USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

# LITESHIELD: Secure Containers via Lightweight, Composable Userspace µKernel Services

Kaesi Manakkal, *The University of Texas at Arlington;* Nathan Daughety and Marcus Pendleton, *Air Force Research Laboratory (AFRL);* Hui Lu, *The University of Texas at Arlington*

https://www.usenix.org/conference/atc25/presentation/manakkal

This paper is included in the Proceedings of the 2025 USENIX Annual Technical Conference.

July 7–9, 2025 • Boston, MA, USA

ISBN 978-1-939133-48-9

Open access to the Proceedings of the 2025 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

# LITESHIELD: Secure Containers via Lightweight, Composable Userspace $\mu$Kernel Services

Kaesi Manakkal, Nathan Daughety[†], Marcus Pendleton[†], Hui Lu
*The University of Texas at Arlington, [†]Air Force Research Laboratory (AFRL)*

## Abstract

This paper introduces LITESHIELD, a new userspace isolation architecture for secure containers that reexamines the boundary between user applications and system services. LITESHIELD decouples traditional guest kernel functionality into modular userspace microkernel ($\mu$kernel) services that interact with guest applications via low-latency, shared-memory-based inter-process communication (IPC). By serving most Linux syscalls in userspace, LITESHIELD enforces a significantly reduced user-to-host interface, with just 22 syscalls, achieving strong isolation comparable to virtual machines (VMs) while avoiding the complexity of hypervisors and hardware virtualization. LITESHIELD further provides a POSIX-compatible runtime with fine-grained syscall interception to support legacy applications and enables composable $\mu$kernel services that can integrate specialized userspace components (e.g., networking and filesystems). Our implementation demonstrates that LITESHIELD delivers strong isolation with performance comparable to traditional containers.

## 1 Introduction

Due to high portability, high density, and low operational cost, containers have been widely used for packaging, isolating, and multiplexing cloud applications. In contrast to virtual machines (VMs) (i.e., hypervisor-based virtualization), containers execute applications directly on the native host OS [8, 53, 61] and leverage kernel-level features, such as namespaces [23], cgroups [16], and seccomp [20], to enforce *isolation* between containerized applications. While the lack of guest OSes and virtual hardware abstraction makes containers *lightweight*, they cannot be directly adopted as the isolation mechanism in multi-tenancy clouds due to *weak isolation* – sharing the same host results in a large attack surface, e.g., 300+ system calls, or syscalls, in Linux. That is precisely why, in today's production systems, containers are deployed within VMs for strong isolation [13, 37].

To address the tension between isolation and performance, recent efforts [28, 56, 62] adopt the technique of *minimization*, including tailoring a VM's kernel with minimal components [28], linking a hosted application into a tiny unikernel image under a single address space [56], and attaching a userspace guest kernel (with a substantial portion of the Linux surface) to a container for VM-like isolation [62]. While these efforts have blurred the isolation boundaries of VMs and containers, they share some common limitations. *First*, maintaining a full guest kernel, even a minimized one, for each hosted entity remains inefficient. *Second*, different applications require access to specific functionalities of the guest kernel, rendering a "one-size-fits-all" guest kernel impractical, if not impossible. *Third*, since some guest kernels have been minimized or even degraded to provide system functions at the same level as userspace applications [56], these approaches rely solely on hypervisors for isolation. However, hypervisors have their share of vulnerabilities [27].

In this paper, we reexamine the isolation boundaries between user/kernel space for applications and kernel/system services and explore a new isolation architecture for secure containers, called LITESHIELD, to achieve *lightweight yet strong isolation*. Inspired by the microservices architecture, LITESHIELD decouples the closely-coupled guest kernel and its hosted applications, or *guest applications*, as loosely-coupled entities. Such decoupling allows the guest kernel to operate as a collection of userspace microkernel ($\mu$kernel) services, each running as regular userspace processes. Communication between $\mu$kernel services and guest applications is facilitated through efficient userspace inter-process communication (IPC), eliminating costly syscalls.

The isolation architecture in LITESHIELD achieves *strong isolation with minimal overhead*. First, LITESHIELD achieves strong isolation by blocking direct host kernel access for guest applications. Instead, they are served by userspace $\mu$kernel services: By serving system services (e.g., networking and filesystems) in the userspace, the *user-to-kernel* interface – i.e., between $\mu$kernel services and the host – is significantly

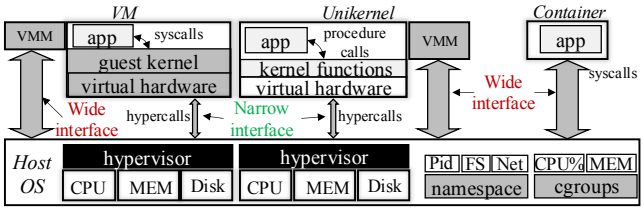



Figure 1: Comparisons of three representative isolation architectures: VMs, unikernels, and containers.

reduced (e.g., from 300+ syscalls to 20+) and comparable to VMs (e.g., 20+ hypercalls and 60+ VMExits). Even if a malicious guest application exploits a $\mu$kernel service (via userspace IPC), its access is limited to a restricted userspace process (i.e., defense in depth like VMs). As communication between guest applications and $\mu$kernel services is facilitated through userspace IPC, LITESHIELD removes the hypervisor and further reduces the attack surface. Further, by hosting system services in userspace, LITESHIELD replaces expensive cross-boundary invocation overheads (e.g., syscalls or VMExits/VMEntries) with fast (cache-to-cache) IPC. Last, LITESHIELD supports composable $\mu$kernel services, enabling seamless integration of specialized userspace approaches [41,44,47,48] to deliver highly efficient userspace system services, instead of relying solely on general-purpose, monolithic, and hard-to-optimize in-kernel services.

To ensure compatibility with existing commodity kernels (e.g., Linux) and legacy applications, LITESHIELD addresses several challenges: First, the Linux kernel's implementation restricts certain syscalls (e.g., process and memory management) from being executed in a separate process, complicating the operation of $\mu$kernel services as independent userspace processes. To address this, LITESHIELD categorizes syscalls into *delegable syscalls*, which are redirected to LITESHIELD's $\mu$kernel services for processing, and *non-delegable syscalls*, which are trapped, monitored, and validated before being executed within the same process via an arbitration mechanism. To further support legacy applications, LITESHIELD provides a POSIX-compatible library that supports runtime injection and fine-grained syscall interception, enabling seamless redirection of *delegable syscalls* from legacy applications to userspace $\mu$kernel services, without any binary modifications. Last, to achieve high performance, LITESHIELD employs a userspace IPC mechanism with a shared memory region and polling-based threads to facilitate low-latency communication between guest applications and $\mu$kernel services.

Our current implementation of LITESHIELD supports most of the Linux kernel syscalls in userspace required by regular guest applications (i.e., those not running with root privileges) and achieves a thin *user-to-host* interface with only 22 syscalls that need the support of the host kernel (compared to 20+ hypercalls and 64 VMExits for KVM-based VMs), while significantly reducing both the software codebase (eliminating the need for a hypervisor and QEMU-based emulator) and hardware complexity (requiring no hardware virtualization). We have ported an existing userspace network stack, `f-stack` [9], and implemented an `ext2`-like userspace filesystem as userspace networking and filesystem $\mu$kernel services. Porting `f-stack` to LITESHIELD only required 400+ lines of code. By leveraging lightweight userspace $\mu$kernel services and fast shared memory-based IPC between guest applications and $\mu$kernel services, LITESHIELD delivers performance comparable to traditional containers.

## 2 Motivation

### 2.1 Cloud Native and Isolation

IT companies are under constant pressure to simplify their product development with shortened production cycles to adapt to changing markets and diverse demands. Cloud-native technologies are poised to tackle this pressing challenge. First, developers decompose a traditional monolithic application into graphs of *single-purpose, loosely-coupled* microservices. As each microservice focuses on a small subset of the monolithic application's functionality, this microservices-based architecture reduces development complexity and increases code velocity [1, 3, 4, 6, 12]. Further, cloud-native platforms deploy, manage, and scale microservices completely for cloud tenants, further liberating them from the management of virtual servers (i.e., server*less* for tenants). The use of cloud-native technologies is becoming pervasive: Companies like Amazon, Netflix, Twitter, Uber, and eBay have adopted the microservices architecture [5, 7, 10, 17, 26]. In addition, a proliferation of serverless platforms enables a simple way, i.e., via Function-as-a-Service (FaaS), to build and execute cloud-native applications [2, 24, 25, 35, 36].

**Isolation** ensures safe resource sharing by preventing cloud tenants from accessing each other's shared resources. Without enforcing isolation, a malicious user could steal sensitive information from victims [54, 60, 63, 64], or an aggressive user might degrade the performance of others [49].

State-of-the-art virtualization techniques provide isolation via full-blown VMs [14, 58], micro-VMs [28, 62], containers [8, 53, 61], microkernels [30, 43, 52], and unikernels [31,40,42,57]. As illustrated in Figure 1, VM-based virtualization achieves isolation through a *virtual hardware interface*, allowing each VM to operate with a fully functional guest operating system (OS). Since the virtual hardware interface differs from real hardware, it is further supervised by another software layer, the *hypervisor*. The virtualization architecture provides a *strong* security boundary between sandboxed applications in VMs due to: 1) a minimal attack surface between VMs and the native host (e.g., tens of hypercalls or VMExits); and 2) defense-in-depth, where both the guest OS and hypervisor contribute to security. However, this defense-in-depth approach introduces non-trivial performance overhead by layering guest kernels and hypervisors, resulting in costly inter-

actions across multiple layers of the virtualization stack for CPU, memory, and I/O virtualization. Microkernels improve isolation by minimizing kernel functionality and shifting OS services, such as file systems and drivers, to userspace, reducing the attack surface and improving fault isolation. Notably, seL4 [43] offers formal verification guarantees, while systems like Barrelfish [30] and Arrakis [52] extend this model to multicore and I/O-optimized architectures. Though well-suited for specialized, high-assurance systems, microkernels often face challenges with performance due to IPC overhead and supporting legacy applications. These factors limit usability in general-purpose cloud-native environments.

In contrast, containers execute applications directly on the native host OS [8, 53, 61]. Kernel-level features, e.g., namespaces [23], cgroups [16], and seccomp [20], enforce isolation between containerized applications. While the lack of virtual hardware abstraction makes containers lightweight, they cannot be directly adopted as the isolation mechanism in multi-tenant clouds due to *weak isolation* – i.e., sharing the same host results in a large attack surface (e.g., 400+ system calls, or syscalls, in Linux). While modern OSes have a "whitelisting" mechanism (e.g., `seccomp`) allowing containers to transition into a restricted mode with a narrowed syscall interface, the common problem is that it is difficult to determine what syscalls an application may use. Default whitelisting policies still tend to be large, e.g., 250+ syscalls [21].

Recent efforts [28, 50, 62] aim to balance isolation and performance through *minimization*. For example, AWS's Firecracker [28] optimizes a VM's kernel by including only essential components and adopting a simplified I/O model. NEC's LightVM [50] integrates a hosted application into a minimal unikernel image within a single address space for greater efficiency. Google's gVisor [62] employs a userspace guest kernel, incorporating a significant portion of the Linux interface, to provide VM-like isolation for containers.

## 2.2 Limitations

Unfortunately, cloud native presents new and pressing challenges to existing virtualization techniques. *1) High performance overhead:* Unlike monolithic applications, a cloud-native application consists of numerous distributed microservices, each requiring sandboxing, which significantly amplifies the performance overhead and memory footprint. Moreover, virtualization overhead becomes more pronounced compared to monolithic applications, as each hosted microservice is typically smaller and more ephemeral (e.g., high startup overhead). *2) Generality vs. specialization:* The increasing diversity of microservices necessitates specialized system services, such as customized network stacks [51] or specialized file systems [47]. However, existing virtualization techniques for isolation continue to rely on general-purpose, one-size-fits-all guest kernels. While these kernels are often highly optimized [28], they struggle to meet the varied and specific requirements of diverse microservices. Furthermore, despite recent advancements in high-performance networking and file systems [41, 47, 51], there is no practical way to seamlessly integrate these technologies with general applications or microservices. *3) Large attack surface.* Although the user-to-host interface for VM-based virtualization is thin, the code responsible for virtualization and isolation remains large. This includes both *the hypervisor*, which resides in the host kernel (i.e., type-2 hypervisor) and interacts with hardware mainly for CPU and memory management, and *the QEMU-based Virtual Machine Monitor* (VMM), which operates in userspace primarily for device emulation. According to the Common Vulnerabilities and Exposures (CVE) database, 184 vulnerabilities have been reported in major hypervisors (e.g., Xen, KVM, Hyper-V) since 2007, with 33% of these occurring in the past 1.5 years. Additionally, the VMM is huge (e.g., QEMU-based VMM has more than 1.4 million lines of code) and can access the host via the whole syscall interface.

## 2.3 Threat Model and Assumptions

In this paper, we focus on security vulnerabilities and isolation mechanisms in general-purpose monolithic kernels (e.g., Linux). We share the common isolation assumptions as VMs/unikernels [56, 62]: We trust fundamental hardware-based protections – such as page tables and CPU execution modes – that ensure strong isolation between different processes and between user and kernel execution within the same process. We focus on software deficiencies in host kernels, guest kernels, and hypervisors that can be exploited through their exposed *user-to-host* interfaces, namely, system calls and hypercalls, which constitute the main attack surface. Therefore, our threat model is that one malicious user could break out of the isolation by compromising the *user-to-host* interface. Two metrics are used to evaluate the attack surface: 1) the size of the user-to-host interface and 2) the amount of code accessible through the interface. Other attacks, such as covert channels and side channels, pose security risks by enabling unauthorized communication and information leakage through unconventional pathways. Existing software-based isolation approaches, including VMs, microVMs, unikernels, and others, are susceptible to these attacks. Although eliminating all side and covert channels is challenging due to the complexity of hardware and software interactions, existing measures, such as secure hardware designs, strict resource isolation, and the introduction of system randomness, can be orthogonally applied to software-based isolation approaches.

# 3 Design of LITESHIELD

We present LITESHIELD, a novel sandboxing architecture providing *strong yet lightweight* isolation for secure containers. Drawing inspiration from the microservice architecture, LITESHIELD offers *on-demand, composable* guest kernel/system services to cloud-native applications, functioning

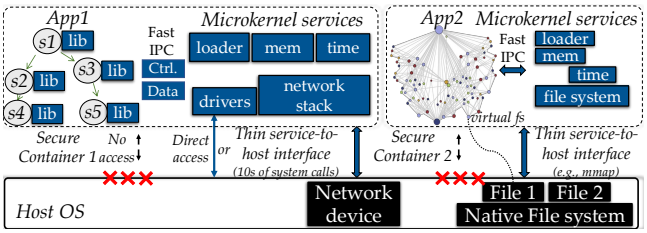



Figure 2: Architecture of LITESHIELD.

as userspace microkernel ($\mu$kernel) services. (Figure 2).

LITESHIELD's core innovation lies in decoupling guest applications from their hosting guest kernels, running them as separate userspace entities. Each application process acts as a client, while all required $\mu$kernel services are combined to provide system services. Clients interact with userspace $\mu$kernel services through fast inter-process communication (IPC) channels (§3.2). System calls made by a guest application are intercepted and redirected to the appropriate userspace $\mu$kernel service, which handles the request mostly within the userspace while only contacting the host kernel for (a limited number of) privileged operations, resulting in a thin *user-to-host* interface (comparable to VMs). To prevent malicious applications from bypassing the interception library and making direct system calls to the host kernel, LITESHIELD employs `seccomp` [20], effectively blocking unauthorized "direct" syscalls.

LITESHIELD inherits advantages from the microservices architecture, such as modular development, flexible deployment, and rapid iteration. Userspace $\mu$kernel services can be gradually and individually developed, extended, replaced, customized, and integrated with other existing/ongoing userspace systems [41, 47, 51]. In addition, while hardware specialization is accelerating in cloud [18], LITESHIELD can quickly support specialized hardware (e.g., GPU, smart NICs, and persistent memory) with userspace support, making new, advanced hardware accessible by specific cloud-native applications (e.g., machine learning tasks).

## 3.1 Strong Isolation via Thin Interface

As shown in Figure 2, LITESHIELD achieves a level of isolation comparable to VMs and unikernels, maintaining a similarly *thin user-to-host interface* (i.e., requiring tens of syscalls to the host, compared to 300+ for containers).

First, guest applications have no direct access to the host kernel. LITESHIELD enforces this restriction by using `seccomp` to block all direct syscalls from guest applications, i.e., by applying a `seccomp` profile that denies all syscalls by default. Second, userspace $\mu$kernel services are permitted to access the host kernel when necessary. To ensure that the *user-to-host* interface remains *thin*, `seccomp` is also applied to these services, allowing only a minimal set of explicitly defined syscalls through a restrictive profile. The rationale is simple: "the more done in userspace, the less needed in the kernel". For example, the unikernel approach [56] reduces

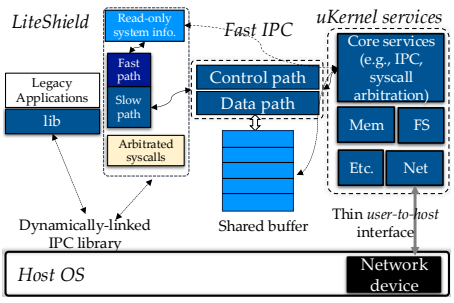



Figure 3: Fast userspace inter-process communication (IPC).

syscall or hypercall usage to fewer than ten by shifting most system functions into userspace. Similarly, LITESHIELD performs most guest kernel functions, such as file and filesystem management, networking, and IPC, entirely within userspace, thereby minimizing interactions with the host kernel and achieving a thin *user-to-host* interface.

Unlike the unikernel approach, which embeds guest kernel functions in the same address space as guest applications, LITESHIELD executes these functions within more flexible and independent $\mu$kernel services. However, certain syscalls, namely *non-delegable* ones, must be executed within the context of the same process, such as process management (e.g., `fork`, `clone`, `wait`, and `exit`) and memory management (e.g., `mmap`, `munmap`, `mprotect`, `msync`, and `madvise`). A straightforward solution would make the kernel LITESHIELD-aware by introducing kernel support to convert *non-delegable* syscalls to *delegable* ones, allowing them to execute in the context of another process. LITESHIELD, instead, focuses on compatibility with the *unmodified* Linux kernel and realizes an *arbitration* mechanism to handle the execution of the non-delegable syscalls. Specifically, LITESHIELD permits guest applications to execute non-delegable syscalls by explicitly allowing them in the `seccomp` profile. However, it leverages Linux's `ptrace` mechanism to trap and monitor these syscalls through LITESHIELD's core $\mu$kernel service (§3.3). When a guest application is launched, it is registered as a tracee of the core $\mu$kernel service before execution begins. Thus, any subsequent invocation of a non-delegable syscall is intercepted by the core service, allowing LITESHIELD to perform sanity checks or other forms of inspection [34, 38] before permitting the syscall to proceed. This mechanism enables fine-grained monitoring of non-delegable syscalls while maintaining compatibility with the existing Linux kernel.

Since guest applications can only access the host kernel via LITESHIELD, even if a malicious guest application exploits a bug in a $\mu$kernel service of LITESHIELD (e.g., through IPC channels), it can only gain access to a restricted userspace process (i.e., defense in depth). As the communication between guest applications and $\mu$kernel services is through userspace IPCs or arbitrated syscalls, LITESHIELD eliminates the hypervisor, further minimizing the attack surface for isolation.

## 3.2 Lightweight Isolation via Fast IPCs

One key goal of LITESHIELD is to provide strong isolation with much *less* overhead than VM/unikernel-based isolation to *legacy* applications, better, with performance comparable to that of "lightweight" containers.

Containerized applications request system services via *syscalls* from the kernel space. This mechanism is actually costly, as it involves same-core user/kernel mode transitions, cache pollution, and data movement across the kernel/user space, consuming *microseconds* [55]. In addition, the "one-size-fits-all" monolithic kernel services could be suboptimal for many new application scenarios [45, 46]. The generality of monolithic kernels makes optimization difficult, and vertical integration efforts are often broken by upstream changes [51]. On the other hand, recent userspace and hybrid approaches [32, 39, 41, 44, 47, 48, 59] have been proposed to mitigate such overhead by redistributing functions between user and kernel space, demonstrating superior performance compared to in-kernel services. Following this direction, LITESHIELD runs most kernel services in userspace, achieving a thin user-to-host interface (and strong isolation) while also enabling the potential for high performance through various specialized userspace approaches.

Guest applications request system services from *userspace* $\mu$kernel services through IPC channels, instead of syscalls. As shown in Figure 3, LITESHIELD develops a *high-performance shared-memory-based IPC mechanism* for fast system service delivery. *First*, each application is assigned a shared memory buffer. When an application makes a request, it places the syscall number and arguments in the shared buffer and toggles a flag to let the $\mu$kernel services know that a request has been made. Once the request has been completed, the $\mu$kernel service puts the response in this buffer and toggles the flag to let the guest application know it is done. This approach draws inspiration from existing userspace communication techniques, e.g., RDMA, and incorporates a *polling-based* mechanism to reduce communication latency: A core *$\mu$kernel service for IPCs* employs a dedicated polling thread to continuously monitor for incoming IPC requests from guest application processes. Upon detection of a request, this thread promptly handles it – by forwarding it to one of the composable userspace $\mu$kernel services (e.g., networking or file systems). Similarly, application processes use another polling thread to actively wait for and immediately process responses from the IPC $\mu$kernel service. *Second*, LITESHIELD leverages a multi-core system (common today) to place application processes and userspace $\mu$kernel services on separate cores – i.e., avoiding same-core context switches. This separation precludes context switching on the same core and further minimizes communication latency between guest applications and $\mu$kernel services. Since the context switch overhead for userspace processes is typically lower than that of virtual CPUs, LITESHIELD's userspace solution is expected to outperform VMs even with core multiplexing. *Finally*, the communication latency under the proposed shared-memory-based IPC mechanism mainly hinges on the memory access latency. If the cores for the application and the userspace $\mu$kernel services are situated on the same CPU, this configuration can capitalize on the last-level cache (LLC) to expedite IPC (i.e., cache-to-cache transfers that typically require only tens of CPU cycles [55]).

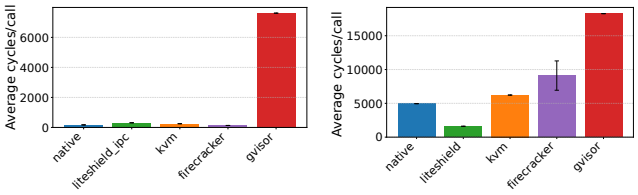
Moreover, LITESHIELD provides a POSIX-compatible library that combines `LD_PRELOAD` and the binary translation capabilities of the `libsyscall_intercept` library [15]. The `LD_PRELOAD` mechanism allows the library to be injected into the address space of legacy applications at runtime, enabling it to override standard library functions (i.e., `glibc`) and intercept system calls. Meanwhile, `libsyscall_intercept` provides fine-grained control over syscall interception by hooking directly into the syscall execution path using inline hooking and binary rewriting techniques. This combined approach enables LITESHIELD to dynamically link the library to legacy applications – without any modifications to the binaries – and intercept delegable syscalls. The intercepted syscalls are redirected to the IPC shared buffer, where LITESHIELD facilitates communication with its $\mu$kernel services.

Like FlexSC [55], LITESHIELD aims to improve syscall performance by rethinking the traditional syscall interface. However, they take different architectural approaches. FlexSC introduces exception-less system calls that batch syscall execution and decouple it from the application thread, reducing kernel traps and improving throughput on multicore systems. In contrast, LITESHIELD decomposes traditional kernel functionality into userspace $\mu$kernel services and uses fast userspace IPC and selective syscall trapping (via ptrace) to intercept and redirect syscalls, maintaining compatibility with unmodified Linux and legacy applications.

## 3.3 Userspace $\mu$Kernel Services

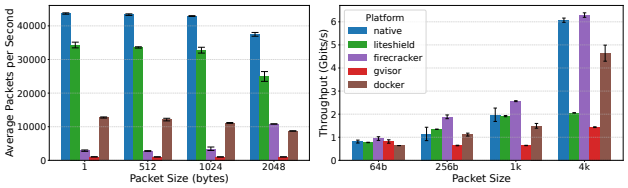
As illustrated in Figure 3, LITESHIELD's $\mu$kernel services are divided into two categories: core services and composable services. Core services, such as IPC, syscall arbitration, time management, and memory management, provide essential $\mu$kernel functionality required by every guest application. In contrast, composable services, including file, device, and networking management, are provided on demand. Integrating existing and new userspace approaches into LITESHIELD as composable $\mu$kernel services is straightforward.

We have integrated the `DPDK`-based userspace network stack, `f-stack` [9], into LITESHIELD. First, we extended LITESHIELD to fetch syscall requests from the IPC shared buffer and enqueue them in a separate operation queue for `f-stack` to process. Further modifications were made to `f-stack`'s main network processing loop to monitor the operation queue and handle any received network requests. Once processed, the results are placed back into the IPC shared buffer, where the guest application can retrieve them.




(a) `getpid`. (b) `read`.

Figure 4: Syscall latency comparisons.




(a) UDP. (b) TCP.

Figure 5: Network performance comparisons, on UDP and TCP protocols, with different packet sizes.

`f-stack` primarily processes packets in userspace, interacting with the underlying network device only when necessary (e.g., for sending packets) through a limited syscall interface (e.g., `read` and `write`). Each instance of `f-stack` is bound to a software `tap` device, enabling the sharing of a physical network card across multiple containers. Last, `f-stack` supports direct use of a physical NIC for scenarios requiring dedicated access. We have also implemented and integrated a userspace filesystem, compatible with `ext2` [22], into LITESHIELD.

Table 1: Ptrace overhead for non-delegable syscalls.

| Syscall | Syscall lat ($\mu$s) | Ptrace lat ($\mu$s) | Ptrace overhead |
|---|---|---|---|
| mmap | 0.220 | 25.485 | 99.1% |
| fork | 40.553 | 35.656 | 46.8% |
| clock_nanosleep | 56.896 | 23.256 | 29.0% |
| futex | 0.254 | 15.476 | 98.4% |

## 4 Evaluation

We have implemented LITESHIELD in approximately 7,000 lines of C/C++ code [1]. Currently, LITESHIELD supports the redirection or arbitration of around 170 Linux kernel syscalls, including 142 delegable and 28 non-delegable ones. Most of the remaining unsupported syscalls (approximately 132) require root privileges from guest applications. LITESHIELD's modular design allows for the incremental addition of support for these syscalls. The *user-to-host* interface in LITESHIELD is thin, requiring only 22 syscalls – compared to 60+ VMExits for KVM-based VMs and 250+ syscalls in the default `seccomp` whitelist for containers. Please refer to Appendix A for a more detailed breakdown. We have evaluated the effectiveness of LITESHIELD by comparing it with state-of-the-art isolation mechanisms, including Docker containers [8], KVM-based VMs [14], Firecracker [28], and gVisor [62].

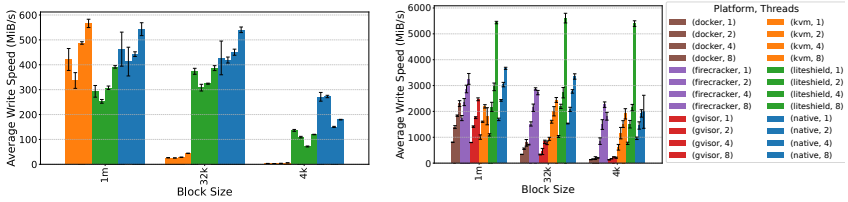
**Testbed.** We conducted our experiments on a platform with an Intel Xeon Gold 6430 CPU, 96GB DDR5 RAM, and a Micron 7450 NVMe SSD (ext4), running Ubuntu 22.04 with Linux kernel 5.15. Hyperthreading was disabled, and resources were configured to avoid bottlenecks. KVM tests used 16 vCPUs, 32GB RAM, and a virtio disk in direct sync cache mode with the same OS as the host. gVisor [62] (v1.10.1) was tested in systrap mode with Docker support, while Firecracker [28] (v1.10.1) was tested with an Ubuntu 24.04 root filesystem on Linux kernel 5.10, configured with 16 vCPUs and 32GB RAM. Docker containers and LITESHIELD were constrained to 16 cores and 32GB RAM using `cgroup` [16].

**Syscall latency.** We evaluated syscall performance using two representative syscalls: a simple syscall, `getpid`, and a complex syscall, `read`. Figure 4a shows the average latency of invoking `getpid` one million times – LITESHIELD achieves significantly lower latency than the user-level isolation mechanism gVisor due to its fast IPC mechanism, while maintaining comparable latency to other VM-based approaches for this simple syscall. Figure 4b illustrates the average time to read 1 byte from each block of a 4GB file. For this complex syscall, LITESHIELD outperforms VM-based approaches, as `read` triggers VMExits in VMs, incurring high context-switch overhead. Further, LITESHIELD surpasses native performance due to its *specialized, lightweight* userspace filesystem.

**Ptrace overhead.** We evaluated the performance impact of LITESHIELD's `ptrace`-based arbitration mechanism for non-delegable syscalls. As shown in Table ??, we selected one representative syscall from each of the four main classes of non-delegable syscalls: process management, memory management, timing, and locking. On average, `ptrace` introduces 15–35$\mu$s of overhead, with the effect being more pronounced for lightweight syscalls (e.g., `mmap` and `futex`) and less significant for heavier ones (e.g., `fork` and `clock`). We note that these syscalls are generally invoked infrequently. In future work (§5), we plan to convert these non-delegable syscalls into delegable ones to eliminate this overhead.

**Userspace networking.** We evaluated the performance of LITESHIELD's userspace network stack, ported from `f-stack`. Figure 5a illustrates UDP network performance (packets/second) with a client sending UDP packets of various sizes to a server hosted under different isolation approaches. We used a pair of connected virtual interfaces (i.e., `veth`) to connect the virtual NICs of the client and server. LITESHIELD outperforms all other isolation mechanisms, with performance slightly below native, due to the highly optimized userspace network stack provided by `f-stack`. Figure 5b shows TCP performance using `iperf` [11], with one instance as the client and another as the server hosted under those same isolation

[1] Project website: https://github.com/kmanakk1/liteshield




(a) Direct IO. (b) Cached IO.

Figure 6: Performance for writing a 4GB file with various threads and block sizes.

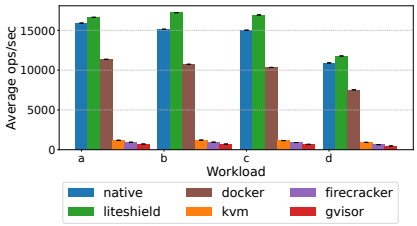



Figure 7: `YCSB` [33] performance on Redis [19] with different workloads.

approaches. While LITESHIELD delivers comparable performance for small packets, it falls behind the microVM approach (e.g., Firecracker) for larger packets because `f-stack` lacks the `GRO` feature. `GRO` reduces packet processing overhead for large packets by merging multiple small packets before processing, but it may introduce latency for small packets, highlighting the lack of a "one-size-fits-all" solution. With LITESHIELD, however, guest applications can select $\mu$kernel services tailored to their specific requirements.

**Userspace filesystem.** We evaluated the performance of LITESHIELD 's userspace filesystem, developed from scratch to emulate the functionality of an in-kernel `ext2` filesystem. Using `fio` [29] (version 3.28), we conducted two types of write I/O tests: 1) cached I/O, where data is written to the page cache and asynchronously flushed to disk, and 2) direct I/O (with `O_DIRECT`), which bypasses the page cache and reaches the disk directly. The `fio` benchmarks were run with multiple threads accessing a single 4GB file using various block sizes.

For VM-based approaches (e.g., gVisor and Firecracker), direct I/O (`O_DIRECT`) bypasses the VM's page cache but can still be buffered by the host's page cache, causing double caching and hindering direct persistence to disk. In contrast, with userspace isolation, LITESHIELD completely eliminates double caching. As shown in Figure 6a, LITESHIELD achieves higher write performance than KVM (we explicitly configured QEMU to enforce direct writes for its disk) and native setups for smaller block sizes, with slightly lower performance for very large block sizes (e.g., 1MB). For cached I/O, Figure 6b shows that LITESHIELD demonstrates better scalability than other approaches as the thread number increases. This is because LITESHIELD's userspace filesystem has a greatly simplified and efficient page cache mechanism.

**Real-world applications.** We evaluated the performance of a real-world application, Redis [19] v6.0.16, on LITESHIELD (with both networking and filesystem $\mu$kernel services) using YCSB v0.18.0 [33] by executing four distinct workloads: Workloads A (50% reads, 50% updates), B (95% reads, 5% updates), C (95% reads, 5% inserts), and D (50% reads, 50% read-modify-write). Figure 7 shows that LITESHIELD achieves higher performance compared to native execution. This improvement is primarily due to the reduced overhead of IPC versus traditional syscalls, particularly for complex operations. In contrast, this overhead in alternative isolation solutions, such as Firecracker and gVisor, is more pronounced, resulting in inferior performance.

## 5 Conclusions and Future Work

We present LITESHIELD, an effort to explore to what extent guest applications can be isolated in userspace without requiring kernel or application modifications. LITESHIELD achieves this by decoupling guest applications from their guest kernels and offering guest kernel services as a collection of userspace $\mu$kernel services. It ensures strong isolation through a thin user-to-host interface and delivers high performance with specialized userspace $\mu$kernel services.

Despite its effectiveness, LITESHIELD has several limitations that suggest directions for future work. First, the `ptrace`-based arbitration mechanism introduces overhead for non-delegable system calls. While we consider this a reasonable trade-off to achieve performance, compatibility, and isolation, future kernel support could eliminate this overhead by converting non-delegable syscalls into delegable ones. Rather than modifying the host kernel to introduce new context-aware variants of these syscalls, we are exploring a more transparent solution: leveraging a kernel module to detect whether non-delegable syscalls originate from LITESHIELD's $\mu$kernel services. When such calls are identified, they could be dynamically converted into context-aware syscalls. Furthermore, statically linked applications, including those with custom `libc` implementations or inline assembly system call instructions, may bypass LITESHIELD's interception library, resulting in failures due to `seccomp` blocking. To address this issue, we are exploring a "hotpatching" technique that disassembles system call instructions in statically linked processes and replaces them with hooks that redirect the calls to LITESHIELD's IPC mechanisms.

## 6 Acknowledgments

We thank our shepherd, Kevin Pedretti, and the anonymous reviewers for their valuable feedback. We also thank Fotis Antonatos for his early contributions to this project. This work was supported by the Air Force Research Laboratory (AFRL) under Awards FA8750-24-2-0001 and FA8750-25-C-B038, and by the National Science Foundation (NSF) under Awards CCF-2415473 and CNS-2415774.

## References


[1] 5 principles for cloud-native architecture—what it is and how to master it. https://cloud.google.com/blog/products/application-development/5-principles-for-cloud-native-architecture-what-it-is-and-how-to-master-it.

[2] AWS Lambda:Run code without thinking about servers. Pay only for the compute time you consume. https://aws.amazon.com/lambda/.

[3] Cloud native applications: Ship faster, reduce risk, and grow your business. https://tanzu.vmware.com/cloud-native.

[4] Cncf cloud native definition v1.0. https://github.com/cncf/toc/blob/main/DEFINITION.md.

[5] Decomposing twitter: Adventures in service-oriented architecture. https://www.infoq.com/presentations/twitter-soa/.

[6] Defining cloud native. https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition.

[7] A design analysis of cloud-based microservices architecture at netflix. https://medium.com/swlh/a-design-analysis-of-cloud-based-microservices-architecture-at-netflix-98836b2da45f.

[8] The docker container. https://www.docker.com/.

[9] f-stack. https://www.f-stack.org/.

[10] Introducing domain-oriented microservice architecture. https://eng.uber.com/microservice-architecture/.

[11] iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/.

[12] Journey to being cloud-native – how and where should you start? https://aws.amazon.com/blogs/apn/journey-to-being-cloud-native-how-and-where-should-you-start/.

[13] Kata containers. https://github.com/kata-containers.

[14] Kernel based virtual machine. http://www.linux-kvm.org/.

[15] libsyscall_intercept. https://github.com/pmem/syscall_intercept.

[16] Linux control groups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[17] Microservices at ebay-what it looks like today. https://www.sayonetech.com/blog/microservices-ebay/.

[18] A new golden age for computer architecture. https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext.

[19] Redis. https://redis.io/.

[20] Seccomp bpf (secure computing with filters). https://www.kernel.org/doc/html/v5.1/userspace-api/seccomp_filter.html.

[21] Seccomp security profiles for docker. https://docs.docker.com/engine/security/seccomp/.

[22] The second extended filesystem. https://docs.kernel.org/filesystems/ext2.html.

[23] Separation Anxiety: A Tutorial for Isolating Your System with Linux Namespaces. https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces.

[24] Serverless. https://cloud.google.com/serverless/.

[25] Serverless computing. https://azure.microsoft.com/en-us/overview/serverless-computing/.

[26] What led amazon to its own microservices architecture. https://thenewstack.io/led-amazon-microservices-architecture/.

[27] Common Vulnerabilities and Exposures: Hypervisors. http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=hypervisor.

[28] AGACHE, A., BROOKER, M., IORDACHE, A., LIGUORI, A., NEUGEBAUER, R., PIWONKA, P., AND POPA, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 419–434.

[29] AXBOE, J. Flexible i/o tester. https://github.com/axboe/fio.

[30] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, Association for Computing Machinery, p. 29–44.

[31] BRATTERUD, A., WALLA, A.-A., HAUGERUD, H., ENGELSTAD, P. E., AND BEGNUM, K. Includeos: A minimal, resource efficient unikernel for cloud services. CLOUDCOM '15, IEEE Computer Society, p. 250–257.

[32] CHEN, Y., SHU, J., OU, J., AND LU, Y. Hinfs: A persistent memory file system with both buffering and direct-access. *ACM Trans. Storage 14*, 1 (apr 2018).

[33] COOPER, B. Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB.

[34] DEMARINIS, N., WILLIAMS-KING, K., JIN, D., FONSECA, R., AND KEMERLIS, V. P. sysfilter: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)* (San Sebastian, Oct. 2020), USENIX Association, pp. 459–474.

[35] ELLIS, A. Introducing Functions as a Service (OpenFaaS). https://blog.alexellis.io/introducing-functions-as-a-service/.

[36] HENDRICKSON, S., STURDEVANT, S., HARTER, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless computation with openlambda. *Elastic 60*, 80.

[37] HUANG, H., LAI, J., RAO, J., LU, H., HOU, W., SU, H., XU, Q., ZHONG, J., ZENG, J., WANG, X., HE, Z., HAN, W., LIU, J., MA, T., AND WU, S. Pvm: Efficient shadow paging for deploying secure containers in cloud-native environment. In *Proceedings of the 29th Symposium on Operating Systems Principles* (New York, NY, USA, 2023), SOSP '23, Association for Computing Machinery, p. 515–530.

[38] JACOBS, A., GÜLMEZ, M., ANDRIES, A., VOLCKAERT, S., AND VOULIMENEAS, A. System call interposition without compromise. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2024), pp. 183–194.

[39] KADEKODI, R., LEE, S. K., KASHYAP, S., KIM, T., KOLLI, A., AND CHIDAMBARAM, V. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, Association for Computing Machinery, p. 494–508.

[40] KANTEE, A. The Design and Implementation of the Anykernel and Rump Kernels, 2nd Edition. http://book.rumpkernel.org.

[41] KAUFMANN, A., STAMLER, T., PETER, S., SHARMA, N. K., KRISHNAMURTHY, A., AND ANDERSON, T. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019), EuroSys '19, Association for Computing Machinery.

[42] KIVITY, A., LAOR, D., COSTA, G., ENBERG, P., HAR'EL, N., MARTI, D., AND ZOLOTAROV, V. Osv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 61–72.

[43] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., HUUCK, R., MURRAY, T. C., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)* (Big Sky, Montana, USA, 2009), ACM, pp. 207–220.

[44] KWON, Y., FINGLER, H., HUNT, T., PETER, S., WITCHEL, E., AND ANDERSON, T. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 460–477.

[45] LEI, J., MUNIKAR, M., SUO, K., LU, H., AND RAO, J. Parallelizing packet processing in container overlay networks. In *Proceedings of the Sixteenth European Conference on Computer Systems* (New York, NY, USA, 2021), EuroSys '21, Association for Computing Machinery, p. 261–276.

[46] LEI, J., SUO, K., LU, H., AND RAO, J. Tackling parallelization challenges of kernel network stack for container overlay networks. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)* (Renton, WA, July 2019), USENIX Association.

[47] LIU, J., REBELLO, A., DAI, Y., YE, C., KANNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 819–835.

[48] LIU, J., REBELLO, A., DAI, Y., YE, C., KANNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Scale and performance in a filesystem semi-microkernel. SOSP '21, Association for Computing Machinery, p. 819–835.

[49] LU, H., SALTAFORMAGGIO, B., KOMPELLA, R., AND XU, D. vFair: Latency-aware fair storage scheduling via per-io cost-based differentiation. In *Proceedings of the 6th ACM Symposium on Cloud Computing* (2015).

[50] MANCO, F., LUPU, C., SCHMIDT, F., MENDES, J., KUENZER, S., SATI, S., YASUKATA, K., RAICIU, C., AND HUICI, F. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 218–233.

[51] MARTY, M., DE KRUIJF, M., ADRIAENS, J., ALFELD, C., BAUER, S., CONTAVALLI, C., DALTON, M., DUKKIPATI, N., EVANS, W. C., GRIBBLE, S., KIDD, N., KONONOV, R., KUMAR, G., MAUER, C., MUSICK, E., OLSON, L., RUBOW, E., RYAN, M., SPRINGBORN, K., TURNER, P., VALANCIUS, V., WANG, X., AND VAHDAT, A. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, Association for Computing Machinery, p. 399–413.

[52] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 1–16.

[53] RAHO, M., SPYRIDAKIS, A., PAOLINO, M., AND RAHO, D. Kvm, xen and docker: A performance analysis for arm based nfv and cloud computing. In *Information, Electronic and Electrical Engineering (AIEEE), 2015 IEEE 3rd Workshop on Advances in* (2015), IEEE, pp. 1–8.

[54] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), pp. 199–212.

[55] SOARES, L., AND STUMM, M. FlexSC: Flexible system call scheduling with Exception-Less system calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)* (Vancouver, BC, Oct. 2010), USENIX Association.

[56] WILLIAMS, D., KOLLER, R., LUCINA, M., AND PRAKASH, N. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2018), SoCC '18, Association for Computing Machinery, p. 199–211.

[57] WILLIAMS, D., KOLLER, R., LUCINA, M., AND PRAKASH, N. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2018), SoCC '18, Association for Computing Machinery, p. 199–211.

[58] XEN. http://www.xen.org/.

[59] YANG, J., KIM, J., HOSEINZADEH, M., IZRAELEVITZ, J., AND SWANSON, S. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 169–182.

[60] YAROM, Y., AND FALKNER, K. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)* (2014), pp. 719–732.

[61] YOUNG, E. G., ZHU, P., CARAZA-HARTER, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The true cost of containing: A gvisor case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)* (Renton, WA, July 2019), USENIX Association.

[62] YOUNG, E. G., ZHU, P., CARAZA-HARTER, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The true cost of containing: A gvisor case study. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)* (2019).

[63] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), pp. 305–316.

[64] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), pp. 990–1003.

## Appendix A

Table 2: System call classification.

| User-host interface (supported) | Delegable (supported) | Non-delegable (supported) | Blocked (not supported) | Blocked (root privilege) (not supported) |
|---|---|---|---|---|
| timerfd_create | read | mmap | msgget | sched_getaffinity |
| timerfd_settime | write | munmap | msgsnd | mincore |
| dup | open | mremap | msgrcv | pause |
| poll | close | brk | msgctl | vfork |
| ioctl | stat | mprotect | gettimeofday | times |
| read | fstat | madvise | setsid | rt_sigpending |
| write | lstat | rt_sigprocmask | getsid | rt_sigtimedwait |
| pread64 | poll | rt_sigsuspend | capget | rt_sigqueueinfo |
| msync | lseek | rt_sigreturn | capset | sigaltstack |
| brk | ioctl | rt_sigaction | acct | utime |
| futex | pread64 | prlimit64 | setdomainname | uselib |
| clock_nanosleep | pwrite64 | prctl | iopl | personality |
| tgkill | writev | clock_nanosleep | ioperm | sysfs |
| epoll_create | pipe | set_robust_list | io_setup | getpriority |
| epoll_ctl | dup | rseq | io_destroy | setpriority |
| epoll_wait | dup2 | fork | io_getevents | sched_setparam |
| readv | getpid | clone3 | io_submit | sched_getparam |
| writev | sendfile | execve | io_cancel | sched_setscheduler |
| wait4 | socket | exit_group | get_thread_area | sched_getscheduler |
| listen | connect | wait4 | lookup_dcookie | sched_get_priority_max |
| fcntl | accept | futex | remap_file_pages | sched_get_priority_min |
| rt_sigreturn | sendto | tgkill | semtimedop | sched_rr_get_interval |
| | recvfrom | msync | settimeofday | mlock |
| | sendmsg | arch_prctl | fanotify_init | munlock |
| | recvmsg | alarm | fanotify_mark | mlockall |
| | shutdown | exit | shmget | munlockall |
| | bind | setitimer | shmat | vhangup |
| | listen | getitimer | shmctl | modify_ldt |
| | getsockname | | getrandom | pivot_root |
| | getpeername | | semget | sysctl |
| | socketpair | | semop | adjtimex |
| | setsockopt | | semctl | setrlimit |
| | getsockopt | | shmdt | chroot |
| | uname | | getrlimit | mount |
| | fcntl | | getrusage | umount2 |
| | flock | | | swapon |
| | fsync | | | swapoff |
| | fdatasync | | | reboot |
| | ftruncate | | | create_module |
| | getcwd | | | init_module |
| | chdir | | | delete_module |
| | rename | | | get_kernel_syms |
| | mkdir | | | query_module |
| | rmdir | | | quotactl |

*Continued on next page...*

| User-host interface | Delegable | Non-delegable | Implementable | Blocked |
|---|---|---|---|---|
| | unlink | | | nfsservctl |
| | getppid | | | getpmsg |
| | epoll_create | | | putpmsg |
| | getdents64 | | | afs_syscall |
| | fadvise64 | | | tuxcall |
| | epoll_wait | | | security |
| | epoll_ctl | | | readahead |
| | openat | | | time |
| | newfstatat | | | sched_setaffinity |
| | unlinkat | | | epoll_ctl_old |
| | pselect6 | | | epoll_wait_old |
| | sync_file_range | | | restart_syscall |
| | timerfd_create | | | clock_getres |
| | fallocate | | | utimes |
| | timerfd_settime | | | vserver |
| | timerfd_gettime | | | mbind |
| | accept4 | | | set_mempolicy |
| | eventfd2 | | | get_mempolicy |
| | epoll_create1 | | | mq_open |
| | pipe2 | | | mq_unlink |
| | statx | | | mq_timedsend |
| | access | | | mq_timedreceive |
| | gettid | | | mq_notify |
| | getdents | | | mq_getsetattr |
| | fchdir | | | kexec_load |
| | creat | | | waitid |
| | link | | | add_key |
| | symlink | | | request_key |
| | readlink | | | keyctl |
| | chmod | | | ioprio_set |
| | fchmod | | | ioprio_get |
| | chown | | | inotify_init |
| | fchown | | | inotify_add_watch |
| | lchown | | | inotify_rm_watch |
| | umask | | | migrate_pages |
| | getuid | | | unshare |
| | getgid | | | get_robust_list |
| | setuid | | | splice |
| | setgid | | | tee |
| | geteuid | | | sync_file_range |
| | getegid | | | vmsplice |
| | setpgid | | | move_pages |
| | getpgrp | | | utimensat |
| | truncate | | | inotify_init1 |
| | setreuid | | | rt_tgsigqueueinfo |
| | setregid | | | perf_event_open |
| | getgroups | | | name_to_handle_at |
| | setgroups | | | open_by_handle_at |
| | setresuid | | | clock_adjtime |

*Continued on next page...*

| User-host interface | Delegable | Non-delegable | Implementable | Blocked |
|---|---|---|---|---|
| | getresuid | | | syncfs |
| | setresgid | | | setns |
| | getresgid | | | getcpu |
| | getpgid | | | process_vm_readv |
| | setfsuid | | | process_vm_writev |
| | setfsgid | | | kcmp |
| | ustat | | | finit_module |
| | statfs | | | sched_setattr |
| | fstatfs | | | sched_getattr |
| | sync | | | memfd_create |
| | sethostname | | | kexec_file_load |
| | mkdirat | | | bpf |
| | mknodat | | | userfaultfd |
| | fchownat | | | membarrier |
| | futimesat | | | mlock2 |
| | renameat | | | pkey_mprotect |
| | linkat | | | pkey_alloc |
| | symlinkat | | | pkey_free |
| | readlinkat | | | io_pgetevents |
| | fchmodat | | | seccomp |
| | faccessat | | | sched_yield |
| | ppoll | | | nanosleep |
| | epoll_pwait | | | ptrace |
| | dup3 | | | tkill |
| | preadv | | | syslog |
| | pwritev | | | set_tid_address |
| | recvmmsg | | | set_thread_area |
| | sendmmsg | | | execveat |
| | renameat2 | | | kill |
| | copy_file_range | | | clock_settime |
| | preadv2 | | | clock_gettime |
| | pwritev2 | | | timer_create |
| | sysinfo | | | timer_settime |
| | select | | | timer_gettime |
| | mknod | | | timer_getoverrun |
| | readv | | | timer_delete |
| | setxattr | | | signalfd |
| | lsetxattr | | | eventfd |
| | fsetxattr | | | signalfd4 |
| | getxattr | | | |
| | lgetxattr | | | |
| | fgetxattr | | | |
| | listxattr | | | |
| | llistxattr | | | |
| | flistxattr | | | |
| | removexattr | | | |
| | lremovexattr | | | |
| | fremovexattr | | | |