Characterizing and Optimizing Kernel Resource Isolation for Containers

Kun Wang[a], Song Wu[a,*], Kun Suo[b], Yijie Liu[a], Hang Huang[a], Zhuo Huang[a], Hai Jin[a]

[a]*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, PR China*
[b]*College of Computing and Software Engineering, Kennesaw State University, Kennesaw, GA 30144, USA*

Abstract

Container-based virtualization has become increasingly popular as a lightweight alternative to hypervisor-based virtualization in cloud computing. Isolation is a fundamental property for consistent and reliable performance for cloud environment. However, the isolation between containers is much weaker than virtual machines as containers on the same host share one underlying host kernel. Existing works have mainly focused on the isolation problems at physical resources (e.g. CPU) level and almost not discussed with kernel resources (e.g. lock). In this paper, we perform a study to quantify kernel resource isolation for containers with a new microbenchmark, *KRIBench*. Then we describe kernel resource isolation issues and identify several kernel resources competition behind the poor isolation. Furthermore, we design and implement *Valve*, a general and flexible system that reduces kernel resources competition through limiting usage of system calls. *Valve* adopts Pareto-based container identification to locate misbehaving containers and supply-demand model to manage usage of system calls. The evaluation results demonstrate that our system can effectively enhance the kernel resource isolation for containers with negligible performance overhead.

Keywords: container, isolation, kernel resource, system call, performance interference, cloud computing

1. Introduction

1.1. Motivation

In recent years, due to its high performance, low footprint, simplicity of design and natural support for the micro-service and serverless computing [1], container-based virtualization, such as Docker [2], is becoming increasingly popular in cloud computing and a lightweight alternative to conventional hypervisor-based virtualization. Unlike traditional virtualization, container-based virtualization shares the same underlying host Operating System (OS) without Virtual Machine Monitor (VMM) and Guest OS. This fundamental difference helps containers eliminate most of the overhead that virtual machines (VM) suffer from. Recent studies [3, 4, 5] have shown that containers achieved near-native performance on CPU, memory and I/O in various workloads.

Isolation is a fundamental property of containers. Unlike VMs, which attain isolation through VMM and Guest OS, containers are isolated from each other through various kernel isolation primitives, such as Linux namespaces [6], control groups (cgroups) [7] and seccomp [8]. Namespaces provide containers with separate views of the file systems, IPC, PID, etc. Cgroups enable resource isolation by allocating, metering and enforcing resource usage, such as CPU, memory, and disk. Seccomp checks system call to protect the OS kernel against untrusted containers. However, these isolation primitives cannot achieve as strong isolation [4, 9, 10] as VMs.

Weak isolation is a key vulnerability that prevents containers from being widely accepted in cloud computing. Compared to VMs, it may introduce serious problems such as unstable performance [11, 12], system crash [13, 14] and security issue [15, 16]. To address the problem, researchers focus on wrapping each container within a dedicated OS kernel [17, 18, 19, 20]. However, these dedicated kernels incur additional significant performance overhead for containerized applications and services [19]. Another trend to enhance isolation is to optimize existing kernel isolation primitives [21, 22, 12] for a specific issue. While these works reserve high efficiency and flexibility of containers, they only realize partial isolation. For example, existing kernel isolation primitives and their optimizations do not isolate kernel resources such as file descriptors, locks and semaphores.

Meanwhile, there is an urgent need to enhance kernel resource isolation for containers. As multiple containers access the shared kernel concurrently through system calls, it may introduce serious kernel resource competition and significantly decrease the overall system performance. Specifically, as existing kernel isolation primitives cannot guarantee the reasonable allocation of kernel resources, one (misbehaving) container can maliciously occupy all the kernel resources decreasing the performance of other (well-behaved) containers. Although real-world applications generally do not occupy so many kernel resources, some malicious operations (e.g.looping system calls) can cause serious kernel resource competition easily. Therefore, the problem is realistic and urgent to be addressed.

However, isolating all kernel resources has lots of challenges.

[*]Corresponding author
Email address: `wusong@hust.edu.cn` (Song Wu)

First, as kernel resources are diverse and their design perspectives as well as details are completely different, isolating all kernel resources requires substantial kernel modifications and engineering effort. Second, there lacks precise and quantitative analysis of varying kernel resources, hence it is challenging to isolate kernel resources comprehensively.

1.2. Contributions

Characterization. This paper studies the research question of the necessity to enhance kernel resource isolation for containerized applications. As the system call interface is the main gateway for containers to access underlying kernel resources and system calls are simpler to observe and measure than kernel resources, this work investigates kernel resource isolation through system call control and management. Specifically, we introduce *KRIBench* (Kernel Resource Isolation Benchmark), a microbenchmark suite that is used to quantify kernel resource isolation. *KRIBench* includes 14 system calls which are collected from several representative containerized applications. We measure the performance degradation of a well-behaved container when deployed with a misbehaving container, which stresses it by looping the 14 system calls. The kernel resource isolation is quantified by such performance degradation, details of which can be found in Section 2.1.

Our study finds that all containerized applications' performance is highly degraded when co-locating with misbehaving containers. Many of the slowdowns are substantial: 62% of the slowdowns are by at least 40% and 37% of them are by at least 50%. In the worst scenario, the `open` system call can even cause 100% performance degradation. We further drill down on the above isolation issues and reveal that the major root cause is competition for kernel resources. These kernel resources fall into two categories: *consumable resource* and *exclusive resource*. One container can exhaust consumable resource, such as index node (`inode`) and file descriptor (`fd`), and significantly slow down other containers. Exclusive resource, such as locks and semaphores, can be preempted maliciously by one container that results in performance degradation of other containers on the same host kernel.

Optimization. Based on the above characterization, we design *Valve* to enhance kernel resource isolation for containers. The key idea is to precisely control the kernel resource competition and effectively manage usage of system calls for different containers. First, *Valve* identifies misbehaving containers using a model named *Pareto-based Container Identification*, which monitors the system call usage of containers and employs Pareto distribution to compute an optimal threshold that separates the well-behaved and misbehaving containers. After identifying misbehaving containers, we propose *supply-demand model* in *Valve* to manage usage of system calls. This model employs supply-driven resource allocation for misbehaving containers to limit usage of system calls and enhance isolation. In contrast, well-behaved containers invoke system calls on a demand-driven (best-effort) basis to ensure good performance.

Evaluation. We evaluate *Valve* in terms of container identification accuracy, isolation improvement and performance overhead through a range of benchmarks. Our results show that the accuracy of misbehaving container identification can reach 98.9%, which demonstrates the effectiveness of *Pareto-based Container Identification*. In addition, *Valve* can significantly improve kernel resource isolation (by a maximum of 98% and an average of 37.7%) with negligible overall performance overhead (by a maximum of 2.5% and an average of 1.3%).

1.3. Organization

The rest of this paper is organized as follows: Section 2 introduces the kernel resource isolation issues and root causes we explored. Section 3 and Section 4 present the design and implementation of `Valve` that we proposed to enhance kernel resource isolation for containers. Section 5 discusses evaluation results and Section 6 reviews the related work. Section 7 concludes this paper.

2. Characteristic Study

In this section, we first describe *KRIBench* and the methodology we use to drive our analysis and illustrate the weak isolation of container-based virtualization. Then, we analyze the container performance degradation and reveal that such the issue is attributed to abuse of *consumable resource* and *exclusive resource*.

2.1. Isolation Measurements

KRIBench. System call interface is the main gateway for containers to access underlying shared kernel resources. So our experiments focus on exploring kernel resource isolation through system calls. Inspired from analysis of Linux's performance [23], we only study system calls that are frequently exercised in container environments as these system calls are most likely to impact the performance of containerized applications and services.

To determine which system calls should be analyzed, we select a set of representative containerized applications whose images are mostly downloaded on DockerHub. Table 1 lists the applications and the workloads we select. These containerized applications include several database applications, such as *Couchbase*, a NoSQL database, *PostgreSQL*, a relational database, and *Redis*, a key-value store. In addition, we also selected three popular web server application, *Nginx*, *Httpd* and *Tomcat*, and two message systems, *RabbitMQ* and *NATS*. Also, we vary the workload sizes to take different usage scenarios into account.

In order to measure CPU time and the execution frequency of each system call invoked by the workloads, we adopt `strace` for such metrics. As listed in Table 2, those system calls which consume the most time across all workloads are selected into *KRIBench*. For each benchmark, we launch a loop process that constantly executes the system call to simulate misbehaving containers and interfere with well-behaved containers.

Methodology. We used Docker 18.06 on Ubuntu 16.04 to deploy two containers on a server, which was equipped with a 20-core Intel Xeon E5-2650 CPU, 128GB memory and 2TB HDD storage. Similar to cloud production environment, the

Table 1: Representative applications and workloads used to choose system calls and test isolation, and each applicaiton's performance metric collected in the test.

Application	Workload	Main performance metric
Couchbase, MongoDB, Memcached, Aerospike, Cassandra	YCSB with 5K, 10K and 50K operations	operations per second
PostgreSQL	pgbench with 1K, 2K and 5K transactions	transactions per second
Redis	redis-benchmark with 10K, 50K and 100K requests	requests per second
MySQL, MariaDB	mysqllap with 50K, 100K and 500K queries	average response time
InfluxDB	influx-stress with 1K, 5K, 25K requests	requests per second
Nginx, Httpd, Tomcat	ApacheBench with 10K, 25K and 50K requests	requests per second
RabbitMQ	rabbitmq-perf-test with 1, 10, 50 producers and consumers	average sending/receiving rate
NATS	nats-bench with 1K, 2k and 5k messages	messages per second

Table 2: Description of KRIBench tests.

Test	Description
`open`, `close`	Loops constantly opening or closing new files. To reduce the impact of competition for physical resources (e.g. disk space), these new files are empty.
`read`, `write`, `stat`	Loops reading 1 byte from, writing 1 byte to or returning information of a file.
`fork`	Loops creating new child processes. We killed the child process as soon as it was created, to release physical resources it occupied.
`mmap`, `munmap`, `mprotect`	Loops mapping 1 page from a file into memeory, or unmapping 1 page into a file, or changing access protections for a memroy page.
`sendto`, `recvfrom`	Testing process loops sending or receiving a message to or from an assisting process with TCP connection. Two processes run on different machines to reduce impact of assisting process. And the message size is 1 byte.
`select`, `poll`, `epoll`	Loops performing `select`, `poll` or `epoll` on a socket file descriptor which is ready on.

container execution platform was configured with multi-tenants. In order to explore interference from OS kernel other than physical resources, cgroups, namespaces and seccomp were enabled to isolate containers. Each container was set as non-privileged, configured with limited hardware resources, and pinned to specific cores. The container was deployed with Docker's default seccomp profile, which allows 358 system calls to the kernel. We assumed that the host environment was trusted, but misbehaving containers can maliciously compete for shared resources, slowing down other well-behaved containers.

The primary goal is to evaluate whether one container can be affected by other misbehaving containers and what the impact would be. The workflow consists of two steps: First, one of the representative workloads ran alone inside one container as well-behaved, while the other remained idle and introduced non-interference between two containers. The performance metric listed in Table 1 of the well-behaved container was collected and used as a baseline (denoted as $performance_{baseline}$). Second, the same workload ran inside a well-behaved container while a misbehaving container ran simultaneously on the same host. The process in the misbehaving container repeatedly executed system calls in *KRIBench* to produce stress and performance interference. Then, the performance metric of the well-behaving container (denoted as $performance_{misbehaving}$) was collected again. The isolation (denoted as ISO, smaller is better), which was quantified by the difference of the performance metrics collected from the above two steps, is denoted as follows:

$$ISO = \left(1 - \frac{performance_{misbehaving}}{performance_{baseline}}\right) \times 100\% \qquad (1)$$

Results. Fig. 1 illustrates the kernel resource isolation of each test across all workloads. The darker the red color is, the worse the isolation would be. As shown in the results, 97% of tests incur significant performance degradation. More specifically, 62% of the results encounter slow down by at least 40%, and 37% of them are reduced by at least 50%. For example, Nginx's performance is degraded by 67% when suffering from `select` stress in misbehaving container. When looping `open` system call in misbehaving container, as shown in 1^{st} row of Fig. 1, well-behaved containers cannot even start up. These results strongly suggest that the kernel resource isolation of container-based virtualization is vulnerable and containers would suffer from unstable performance. In what follows, we further analyze and explain root causes behind the performance degradation.

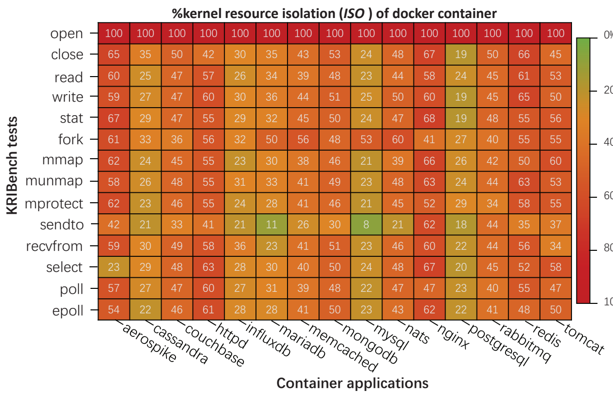


Figure 1: Results of kernel resource isolation for containers(lower is better).

2.2. Root Causes

The isolation issues mentioned above are mainly caused by concurrent accesses to shared kernel resources. Generally, containerized applications need to access kernel resources to com-

Table 3: Root causes of Linux container's kernel resource isolation issues.

System calls	Root causes	Description	Slowdown
Consumable resource: when this kind of resource is used up, other containers will be failed to request the resource.			
`open`	`fd`	Firstly, allocates a file descriptor to an opened file, consuming fd resources.	100%
	`inode`	Secondly, allocates an index node to a new file, consuming inode resources.	100%
`mmap`	`vitual memory area`	Alocates memory space, consuming viual memory area resources.	29.3%
`epoll`	`epoll user watch`	Inits epoll, consuming user watch resources.	42.1%
Exclusive resource: concurrent use of shared exclusive resource causes waiting time.			
`close`, `read`, `write`, `stat`	`mutex_lock` (or `down_write`)[a]	Acquires a mutex lock (or read/write semaphore)[a] to execute file operations.	60.69%
`fork`	`write_lock_irq`	Acquires a reader-writer lock to update task list.	54.92%
`mmap`, `mprotect`, `munmap`	`down_write`	Acquires a reader-writer semaphore to access memory.	38.05%
`sendto`	`lock_sock_nested`	Acquires a spinlock to use socket.	47.12%
`recvfrom`	`begin_current_label_crit_section`	Enters critical section to execute apparmor profile.	52.69%
`select`, `poll`	`rcu_read_lock`	Acquires a reader-writer lock to search fd list.	33.64%
`epoll`	`spin_lock_irqsave` (or `write_lock_irq`)[a]	Acquires a spinlock(or read/write lock)[a] to search events.	19.84%

[a] To observe whether the root causes change as kernel version is updated, we analyzed two generations of Linux kernel, 4.4.15 and 5.6.0. The contents in brackets are based on Linux kernel 5.6.0 and others are based on Linux kernel 4.4.15.

plete system calls. When multiple containers access shared kernel resources concurrently, it will introduce intensifying competition and overall system performance will significantly decrease due to poor isolation. Table 3 presents the root causes of each *KRIBench* test and the maximal slowdown of container application workloads. Based on our observation, kernel resources can be classified into two categories: *consumable resource* and *exclusive resource*. Consumable resource is shared among containers on one host. If it is exhausted by one container, other containers fail to request, and dramatic performance slowdown might happen. Exclusive resource can only be used by a single container each time, which would further prolong the waiting time and increase the overall system overhead.

2.2.1. Consumable Resource

Similar as physical resources (e.g. CPU, memory), there is a fixed total amount of consumable resources in the kernel, which are not isolated by existing cgroups mechanism. Unfair sharing of consumable resources can cause serious isolation problems. Specifically, one misbehaving container could greatly occupy a large amount of consumable resources, significantly slowing down or even suspending other well-behaved containers. For example, `open` system call can exhaust *inode* and *fd* resource, which are both limited in the OS kernel. Each file has an inode, which the filesystem uses to identify the file. An inode contains metadata about a file and is stored on disk. Therefore, the total amount of inode depends on the size of disk. Similarly, a fd identifies an opened file and its total amount depends on system memory size.

As depicted in Fig. 2, file operation such as `open` consumes

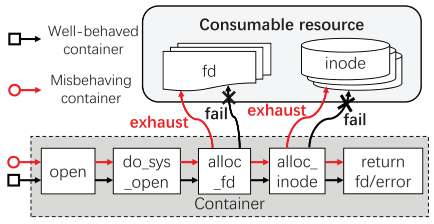
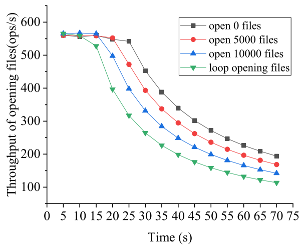


Figure 2: An example of consumable resource: competition for fd and inode causes isolation issues.

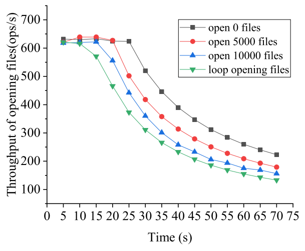
inode and fd in the kernel. However, when one misbehaving container competitively and continuously executes `open` system call, the host inode or fd resource will be quickly exhausted. In this case, other containers on the same host would fail to request for inode or fd resource, and cannot perform file operations anymore. As a consequence, the performance of well-behaved containers will be inevitably degraded.

It is possible to set inode and fd limits per user or process with existing kernel primitives (e.g. Linux `ulimit`). These limits can prevent inode and fd resources being exhausted, but cannot be used to deal with container isolation. The reasons are as follows: (1) We can set the maximum resources usage for a user, not for a container, so containers created by the same user will still share the available resources. (2) Generally, there are a group of processes running in a single container, so set limits per process is obviously not suitable for container isolation.

We further detailedly analyzed the performance interference of inode and fd competition. In this experiment, we used Linux kernel, and set inode and fd limits to the maximums with `ulimit`.



(a) Results of inode competition



(b) Results of fd competition

Figure 3: Experimental results of inode and fd competition analysis.

We ran `filebench` [24] benchmark, which is widely used to evaluate the performance of file system operations, including copying files, creating files or opening files in well-behaved container. We collected the throughput of opening files every 5 seconds. In the misbehaving container, we opened a different number of files to vary degrees of resource competition.

Fig. 3(a) shows the performance of the well-behaved container under different degrees of inode competition. When we opened 0 files in the misbehaving container (no resource competition), the performance dropped sharply from 25 seconds. This suggests that the host inode resource is exhausted by the well-behaved container in 25 seconds. As the number of opened files increased (competition increases), the throughput dropped earlier, and the average throughput was degraded. This indicates the poor isolation of containers and the inode competition would severely impact the overall performance. Similar results of fd competition analysis can be observed from Fig. 3(b).

2.2.2. *Exclusive Resource*

Different from consumable resource, exclusive resource can only be used by a single container each time, which might be more dangerously used by misbehaving containers. For instance, current operating systems (e.g. Linux) normally deploy a unified task list to provide reliable and centralized processes management. Task list is both updated and searched in host kernel. When the list is updated, it is essential that no other threads concurrently write to or read from the list. Writing shared resource demands mutual exclusion. When the list is searched or read, it is also important that no tasks write to the list. Therefore, task list is protected by a reader-writer spin lock. One or more readers can concurrently hold the reader lock. The writer lock, conversely, can be held by at most one writer with no concurrent readers [25]. Due to the shared host, multiple containers share a single task list lock.

As shown in Fig. 4, the shared lock introduces serious competition and one container, which is creating a child process by invoking system call `fork`, would hold the lock to update task list. When one misbehaving container maliciously keeps forking, the well-behaved containers have to wait for task list lock to update the list. Therefore, performance of well-behaved containers will be greatly reduced due to such exclusive resource competition.

Here we used concrete examples to demonstrate the performance interference of task list lock competition. In this experiment, we compared the performance degradation of two cases: (1) with task list lock competition, and (2) without such competition. In order to completely eliminate the competition of task list lock, we designed and implemented a hierarchical task list, which makes containers running on the same host no longer share the task list lock. As shown in Fig. 5, the hierarchical task list has a two-level hierarchy, where a public task list is distributed across containers while every container used a private task list.

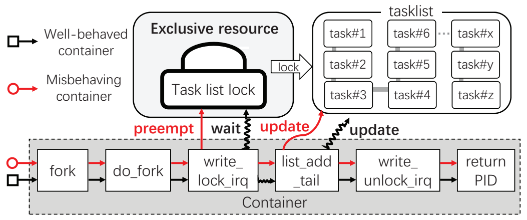


Figure 4: An example of exclusive resource: competition for task list lock causes isolation issues.

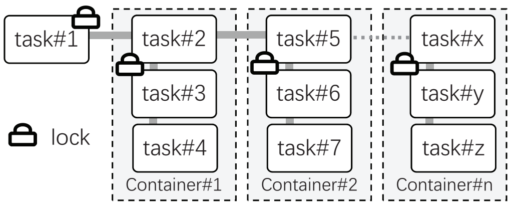
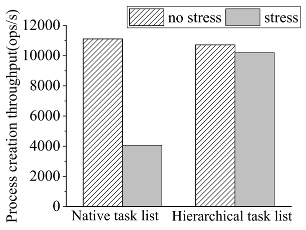


Figure 5: Architecture of hierarchical task list.

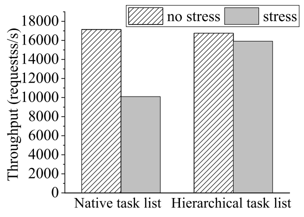
The public task list serves for two purposes. First, it is used to manage host processes that do not belong to any containers. Therefore, our hierarchical task list design has no impact on these host processes. Second, the `init` process of each container is organized into the public task list. In this way, host kernel would not lose control of processes running in containers. The private task list in each container is designed to manage processes running in the same container. As the `init` process of a container is placed into list head, host kernel can manage all processes in the container through `init` process organized into the public task list. When one container updates the task list, it only needs to hold its own private task list lock without suffering from lock waiting caused by contention of the public task list lock.

Same experiments were performed on native task list and hierarchical task list. We created child processes by system call `fork` in a well-behaved container. Then, we collected the completion time of the program to calculate the number of operations executed per second. We also ran Nginx as a macrobenchmark in the well-behaved container. In the misbehaving container, we looped many process creation instructions to stress the well-behaved container. In addition, to mitigate the impact of competition for resources other than the lock, we released the resources by killing the child process once it was created.

Fig. 6 illustrates the performance comparison between two task lists. As shown in Fig. 6(a), the throughput of `fork` degrades by 63.41% due to the misbehaving container when they run with native task list. In comparison, the performance degra-



(a) Results of fork



(b) Results of Nginx

Figure 6: The performance comparison between native task list and hierarchical task list.

dation of hierarchical task list is negligible (4.78%). Such performance improvement is due to the fact that we eliminate task list lock competition using hierarchical task list. Similar results of Nginx can also be observed from Fig. 6(b). The performance degradation of the two cases is 41.08% and 4.52%, respectively. The above results clearly suggest that the poor isolation among containers and the task list lock competition could introduce significant performance degradation.

2.3. Observation

To summarize, we have made the following observations:

Observation 1: *Competition for kernel resource can cause significant performance degradation and isolation issues.* Our study shows that well-behaved containers can be slowed down by misbehaving containers due to competition for kernel resources, which suggests necessity of kernel resource isolation.

Observation 2: *Kernel resources that cause isolation issues fall into two categories: consumable resource and exclusive resource.* Consumable resource can be exhausted and exclusive resource can be preempted, so other containers may be affected when they request these resources. This sheds light on our solution and corresponding optimization.

3. Design

Based on the above observations, we design *Valve* for containers to reduce kernel resources competition and enhance kernel resources isolation. In this section, we first provide a system overview and then describe two key techniques of our design in detail.

3.1. Overview

Key idea. Normally, it is extremely difficult to a general design to reduce competition of kernel resources. There are two reasons: First, kernel resources are diverse and their design perspectives and details are completely different. So a dedicated design (e.g. the hierarchical task list described in Section 2.2.2) works only for one kernel resource not for others. Second, with updates of kernel version, the resource involved in one system call might change. For instance, the cause of `write` system call, as shown in Table 3, is `mutex_lock` in Linux kernel 4.4.15 and `down_write` in Linux kernel 5.6, respectively. To address the above challenges, we propose to limit usage of system calls for misbehaving containers in *Valve*. Based on the fact that the

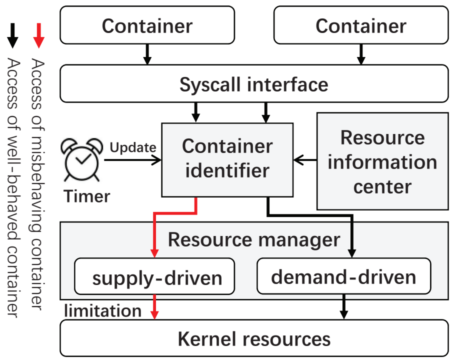


Figure 7: Overview of *Valve*.

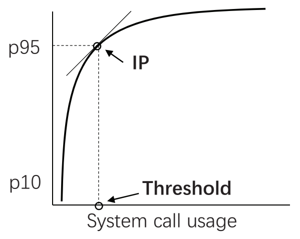
amount of kernel resources used by a container mainly depends on the times or rate of system calls called by the container, it is effective to control usage of system calls. This approach has two major advantages: First, it has great generality, which can be applied to all system calls and kernel resources without system modification. Second, it provides high flexibility for end users to enable or disable any system calls and kernel resources limit on purpose.

Architecture. Fig. 7 provides an overview of the interaction between the various components in *Valve*. *Resource information center* is designed to measure the kernel statistics and account for the status of system calls for each container. The status includes current usage, historical usage, maximal usage, etc. When one container requests kernel resource through a system call, *container identifier* intercepts this system call and checks whether the container is misbehaving or well-behaved based on the status of the system call. *Resource manager* guarantees resource fair shares among containers. It limits usage of system calls for misbehaving containers to enforce isolation and satisfies the system calls needs of well-behaved containers to reduce the performance overhead.

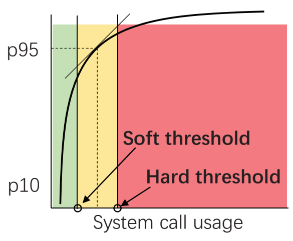
Challenges. Here *Valve* faces two main challenges: First, as there are a large amount of containers running on a same host, how to effectively and accurately check whether one container is misbehaving or not? Second, after we locate misbehaving containers, how to limit resource usage of them and reduce the impact on well-behaved containers? To address the first challenge, *Valve* proposes a key technique called *Pareto-based container identification*. We assume system calls' usage of containers forms a Pareto distribution [26] with a high alpha number. In other words, most containers are well-behaved and have a small amount of usage of system calls. However, a few containers (e.g., <5%) are misbehaving and invoke too many system calls. Based on Pareto distribution, *Valve* monitors the system calls' usage distribution of current containers running on one host and computes a roughly optimal threshold that separates the well-behaved and misbehaving ranges. To address the second challenge, *Valve* provides a *supply-demand model* to manage resources. In this model, supply-driven resource allocation is employed for misbehaving containers to limit usage of system calls and enhance isolation. In contrast, well-behaved containers invoke system calls on a demand-driven (best-effort)

Table 4: Maximum increase ratio of system calls with various containerized applications.

System calls	`close`	`read`	`write`	`stat`	`fork`	`mmap`	`munmap`	`mprotect`	`send`	`recv`	`select`	`poll`	`epoll`
Increase ratio	31%	32%	34%	33%	28%	31%	30%	25%	30%	29%	27%	29%	32%



(a) System call usage distribution and inflection point (IP).



(b) Classify containers into well-behaved, doubtful and misbehaving

Figure 8: System call usage CDF.

basis to ensure good performance.

3.2. *Pareto-based Container Identification*

Identification. As our idea is to limit the usage of system calls for misbehaving containers, they must be identified first before further limit actions. The major characteristic of misbehaving containers is that they consume a large amount of kernel resources and invoke too many system calls. Therefore, we check whether a container is misbehaving according to its usage of system calls. A simple threshold-based method is effective. Specifically, a container is classified as well-behaved, if its system call usage is less than a threshold value. Otherwise, it is considered misbehaving.

However, the main challenge here is how to find an optimal threshold to separate the well-behaved and misbehaving ranges. If we set the threshold too low, there might exist too many limited containers, including those that are supposed to be well-behaved, which incurs unnecessary performance loss. On the other hand, setting a too high threshold will make some misbehaving containers treated as well-behaved, which reduces the limitation scope and fails to enhance the isolation.

Based on the fact that there are far more well-behaved containers than misbehaving containers in actual cloud environment, we assume that system calls' usage of containers forms a Pareto distribution with a high alpha number. As an example shown in Fig. 8(a), 95% of containers likely invoke a small amount of system calls, but the other 5% of containers invoke too many system calls. Such a Pareto distribution clearly contrasts the well-behaved and misbehaving regions.

To separate the two regions, we need to find the best inflection point (IP) for maximizing the isolation enhancement. There are different methods to find an optimal IP. We pick a value where the slope of the CDF is one (likely being misbehaving) as our IP. This method has been proven effectiveness in addressing diverse problems [27]. Unlike LinnOS [27] which reports the IP based on the measured latency, we report our IP by counting the number of system calls. The reasons are as follows. First, we can not identify containers based on latency or actual execution time because the performance of both misbehaving containers and well-behaved containers may be degraded when misbehaving containers appear. Second, based on the analysis in Section 2, we find that the major characteristic of misbehaving containers is invoking a large number of system calls. Consequently, we can locate an optimal threshold based on Pareto distribution. For example, if the CDF's 45-degree slope is at $y=p95$ and $x=100$, then the threshold is set as 100.

Improving accuracy. Unfortunately, using one constant threshold to determine whether a container is misbehaving or not might be inaccurate. The system calls' usage distribution of containerized applications changes dynamically over the time. Using the same threshold calculated 10 minutes ago inevitably introduces unfaithful identification. In addition, the two-class approach (well-behaved or misbehaving) based on a constant threshold may also output different identification for highly similar containers. For instance, two containers whose usage of system calls are 99 and 101, will be identified as well-behaved and misbehaving, respectively, if the threshold is set as 100.

To achieve high accuracy, we first periodically calculate threshold according to the system calls' usage of a container in real time. High accuracy depends on the calculating period.But there is a trade-off between accuracy and performance. Specifically, the more frequently threshold is updated, the more accurate it is for current containers. However, too frequent updating can also contribute to unnecessary performance overhead. In our current design, the period is not changed automatically at runtime and *Valve* allows users to change the period manually when and if necessary. For example, if users need high accuracy, they can reduce the period. And if users want low overhead, they can increase the period. We discuss and evaluate some possible trade-offs between accuracy and performance in Section 5.1 to get the "best" calculating period in *Valve* and demonstrate the effectiveness of the period for improving accuracy. Using an analytical model (e.g. compromise programming) to formulate the trade-off and determine the best period value is under consideration and left as future work. The threshold may vary with the application and the capacity (e.g. memory size and the number of cores), but the Pareto-based model and the period, which are used to determine the best threshold value, are neither application-dependent nor affected by the capacity.

To deploy *Valve*, we do not need to generate the system call usage distribution and inflection point (IP) for different applications with different host (or container) configuration. Because the system call usage distribution and IP are generated without any prior knowledge of the application characteristics and host (or container) configuration. Hence, *Valve* is easy to configure and deploy. We empirically set the IP to 95% at system startup. As the system runs, the distribution and IP are updated periodically in a live fashion.

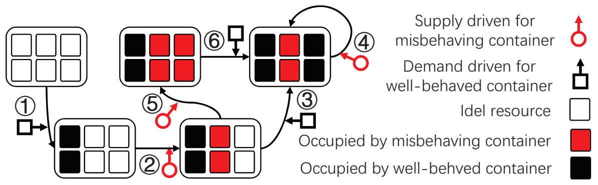
Second, instead of two types, we further classify containers into three categories: well-behaved, doubtful and misbehaving. As shown in Fig. 8(b), a container is classified as well-behaved (green area), if its usage of system calls is less than *soft threshold*. A container is classified as doubtful (yellow area), if its usage of system calls is between *soft threshold* and *hard threshold*. If its usage of system calls is higher than *hard threshold*, the container is considered misbehaving (red area). Now we need to set an optimal *soft threshold* and *hard threshold* based on the threshold that we calculate with Pareto distribution (defined as *based threshold*). We first measure the maximum usage increase ratio of each system call using the representative containerized applications and workloads (shown in Table 1). First, each time we run one application with various workloads that can cover most usage scenarios of the application and collect the system calls' usage for each workload with a tool named *strace*. Then the maximum increase ratio of each system call is calculated with the minimum usage and the maximum usage. For example, if minimum and maximum system call usage are 100 and 110, respectively, then the increase ratio is 10%. Last, for each system call, we calculate the average maximum increase ratio of all applications. Results in Table 4 show that the maximum usage increasing ratio of each system call is close to 30%. Based on this results, we set *soft threshold* to 0.85× of based threshold and *hard threshold* to 1.15× of based threshold. Such that well-behaved containers will never be classified as misbehaving.

After obtaining *soft threshold* and *hard threshold* and classifying containers into three types, we need to further deal with doubtful containers. We determine whether a doubtful container is well-behaved or misbehaving according to its usage increase ratio of system calls. Specifically, if the increase ratio is larger than 30%, the doubtful container is considered misbehaving. Otherwise, it is well-behaved. This is effective and reasonable. Because previous experimental results show that the maximum usage increase ratio of system calls for all representative applications is about 30%. As a result, for well-behaved containers, the increasing ratio of system calls' usage will not be larger than 30%.

Summary. We employ Pareto distribution to set an optimal threshold which is used to identify whether a container is misbehaving. To improve accuracy, we dynamically update threshold in real time and classify containers into three types: well-behaved, doubtful and misbehaving, instead of two types. Then we determine whether a doubtful container is misbehaving according to its usage increase ratio of system calls. After identifying misbehaving containers, we limit their usage of system calls with another design, named *supply-demand model*, which is described in what follows.

3.3. Supply-demand Model

In this section, we propose *supply-demand model* to manage the usage of system calls for containers. Our approach is able to enhance kernel resources isolation while introducing negligible performance overhead. Specifically, in order to ensure good performance for well-behaved containers, we use a demand-driven management strategy to make the best effort to satisfy their demand for system calls. If kernel resource is exhausted, we release those resources occupied by misbehaving containers to meet the demand of well-behaved containers. First, we can infer the resource that the well-behaved container needs from the invoking system call based on the root causes in Table 3. Then, we abort the same system call of misbehaving containers and free up the corresponding resource occupied by misbehaving containers. So we do not need to abort all system calls and free up all resources belonging to misbehaving containers. For example, if a well-behaved container is executing an `open` system call but the kernel returns "no resources are available", we will abort the `open` system call of misbehaving containers and delete the files belonging to misbehaving containers to release the *inode* and *fd* resources for the well-behaved container.


①
②
③
④
⑤
⑥
Supply driven for misbehaving container
Demand driven for well-behaved container
Idel resource
Occupied by misbehaving container
Occupied by well-behved container
Figure 9: An example of supply-demand model.

For misbehaving containers, our supply-driven management strategy provides misbehaving containers with a limited amount of system calls. In addition, it prohibits them from competing for resources occupied by well-behaved ones, which can effectively enhance the kernel resource isolation. Using this method, the performance of well-behaved containers will not be affected dramatically and the reasons are as follows. First, limitation prevents kernel resources from being exhausted quickly and avoids releasing resources for well-behaved containers frequently, which may slow down well-behaved containers due to waiting for releasing resources. Second, we set the limit to the smaller value of the average system calls' usage of well-behaved containers and the remaining idle resources. Therefore, if a well-behaved container is misidentified as misbehaving, its performance will not be affected significantly. In our design, we collect the historical system calls' usage of well-behaved containers into a component named *resource information center*. Thus, we can calculate the average system calls' usage of well-behaved containers with the historical data. To ensure the accuracy, we use the data collected in the current period (the threshold updating period).

An example is illustrated in Fig. 9. First, a well-behaved container executes system calls and consumes part of kernel resources (highlighted by the black area, step ①). Next, a misbehaving container loops substantial system calls and attempts to exhaust kernel resources (step ②). However, *Valve* only allows them to execute a limited amount of system calls, which is the average system calls' usage of well-behaved containers. Therefore, the misbehaving containers can only consume part of kernel resources, as highlighted by the red area, instead of exhausting the overall resources. At this time, if the well-behaved container requests resource, it can use the idle resources without waiting for others to release (step ③). Then even if the misbehaving container re-accesses the kernel resource again (step ④), there is no idle resource that can be supplied to the container and this access will be denied by supply-driven management strategy. When there exist idle resources that are not more than the average usage of well-behaved containers, kernel resource can be exhausted by misbehaving containers(step ⑤). Then, if well-behaved container re-accesses kernel resource again (step ⑥), the demand-driven management strategy releases the resource occupied by misbehaving containers and reallocates it to well-behaved container.

4. Implementation

We used cgroups to implement a prototype of *Valve*. In Linux, cgroups mechanism is stable and popular with developers to realize resource isolation. As mentioned before, with updates of kernel version, the resource involved in one system call might change. To check if our design is general, we implement a new cgroup subsystem called *syscall cgroup* based on two generations of Linux kernel, 4.4.15 and 5.6.0.

In *syscall cgroup*, usage of system calls is calculated and stored into *resource information center*, which is implemented as a piece of kernel memory (4KB for one container). For consumable resource, we account for the system calls times used in each container as sage of system calls. For exclusive resource, system calls rate is calculated based on system calls times and how long containers have run. To improve the statistical accuracy, we use a classic rate limit algorithm, named `sliding window` [28], to implement the calculation.

Syscall cgroup enforces strict isolation by dropping system calls of misbehaving containers. System calls are funneled from user space to the appropriate endpoints in kernel space through interfaces. We first intercept system calls at the point of system call interface before the appropriate kernel functions execute. After obtaining the incoming system call, *container identifier* (implemented as a kernel thread) reads its usage from *syscall cgroup* and checks whether this call is from a misbehaving container. If yes and there is not enough kernel resource that can be supplied to this call, the system call will be denied by returning `ENOSYS` (“function not implemented”) instead of `EAGAIN` (“resource temporarily unavailable”). We do not recommend resuming the system call again immediately because frequently triggering failed system calls could introduce overhead. Later, if the resources are enough, the dropped system calls can be resumed and executed again. Otherwise, the system call executes normally.

Dropping system calls of misbehaving containers can effectively prevent misbehaving containers from competing for kernel resources with well-behaved containers, enhancing isolation. However, if a well-behaved container is misidentified as misbehaving, dropping system calls of misbehaving containers will hurt the performance of this well-behaved. But based on our high accuracy of container identification (96.5%), the performance of well-behaved containers will not be affected significantly. We argue that this is acceptable for enhancing isolation and ensuring stable performance of the whole system.

Table 5: Effectiveness of Pareto-based container identification

NoMC	0	2	4	8	16
Theoretic IP	p100	p98	p96	p92	p84
Computed IP	p99.6	p97.8	p96.3	p91.4	p84.1

In this table, “NoMC” implies “Number of Misbehaving Containers”.

This implementation provides high generality and flexibility. First, *syscall cgroup* supports all system calls as our solution does not rely on any syscall-specific features and we leverage stable system call interface to account for usage of system calls. Second, users can enable or disable kernel resource isolation for any system call and container based on existing cgroups mechanism. In addition, users can also flexibly limit the rate or the times of system calls instead of just allowing or denying system calls (like seccomp).

5. Evaluation

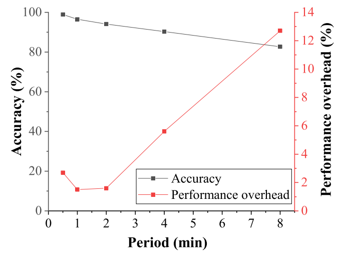
In this section, we present the experimental evaluation of *Valve*. Experiments were performed in Linux version 4.4.15 and 5.6.0. The results of both kernels are similar, demonstrating the generality of our design. So we mainly discuss the results based on Linux kernel 4.4.15. Our evaluation answers the following questions:

- Is the Pareto-based approach effective for container identification and, how accurate is it? (Section 5.1)
- Dose *Valve* successfully enhance kernel resource isolation for containers? (Section 5.2)
- What is the performance overhead of our system? (Section 5.3)

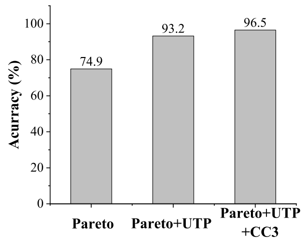
5.1. Container Identification

Effectiveness. We first demonstrate the effectiveness of our Pareto-based approach on identifying misbehaving containers. For this experiment, we ran 100 containers on a host, of which 0-16 were misbehaving. Hardware is the same as described in Section 2.1. Table 5 shows the well-behaved/misbehaving inflection point (IP) value we computed based on Pareto distribution. In this table, the theoretic IP value depends on the number of misbehaving containers. For example, if there are 2 misbehaving containers, the theoretic IP value should be p98. As shown in the table, the computed IP is very close to the theoretic value, which suggests the Pareto-based approach can effectively identify misbehaving containers.

Accuracy. As depicted in Table 5, the computed IP values widely range from p84.1 to p97.8 for different number of misbehaving containers, which highlights why a constant IP (threshold) value is not accurate. We periodically update the threshold value, but we need to select the “best” period to balance the accuracy and performance. Here we totally ran 100 containers, including Nginx containers and misbehaving containers for 24 hours. We randomly adjusted the number of misbehaving containers (from 0 to 16). We measured accuracy by counting the number of containers that were identified falsely.



(a) Accuracy and performance vary with period.



(b) Accuracy improved by updating threshold periodically (UTP) and classifying container into three types (CC3).

Figure 10: The “best” period and accuracy improvement.

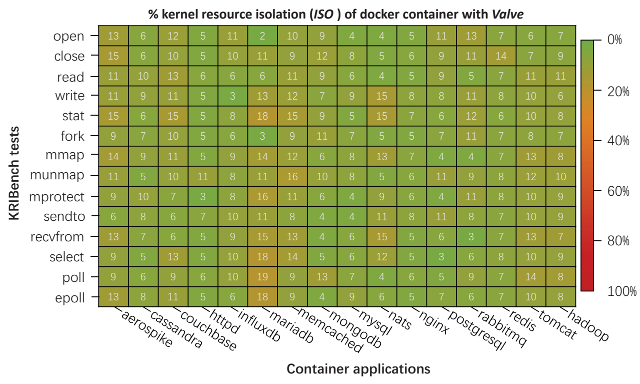


Figure 11: Enhanced kernel resource isolation (lower is better).

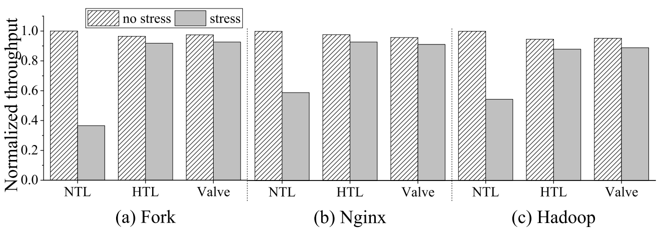


Figure 12: The normalized performance comparison of native task list (NTL), hierarchical task list (HTL) and *Valve*

We used `Apache benchmark` to measure the average throughput of all Nginx containers, then performance overhead was calculated based on the average throughput of Nginx containers that ran without misbehaving containers.

As shown in Fig. 10(a), the accuracy decreases as the period gets longer. The performance rises to a high point when the period is 1.5 minutes. This is because updating overhead is significant if the period is too short. However, if the period is too long, the accuracy is so low that isolation cannot guarantee the performance while leaving lots of system overhead. Based on our experiments, 1.5 minutes is the “best” period to achieve both high accuracy and low overhead. In addition, we also evaluate accuracy improvement by updating threshold periodically (denoted as “UTP” in Fig. 10(b) and period is set to 1.5 min) and classifying container into three types (denoted as “CC3” in Fig. 10(b)). Results in Fig. 10(b) illustrates that updating threshold periodically can improve accuracy by 18.3% and classifying container into three types helps to improve the accuracy by 3.3% further.

The results show that a static period can improve the accuracy to 96.5%. So there is little optimization space to motivate us to design a dynamic period. And the threshold is changed periodically based on the period value. We detect the misbehaving containers based on the threshold. Once the system call usage is higher than a threshold, we can identify it as misbehaving. However, when the threshold becomes invalid and is not updated in time (this is decided by the period), the high system call rate needs to appear multiple times before we take action. This is why the accuracy we achieve is 96.5% rather than 100%. We argue that such little accuracy loss is acceptable and does not reduce the practicability and effectiveness of our system.

5.2. Isolation

Next, we demonstrate the effectiveness of our design on enhancing the kernel resource isolation of container-based virtualization. We used *KRIBench*, the 15 representative real world containerized applications and Hadoop to evaluate kernel resource isolation for containers. Hadoop is a typical workload on cloud computing and is used to check if our design works for applications other than the 15 applications we analyzed before. Testbed and methodology are the same as described in Section 2.1, except that we modified the source code of Docker to configure our *syscall cgroup* for a container when it launches. To answer the second question, we repeat the *KRIBench* tests on *Valve*.

As shown in Fig. 11, all *KRIBench* tests have a lighter performance impact on container applications, which suggests a higher degree of kernel resource isolation. Overall, the kernel resource isolation is improved by a maximum of 98%, and an average of 37.7%, across all tests and workloads. More specifically, when running `open` test in misbehaving container, the average performance degradation of well-behaved container is improved significantly, from 100% to 7.9%. The improvement comes from the fact that our system can effectively reduce consumable resource competition in kernel. When misbehaving container competes for exclusive resource, the maximum and average slowdown of well-behaved container are 19% and 8.7%, respectively. Compared to the native kernel performance (67% and 42.2%, respectively), our design can also help isolate exclusive resource inside the kernel. The results of Hadoop (the last column) reveal that the conclusions are still relevant. All the above results clearly prove that *Valve* can restrict misbehaving containers, reduce competition of kernel resources, improve performance degradation of well-behaved containers, and enhance kernel resource isolation.

We further use `fork` as an example to demonstrate the enhanced isolation by comparing *Valve* and hierarchical task list (described in Section 2.2.2). Methodology is same as described in Section 2.2.2. We used `fork` as a microbenchmark, also Nginx and Hadoop as macrobenchmarks. Fig. 12 shows the relative performance of them normalized to native task list without stress. As shown in Fig. 12(a), for native task list, the

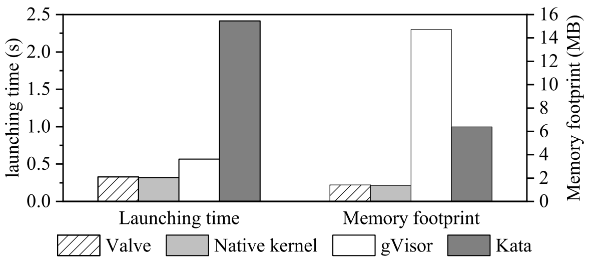


Figure 13: Launching time and memory footprint (lower is better).

throughput of `fork` degrades by 63.41% due to the stress. In comparison, the performance degradation of *Valve* is much less (4.97%). Such performance improvement clearly suggests that the kernel resource isolation is enhanced. And the effectiveness of our design is comparable with hierarchical task list (4.78%) but is more general than hierarchical task list. As shown in Fig. 12(b) and Fig. 12(c), tests on Nginx and Hadoop show a similar trend.

5.3. Performance Overheads

In this section, we evaluate the overhead of our system on launching time, memory footprint and overall performance, compared with Docker running on native kernel. As gVisor and Kata containers are widely used methods for isolation, we also compared *Valve* with these systems. In the experiments, we execute the same operations on the above platforms.

Launching time. To evaluate the launching time, we started a Docker container on Ubuntu 16.04 and used the *time* command to print out how long it takes for launching that container. Fig. 13 depicts the time of launching a new container on various platforms. Compared to the vanilla kernel, it takes 0.3287s to launch a new Docker container on *Valve*, which only adds 0.009s additional time. The result indicates that our optimizations incur negligible overhead (2.86%), which mostly spends on initializing per-container cgroup. In contrast, gVisor takes 1.72× time to boot a dedicated user-space kernel while Kata container needs 7.34× time to launch a new container. The reason is that the overhead of per-container kernel and hypervisor add more than 2s to the total startup.

Memory footprint. To measure the memory footprint, we ran an Nginx container and leveraged the *docker stats* command to collect memory usage of the container. As illustrated in Fig. 13, each container running on our optimized kernel consumes 1.394 MB memory, which only adds 0.02 MB (1.46%) compared to the container running on native kernel. Contrast that with *Valve*, gVisor and Kata containers introduce 10.56× and 4.58× memory footprints on the Docker container. As our solution builds on existing cgroups mechanism and takes advantage of the lightweight containers, the memory consumption overhead of *Valve* is remarkably negligible compared to existing popular container isolation methods.

Overall performance. To evaluate the overall performance of our system, we executed the representative applications and workloads shown in Table 1 and Hadoop, then collected their performance metrics. Fig. 14 illustrates the relative performance of applications normalized to Docker running on native kernel. Compared to the vanilla throughput, our system provides significant isolation on kernel resources while only introduces the maximum and average performance overhead by 2.5% and 1.3%, respectively. As shown in the figure, our system performs much better than gVisor and Kata containers. For example, compared to gVisor, the throughput of *Nginx*, *PostgreSQL* and *Redis* in *Valve* is 10.3×, 4.1× and 12.8×, respectively. In contrast with Kata containers, the throughput of *Nginx*, *PostgreSQL* and *Redis* in *Valve* is 6.9×, 5.6× and 14.3×, respectively. The overhead in gVisor and Kata containers mainly suffers from the per-container kernel and (or) heavy hypervisor.

6. Related works

Isolation analysis. Container-based virtualization allows multiple lightweight container instances to run on a sharing host OS. There are various implementations of container-based virtualization, such as Docker [2], LXC [29], OpenVZ [30], FreeBSD Jails [31] and Solaris Zones [32]. Generally, these systems leverage some OS mechanisms to provide isolation between containers. Some works focusing on evaluating the performance and isolation of containers [3, 4, 5, 9, 10] demonstrate that isolation is still a concern due to the sharing of host kernel. However, these works evaluate isolation at a coarse-grained application level and do not analyze the root causes. In this work, we quantify container isolation at system call level and identify several kernel resources competition to explain the isolation issues.

Kernel mechanism optimization. Several works focus on addressing specific isolation problems by optimizing kernel mechanisms. Huang *et al.* [21] analyzed the semantic gap on resources view and developed a sysfs namespace to export effective resources, including the number of cores and the amount of memory that are available to a container. Similarly, Sun *et al.* [22] proposed security namespace to enable containers to have an autonomous control over their security policies. Gao *et al.* [15] designed a power-based namespace partitioning power consumption at a container level to defend against the synergistic power attack, which can maximize power attack effects by exploiting leaked host information. Gao *et al.* [16] also explored several exploiting strategies to escape the resources control by de-associating processes from their original cgroups. To address this problem, Khalid *et al.* [12] optimized Linux scheduler to charge CPU resource for softirq processing which is one of the strategies to escape cgroups. PINE [11] was proposed to address the issue of misallocation of storage resource. However, even with the above optimizations, container performance is still affected by poor kernel resource isolation. This work reveals several kernel resource isolation issues that are never discussed before and proposes a general system to address these issues.

Container runtime. Various runtimes were proposed to address isolation problems between containers. Hyper [33], Hyper-V containers [34], Clear containers [35], Kata containers [17], VMware vSphere Integrated containers [36], LightVM

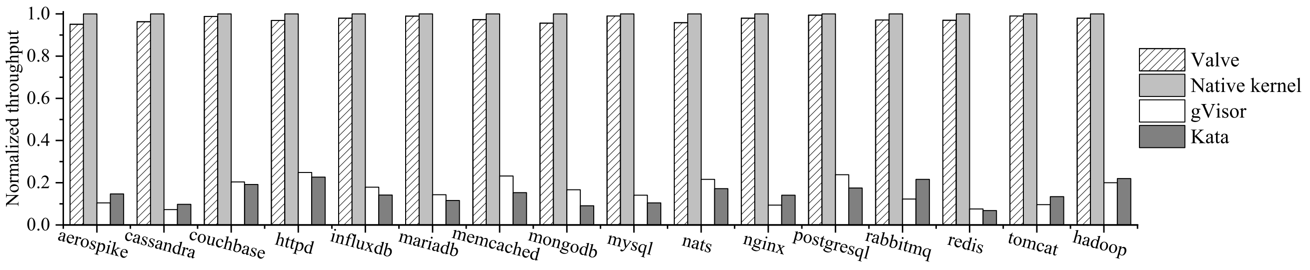


Figure 14: Normalized throughput of applications (higher is better).

[18], Mavridis [37] and X-Containers [19] all leverage hypervisor to wrap containers with a dedicated kernel. Similarly, gVisor [20] runs containers within a user-space kernel written in Go. However, these platforms can cause significant performance overhead because of the additional kernels. Instead, *Valve* is much lighter than a native OS kernel, which incurs negligible performance overhead.

Kernel isolation. Some other approaches were proposed to enhance container isolation, assuming a threat model different from ours where even the host kernel cannot be trusted. On the one hand, SCONE [38] proposed to run containers inside Intel SGX enclaves. On the other hand, because attacks against containers often leverage kernel exploits through system call interface, Wan *et al.* [39] minimized the set of system calls for containers and Win *et al.* [40] used system call interception to ensure the isolation between a compromised host and containers. However, none of the above techniques were designed to enhance kernel resource isolation for containers. Instead, *Valve* protects the performance of well-behaved containers from being interfered by misbehaving containers.

7. Conclusion

In this paper, we first introduced *KRIBench*, a microbenchmark, which is consisted of a set of representative real-world containerized applications and can quantify kernel resource isolation for containers. Next, we revealed several isolation issues of Docker containers using *KRIBench*, and further identified and analyzed two categories of kernel resources competition: *consumable resource* and *exclusive resource*. Finally, we proposed a general and flexible design, named *Valve*, which enhances kernel resource isolation containers by limiting usage of system calls for misbehaving containers. *Valve* employs Pareto distribution to identify misbehaving containers and uses *supply-demand model* to enforce isolation and decrease performance overhead. The experimental results show that our approach effectively isolates kernel resources among containers and performance overhead is negligible compared to the native systems and existing solutions.

References

[1] K. Suo, Y. Zhao, W. Chen, J. Rao, An analysis and empirical study of container networks, in: Proceedings of IEEE Conference on Computer Communications (INFOCOM), 2018, pp. 189–197.

[2] Docker, `https://www.docker.com/` Accessed April 4, 2022.

[3] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance comparison of virtual machines and Linux containers, in: Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015, pp. 171–172.

[4] P. Sharma, L. Chaufournier, P. Shenoy, Y. Tay, Containers and virtual machines at scale: A comparative study, in: Proceedings of International Middleware Conference (Middleware), 2016, pp. 1–13.

[5] M. Plauth, L. Feinbube, A. Polze, A performance survey of lightweight virtualization techniques, in: Proceedings of European Conference on Service-Oriented and Cloud Computing (ESOCC), 2017, pp. 34–48.

[6] Linux namespaces, `http://man7.org/linux/man-pages/man7/namespaces.7.html` Accessed April 4, 2022.

[7] Linux control groups, `http://man7.org/linux/man-pages/man7/cgroups.7.html` Accessed April 4, 2022.

[8] D. Skarlatos, Q. Chen, J. Chen, T. Xu, J. Torrellas, Draco: Architectural and operating system support for system call security, in: Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020, pp. 42–57.

[9] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, J. Owens, Quantifying the performance isolation properties of virtualization systems, in: Proceedings of workshop on Experimental Computer Science (ExpCS), 2007.

[10] M. G. Xavier, I. C. De Oliveira, F. D. Rossi, R. D. Dos Passos, K. J. Matteussi, C. A. De Rose, A performance isolation analysis of disk-intensive workloads on container-based clouds, in: Proceedings of International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2015, pp. 253–260.

[11] Y. Li, J. Zhang, C. Jiang, J. Wan, Z. Ren, PINE: Optimizing performance isolation in container environments, IEEE Access 7 (2019) 30410–30422.

[12] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, A. Akella, Iron: Isolating network-based CPU in container environments, in: Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2018, pp. 313–328.

[13] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, L. Peterson, Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors, in: Proceedings of European Conference on Computer Systems (EuroSys), 2007, pp. 275–287.

[14] O. Laadan, J. Nieh, Operating system virtualization: practice and experience, in: Proceedings of Annual Haifa Experimental Systems Conference (SYSTOR), 2010, pp. 1–12.

[15] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, H. Wang, Containerleaks: Emerging security threats of information leakages in container clouds, in: Proceedings of International Conference on Dependable Systems and Networks (DSN), 2017, pp. 237–248.

[16] X. Gao, Z. Gu, Z. Li, H. Jamjoom, C. Wang, Houdini's escape: Breaking the resource rein of Linux control groups, in: Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS), 2019, pp. 1073–1086.

[17] Kata Containers, `https://katacontainers.io/` Accessed April 4, 2022.

[18] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, F. Huici, My VM is lighter (and safer) than your container, in: Proceedings of Symposium on Operating Systems Principles (SOSP), 2017, pp. 218–233.

[19] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Re-

nesse, H. Weatherspoon, X-Containers: Breaking down barriers to improve performance and isolation of cloud-native containers, in: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019, pp. 121–135.

[20] gVisor: Container runtime sandbox, `https://github.com/google/gvisor` Accessed April 4, 2022.

[21] H. Huang, J. Rao, S. Wu, H. Jin, K. Suo, X. Wu, Adaptive resource views for containers, in: Proceedings of International Symposium on High-Performance Parallel and Distributed Computing (HPDC), 2019, pp. 243–254.

[22] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, T. Jaeger, Security namespace: making linux security frameworks available to containers, in: Proceedings of USENIX Security Symposium (USENIX Security), 2018, pp. 1423–1439.

[23] X. Ren, K. Rodrigues, L. Chen, C. Vega, M. Stumm, D. Yuan, An analysis of performance evolution of linux's core operations, in: Proceedings of Symposium on Operating Systems Principles (SOSP), 2019, pp. 554–569.

[24] V. Tarasov, E. Zadok, S. Shepler, Filebench: A flexible framework for file system benchmarking, login: The USENIX Magazine 41 (2016) 6–12.

[25] R. Love, Linux Kernel Development, 3rd Edition, Pearson Education, 2010.

[26] Pareto distribution, `https://en.wikipedia.org/wiki/Pareto_distribution./` Accessed April 4, 2022.

[27] M. Hao, L. Toksoz, N. Li, E. E. Halim, H. Hoffmann, H. S. Gunawi, Linnos: Predictability on unpredictable flash storage with a light neural network, in: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2020, pp. 173–190.

[28] Sliding window based rate limiter, `https://www.codementor.io/@arpi-tbhayani/system-design-sliding-window-based-rate-limiter-157x7sburi/` Accessed April 4, 2022.

[29] LXC: Linux Containers, `https://linuxcontainers.org/` Accessed April 4, 2022.

[30] OpenVZ Containers, `https://openvz.org/` Accessed April 4, 2022.

[31] P.-H. Kamp, R. N. Watson, Jails: Confining the omnipotent root, in: Proceedings of International System Administration and Networking Conference (SANE), 2000, pp. 116–127.

[32] Solaris Zones, `https://en.wikipedia.org/wiki/Solaris_Containers` Accessed April 4, 2022.

[33] Hyper Containers, `https://hypercontainer.io/` Accessed April 4, 2022.

[34] Hyper-V Containers, `https://docs.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/hyperv-container` Accessed April 4, 2022.

[35] Intel Clear Containers, `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-clear-containers-1-the-container-landscape.html` Accessed April 4, 2022.

[36] VMware vSphere Integrated Containers, `https://www.vmware.com/products/vsphere/integrated-containers.html` Accessed April 4, 2022.

[37] I. Mavridis, H. Karatza, Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing, Future Generation Computer Systems (FGCS) 94 (2019) 674–696.

[38] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, et al., Scone: Secure Linux containers with Intel SGX, in: Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016, pp. 689–703.

[39] Z. Wan, D. Lo, X. Xia, L. Cai, S. Li, Mining sandboxes for Linux containers, in: Proceedings of International Conference on Software Testing, Verification and Validation (ICST), 2017, pp. 92–102.

[40] T. Y. Win, F. P. Tso, Q. Mair, H. Tianfield, PROTECT: Container process isolation using system call interception, in: Proceedings of International Symposium on Pervasive Systems, Algorithms and Networks & International Conference on Frontier of Computer Science and Technology & International Symposium of Creative Computing (ISPAN-FCST-ISCC), 2017, pp. 191–196.