research interests include computer architecture, parallel and distributed computing, big data processing, data storage, and system security.