This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383988

vKernel: Enhancing Container Isolation via Private Code and Data

Hang Huang, Honglei Wang, Jia Rao *Member, IEEE*, Song Wu *Member, IEEE*, Hao Fan, Chen Yu *Member, IEEE*, Hai Jin *Fellow, IEEE*, Kun Suo *Member, IEEE*, Lisong Pan

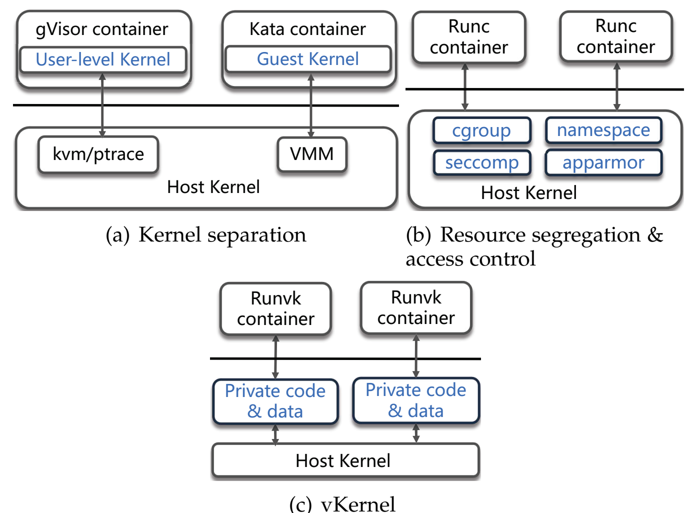
Abstract—Container technology is increasingly adopted in cloud environments. However, the lack of isolation in the shared kernel becomes a significant barrier to the wide adoption of containers. The challenges lie in how to simultaneously attain high performance and isolation. On the one hand, kernel-level isolation mechanisms, such as *seccomp*, *capabilities*, and *apparmor*, achieve good performance without much overhead, but lack the support for per-container customization. On the other hand, user-level and VM-based isolation offer superior security guarantees and allow for customization since a container is assigned a dedicated kernel, however, at the cost of high overhead. We present *vKernel*, a kernel isolation framework. It maintains a minimal set of code and data that are either sensitive or are prone to interference in a virtual kernel instance (vKI). vKernel relies on inline hooks to intercept and redirect requests sent to the host kernel to a vKI, where container-specific security rules, functions, and data are implemented. Through case studies, we demonstrate that under vKernel user-defined data isolation and kernel customization can be supported with a reasonable engineering effort. An evaluation of vKernel with micro-benchmarks, cloud services, real-world applications show that vKernel achieves good security guarantees, but with much less overhead.

Index Terms—container, kernel, isolation, performance.

1 Introduction

Containers, also known as operating system (OS)-level virtualization, are increasingly adopted in data center management due to their high performance compared to hypervisor-based virtualization, i.e., virtual machines (VMs) [1]. While OS-level virtualization offers near-native performance, it does not provide adequate isolation between containers since all on one host share the same OS kernel [2]. The weak isolation has been shown to affect both the security and performance of containers in shared environments [3]. On the one hand, the ability of containers to directly access the shared kernel opens up opportunities for attackers to cause information leakage [4], privilege escalation [5], and denial of services [6]. On the other hand, sharing the host kernel not only leads to contentions on shared data structures that cause performance interference but also disallows application-specific customizations or optimizations to the kernel [7]. The lack of isolation in the shared kernel has become a barrier for container adoption in new computing paradigms [8], such as serverless [9].

As shown in Figure 1, there exist several mechanisms for inter-container kernel isolation. One approach is to deploy a dedicated kernel different from the host kernel in each con-



(a) Kernel separation

(b) Resource segregation & access control

(c) vKernel

Fig. 1: The comparison of kernel isolation approaches

tainer. Since containers share nothing with the host kernel or other containers, kernel separation offers superior isolation. VM-based kernel separation deploys each container to a separate VM running a full-fledged guest kernel. While VM-based isolation provides strong protection and compatibility to legacy applications, it requires a virtual machine monitor (VMM) to expose virtualized hardware to guest kernels, resulting in a larger per-container resource footprint and slower startup times. Lightweight VMs, such as Kata [10] and Firecracker [11] devise a minimal guest kernel and VMM to reduce the memory footprint of containers but still incur non-negligible overhead compared to native containers due to the additional layer of indirection at the VMM. Application or user-level kernels, such as gVisor [12], [13], intercept application system calls to create a system interface

- *H. Huang, H. Wang, S. Wu, H. Fan, C. Yu, H. Jin, and L. Pan are with 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, China. H. Fan, S. Wu, and H. Jin are also with Jinyinhu Laboratory, Wuhan, China. Jia Rao and Hao Fan are the corresponding authors. Email: jia.rao@uta.edu, haofan@hust.edu.cn.*
- *J. Rao is with the Department of Computer Science, University of Texas, Arlington, TX 76019 USA.*
- *K. Suo is with the Department of Computer Science, Kennesaw State University, Kennesaw, GA 30144 USA.*

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383988

similar to the host kernel without the need for hardware virtualization. However, request interception inevitably causes excessive context switches and hence substantial overhead.

Another approach to kernel isolation is to leverage the existing resource management and security mechanisms in the OS kernel, such as *cgroups*, *namespaces*, *capabilities* [14], *seccomp* [15], and *apparmor* [16], to provide containers with segregated views of system resources and restrict their accesses to system calls, privileged functions, and sensitive files. While this approach achieves near-native performance due to its tight coupling with the host kernel, it does not provide adequate isolation or allow applications to customize kernel configurations or policies. To avoid the cost of request indirection, such as context switches, unikernels [17], [18], [19] are proposed to run a container and the guest kernel in the same address space. Although this approach helps mitigate the overhead, it requires significant engineering efforts to port legacy applications to unikernels.

This paper proposes, *vKernel*, a kernel isolation framework for containers. Unlike the existing approaches that main separate kernels for a container and the host, vKernel maintains a minimal set of private code and data for each container that is necessary for isolation while shares the remaining with the host kernel. The private code and data includes that involved in the existing kernel security checks, such as system calls, as well as functions and data that cause interference between containers. At heart, vKernel relies on inline hooks to intercept and redirect requests sent to the host kernel to a *vKernel instance* (vKI), a Rust-based kernel module where a container-specific system call table, capabilities, file permission lists, and other user-defined functions and data are implemented and stored. The vKI can be dynamically loaded and updated as a kernel module and is independent from the host kernel. We demonstrate that vKernel supports the same types of security checks the existing kernel security mechanisms offer but with less overhead. We further showcase how users can customize vKernel to improve data isolation in the commonly-used `futex` system call, enable different configurations of shared kernel parameters, and support customized scheduling that only takes effect in a particular container.

This paper makes the following contributions:

- A comprehensive study of the existing kernel isolation approaches and identify their limitations on usability, performance, and specialization.
- A novel kernel isolation framework that allows individual containers to maintain private code and data for stronger, customizable, and more efficient isolation.
- An evaluation of vKernel with micro-benchmarks, cloud services, real-world applications, case studies on user-defined isolation and customization, and several recently reported vulnerabilities shows the effectiveness and efficiency of vKernel.

2 Background and Motivation

Container isolation has recently attracted much attention in industry and academia due to not only security but also a growing concern of performance interference among containers. An analysis of the Alibaba cloud trace [20] found that only 1.63% of the servers run one container per node while more than 80% of the servers run more than 6 containers. The lack of isolation in native containers, such as Docker, impedes high-density container deployment due to inter-container interference on the shared host kernel. In what follows, we discuss the limitations of user-level kernels, VM-based kernel isolation, and the existing isolation in the Linux kernel.

2.1 User-level Kernel Isolation

Kernel isolation at the user-level redirects the requests to the host kernel to the application-specific kernel implemented at the user-level. The key to redirecting user-level requests and the major source of overhead is to intercept requests to the host kernel. Among a variety of mechanisms for request interception, `ptrace` is a tracing technique widely used to implement user-level OS kernels. In `ptrace`, one process (the "tracer") observes and controls the execution of another process (the "tracee"). A tracer can emulate an entire foreign kernel with mutated system calls. The popular user-level kernel framework gVisor uses a sentry process to trace application processes. System calls issued by application processes are intercepted by `PTRACE_SYSEMU` and handled by the sentry process. Sentry emulates most system calls and replaces the native system call with a user-level implementation. It also redirects I/O operations to a file proxy and implements task scheduling using Go.

While user-level kernel isolation achieves strong data isolation, it suffers from high context switch and user-kernel mode switch overhead. `Ptrace` requires multiple context switches between the tracer and the tracee and gVisor may incur additional context switches due to I/O redirection and task scheduling.

2.2 VM-based Kernel Isolation

An alternative way to kernel isolation is to host containers in separate VMs, each running a dedicated guest kernel. VM-level kernel isolation offers strong protection between containers as it is difficult for malicious users to escape the guest kernel and compromise the host kernel or the hypervisor. Although there have been significant efforts dedicated to optimizing VMs to reduce virtualization overhead and memory footprint, lightweight VMs, such as those employed in Kata [10] and Firecracker [11], still incur non-negligible overhead compared to native containers. X-container [21] uses a unikernel as the guest kernel to further close the performance gap. However, it requires a significant engineering effort to port legacy applications to unikernels and hence unikernel-based VM isolation is not readily available in production systems.

2.3 Isolation mechanisms in the Linux Kernel

The existing container isolation mechanisms in the Linux kernel are based on isolated resource views and security checks. `Namespaces` provide containers with isolated views of process IDs, file systems, and network interfaces while `cgroups` impose hard and soft limits on container resource allocation. Security checks, which are often based on a user-provided security profile, restrict container access to specific system calls, privileged code, and sensitive files

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383988

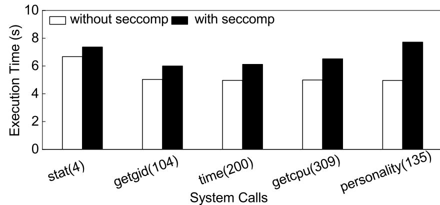


Fig. 2: The performance of system calls w and w/o *seccomp*. The number in the parenthesis is the system call ID.

according to either a white or black list. Neither the isolated resource view nor the access control imposed by security checks can provide adequate performance isolation between containers that make legitimate requests.

Seccomp is an eBPF-based system call filtering mechanism. It implements fine-grained restrictions over dangerous system calls before the actual system calls are invoked. Users define a system call white-list through a seccomp profile, and the container engine loads the seccomp filter generated based on the profile to the host kernel. The seccomp filter applies to all processes belonging to a container. Once the container invokes a system call, the BPF interpreter intercepts the request and executes the corresponding seccomp filter. The seccomp filter checks whether the system call is in the white-list and if the arguments for the call meet predefined requirements. The filters return the signal "`KILL`" if the requested system call is excluded from the white-list and return "`ERRNO(1)`" if the parameters are invalid. Signal "`ALLOW`" lets the container invoke the requested system call. With the help of seccomp, containers are restricted to access a few predetermined system calls that are deemed safe.

However, seccomp has several limitations. First, BPF does not support dynamic memory allocation in constructing the filter and hence seccomp has to statically write the rules for system call check into an eBPF program. As the eBPF filter is a generic program that is executed whenever a system call is invoked, the filter program has to check the invoked system call sequentially. The sequential check incurs increasing overhead to system calls that reside at the bottom of the profile. To quantify the overhead, we wrote a micro-benchmark that repeatedly invokes various system calls with different system call IDs for 10 million times and measured their invocation time with and without seccomp enabled. As shown in Figure 2, seccomp introduces non-negligible overhead to system call invocations and the performance slowdown ranges from 10% to 55.5%.

Capability is a permission check mechanism for privileged functions in the Linux kernel. *Capability* works at a per-process level and compares a process' capabilities with the privilege level of the functions it intends to invoke. A process may possess multiple capabilities but the *effective capability* is the one that takes effect. Upon a function call, the Linux kernel performs the permission check by invoking `capable()` to check the permission bits in a process' effective capability against the function. Since capability check is a bit operation, it does not cause noticeable overhead.

However, Capability works at the process-level based on inheritance rather than at the container-level. Therefore, if a process escapes the permission check by tampering with its effective capability, it can bypass the security checks imposed by its host container. For example, a possible way to escape is to invoke *commit_creds()* in the kernel, which rewrites the process's effective capability [22], [23]. Once the process obtains full capabilities, the isolation enforced by the Capability mechanism fails.

TABLE 1: Shared kernel data structures and parameters.

Data structures	syscalls	Kernel parameters	syscalls
dentry_hashtable	79	overcommit_memory	45
mount_hashtable	67	tainted	34
mm_percpu_wq	7	nr_open	19
system_unbound_wq	7	max_map_count	14
kblockd_workqueue	5	vfs_cache_pressure	13
system_power_efficient_wq	3	wmem_default	8
idents_hash	2	rmem_default	7
mountpoint_hashtable	2	protected_symlinks	5
futex_queues	1	protected_fifos	4

Apparmor restricts programs' access to sensitive files based on path-based access check. Only the matched paths in a white-list are allowed to be accessed. An example apparmor profile (black list) may deny any write access to files in folders `/proc` and `/sys` as those operations could alter system-wide configurations affecting other containers. At container startup, the container engine loads the profile, and apparmor analyses the profile and generates a deterministic finite automation engine, which verifies on every file access whether the request violates the paths denied in the profile.

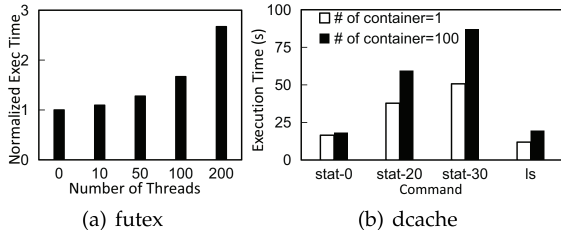
Compared to *seccomp* and *capability* that use white lists for security and permission checks, Apparmor uses a blacklist to check file access. Since *Apparmor* lacks the awareness of containers and files associated with them, every file access, including those to sensitive and non-sensitive files, needs to go through *Apparmor* check. This inevitably introduces slowdowns to overall file system performance. We used a micro-benchmark to resolve 10 millions file paths and tested the completion time with and without *Apparmor* enabled. Results show that apparmor incurs a consistent overhead ranging from 11.5% to 22.2% on all file accesses.

Strengths and weaknesses The three discussed security mechanisms are executed in the host kernel whenever a container enters the kernel mode and hence do not suffer from the request interception and redirection overhead as do in user-level kernels and VM-based isolation. However, they share some common weaknesses: **1)** Whitelisting and blacklisting-based kernel-level isolation are not as strong or flexible as approaches that maintain separate kernels for containers. Whitelisting can be overly restrictive and blacklisting is not effective against unknown threats. Most importantly, except for the security check, there is no physical isolation between containers, which may lead to evasion of the permission check. **2)** The existing security mechanisms fall short of supporting container-specific kernel customization or **3)** data isolation among containers.

Lack of data isolation. Many kernel data structures are allocated at kernel initialization and globally shared in the kernel space. This allows for fast memory allocation and deallocation as well as facilitating data reuse. Typically, these data structures are allocated from fixed sized memory blocks, which cannot be expanded after kernel boot. Performance interference due to such shared data structures can manifest in two ways. First, concurrent updates to shared data can lead to severe contentions on locking, which can lead to drastic performance degradation. Second, contain-

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383988



(a) futex (b) dcache

Fig. 3: Slowdown due to contentions in `futex` and `dcache`. The suffix in `stat` indicates the depth of file paths.

ers may unintentionally or intentionally exhaust the fixed memory blocks needed for shared data and cause denial of services or out-of-memory errors. We used the Linux system called fuzzer system [24] to monitor data accesses to shared kernel structures from all Linux system calls as shown in Table 1. `Dentry_hashtable` and `mount_hashtable`, which are lock-protected and allocated from a fixed pool of memory, can be accessed from more than 60 different system calls. Similarly, we also identified several kernel parameters that are commonly referenced in many system calls with `overcommit_memory` as the most referenced parameter. The existing container isolation mechanisms in the kernel do not isolate the shared data structures or parameters.

To demonstrate the sophistication of data isolation and the severity of performance interference, we examined the sharing of the `futex_queues` structure in system call `futex`. Threads that fail to acquire a lock are placed in a sleep state in the `futex_queues` where threads from different containers may collide in the same bucket. There are a slew of implications of performance interference between threads in the same bucket, including the order of wakeup, the selection of CPUs to execute the threads after wakeup, and data locality. To show the severity of the problem, we placed threads from two containers in the same `futex_queues` bucket and measured one container's performance of `futex` operations as the number of threads in another container that occupied slots in the `futex_queues` bucket gradually increased. Figure 3 shows as much as 167.6% performance slowdown as interference ramped up. Experiments with the `ls` and `stat` file operations also show significant performance degradation due to contentions from colocated containers.

3 vKernel: Design and Implementation

Overview. To simultaneously achieve data/performance isolation, kernel customization/specialization, and low overhead, we propose *vKernel*, a generic container isolation framework. Unlike the existing kernel isolation mechanisms, vKernel enhances isolation by maintaining private copies of sensitive code (e.g., system calls and privileged functions) and shared data that may cause interference (e.g., files and kernel data structures), and kernel configurations into a per-container *vKernel instance* (vKI). Note that vKernel does not seek to improve security and isolation beyond user-level kernels and VM-based kernel isolation. The objective is to achieve similar strong isolation with a minimal set of private code and data and without the high cost of duplicating kernels in each container. vKernel begins with isolating sensitive code and data identified by *seccomp*, *capability*, and *apparmor* and uses a security profile to specify them. Furthermore, vKernel allows users to define customized isolation rules and kernel configurations for each container.

TABLE 2: Comparison of LKM with eBPF

	features	LKM	eBPF
Usability	memory allocation	✓	
	kernel function access	✓	limited
	kernel function hook	✓	hard
safety	program verifier		✓
	runtime isolation		✓

Figure 4 shows the vKernel design. vKernel consists of a container runtime (*runvk*), a vKernel builder (*vkernel-builder*), a system-wide vKernel manager (vKM), and per-container vKernel instances (vKI). *runvk* is modified from the widely-used container runtime (*runc*), which is responsible for loading, updating, and unloading vKIs for containers, and registering the container and its corresponding vKI in vKM. *runvk* is consistent with OCI (Open Container Initiative) specifications, ensuring compatibility with existing container development tools. Users simply need to select runvK as their runtime when creating containers with container development tools such as Docker, or container orchestration tools like Kubernetes. *vkernel-builder* analyzes container security profiles, verifies the safety of the vKI code, and finally generates a loadable module (*vKI.ko*). Notably, as shown in the table2, *eBPF* does not support memory allocation, can only access limited kernel functions, and is hard to hook kernel functions, therefore loadable kernel module is more suitable for vKernel. vKM is a loadable kernel module responsible for redirecting container kernel requests to corresponding vKIs. It relies on inline hooks to intercept system call and privileged function invocations. vKI is also a kernel module loaded when a container is launched and bound to the container based on the container's PID namespace. Both vKM and vKI are implemented using the rust language to ensure code safety. vKernel strikes a good balance between performance and isolation. Unlike VM-based methods that employ an additional kernel to achieve full isolation, vKernel only isolates the necessary data and code for container execution without redundant isolation, resulting in lower performance overhead. Compared to traditional Docker Runc, vKernel offers more comprehensive isolation that can achieve parameter isolation and code customization.

3.1 vKernel Builder

vKernel Builder (*vkernel-builder*) is a rust-based, automatic tool for building user-customized vKI. *vkernel-builder* by default uses a security profile that restricts container access

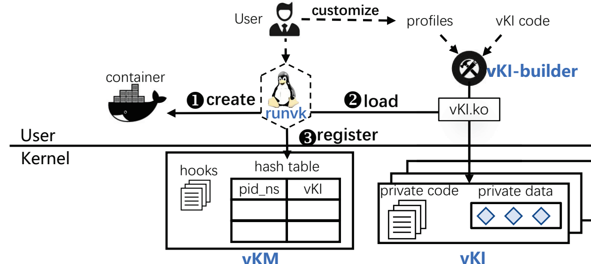


Fig. 4: The architecture of vKernel

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383988

to system calls, privileged functions, and sensitive files, as in *seccomp*, *capacity*, and *apparmor*. It also allows users to specify kernel code, data, and configuration to be included in a vKI. The output of *vkernel-builder* is a loadable rust kernel module `vKI.ko` if vKI passes rust compilation and contains no unsafe code. Note that `vKI.ko` can be reused for any containers sharing the same isolation requirement. Since building a vKI is done offline, it does not add any delays to container startup.

3.2 vKernel Manager

The responsibility of vKM is to intercept kernel requests, such as system calls and privileged functions, issued by a container and redirect them to the corresponding vKIs based on a hash table. At boot time, the host OS loads vKM as a loadable kernel module, which registers inline hooks for the default security checks and user-defined isolation. Note that vKM does not use the existing in-kernel monitoring hook mechanism (`ftrace`) due to its high indirection overhead. Instead, vKM uses inline hooks. It first looks up the addresses of the functions that need to be redirected and then builds a stub function by invoking `text_poke` to redirect them to new function implementations in vKM. Figure 5 shows examples of hooking the system call interface `do_syscall_64`, the entry function of the dentry cache `d_hash`, and the function that manages memory over-commitment `vm_enough_memory`. When the tracer function is invoked in the container, the request is redirected to registered call-back functions, e.g., replacing `do_syscall_64` with `vkm_do_syscall_64`. The call-back functions invoke the corresponding implementations of the intercepted functions in a container's vKI.

3.3 vKernel Instance

A vKernel instance (vKI) is a per-container rust kernel module responsible for container-specific security checks, data isolation, and user-defined customization. As vKI has its own code and data, it essentially serves as a minimal virtualized interface on top of the shared host kernel. With the help of `vkernel-builder`, users can define profiles based on their requirements and generate a vKI, i.e., a container-specific `vki.ko`. The profile specifies what system calls, privileged functions, and files the container is allowed to access as well as user-defined data isolation and other resource management policies. vKM stores the pairs of a container's PID namespace and a pointer to its corresponding vKI in a hash table. Note that a vKI can be

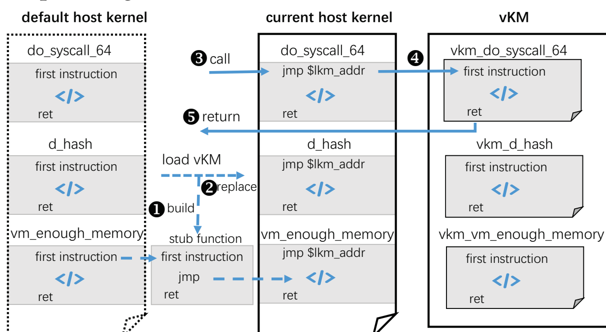


Fig. 5: The inline hooks in vKM

updated without restarting the container. A newly loaded vKI can be bound to the container's namespace, replacing the existing vKI, and subsequently become effective for the container. In what follows, we explain how vKI achieves more efficient isolation for system calls, permission checks, and file accesses as in *seccomp*, *capability*, and *apparmor*.

System call isolation. Linux kernel saves the addresses of all system calls in a global system call table (`sys_call_table`) and locates the implementation of a system call based on its ID. As the system call table is shared among all containers in a host, the existing security mechanisms for system call permission check, e.g., *seccomp*, use a permission filter to check a request system call against all calls in the table until a match is found. Contrary to this design, vKI keeps a private system call table `vki_sys_call_table` for each container, which only contains the system calls the container is allowed to access. The entries for all other system calls are marked as *NULL* in the private system call table, which will be denied by a `KILL` signal. The per-container private system call table ensures that the permission check can be completed in constant time regardless of the call ID.

Privileged function isolation.

For *capability*, if a process obtains elevated capabilities by exploiting the existing vulnerabilities of the kernel, it can evade the permission check. Inspired by the security isolation in VMs which leverages malicious processes' unawareness of hardware virtualization to prevent them from escaping from the guest kernel, we impose an additional permission check for privileged functions at the vKI. Each vKI maintains a read-only effective capability `vki_caps_effective` for a container. The container-wide capability is an upper bound on what processes can do within a container and overrides per-process capabilities if there is a conflict. When the `cap_capable` function is intercepted, the vKI first checks the request against the container's capability. If passed, vKI performs per-process permission check as done in *Capability*. Otherwise, a process evasion is detected and the request is denied. Since the container-wide capability is read-only and is not visible to processes within a container, vKI is more secure than the *capability* mechanism.

File isolation. For *Apparmor*, since all file accesses need to be checked, a majority of which does not involve sensitive files, *apparmor* imposes unnecessary overhead. To address this issue, vKI employs a two-step process for file permission checks. First, vKI leverages an unused bit in the file inode's `i_opflags` to indicate if a file is sensitive. The sensitive bit is set to 1 if any container's *apparmor* profile includes its path in its black-list. Second, at initialization, vKI scans a container's *apparmor* profile to identify sensitive files that should be checked upon file accesses and builds a hash table mapping from the sensitive files' inode numbers to the corresponding access permissions specified in the profile. Upon a file access, vKI intercepts the `generic_permission` function and checks the sensitive bit in the `i_opflags` in the requested inode. If the bit is clear, vKI immediately returns `ALLOW`. Otherwise, it looks up the access permissions of the inode in the hash table. If any match is found in the black-list, the file access is denied. vKI helps remove permission checks in accessing non-sensitive files.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383988

4 USER-DEFINED CUSTOMIZATION

vKernel is designed to provide a lightweight framework for isolation and customization. In addition to the three standard access controls, vKernel allows users to define customized rules for data isolation and kernel customization. The containers can achieve better performance with customization. In the following, we present four case studies. The case study on isolating Linux `dcache` demonstrates how to isolate shared kernel data structures without much change to the code accessing them while the `futex` system call study shows sophisticated isolation involving both code and data. We further present a case for allowing containers to configure their own kernel parameters. Last, we show how to enable a customized task scheduling policy for a particular container. Here, we leverage the isolation framework provided by vKernel to tailor the parameters and codes extensively utilized by containers. Additionally, we can employ profiling tools [25], [26] to pre-run the container, extract the code and parameters involved in its execution, and further customize the kernel specifically for containers.

4.1 Kernel data structure isolation

The Linux `dcache` mechanism caches the mappings between file paths and inodes in a `d_entry` cache. It is critical to fast file system operations that involve a large number of inode accesses, such as `ls` and `stat`. The key operation in `dcache` management is the allocation of a `d_entry` from the `dentry_hashtable`, which is a structure referenced by 79 system calls and a source of interference. vKernel allocates a private `dentry_hashbable` for each container to cache inodes specific to the container's file access. No significant changes to the kernel code are necessary unless replacing the original `d_hash` with `vkm_d_hash` in the vKM to redirect a container's file access.

4.2 `Futex` isolation

Fast userspace mutex (`futex`) is a widely-used system call applications use to implement efficient synchronization, such as POSIX mutex and barrier. Threads that fail to acquire a lock (i.e., waiters) are put to sleep in a wait queue. `Futex` maintains a single, system-wide `futex_queues` with multiple buckets. Threads waiting on the same lock are placed in the same bucket. `Futex` uses the address of the userspace lock as the key to `hash_futex` to select the bucket. Since `futex_queues` is shared among all containers, as discussed in Section 2.3, threads from multiple containers may collide in the same bucket. Performance interference manifests in two ways. First, interleaving threads from different containers in the same bucket compromises wake-up efficiency as one container need to scan other threads before locating a thread to wake up. Second, sleeping threads on the `futex_queues` are removed from CPU run queues and later will be inserted back into a run queue when they wake up. It is difficult to preserve threads' data locality, i.e., placing them back to the CPUs where they were running before sleep if multiple threads are simultaneously waking up from a shared queue.

vKernel provisions a dedicated `futex_queues` for each container in its vKI and devise private `futex` functions that operate only on the container-local queue. For example, the private `vki_hash_futex` function only maps a thread to a container local queue. It should be noted that the private kernel data allocated for containers exclusively consists of the management structure of the hash table. The memory overhead associated with this component remains at the KB level. As the number of containers grows, the memory overhead does not become excessively prominent. Container-specific wake up is more challenging as multiple containers may simultaneously wake up threads and attempt to insert them to CPU run queues. This may lead to locking on the same run queue to prevent simultaneous insertion. To completely isolate the wakeup process of different containers and preserve thread locality, we also isolate threads from different containers in their own CPU runqueues. With the help of `cgroups`, a reasonable change to the Linux completely fair scheduler (CFS) with approximately 200 lines of code ensures that threads are scheduled on their container-local CFS runqueue. As such, the thread is guaranteed to be on the same CPU where it ran, not only preserving data locality but also avoiding inter-container run queue contention. To enable this new wakeup mechanism, we override the generic `futex_wait` and `futex_wake` functions with new implementations in vKI and add a `is_waking` flag to each thread. `Vki_futex_wait` does not put a thread into sleep but forces the CPU scheduler to bypass the thread with the `is_waking`(0), emulating sleeping on the futex queue.

4.3 Kernel parameter isolation

The Linux kernel includes many tunable parameters for users to control its runtime behavior. Most of these parameters are global and shared among all containers on the same host. The change to a shared parameter will take effect for all containers. Although parameters local to a namespace is private to a container, there exist a vast majority of kernel parameters, some performance critical, need to be isolated.

For example, parameter `overcommit_memory` specifies whether an application can allocate a memory region in its virtual address space that exceeds the amount of available physical memory, and `overcommit_kbytes` and `overcommit_ratio` determine whether the current memory usage deems to be an overcommitment. While memory overcommitment leads to more flexible memory allocations, it also could result in memory thrashing. Setting this parameter indistinguishably for all containers inevitably leads to suboptimal performance. Parameter isolation not only requires the duplication of parameters per container but also needs to override kernel functions that report statistics associated with the parameters as well as those implementing the corresponding resource management policy. Specifically, to allow per-container memory overcommitment configuration, vKI replicates the three parameters for each container, overrides the handlers for the `procfs` to report per-container memory usage, and replaces the generic memory management functions, such as `vm_memory_committed`, with per-container vKI implementations.

4.4 Scheduling customization

The Linux kernel is equipped with four CPU schedulers – the default CFS, first-in-first-out (FIFO), round-robin (RR),

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383988

TABLE 3: The lines of code (LOC) of vKernel

	line of code	
runvk	1060+/72-	
vkernel-builder	718+	
	vKM	vKI
system call isolation	421	47
privileged function isolation	30	82
file isolation	202	94
futex system call isolation	46	209
kernel parameter isolation	42	392
dcache isolation	40	35
scheduling customization	10	276
total	751	1135

and deadline scheduler. In native Linux, scheduler selection can be made on a per-process basis but the configuration requires root privilege. Since containers are unprivileged and reside in userspace, they can only select the CFS scheduler. Different schedulers are desirable in different situations. For example, FIFO scheduling avoids frequent context switches and benefits throughput-oriented workloads. The challenges in enabling scheduling customization for containers are twofold: 1) the customized scheduling policy should only take effect on processes belonging to one container; 2) elevating container privilege to alter host-level scheduling is risky and hence should be forbidden.

To this end, we demonstrate how vKI can help derive customized scheduling for containers without changing host-level scheduling. Specifically, the objective is to emulate the effect of FIFO scheduling in a container on top of CFS scheduling in the host kernel. The methodology remains the same – intercepting generic scheduling functions and overriding them with container-specific functions implemented in vKI. Schedulers differ in the way they select the next task to run and where to insert a completed task back into the run queue. The CFS employs a global cfs_rq (a red-black tree) to manage all runnable threads, where threads from the same container are grouped and managed in a distinct sub-cfs_rq (a sub-tree). With this design in mind, we can implement container-specific scheduling strategies within these sub-cfs_rq, ensuring that such custom strategies affect only the threads contained within them. To emulate FIFO, vKI intercepts CFS functions `__enqueue_entity` and `pick_next_entity` and manipulates scheduling to ensure a process is always kept running until it exits and newly admitted processes are inserted to the tail of the partial CFS run queue of the container. Since the customized policy only works on the partial cfs_rq of the container, it can promise effectiveness inside the container, without affecting the scheduling of global cfs_rq and the fairness between containers.

5 Evaluation

In this section, we present and discuss the experimental results. We seek to answer three questions: (a) How is vKernel's performance compared with that of other kernel isolation approaches? (b) What are the benefits of user-defined isolation and customization enabled by vKernel? (c) How well does vKernel address the existing container vulnerabilities? Our experiments are performed on a PowerEdge R730 server equipped with dual 10-CPU Intel Xeon 2.30 GHz processors, 128 GB memory, and a 1.8TB SATA hard drive. We used Ubuntu 20.04 64bit and Linux kernel version

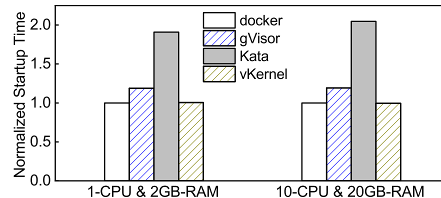


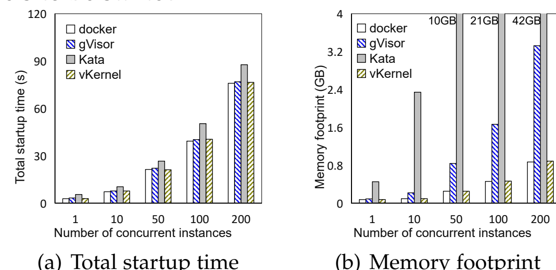
Fig. 6: Startup time of a minimal container image

6.0-rc7 as the host kernel. Docker 20.10.13, gVisor release-20210927.0, and kata 1.13.0 were used as the container technology. For comparison, we evaluate the following cases: the vanilla docker container with *seccomp*, *capability*, and *apparmor* enabled (*docker*), gVisor container with user-level kernel isolation (*gVisor*), container in a lightweight VM (*Kata*), and the docker container with vKernel enabled (*vKernel*). Each result was an average of 10 runs. The engineering effort to implement vKernel is summarized in Table 3. Except for the implementation of `runvk` using the Go language, other components of vKernel are implemented using rust.

5.1 Container startup

The efficiency of startup is important for short-lived containers, and the mechanisms used for kernel isolation may negatively impact container startup time. We test the startup time of the *alpine* OS [27] in containers with different kernel isolation approaches. The startup time is normalized to that of the docker and two hardware configurations were tested. As shown in Figure 6, vKernel does not cause a noticeable increase in containers' startup time. Note that the virtual kernel instances in vKernel were built offline and thus the result only included vKI's load time. As vKI can be reused by containers with similar security profiles, we do not expect vKI build time to be on the critical path of container startup. On average, it takes approximately 2.5s to build a vKI offline based on the default security profile. gVisor increases the startup time by 18.7% mainly due to the initialization of user-level tracer processes. In contrast, Kata needs to boot a VM before a container can be started which increases 90% startup time.

We also test the startup time and memory footprint when multiple instances are simultaneously started as shown in Figure 7. In terms of the total time taken to batch start multiple instances, docker, vKernel, and gVsior show similar results, because the startup process is executed in parallel on multiple cores. Additionally, vKernel exhibits memory consumption consistent with docker due to its loading of only a small amount of code necessary for container execution, whereas gVisor and Kata both require independent kernels to be started.



(a) Total startup time (b) Memory footprint

Fig. 7: Scalability when running multiple instances

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383988

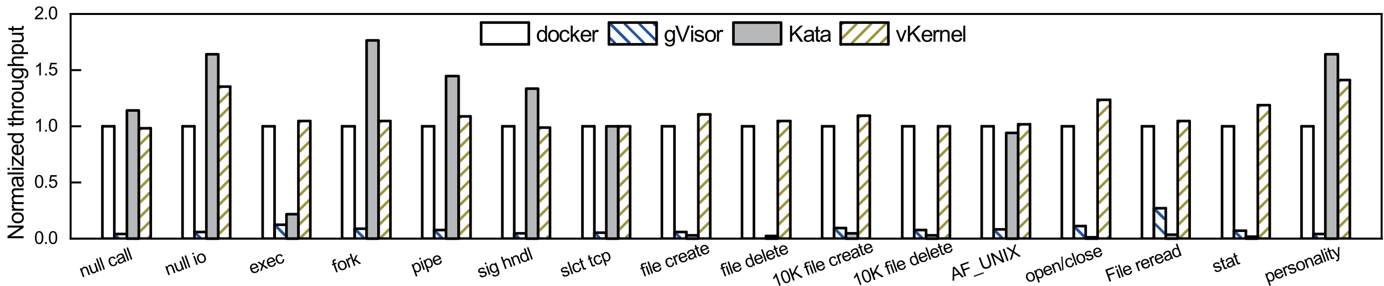


Fig. 8: The performance of system calls

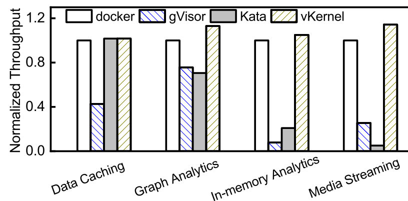


Fig. 9: The performance of cloud benchmarks

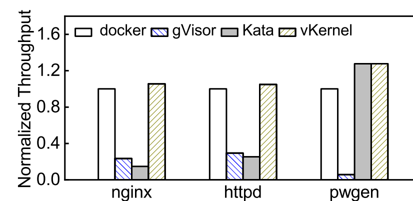


Fig. 10: The performance of real-world applications

5.2 System call

The security mechanisms for kernel isolation mainly affect the performance of system calls and those also with file accesses. We use LMbench [28] to test the performance of various system calls. We also test the bitmap mechanism proposed by Draco [29] which accelerates system call checking by using bitmap to cache always-allowed system calls. We begin with system call without much computation, I/O accesses, or parameters, such as `getpid` and `getgid`, and denote them as *null call*. To evaluate system calls that require file access checking, we configure LMBench to issue various `stat` system calls to sensitive and non-sensitive files without data operation. This is to separate system call invocation time from data transfer time. It is labeled as *stat*. We also includ system call `personality`, which requires parameter check, and commonly-used system calls `exec`, `fork`, `pipe`, and et.al. As shown in Figure 8, *vKernel* achieve an average 10.5% performance improvement compared with *docker*, especially 41% for `personality` by avoiding the sequential parameter check in *seccomp*, and 35% for `null io` by eliminating unnecessary permission check on non-sensitive files. Our results also show that the overhead due to *docker* would have been much higher if LMBench is configured to only scan sensitive files. As expected, *gVisor* has the worst performance with as much as 40x slowdown because each system call invocation requires multiple context switches. The kernel of the *Kata* is specifically designed to be lightweight, and the version of the kernel is high. These ensures that system calls with minimal resource usage are fast for *Kata*, such as `null call` and`numll io`. However, Kata suffers high overhead of file operations because of a longer IO stack, and performs worse with `exec`, which requires frequent page table creations, an operation known to incur high overhead due to memory virtualization.

5.3 Cloud benchmarks

Then, we test the performance of cloud services from CloudSuite with different containers. As shown in Figure 9, *vkernel* outperformed *docker* on cloud services by about 8.5% on average. The reason is that cloud services neither enter the shared kernel and nor trigger security checks as frequently as micro-benchmarks. Cloud services in *gVisor* show at least 30% performance degradation (Graph Analytics), again proving that user-level kernel isolation does not apply to real-world service deployment. *Kata* leads to 20x slowdown of media streaming due to IO virtualization. Virtio can help alleviate the io virtualization overhead of *kata*, but the overhead cannot be completely removed. The results show that vkernel can serve all cloud services well without any performance impact.

5.4 Real-world applications

Next, we evaluate vKernel's performance with three real-world applications. *Nginx* and *Httpd* are popular web servers with frequent network-related system call invocations and file retrievals. We use the workload generator *ab* to emulate 20 concurrent users making a total number of 3000k requests. As shown in Figure 10, *gVisor* causes a dramatic 77% and 71% throughput loss in Nginx and Httpd, respectively; *Kata* performed even worse. In contrast, *vKernel* does not affect throughput. *Pwgen* is a widely-used password generator. It is mostly computationally intensive with little I/O activity but rich of `malloc`-like memory allocations. Except for gVisor, all other approaches achieve acceptable performance in *Pwgen*. The major source of overhead in gVisor is the tracer that frequently intercepts memory system calls and causes context switches. Note that function interception in vKernel is entirely in kernel mode and does not cause noticeable overhead.

5.5 User-defined isolation and customization

Dentry cache. We use two representative file operations `stat` and `ls` to evaluate the performance of the isolated dentry cache. The baseline is native docker with a globally shared dentry cache. The `stat` test instructs each container to repeatedly display file information at different directory depths i.e., 0 (current directory), 20, and 30, for 800 thousand times. For the `ls` test, we recursively listed directory information until reaching a directory depth of 20 and repeated

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383988

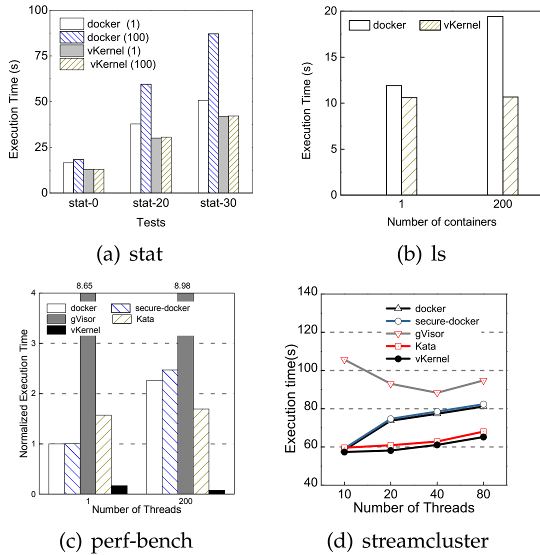


Fig. 11: Effectiveness of vKernel for data isolation. vKernel is the base for normalization.

the operation. Figure 11 (a) and (b) show the performance due to vKernel against the baseline docker. The results suggest that the contention on the dentry cache significantly degrades performance when 100 containers simultaneously request `d_entry` from the `dentry_hashtable`. In contrast, vKernel delivered consistent performance regardless of activities in colocated containers, indicating good isolation on the `dcache`.

Futex. We evaluate how well vKernel enforces isolation on the shared in-kernel `futex_queues` and preserves data locality. We first use `perf-bench` to stress test the `futex` subsystem with two containers. The container under test runs `perf-futex` with a single thread while a malicious container launches a large number of threads to continuously occupy the buckets in `futex_queues`. We controlled the malicious container to place on average 1 or 200 threads in each bucket to cause different levels of contention. Figure 11 shows the performance of the container under test with different kernel isolation approaches. *Docker* experiences significant slowdowns when contention is high, suggesting no isolation on the shared `futex_queues`. In comparison, *vKernel* offers effective isolation with container-local `futex_queues` and reduces the time of the wake up from 10000 `ns` to 100 `ns`.

In addition, we use benchmark *streamcluster* in the PARSEC [30] suite to evaluate thread locality in `futex`. *Streamcluster* is a barrier-intensive workload that has a large number of threads waking up simultaneously when exiting a barrier. Without isolation, the placement of these threads back on CPU run queues is nondeterministic, likely causing loss of locality. Figure 11(d) shows that vKernel with locality optimization (vKernel futex in the figure) effectively preserved thread locality via vKI's mechanism for futex wait and wakeup, in which data locality does not deteriorate with the thread count. In contrast, *docker* with no isolation and *vKernel* without locality optimization suffered much worse performance.

FIFO Scheduling. We compare workload performance under CFS and FIFO to confirm that vKernel truthfully emulates FIFO scheduling in userspace with the help of vKI. The hackbench [31] benchmark spawns a large number of processes, each aggressively communicating with others via pipes and performing little computation. Hackbench performance is largely determined by the number of context switches and benefits from FIFO scheduling. Figure 12 shows the performance of hackbench under different scheduling policies and different types of containers. Note that the docker has no root privilege and is unable to use the FIFO scheduler. In the figure, the "docker-fifo" component is running within a privileged Docker container. This privileged container can have an impact on the scheduling policies of other containers running alongside it. On the other hand, the "vkernel" supports the use of FIFO scheduling policies within the container. The results demonstrate that vKernel's performance using the emulated FIFO scheduling faithfully reflected FIFO performance.

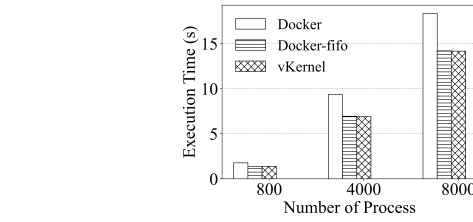
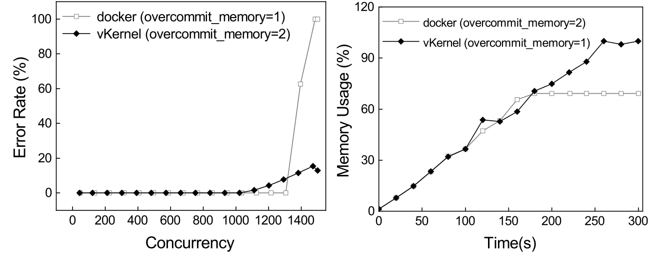


Fig. 12: CFS vs. FIFO scheduling in hackbench



(a) Postgresql errors when Redis is optimal (setting to 1) (b) Redis memory when Postgresql is optimal (setting to 2)

Fig. 13: Performance of Postgresql and Redis under different `overcomit_memory` settings. Docker shares kernel parameters, while vKernel enables customization.

Container-specific memory overcommitment. We demonstrate that vKernel containers can configure different values for globally shared kernel parameters, which lead to superior performance for different types of workloads. We selected Postgresql and Redis for evaluation as they have distinct preferences for memory overcommitment in the kernel. Postgresql is a highly concurrent database that supports a large number of worker threads. It prefers not to aggressively request memory beyond the physical memory size. As Linux employs on-demand memory allocation, aggres-

TABLE 4: The kernel vulnerabilities of container escalation

CVE-ID	should be disabled (seccomp / capability / apparmor)	docker	gVisor	Kata	vKernel
2023-0045	⊘prctl		✓	✓	✓
2022-0185	⊘unshare		✓	✓	✓
2020-8835	⊘bpf	✓	✓	✓	✓
2019-13272	⊘sys_admin+⊘ptrace	✓	✓	✓	✓
2018-18955	⊘setgid		✓	✓	✓
2017-7308	⊘net_raw		✓	✓	✓
2017-5123	⊘waitid		✓	✓	✓
2016-1583	⊘/proc/environ		✓	✓	✓

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383988

sive memory requests that overcommit memory will lead to memory thrashing when concurrency increases, though the requests could be successfully due to the overcommitment configuration. This translates to high error rate in Postgresql. In contrast, a background process (`bgsave`) in Redis occupies a large amount of virtual memory in case the database needs to be dumped on disk. `Bgsave` does not actually always consume the requested memory but needs to provision for the peak demand in case the entire database has to be dumped. Redis prefers to enable memory overcommitment otherwise the foreground database engine may not use the memory occupied by `bgsave`.

We create two containers each with a 2 GB memory limit set by cgroups. The first container runs docker and configures the system-wide `overcommit_memory` parameter. A value of 1 in `overcommit_memory` allows memory overcommitment while a value of 2 disables it. In order to achieve the optimal performance of Redis, under Docker and vKernel, `overcommit_memory` of Redis is set to 1. Redis performs equally well under Docker and vKernel (not shown). In this case, as shown in Figure12 13(a), for Docker with shared kernel, `overcommit_memory` (fixed to 1) cannot be adjusted for Postgresql, resulting in a sudden increase in error rate as concurrency increases. For vKernel, which can customize parameters, `overcommit_memory` corresponding to Postgresql can be set to 2, and the error rate remains stable. To optimize Postgresql performance, `overcommit_memory` for Postsql is set to 2 in Docker and vKernel. In this case, as shown in Figure 13(b), for shared kernel Docker, `overcommit_memory` cannot be adjusted, while vKernel can adjust the parameter to 1 for Redis.

5.6 Security isolation

We demonstrate that vKernel is more efficient than the existing isolation mechanisms using user-level kernels and VMs. Next, we evaluate whether vKernel achieves a similar level of security for the data and code that are required during the execution of a container. Accordingly, we only test whether the data and code required for the container's execution can be vulnerable to attacks. We used the POCs (proof of concept) selected from [22] to test eight known container-related kernel vulnerabilities. The vulnerabilities mostly manifest as privilege escalation. Table 4 shows details of the vulnerabilities, whether a particular kernel isolation approach is vulnerable, and the potential fix if one exists. Note that all vulnerabilities and fixes have been tested on our testbed. Table 4 suggests that vKernel can defend against the listed known threats, at a similar level of security as gVisor and Kata.

6 Limitation and Discussion

vKernel is a virtualization framework that aims to present required kernel isolation for containers. vKernel takes effect for containers based on the cooperation between a system-wide vKM and multiple vKIs. To promise extreme performance for containers, vKernel implements vKM and vKI as LKMs, and strips the changes to the host kernel by way of inline hooks. In such a design, vKernel may introduce additional security implications, because it extends the host kernel with more code and data through LKMs. Rigorous code checks are required through vkernel-buidler when building a customized vKI, but it's still far from enough. In the future, we will implement the fault isolation for vKIs and learn from KPTI to achieve vKI address space isolation to further eliminate the security risks. Having said that, vKernel bravely introduces a new kernel isolation approach that does not give up the host kernel. Based on the design of vKM and vKI, vkernel requires no changes to host kernel, therefore supporting live upgrades and multiple versions of the Linux kernel with strong usability. It allows users to customize the vKI for containers and obtain extreme performance while promising better security guarantees than the commonly-used secure docker container. Currently, vKernel is not as secure as user-level or VM-based isolation. However, it may be a good choice for performance-sensitive services without that strong security requirement, especially those deployed based on secure docker previously.

7 Related Work

We discuss additional prior work related to vKernel.

Kernel-level isolation. Containervisor [32] focuses on isolating containers' memory, while Slim [33] implements an isolated network stack for containers. ContainerLeaks [34] identifies security issues caused by kernel data leakage in a container environment. Huang et al. [35] introduce *sys_namespace* to provide dynamic private memory views for container applications. Song et al. [36] isolate file system data structures to reduce IO competition among containers. In contrast, vKernel offers a low-overhead isolation framework that does not specifically enhance the isolation of individual resources. By implementing these aforementioned isolation methods within a vKernel instance, we can achieve customization for a container without requiring extensive code modifications in the kernel.

Kernel specialization. An emerging trend of kernel isolation is to reduce kernel based on kernel specialization. Confine [37], SPEAKER [38], and sysfilter [26] as well as temporal specialization [39] customize and minimize the kernel interface for containers. SHARD [40] implements a practical framework to enforce fine-grain kernel specialization and kernel reduction. Shadow-kernels [41] provides a primitive for an individual application to access the dedicated kernel text sections at the kernel. Similarly, other approaches [42], [43] based on virtualization can achieve kernel reduction by building upon a minimized kernel view. The kernel specialization offered by vKernel instance supports both kernel reduction and customization for users.

Hardware-based isolation. Recent work [29], [44], [45], [46], [47], [48], [49] explores new hardware to implement additional isolation for containers. SCONE [44] and ARMlock [47] place the container inside the trusted execution domain based on the Intel SGX and ARM TrustZone. FastPass [48] and Iron [46] further isolate memory management and network stack for containers. They are efficient without noticeable overhead, but lack versatility and comprehensiveness. In contract, vKernel is a generic kernel isolation framework, which does not require specific hardwar.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383988

8 Conclusion

In this paper, we present *vKernel*, a kernel isolation framework for containers. Compared to the existing kernel security mechanisms, user-level and VM-based kernel isolation, vKernel is able to simultaneously achieve near-native performance and strong isolation. The key to vKernel isolation is an additional layer of indirection between the container and the host kernel, namely the proposed virtual kernel instance (vKI). vKI allows for efficient implementation of the existing kernel isolation mechanisms as well as user-defined functions and policies. The layer of indirection effectively prevents users from obtaining escalated privilege or escaping from a container.

References

[1] D. Merkel *et al.*, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.
[2] S. Y. Lim, B. Stelea, X. Han, and T. Pasquier, "Secure namespaced kernel audit for containers," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 518–532.
[3] W. Viktorsson, C. Klein, and J. Tordsson, "Security-performance trade-offs of kubernetes container runtimes," in *Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2020, pp. 1–4.
[4] D. Zahka, B. Kocoloski, and K. Keahey, "Reducing kernel surface areas for isolation and scalability," in *Proceedings of the 48th International Conference on Parallel Processing (ICPP)*, 2019, pp. 1–10.
[5] O. Tunde-Onadele, J. He, T. Dai, and X. Gu, "A study on container vulnerability exploit detection," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2019, pp. 121–127.
[6] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: a qualitative analysis," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE)*, 2014, pp. 421–432.
[7] P. Franz, T. Berger, I. Fayaz, S. Nadi, and E. Groshev, "Configfix: Interactive configuration conflict resolution for the linux kernel," in *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 91–100.
[8] N. G. Bachiega, P. S. Souza, S. M. Bruschi, and S. D. R. De Souza, "Container-based performance evaluation: A survey and challenges," in *IEEE International Conference on Cloud Engineering (IC2E)*, 2018, pp. 398–403.
[9] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 2017, pp. 405–410.
[10] *Kata container*, https://katacontainers.io/, 2022.
[11] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020, pp. 419–434.
[12] *gVisor*, https://gvisor.dev/, 2022.
[13] A. Lingayat, R. R. Badre, and A. K. Gupta, "Integration of linux containers in openstack: An introspection," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 12, no. 3, pp. 1094–1105, 2018.
[14] S. E. Hallyn and A. G. Morgan, "Linux capabilities: Making them work," 2008.
[15] J. Edge, "A seccomp overview," *Linux Weekly News*, 2015.
[16] A. Gruenbacher and S. Arnold, "Apparmor technical documentation," 2007.
[17] A. Raza, P. Sohal, J. Cadden, J. Appavoo, U. Drepper, R. Jones, O. Krieger, R. Mancuso, and L. Woodman, "Unikernels: The next stage of linux's dominance," in *Proceedings of the Workshop on Hot Topics in Operating Systems (Hotos)*, 2019, pp. 7–13.
[18] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 461–472, 2013.
[19] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, Ş. Teodorescu, C. Răducanu *et al.*, "Unikraft: fast, specialized unikernels the easy way," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 376–394.
[20] *alibaba clusterdata.*, https://github.com/alibaba/clusterdata/, 2022.
[21] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon, "X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 121–135.
[22] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A measurement study on linux container security: Attacks and countermeasures," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC))*, 2018, pp. 418–429.
[23] W. Wu, Y. Chen, X. Xing, and W. Zou, "{KEPLER}: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities," in *28th USENIX Security Symposium (Security)*, 2019, pp. 1187–1204.
[24] *Trinity: Linux system call fuzzer.*, https://github.com/kernelslacker/trinity, 2022.
[25] Q. Wang, R. Wang, Y. Hu, X. Shi, Z. Liu, T. Ma, H. Song, and H. Shi, "Keentune: Automated tuning tool for cloud application performance testing and optimization," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, p. 1487–1490.
[26] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "Sysfilter: Automated system call filtering for commodity software," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020, pp. 459–474.
[27] *Alpine Container*, https://hub.docker.com/alpine, 2022.
[28] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.
[29] D. Skarlatos, Q. Chen, J. Chen, T. Xu, and J. Torrellas, "Draco: Architectural and operating system support for system call security," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 42–57.
[30] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 72–81.
[31] R. Russell, "Hackbench: A new multiqueue scheduler benchmark," *Message to Linux Kernel Mailinglist: http://www. lkml. org/archive/2001/12/11/19/index. html*, 2001.
[32] T. Li, K. Gopalan, and P. Yang, "Containervisor: Customized control of container resources," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2019, pp. 190–199.
[33] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson, "Slim:{OS} kernel support for a low-overhead container overlay network," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 331–344.
[34] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "Containerleaks: Emerging security threats of information leakages in container clouds," in *2017 47th Annual IEEE International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 237–248.
[35] H. Huang, J. Rao, S. Wu, H. Jin, K. Suo, and X. Wu, "Adaptive resource views for containers," in *28th International Symposium on High-Performance Parallel and Distributed Computing*. IEEE, 2019, pp. 243–254.
[36] S. Wu, Z. Huang, P. Chen, H. Fan, S. Ibrahim, and H. Jin, "Container-aware i/o stack: Bridging the gap between container storage drivers and solid state devices," in *18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 2022, pp. 18–30.
[37] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020, pp. 443–458.
[38] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, "Speaker: Split-phase execution of application containers," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2017, pp. 230–251.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383988

[39] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *29th USENIX Security Symposium (Security)*, 2020, pp. 1749–1766.

[40] M. Abubakar, A. Ahmad, P. Fonseca, and D. Xu, "{SHARD}: Fine-grained kernel specialization with context-aware hardening," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[41] O. R. Chick, L. Carata, J. Snee, N. Balakrishnan, and R. Sohan, "Shadow kernels: A general mechanism for kernel specialization in existing operating systems," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 1, pp. 3–8, 2016.

[42] R. Ta-Min, L. Litty, and D. Lie, "Splitting interfaces: Making trust between applications and operating systems configurable," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 279–292.

[43] Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu, "Face-change: Application-driven dynamic kernel view switching in a virtual machine," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 491–502.

[44] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 689–703.

[45] D. Skarlatos, U. Darbaz, B. Gopireddy, N. S. Kim, and J. Torrellas, "Babelfish: Fusing address translations for containers," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 501–514.

[46] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella, "Iron: Isolating network-based {CPU} in container environments," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 313–328.

[47] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "Armlock: Hardware-based fault isolation for arm," in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, 2014, pp. 558–569.

[48] W. Zhang, A. Sharma, K. Joshi, and T. Wood, "Hardware-assisted isolation in a multi-tenant function-based dataplane," in *Proceedings of the Symposium on SDN Research*, 2018, pp. 1–7.

[49] Z. Hua, Y. Yu, J. Gu, Y. Xia, H. Chen, and B. Zang, "TZ-Container: protecting container from untrusted OS with ARM TrustZone," *Science China Information Sciences*, vol. 64, no. 9, pp. 1–16, 2021.

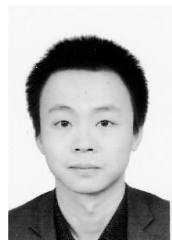
Hang Huang received the BS degree from the Huazhong University of Science and Technology (HUST), in 2016. He is currently working toward the PhD degree with Service Computing Technology and System Lab (SCTS), Cluster and Grid Computing Lab (CGCL), and Big Data Technology and System Lab (BDTS), Huazhong University of Science and Technology (HUST), China. His current research interests include container technology and kernel scheduling.

Honglei Wang received his B.S. degree from the Huazhong University of Science and Technology in 2020 and is currently pursuing his M.S. degree in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Computing Lab (CGCL), HUST. His research interests include operating systems and virtualization.

Jia Rao received the BS and MS degrees in computer science from Wuhan University, in 2004 and 2006, respectively, and the PhD degree from Wayne State University, in 2011. He is currently an associate professor of computer science with the University of Texas, Arlington. His research interests include the areas of distributed systems, resource auto-configuration, machine learning, and CPU scheduling on emerging multi-core systems. He is a member of the IEEE.

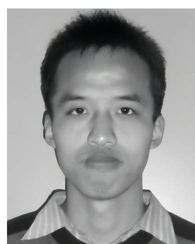
Song Wu received the PhD degree from Huazhong University of Science and Technology (HUST) in 2003. He is a professor of computer science at HUST in China. He currently serves as the vice dean of the School of Computer Science and Technology and the vice head of Service Computing Technology and System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

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



Chen Yu received the PhD degree in information science from the Tohoku University in 2005. From 2005 to 2006, he was a Japan Science and Technology Agency postdoctoral researcher with the Japan Advanced Institute of Science and Technology. He is with the School of Computer Science and Technology, Huazhong University of Science and Technology (HUST), where he is currently a professor working in the areas of cloud computing, ubiquitous computing, and green communications. He is a member of IEEE.

Hai Jin received the PhD degree in computer engineering from Huazhong University of Science and Technology (HUST) in 1994. He is a chair professor of computer science and engineering at HUST in China. He was awarded the Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China. His research interests include computer architecture, virtualization technology, cloud computing, peer-to-peer computing, and network storage. He is a fellow of the IEEE and a member of the ACM.



Kun Suo received the BS degree in software engineering from the Nanjing University, China, in 2012, and PhD degree from the University of Texas, Arlington, in 2019. He is currently an assistant professor with the Department of Computer Science, Kennesaw State University. His research interests include the areas of cloud computing, virtualization, operating systems, Java virtual machines, and software defined network. He is a member of IEEE.

Lisong Pan is working toward the undergraduation degree with the Huazhong University of Science and Technology. He is currently doing an internship with Service Computing Technology and System Lab (SCTS), Cluster and Grid Computing Lab (CGCL), and Big Data Technology and System Lab (BDTS). His research interests mainly include virtualization and cloud computing.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/