LITESHIELD: Secure Containers via Lightweight, Composable Userspace µKernel Services

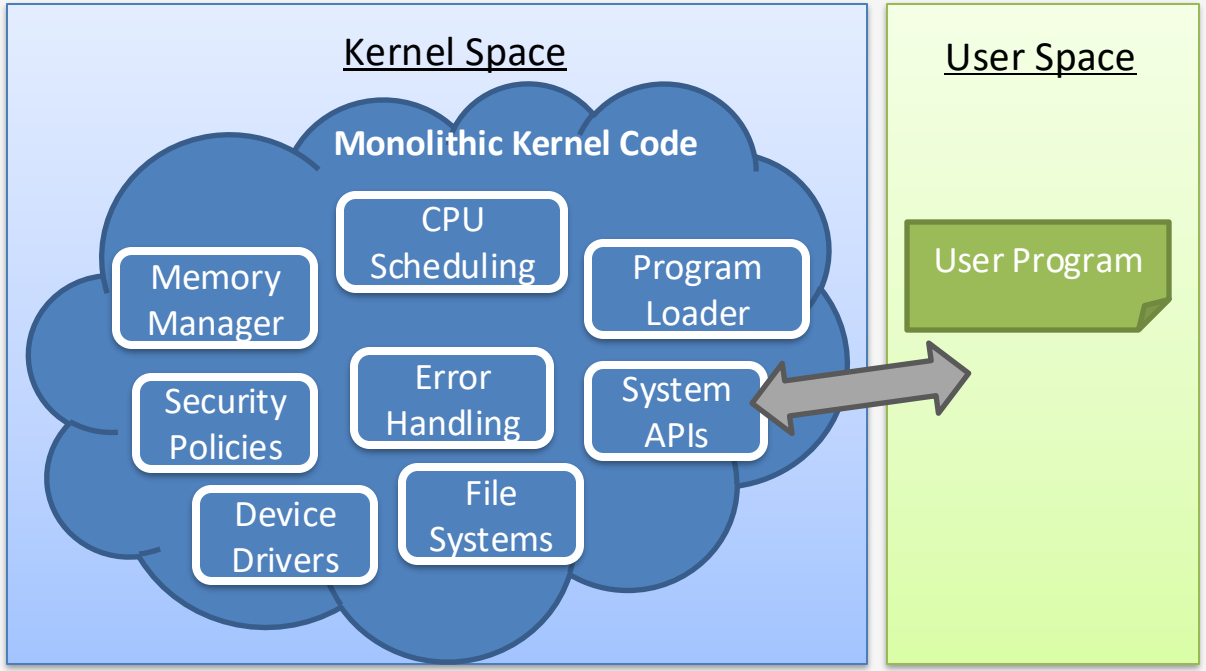
USENIX ATC'25

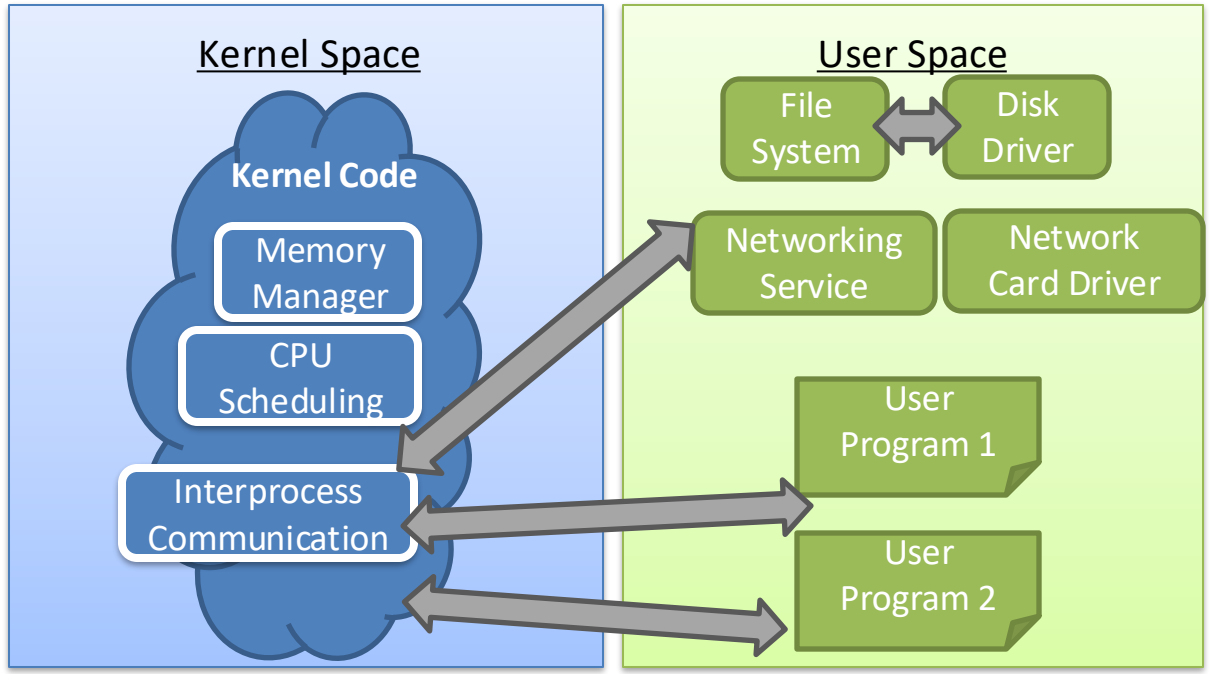
Kaesi Manakkal Nathan Daughety† Marcus Pendleton† Hui Lu

The University of Texas at Arlington, †Air Force Research Laboratory (AFRL)

Microkernel vs. Monolithic Kernel

Secure applications running atop monolithic kernels with the microkernel concepts





Monolithic Kernels:

Huge code base – hard to make it secure

Many features – easy to develop programs

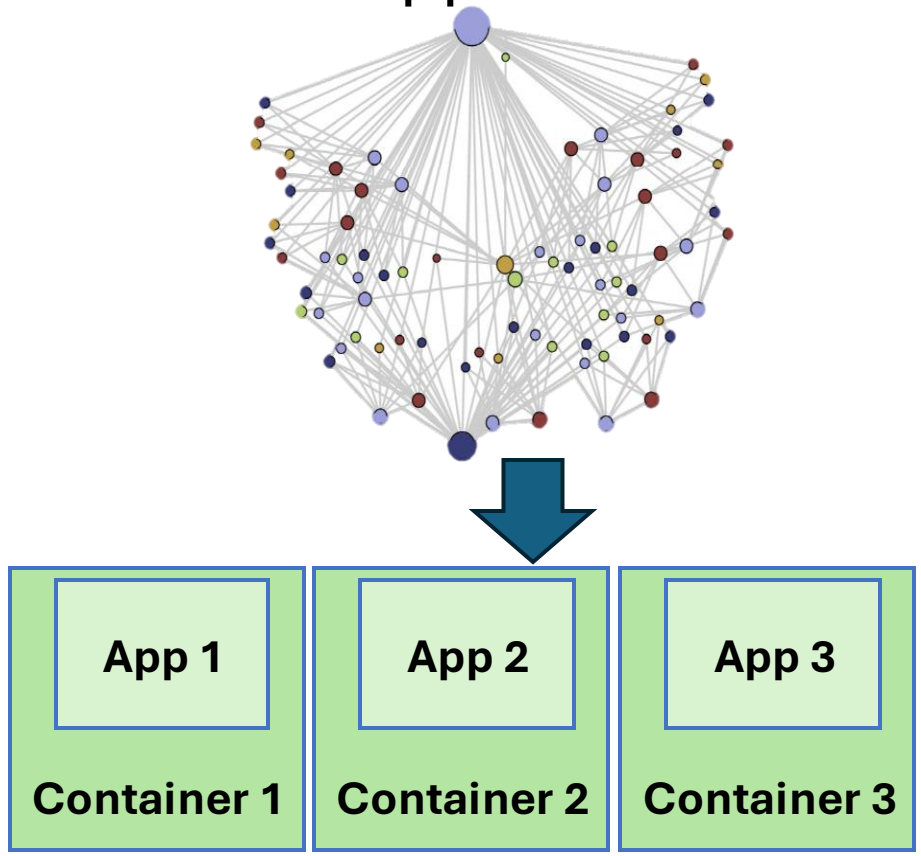
Microkernels:

Small code base – formally verified (e.g., seL4)

Few features – hard to develop user programs

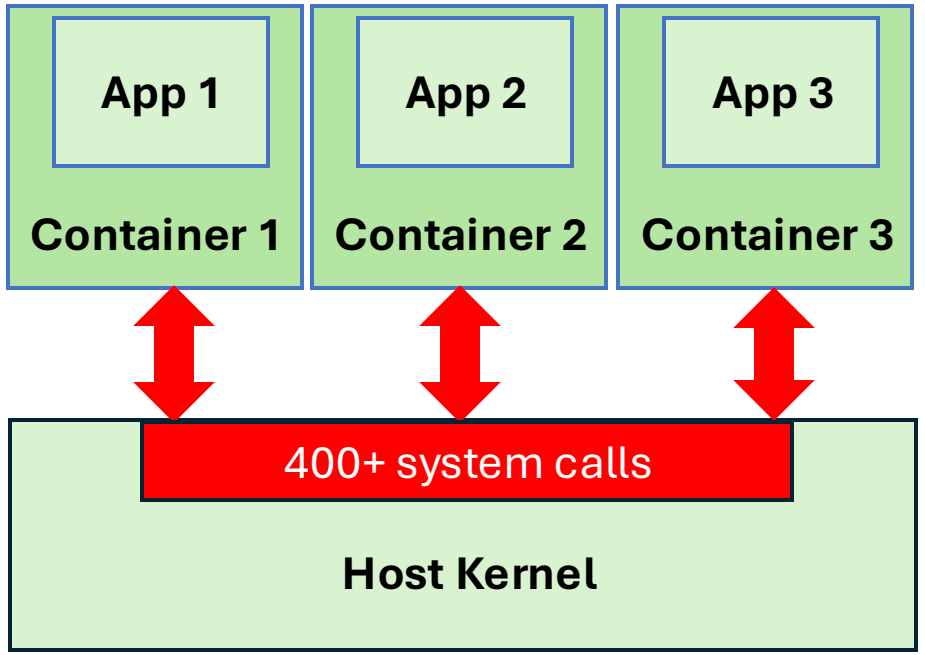
Research Background

- Most modern applications leverage a **microservices** architecture
  - An application consists of numerous small, interconnected components that collectively operate as a unified application.
  - Many benefits include straightforward **horizontal scaling**, **flexibility in deployment**, and **fault tolerance through redundancy**.
  - To deploy these microservices, containers are commonly used, as they are lightweight, and easy to provision on demand



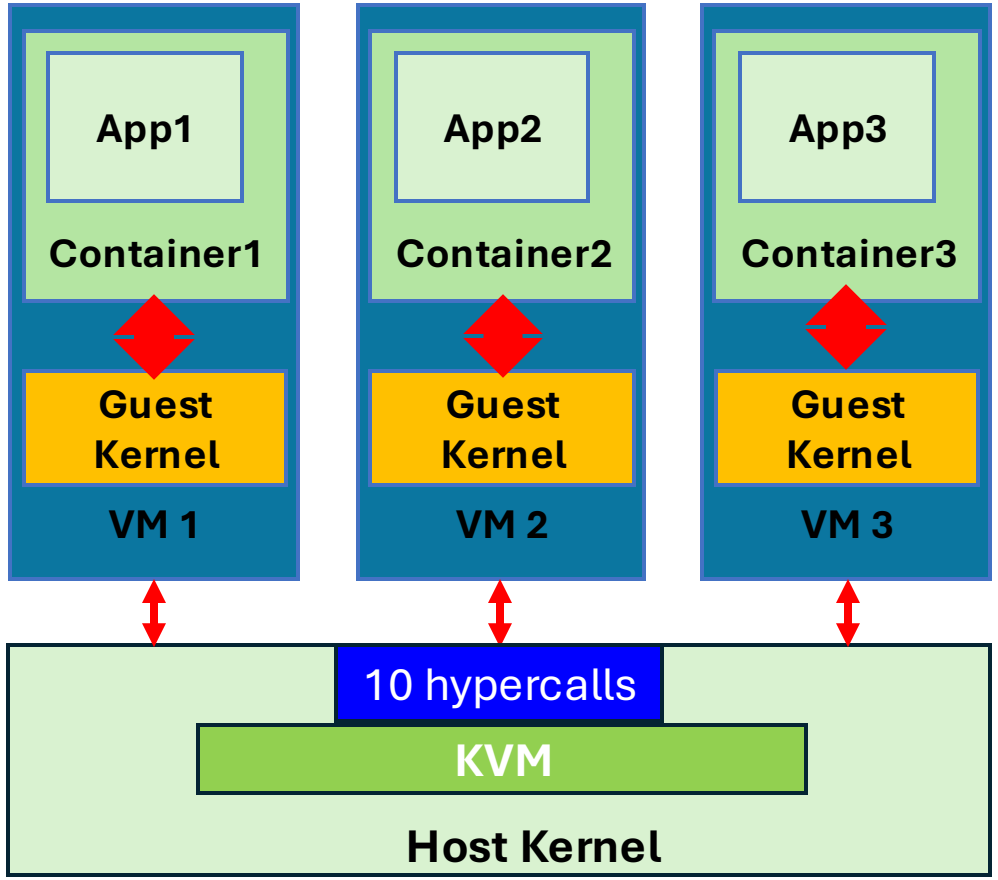
Isolation Problem

- While containers incur less overhead when compared to other isolation mechanisms, e.g. Virtual Machines (VM), they provide **weaker secure isolation**
- The main reason lies in the **large attack surface** between a container and the host kernel (e.g., 400+ system calls)
- Any kernel-related vulnerabilities, exploited via the large attack surface, could break the isolation layer

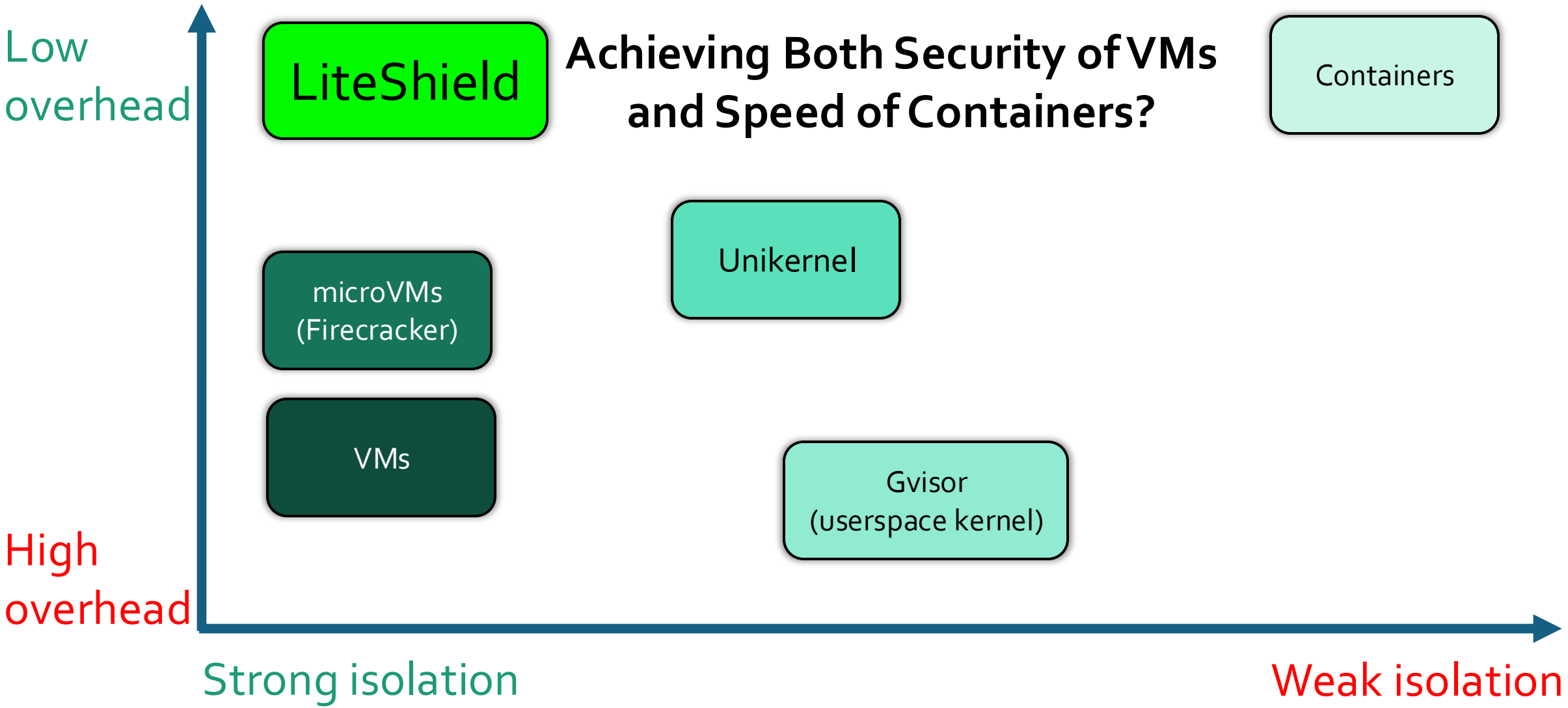


State-of-the-art: VM-based Secure Containers

- The current practical solution to provide **strong isolation** in the public cloud is to **run containers inside a VM** like KVM to separate clients from each other
- This provides a **secure isolation** layer, at the cost of the **high overhead** of running an entire guest OS for both high CPU utilization, large memory footprint, and long I/O latency



Isolation vs. Performance

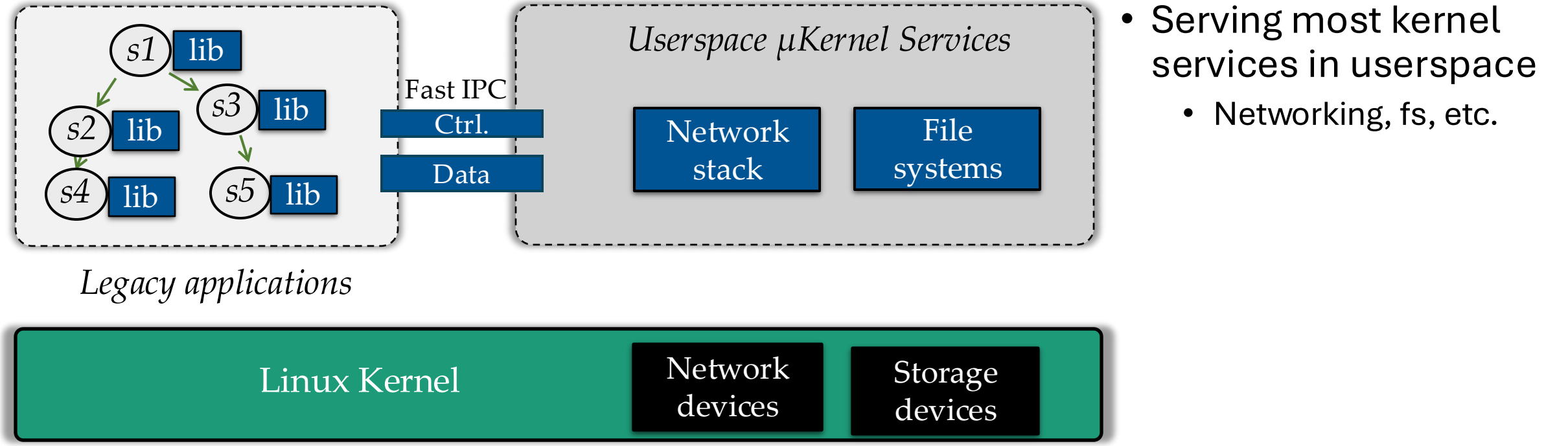


Key Idea: Userspace Microkernel Architecture

- Decouple guest applications from their guest kernels
- Guest kernel runs as a set of userspace microkernel (**μKernel**) services

Key Idea: Userspace Microkernel Architecture

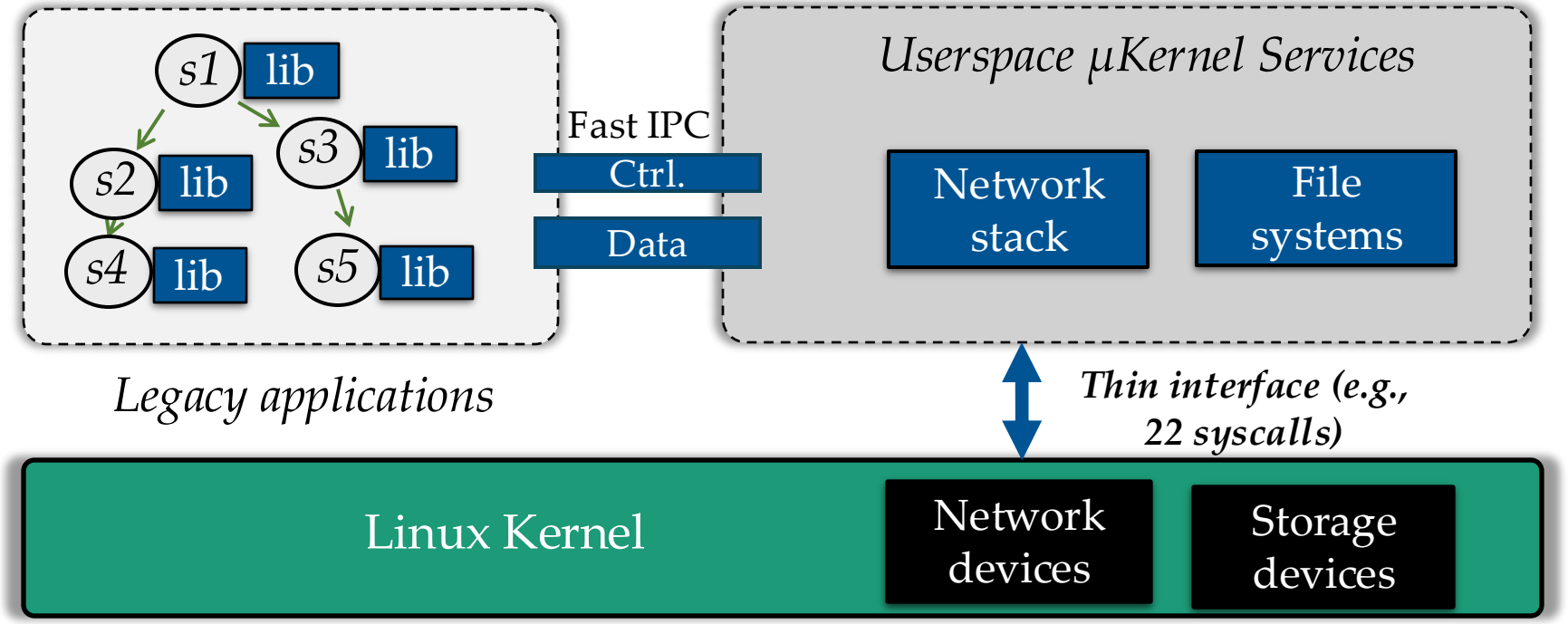
- Decouple guest applications from their guest kernels
- Guest kernel runs as a set of userspace microkernel (**µKernel**) services



- Serving most kernel services in userspace
  - Networking, fs, etc.

Key Idea: Userspace Microkernel Architecture

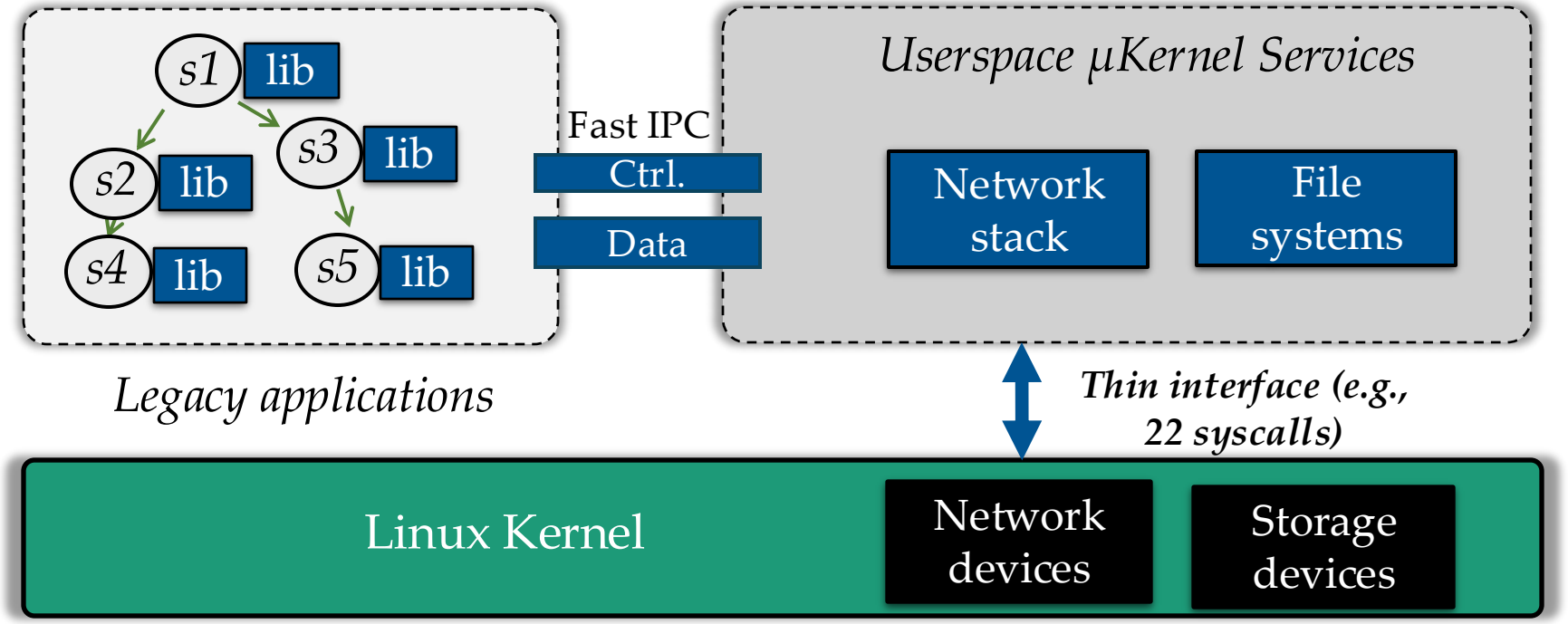
- Decouple guest applications from their guest kernels
- Guest kernel runs as a set of userspace microkernel (**µKernel**) services



- Serving most kernel services in userspace
  - Networking, fs, etc.
- Thin **user-to-host** interface (e.g., 22 syscalls)
  - VM-like isolation

Key Idea: Userspace Microkernel Architecture

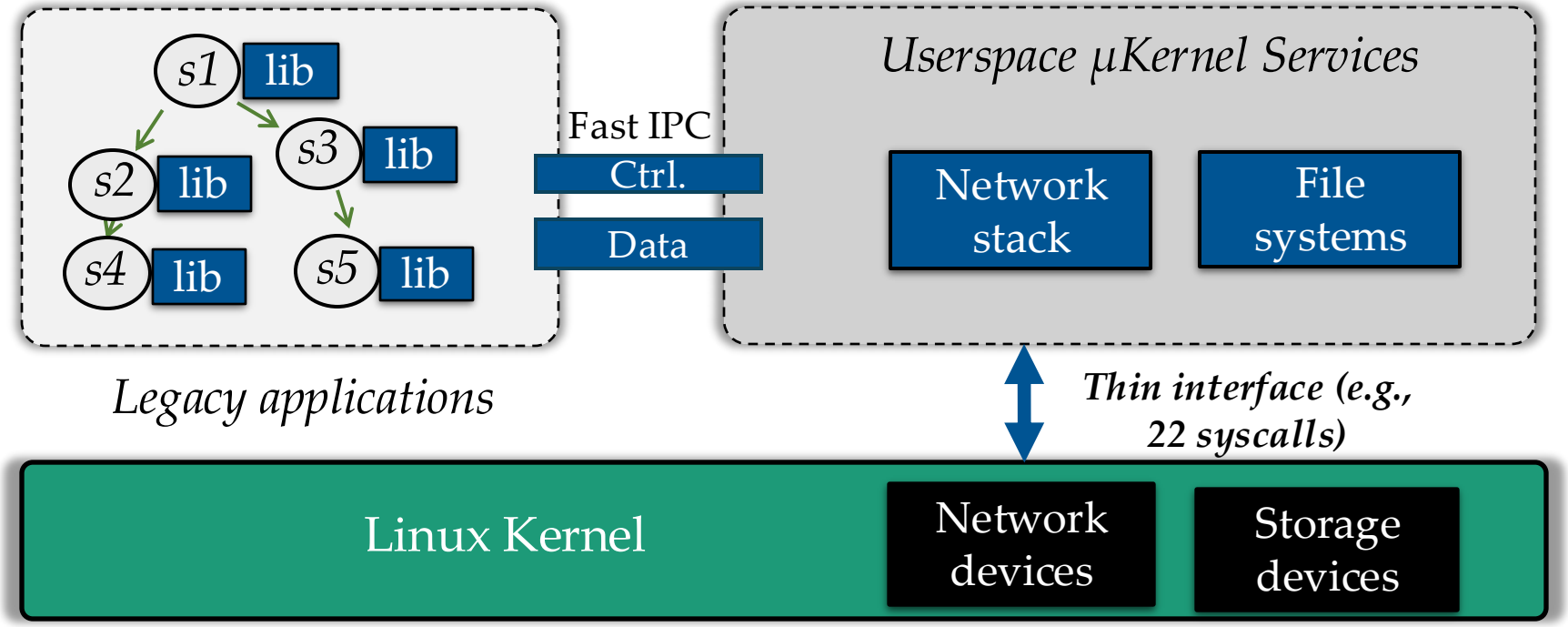
- Decouple guest applications from their guest kernels
- Guest kernel runs as a set of userspace microkernel (**µKernel**) services



- Serving most kernel services in userspace
  - Networking, fs, etc.
- Thin **user-to-host** interface (e.g., 22 syscalls)
  - VM-like isolation
- Fast userspace IPC between apps and µKernel
  - High performance

Key Idea: Userspace Microkernel Architecture

- Decouple guest applications from their guest kernels
- Guest kernel runs as a set of userspace microkernel (**µKernel**) services

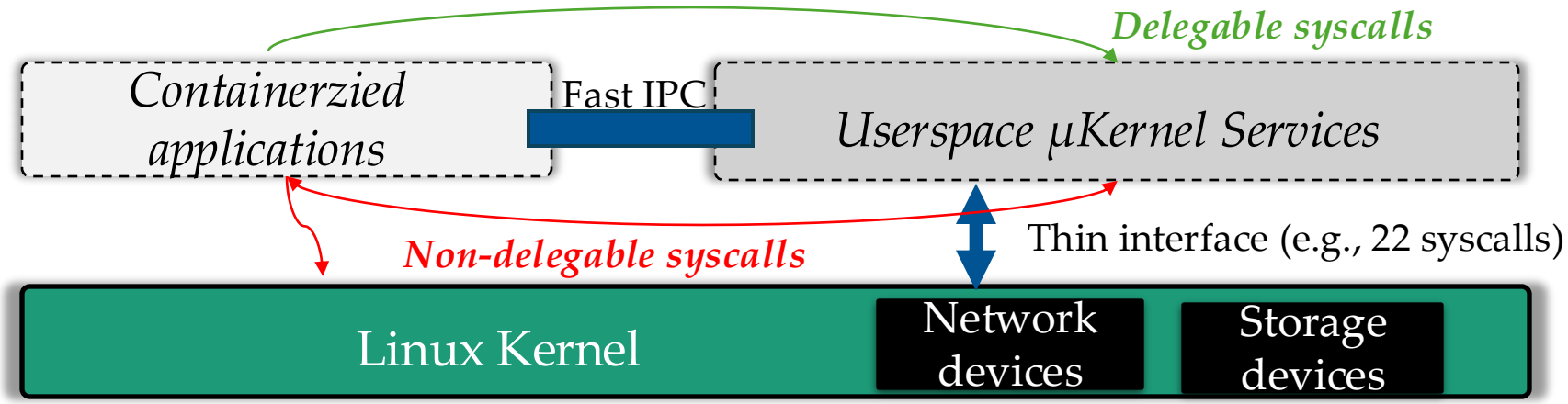


- Serving most kernel services in userspace
  - Networking, fs, etc.
- Thin **user-to-host** interface (e.g., 22 syscalls)
  - VM-like isolation
- Fast userspace IPC between apps and µKernel
  - High performance

The **more** we can accomplish in **userspace**, the **less** that we must do in **kernel** space

Key Challenge - 1

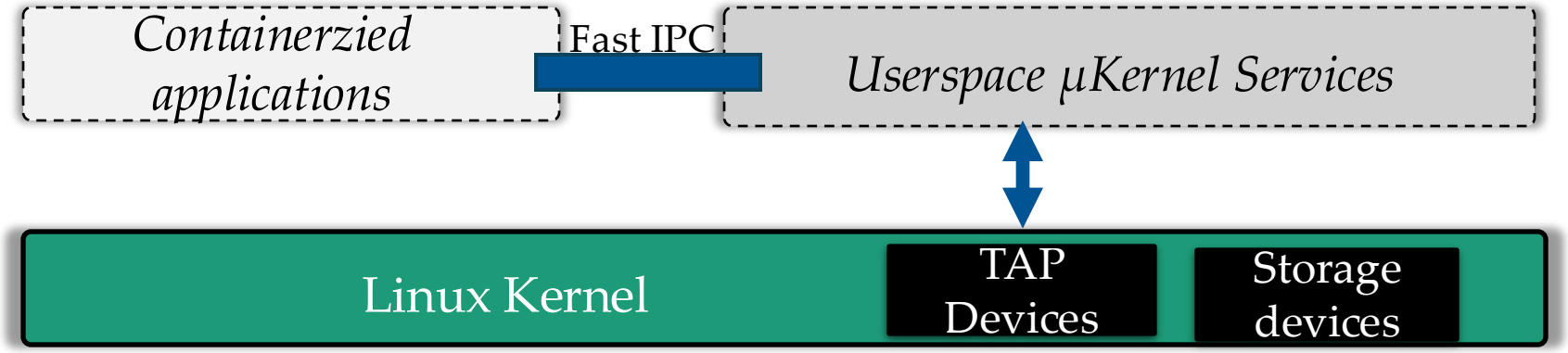
- **No-delegable syscalls:** some syscalls cannot be completely implemented in userspace
  - Must execute **under the same address** space of the guest applications
    - Limited by the Linux kernel's implementation
    - E.g., memory or process management
- **Solution**:
  - Allowed but monitored → use ptrace to enforce correctness



Delegable (140+)	Non-delegable (28)
read	mmap
write	execve
open	fork
close	clone
fstat	exit
accept	kill
…	…

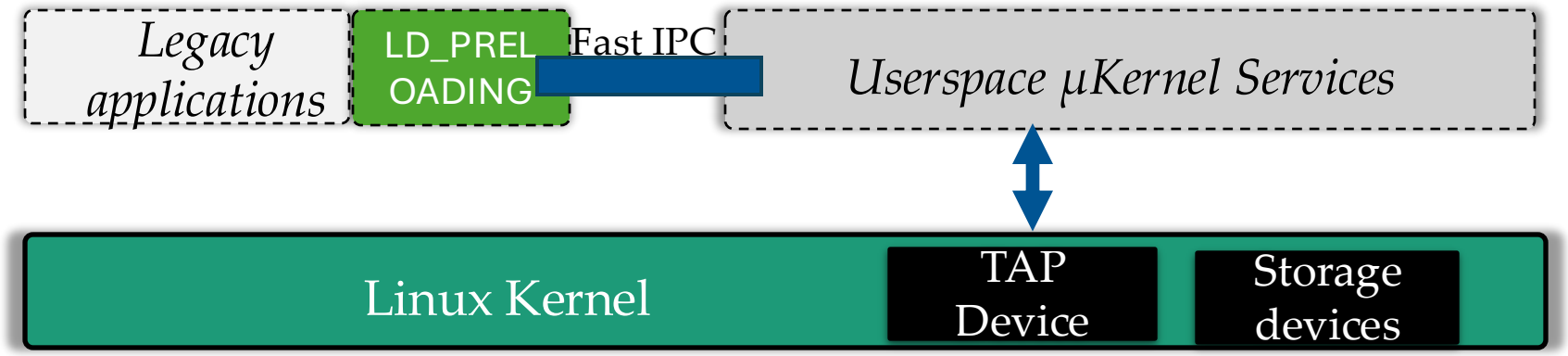
Key Challenge - 2

- **Complete kernel-bypass:** userspace kernel-bypass solutions completely bypass kernel
  - I.e., **no sharing**, each app needs its own NIC
    - Not scalable, infeasible in cloud environments
- **Solution**:
  - Partial kernel bypass: use userspace bypass solutions with emulated devices
    - E.g., use a TAP device as a NIC for DPDK
    - Still benefit from reduced interface (e.g., packet processing in userspace) with **minimal host kernel involvement**



Key Challenge - 3

- **Supporting legacy apps:** traditional Linux apps use syscalls
  - Rewriting apps would require large effort
- **Solution**:
  - LD_PRELOADing: use a preloaded library to intercept requests at syscall granularity



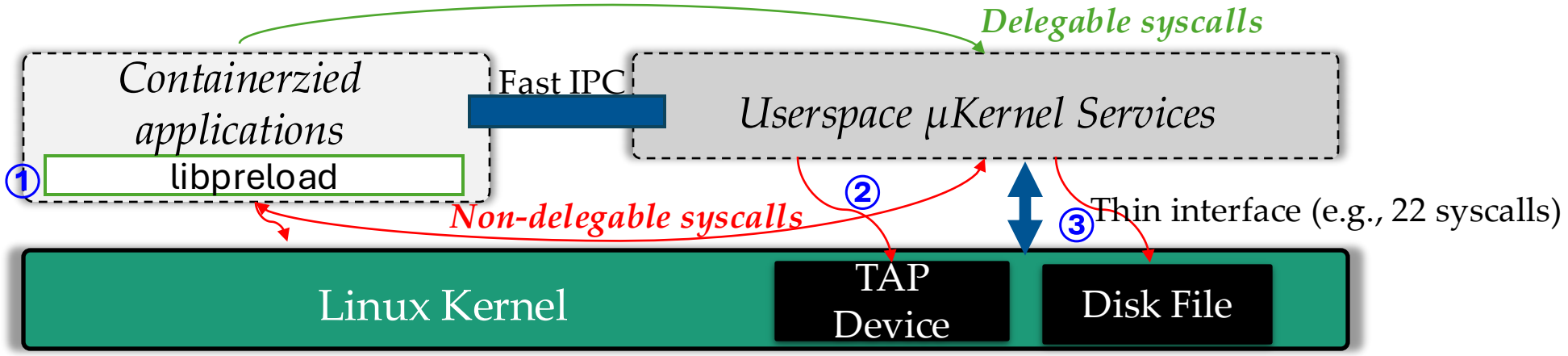
Implementation

① Shared memory IPC calls used to replace traditional syscalls for guest apps using LD_PRELOADING

② A userspace **networking stack** with **DPDK support**

- Based on an existing solution – fstack

③ Userspace **storage stack** with ext2 support backed by a memory-maped file

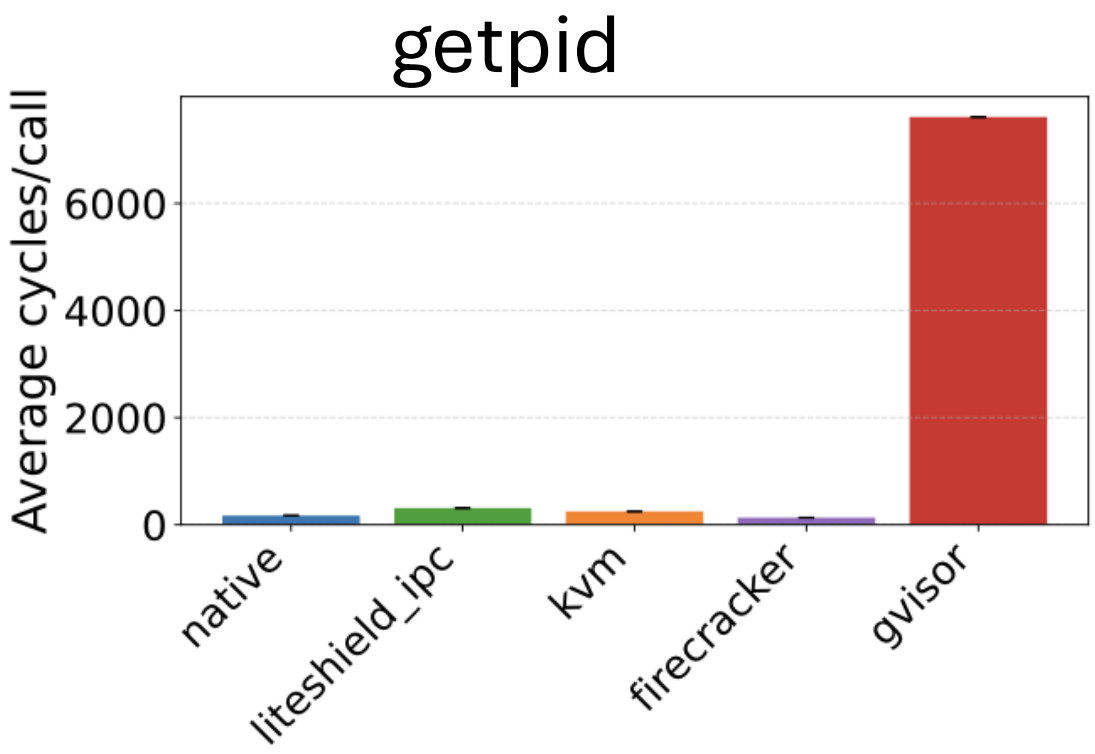


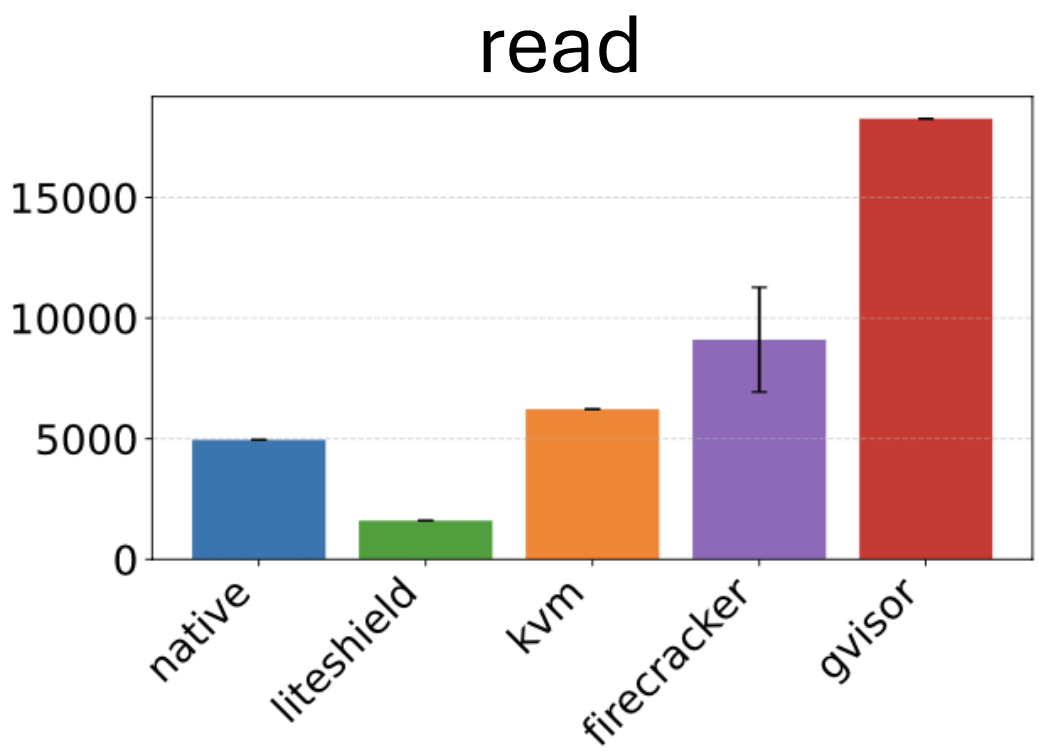
Evaluation

- **~7,000** lines of code for **core services**
- Reduced interface from **400+** to **22**
- **Testbed**
  - **Kernel**: Linux 5.15
  - **CPU**: Intel Xeon Gold 6430 (Hyperthreading disabled)
  - **Memory**: 96GB DDR5 RAM
  - **Disk**: Micron 7450 NVMe SSD (ext4)
- **Baselines** for comparison:
  - Docker containers
  - KVM-based VM
  - Firecracker: a microVM
  - gVisor: a userspace VM

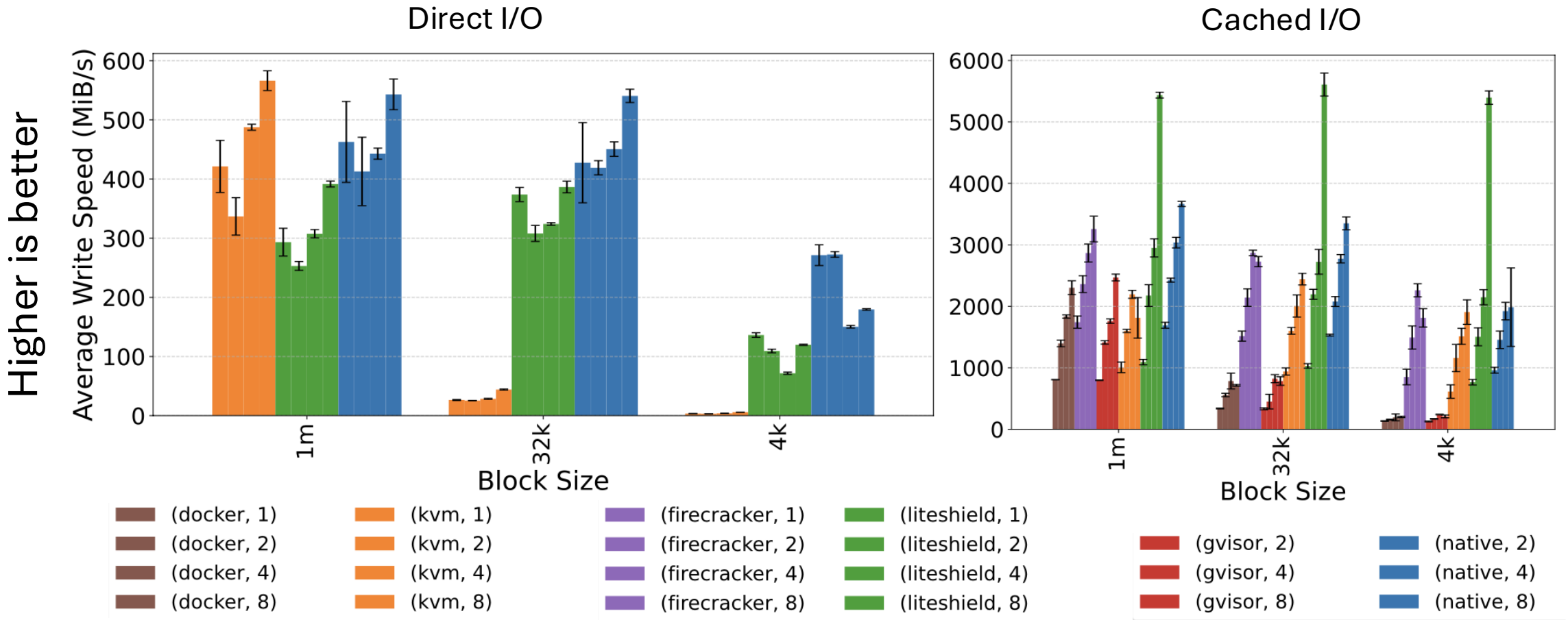
Evaluation: Syscall latency comparison

Lower is better





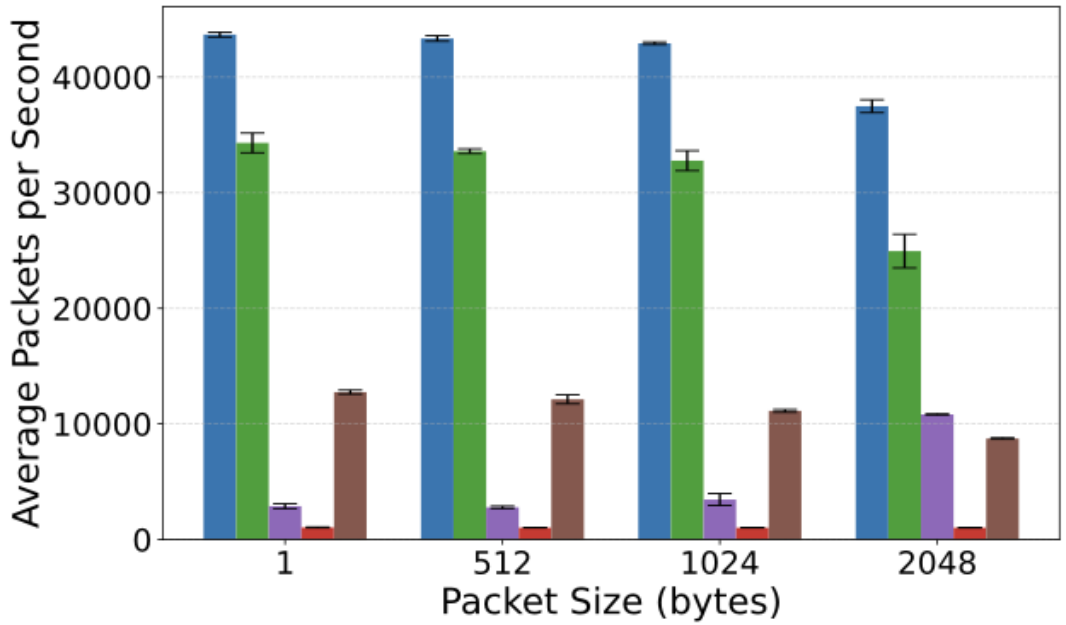
Evaluation: Disk I/O with FIO



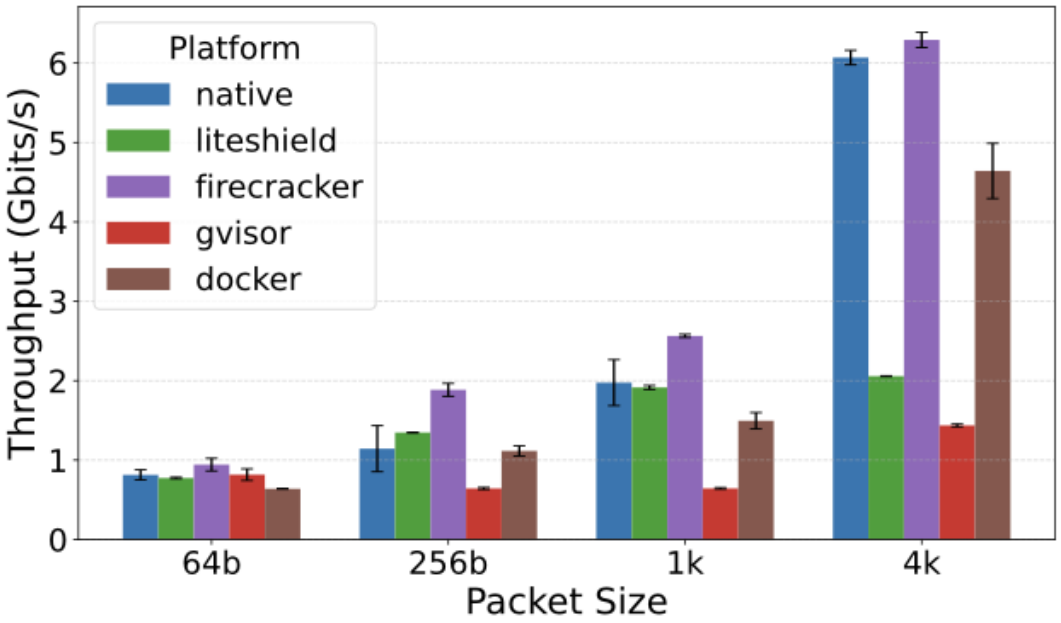
Performance comparisons for writing a 4GB file with various threads and block sizes

Evaluation: Networking Performance

Higher is better



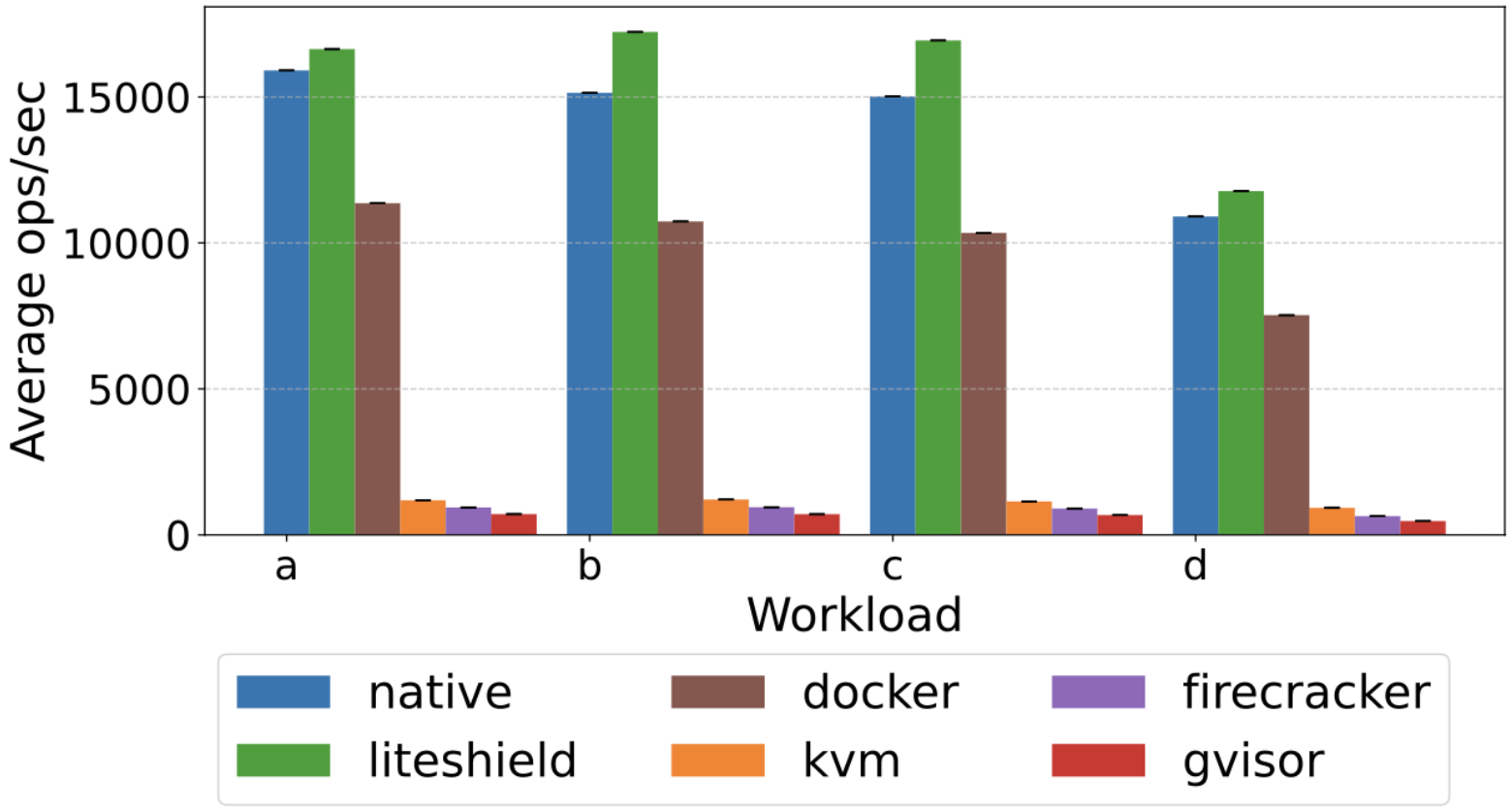
(a) UDP.



(b) TCP.

Evaluation: YCSB on Redis

Higher is better



A	B	C	D
50%/50% Read/Update	95%/5% Read/Update	95%/5% Read/Insert	50%/50% Read/Read-Modify-Write

Summary

- In this talk, we presented **LITESHIELD**, a secure container solution
  - Reduces host-guest interface from **400+** syscalls to **22** by moving kernel services to userspace
  - Replaces traditional syscalls with fast IPC calls
    - Using LD_PRELOAD techniques
  - Uses lightweight kernel abstractions (e.g. TAP) to allow **sharing**
  - Runs legacy apps with **no porting required**

- Evaluation shows that LITESHIELD has comparable performance to the baseline (containers) and is generally faster than VM-based solutions

Thank you!

Kaesi Manakkal Nathan Daughety† Marcus Pendleton† Hui Lu

kxm7920@mavs.uta.edu