CAPIO: Safe Kernel-Bypass of Commodity Devices using Capabilities

Friedrich Doku, Jonathan Laughton, Nick Wanninger, Finn McMillan, Arthur Zakrzewski, Peter Dinda
Northwestern University, Evanston, IL, USA
friedy@u.northwestern.edu, laughtonjonathan@gmail.com, ncw@u.northwestern.edu
finnmcm@u.northwestern.edu, arttomzak@u.northwestern.edu, pdinda@northwestern.edu

***Abstract*—Securing low-latency I/O in commodity systems forces a fundamental trade-off: rely on the kernel's high-overhead mediated interface, or bypass it entirely, exposing sensitive hardware resources to userspace and creating new vulnerabilities. This dilemma stems from a hardware granularity mismatch: standard MMUs operate at page boundaries, making it impossible to selectively expose safe device registers without also exposing the sensitive control registers colocated on the same page. Existing solutions to driver isolation enforce an isolation model that cannot protect sub-page device resources.**

This paper presents CAPIO, the first architecture to leverage hardware capabilities to enforce fine-grained access control on memory-mapped I/O. Unlike prior page-based protections, CAPIO utilizes unforgeable capabilities to create precise, sub-page "slices" of device memory. This mechanism enables the kernel to delegate latency-critical hardware access to userspace applications while strictly preventing interaction with co-located privileged registers.

We implement CAPIO based on CHERI on the ARM Morello platform and demonstrate proof-of-concept safe-access drivers for three commodity network cards. We demonstrate that CAPIO achieves the latency improvements of kernel bypass while enforcing byte-level access control of privileged resources.

***Index Terms*—Access control, computer architecture, operating systems, kernel, memory management, peripheral interfaces, network interfaces, hardware security**

I. Introduction

The kernel often stands in the way of low-latency access to device hardware through expensive ring transitions, abstractions, and memory copy overheads [1]–[3]. Modern commodity systems like Linux or FreeBSD expose their devices through a system call interface, requiring applications to cross the user/kernel boundary for nearly every I/O operation. Because the kernel is ultimately the only entity interacting with the device, this *kernel-mediated device access* is simple, abstract, general, and *safe.* A userspace process cannot use the device to mount an attack on the kernel or another process. However, kernel-mediated device access imposes a significant performance overhead. This overhead is particularly problematic for high-performance I/O workloads where microseconds matter, such as in high-frequency trading or high-performance computing.

In contrast, *kernel-bypass device access* allows the userspace process to directly drive the device, choosing performance over safety. High-end network cards such as NVIDIA Infiniband or AMD Solarflare support user-level access to give developers direct control over the entire network stack and the card itself to create a bespoke design that squeezes out every last cycle. Kernel-bypass device access is desirable beyond networking. For example, databases aiming for maximum performance directly interact with storage controllers according to their needs [4].

Ideally, we would not have to choose between safety and performance. A core problem of achieving *safe kernel-bypass device access* is that of exposing device state and registers to the userspace in a manner that cannot violate safety and security requirements. With existing systems, userspace access to the device is achieved by mapping the device's resources into userspace virtual addresses with page granularity (typically 4KB). As a result, registers and memory that happen to co-reside in a physical page frame in the hardware design are either *all* fully readable and writable or *none* of them are. As an example of this *protection granularity problem*, a status register that userspace might legitimately and safely read might co-exist in the same page frame with a register that a malicious userspace could use to enable unfettered access to all of physical memory through DMA transfers.

High-end network cards achieve safe kernel-bypass device access by essentially providing an additional *alternative user-safe interface* that can instead be mapped into the userspace, with the full-functionality interface being reserved for the kernel. A common model is to leverage hardware features like SR-IOV [5] along with proprietary hardware mechanisms to expose many virtual device queues which can be safely modified and updated by the userspace applications. For example, on AMD Solarflare cards, the user creates a "virtual interface" which allows the application to send and receive raw Ethernet frames over various ports through a virtual command queue safely mapped into their address space [6]. The hardware then pools these queue requests and dispatches them as needed from all virtual interfaces.

The alternative user-safe interface approach is extremely complicated to implement on a device like a network card. It is also overkill in the common case where only a single userspace process needs direct access to the device. The reality is that commodity devices, including network cards, storage controllers, GPUs, accelerators and others, do not provide an alternative user-safe interface, and thus cannot be safely exposed to userspace due to the protection granularity problem.

Device-independent hardware support for protection, such

SC26 Workshops, November 15-20, 2026, Chicago, Illinois, USA
979-8-3195-1221-5/26/$31.00 ©2026 IEEE

as trusted execution environments (TEEs) and virtualization do not solve the protection granularity problem. TEEs, such as Intel SGX and ARM TrustZone, provide isolation for code and data, but do so at a page granularity, generally enforced by the system MMU and IOMMU. Porting drivers or applications to a TEE OS is also a major engineering effort. Virtualization also has the same issue because, once again, protections for memory-mapped device resources operate at the page granularity. Indeed, this is such an issue that some core-local devices like the x64 APIC provide model-specific register (MSR)-level access to allow per-MSR access to be extended to the guest, for example to allow the guest to EOI an interrupt, but not launch one. Only a very tiny set of devices are core-local; per-MSR access control is irrelevant to the vast majority of devices. Of course, a trap-and-emulate approach also allows finer control, but each trap is an entry into the kernel, obviating the point of kernel-bypass device access in the first place.

In this work, we identify the protection granularity problem as the key challenge to achieving safe kernel-bypass device access, and we demonstrate that the problem can be readily solved on a system that provides hardware capabilities. More specifically, we show how to use a CHERI-enabled [7] memory system to solve the problem. CHERI (Capability Hardware Enhanced RISC Instructions) is an architectural extension that replaces traditional integer-based pointers with unforgeable, hardware-enforced capabilities. These capabilities are bounded, tagged references that enforce spatial safety by defining precise access permissions for memory regions, including the base address, length, and allowed operations (read, write, execute).

Capabilities offer fine-grained, *sub-page* access control (down to byte granularity) enforced by the processor with almost no changes required at the application level [7]. With CHERI, the kernel simply maps the entirety of a device's resources into userspace virtual addresses without fear: absent the correct capabilities the application cannot dereference these addresses in any way. The kernel doles out capabilities for specific *byte slices* of the device as it sees fit, tightly controlling the access to the device's resources in order to enforce a security policy. This lets the kernel retain exclusive control over privileged operations like DMA configuration and interrupt management at the "control plane", but defer to the userspace application for "data plane" activities like sending and receiving packets according to its bespoke design. In this model, capabilities serve as unforgeable I/O access tokens: applications cannot forge or expand capabilities to access restricted device registers, even if they exploit memory safety vulnerabilities.

Our approach, CAPIO, consists of three key components: the kernel stub, the device manifest, and the slicer. The kernel stub is responsible for mapping the device resources into the userspace and for handling "control plane" operations that the userspace cannot be trusted to do. The device manifest is essentially an access policy for the device resources, indicating how each range of MMIO addresses mapped by the kernel stub into the userspace is allowed to be accessed. Finally, the slicer transforms the manifest into bounded capabilities that are returned to the user process and can be used by it to access specific runs of device registers and other resources.

Critically, CAPIO works with *any* existing, unmodified commodity device that is accessed via MMIO. CAPIO also has the potential to simplify the design of future high-performance devices, eliminating the need for them to have an alternative user-safe interface. By addressing the protection granularity problem through hardware capabilities implemented in the processor, CAPIO decouples safety from specific device features. It also transforms the kernel from being a data-path bottleneck into being a precise policy administrator, enabling a new class of performant userspace drivers that are safe to use.

We present the design, implementation, and evaluation of a CAPIO prototype for the ARM Morello platform that provides an existence proof that safe kernel-bypass device access to commodity devices can not only be achieved, but can be performant. We target three network cards that span the design space: an off-the-shelf Intel 82574L (e1000e) 1 GbE card, which was *never designed for kernel bypass*; the AMD Solarflare SFN7322F-R2, a dual-port 10 GbE EF10-family NIC whose native ef_vi bypass path enforces isolation only at page granularity; and the Mellanox ConnectX-4, a 100 GbE NIC with a substantially more complex register layout. We provide safe kernel-bypass device access to the e1000e for a FreeBSD process that includes a simple user-level embedded network stack (LwIP), and to the Solarflare for a userspace driver that mirrors ef_vi's virtual-interface abstraction over descriptor rings and doorbells, with per-slice CHERI bounds replacing the NIC's built-in page-level protections. Both setups show considerable reductions in packet round-trip latency compared to alternatives that use safe kernel-mediated device access via the FreeBSD network stack. These microbenchmark results are reinforced by application-level latency reductions of up to 28% for Memcached driven by a YCSB workload. These results suggest that any overheads in the CAPIO approach to safe kernel-bypass device access are likely to be dominated by the performance gains of making user-level device access possible.

Our specific contributions are:

1) We identify the protection granularity problem as key to achieving safe kernel-bypass device access for unmodified commodity devices.
2) We show how to address the protection granularity problem through the mechanism of a hardware capability system.
3) We describe CAPIO, the first architecture to leverage hardware capabilities for enforcing I/O access control at the granularity of individual device registers instead of pages of registers. In our design, kernel/user cooperation results in exposing exactly the safe subset of device resources required to achieve an application goal. We use low-latency networking as an example goal (§III).

4) We give a detailed breakdown of the design and implementation of the CAPIO prototype for the ARM Morello platform, and exemplify the design using the e1000e network card (§IV).
5) We create a user-level driver for the e1000e and couple it with a simple user-level network stack to create a complete example of safe kernel-bypass user-level networking based on the CAPIO approach.
6) We port CAPIO to the Solarflare SFN7322F-R2, an EF10-family 10 GbE NIC whose native kernel-bypass API (ef_vi) enforces isolation only at page granularity, and show that CHERI capabilities can replace that page-based mechanism with byte-granular protection (§IV-E).
7) We evaluate the resulting user-level networking setups, showing that they reduce microbenchmarked UDP latency by up to 46% on the e1000e and port-to-port round-trip latency by up to 43% on the Solarflare, while application TCP latency drops by up to 19% (§V).

We believe that the CAPIO approach is quite general and shows an unexpected benefit of the adoption of hardware capability machines. We also think that our CAPIO prototype generalizes beyond the e1000e or even network cards. Most commodity devices of note operate on a similar descriptor-and-queue model [8]–[10], and, as a result, even the specific scheme we used for the e1000e is likely to be applicable across a large part of the hardware landscape.

II. Background and Related Work

We define *Trusted I/O* as the property by which exchanges between an application and a device are protected such that only authorized parties may initiate or observe transactions, and data cannot be tampered with or intercepted by unprivileged components [11]–[14]. In commodity systems this is an open challenge: existing mechanisms trade access-control granularity against raw performance. Prior work bifurcates into *isolation-centric* systems (e.g., TEEs) that enforce strict logical separation at the cost of performance, and *page-based protection* (virtualization and kernel bypass) that prioritizes performance but relies on coarse-grained MMUs. CAPIO occupies a novel intersection, using CHERI capabilities to enforce strict sub-page access control without either penalty.

A. Trusted Execution Environments

Hardware-based TEEs create Trusted I/O paths by isolating driver execution to a protected memory region. Attestation can be used to verify that the device logic accesses registers as expected, but this comes with porting challenges and performance costs.

Intel SGX enclaves are designed primarily for computation and explicitly lack direct access to device MMIO [13], [14]. Solutions like SGXIO [14] and Aurora [13] must construct synthetic trusted paths using software proxies. Aurora, for instance, leverages System Management Mode (SMM) to intercept I/O interrupts. While effective for isolation, this introduces significant latency due to the heavyweight nature of SMI handlers and the inability of the OS to preempt SMM execution [13].

ARM TrustZone partitions the processor into Secure and Normal worlds. "Minimum Viable Device Drivers" (MVD) [12] leverages this to run security-critical driver logic in the Secure World. While MVD reduces engineering effort via record-replay, it still necessitates a synchronous *World Switch*, a hardware context switch involving pipeline flushes, whenever the Normal World interacts with the device [12]; this is typically more expensive than a switch from user mode to kernel mode.

B. Virtualization, IOMMUs, and Kernel-Bypass

Hardware virtualization protects system memory from malicious devices by enforcing access control at the bus level.

Systems like SUD (Safe User-level Drivers) [15] run unmodified drivers in userspace, relying on the IOMMU to restrict the device's DMA capabilities to specific memory regions. This successfully isolates the kernel from driver crashes, but relies entirely on page-based protection mechanisms to restrict the driver's view of the device [15].

AMD SEV-TIO (Trusted I/O) extends memory encryption to the I/O subsystem [11]. SEV-TIO allows guests to share encrypted memory directly with trusted devices, eliminating software bounce buffers. It leverages the TDISP and IDE standards to enforce isolation at the PCIe link level. However, SEV-TIO targets the virtualization model, assigning devices (or Virtual Functions) to entire Virtual Machines rather than individual applications or processes.

To eliminate system call overhead, high-performance applications utilize kernel-bypass frameworks such as DPDK [16], SPDK [17], and netmap [18]. These architectures map device MMIO regions and DMA buffers directly into the application's virtual address space. While effective for latency, these approaches historically necessitate a binary trade-off: to gain direct access, the application must be fully trusted (often running as root) because the hardware provides no mechanism to restrict access to specific registers within the mapped region.

C. The Protection Granularity Problem

While the approaches above differ in implementation, SUD, SEV-TIO, and DPDK all share a fundamental limitation: they rely on the IOMMU or MMU for protection, which operates at page granularity (typically 4KB).

Commodity device register maps are rarely designed with page-aligned security boundaries. Privileged configuration registers (e.g., interrupt masking, DMA root pointers) frequently reside on the same physical page as the safe I/O registers (e.g., packet tail pointers) required for operation [15]. Consequently, page-based systems cannot expose performance-critical I/O paths to userspace without inadvertently exposing the control registers.

CAPIO resolves this tension by overlaying CHERI capabilities on top of these memory-mapped regions. By bringing hardware-enforced isolation down to the pointer level, CAPIO

can safely expose specific sub-page register ranges to user-space.

D. CHERI and the ARM Morello Platform

To achieve this sub-page isolation, CAPIO relies on CHERI, an architectural extension that enforces fine-grained memory protection through hardware capabilities [7]. At its core, a CHERI capability is an unforgeable token encoding a base address, length, permissions, and a hardware-protected validity tag. Any attempt to access memory beyond these precise bounds, or to maliciously expand a capability, triggers a hardware exception, thereby guaranteeing spatial memory safety by construction.

Crucially, CHERI also provides *sealed capabilities*, which act as opaque, immutable authentication tokens. CAPIO leverages these sealed capabilities to definitively prove that a userspace `mmap` request originates from an authorized process. We implement CAPIO on CheriBSD [19] running atop the ARM Morello platform [20], a prototype SoC that provides native hardware support for these advanced capability and sealing instructions.

E. Software Compartmentalization on CHERI

Prior work has explored CHERI for network and driver security, but focused on software compartmentalization rather than hardware I/O enforcement. Ferraro et al. ported DPDK to Morello, using capability-based VMs to isolate the TCP/IP stack from the main application [21]. While this prevents memory bugs within the stack from corrupting the application, the isolated compartment still retains full page-level access to the entire NIC MMIO region. CAPIO complements this by enforcing register-level access control, ensuring that even a compromised compartment cannot reach privileged device control registers. Similarly, while CHERI has been used to restructure and deprivilege specific GPU drivers [22], CAPIO provides a generalized OS framework for delegating fine-grained MMIO access across commodity devices.

III. Design of CAPIO

CAPIO addresses the protection granularity problem (§II-C) by shifting where access control is enforced. Modern devices routinely pack unrelated control-plane and data-plane registers onto the same page (e.g., interrupt masks colocated with DMA queues). Traditional MMU-based enforcement operates at 4 KB page granularity [23] and cannot separate them, forcing designers to either expose sensitive control registers alongside safe data interfaces or leave all access in the kernel. CAPIO instead enforces access control at the granularity of individual registers.

A. Threat Model

We assume a trusted, CHERI-aware operating system kernel but untrusted or compromised userspace applications. The attacker has arbitrary code execution within their process and may already possess legitimate access to specific data-plane device resources (e.g., a transmit queue). The attacker's goal is to exploit page-granularity mismatches to access privileged

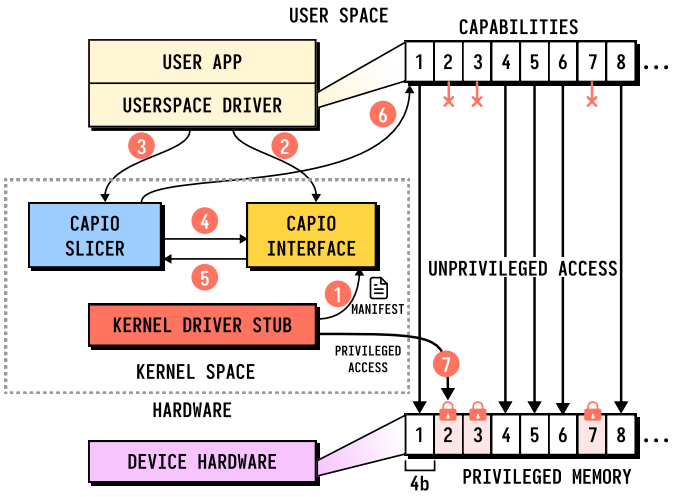


Fig. 1. CAPIO system architecture. The kernel generates bounded capabilities for the authorized driver, enabling direct access to device registers. A compromised or malicious application attempting to access privileged control registers on the same page is architecturally blocked by CHERI hardware bounds checks, preventing unauthorized device reconfiguration.

control registers residing on the same physical page, forge access tokens, or interfere with other device contexts.

We trust the processor to correctly enforce CHERI semantics (bounds, permissions, sealing) and assume the underlying device hardware is non-malicious. The kernel defines access policy, while the hardware enforces it on the critical path. Crucially, standard raw memory interfaces (e.g., `/dev/mem`) are disabled, ensuring userspace processes cannot bypass CAPIO to map arbitrary MMIO regions. Physical tampering, microarchitectural side-channels, and denial-of-service attacks are out of scope.

Against this attacker, CAPIO enforces the following properties:

1) **Sub-Page Device Isolation:** Hardware restricts access to explicitly authorized registers, preventing an application from touching control-plane registers that share the same physical page as data-plane registers.
2) **Unforgeable Authorization:** Access rights are encoded in sealed capabilities, making them impossible for an attacker to fabricate, artificially widen, or bypass.
3) **I/O Provenance:** The kernel can definitively verify that device mapping requests originate from properly authorized processes.
4) **Driver Containment:** A userspace driver is strictly confined to its assigned resources, preventing it from altering global DMA settings, resetting the device, or affecting other hardware contexts.

B. Overview of CAPIO

In the CAPIO architecture, the MMU handles address translation and process isolation, while CHERI hardware capabilities assume the role of fine-grained access enforcement. This separation of concerns allows the kernel to act as a precise policy administrator, defining access rights at the **byte** level. Figure 1 gives an overview of CAPIO system architecture.

The CAPIO architecture consists of five primary components: a pure capability kernel, the kernel stub, the Interface, the userspace driver, and the Slicer. CAPIO requires a **pure capability kernel**, which serves as the root of trust and governs access to all device resources. In CAPIO, the kernel itself runs in pure-capability mode. Crucially, it does not allow unprivileged userspace processes to map arbitrary physical memory (e.g., via `/dev/mem`).

The life cycle begins when the associated **kernel stub** is loaded. The stub is in charge of allocating the device memory, initializing the device, and defining the *Device Manifest*, a specification of the device's register layout and access permissions to these registers. It then registers these with the **Interface** (1). The Interface provides an API to mediate userspace access to devices, and manage mappings defined by the driver.

A **userspace driver** then attaches to the device through CAPIO (2). The userspace driver is an unprivileged, capability-aware process responsible for the high-level device logic. The userspace driver requests mappings to device memory (registers, descriptors, buffers, etc.) through the **Slicer** (3). The Slicer creates a trust boundary by interposing on the standard `mmap` system call for devices. Instead of allowing the userspace driver to map raw physical pages, the Slicer intercepts `mmap` requests, and assigns fine-grained permissions based on the Device Manifest.

To fulfill these requests the Slicer calls into the Interface (4) to request the mapping definitions for a specific device (5). The Slicer provides a runtime policy specifying the exact offsets, lengths, and permissions for each device's register map. It carves a page of registers into slices, each of which is associated with a capability. The Slicer then creates bounded capabilities based on this information and returns them to userspace (6) for use in the userspace driver. Because these capabilities are unforgeable, the driver is architecturally prevented from accessing any system state, such as unrelated registers, that lies outside its explicitly granted slices.

Once this setup is complete, the userspace driver can perform unprivileged accesses through these bounded capabilities with no further kernel interaction in the common (allowed) case. Such accesses may include device register reads/writes, buffer reads/writes, DMA descriptor reads/writes, etc. If the userspace driver needs privileged access (to something for which it has no capabilities), it invokes the kernel stub to perform this access on its behalf, mediated by the kernel in the usual manner (7). An example of such privileged access is configuring DMA registers, as allowing the userspace to write them gives unbounded access to all of physical memory.

IV. Implementation of CAPIO

We implemented CAPIO on the ARM Morello Platform, a functional CHERI extended ARMv8-A SoC [20], running CheriBSD [19]. Our implementation consisted of the *Slicer* kernel module, the CAPIO interface, a kernel stub for a 1GbE e1000e network card, a userspace driver for that same card, and the LwIP network stack. This card was chosen as it has a relatively simple device driver model, while maintaining a "modern"[1] interface of queues and descriptors that are used to issue commands such as send and receive. It was also chosen explicitly *because* it is not amenable to safe kernel bypass as privileged registers are placed interleaved with unprivileged ones, and it requires the kernel to manage privileged DMA memory configurations. We believe that, despite the card's simplicity, its programming interface is in line with a majority of hardware network interfaces, and even beyond that to devices like disks through AHCI or NVMe[2].

A. CheriBSD

To build CAPIO, we leveraged the CheriBSD operating system, an adaptation of FreeBSD for the Morello architecture [19]. Our prototype requires a kernel capable of managing tagged capability memory. CheriBSD provides the necessary *capability-aware* primitives. To implement CAPIO the CheriBSD kernel required minimal changes (∼50 lines of code). These were primarily just modifications to the main `mmap` function in the kernel to allow for requests with an extra pointer, and defining a new device `mmap` callback to accept this extra pointer, and providing an entry function for the Slicer. Importantly, unrelated `mmap` requests were not broken and compatibility is maintained.

B. Slicer

The core enforcement mechanism, *Slicer*, was implemented as a CheriBSD kernel module in ∼416 lines of C code. The primary function of the *Slicer* is to allow for the passing of extra information into mmap requests, handle callback registration for a specific character device, and do the slicing of a memory region based on the manifest returned from the callbacks. Slicing is done simply by calling CHERI intrinsics to set the bounds and permissions of a given root capability (in this case the whole device memory region) and returning an array of those sliced and permission restricted capabilities to userspace. A sealed version of the root capability is returned in the address field of the mmap request, so that memory unmapping is still possible without breaking the security model by returning the original capability.

C. Interface

The CAPIO interface exists to abstract away the complexities of device access and managing the mapping of device memory. Device access is handled through sealed capabilities to create unforgeable tokens that identify the calling process. These sealed capabilities are generated on attach and created by the CAPIO interface. All future requests (mapping or ioctl calls) require this token to be provided.

Device memory mapping is managed through CheriBSD's character device pager interface. It uses the region information provided by the kernel stub to create virtual memory objects, and in turn handle page faults for demand paging of the device

[1]Devices programmed through MMIO-based command queues

[2]These interfaces might have more complexity with restricting access to the DMA "paddr" registers, though.

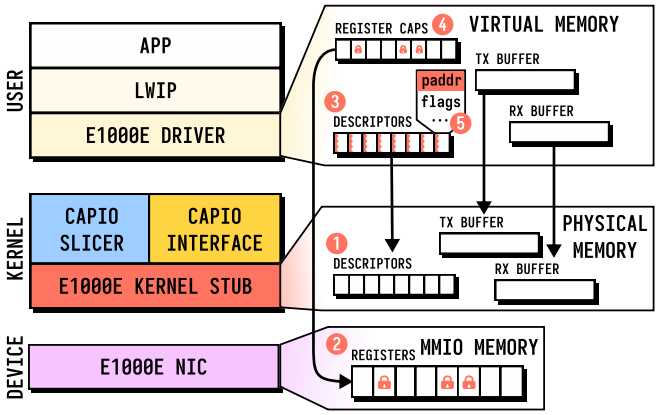


Fig. 2. CAPIO implementation.

memory regions. The memory mapping logic is sufficiently general for common device memory patterns such as MMIO registers, command queue descriptors, and device buffers.

D. e1000e Kernel Stub

The purpose of the kernel stub is to allocate DMA memory, handle the initial setup of the device, and define the shared memory regions that will be exposed to the CAPIO interface. In the case of the e1000e this is handling the PCI probe, attach, and detach sequence, as well as allocating memory (1) for the receive and transmit buffers and their corresponding descriptor rings. Critically, due to granular byte protections, instead of mapping the whole MMIO region (2), we can define individual slices for each MMIO register with their own unique permissions that are returned to the user (4). For example, the STATUS register is defined as read-only, but the CTRL register is not. In the case of receive and transmit descriptors (3), we can carve out the physical address portion of the descriptor struct (5). This prevents arbitrary memory access with the device, but userspace can still modify the flags as needed in the transmit function, for example. To avoid going to the kernel for every receive and transmit, the descriptor rings are preprogrammed to a corresponding device buffer. Additionally, an ioctl call is exposed to allow for a privileged write to allowable regions. This could be updating the address in the transmit descriptor to point to a buffer the userspace process owns. In this case, the stub would verify that this is a valid address by checking the tag and extracting the physical address.

E. Solarflare Kernel Stub

The Solarflare SFN7322F-R2 stresses the CAPIO architecture in a very different direction from the e1000e. It is an EF10-family adapter [24] whose native kernel-bypass API, ef_vi, already isolates userspace clients from privileged device state, but does so at 4 KB page granularity: each *virtual interface* (VI) is assigned an 8 KB window in the device's register region, and control-plane configuration is separated onto the on-card firmware's Management Controller to Driver Interface (MCDI) rather than being placed on the same page as the data-plane doorbells. Porting CAPIO to this card asks whether a purely host-side capability mechanism can replace those hardware-enforced page boundaries with byte-granular ones.

The Solarflare kernel stub is a thin driver that owns the PCI lifecycle, brings the adapter up through MCDI, allocates one VI (event queue, transmit and receive descriptor rings, doorbell window, and a DMA-coherent packet buffer), and registers the resulting regions with the CAPIO interface. Only the control-plane event handler stays in the kernel; every packet the userspace driver sends or receives is a purely userspace operation against CAPIO-bounded capabilities. Because the Solarflare's data path is event-driven rather than descriptor-status-driven, the stub also populates a separate userspace-visible event queue that the driver polls in the hot path, isolating rare control events from per-packet work.

F. Userspace Network Drivers

To demonstrate full-featured, capability-isolated networking, we ported the LwIP network stack [25] to CHERI-extended AArch64 and developed two userspace network drivers. Porting LwIP primarily required retrofitting its memory model to accommodate CHERI's 128-bit capabilities; because LwIP's tightly packed packet buffers and control blocks initially violated CHERI alignment rules, we reconfigured all memory pools to enforce strict 128-bit alignment. The network stack interfaces with our drivers via the standard Network Interface (NETIF) API, and to minimize latency, we copy LwIP's internal buffers directly into the device's DMA regions.

Intel e1000e Driver: Our first driver targets the Intel 82574L (e1000e) Gigabit Ethernet controller and is implemented in just 601 lines of C. The driver authenticates with the device via the CAPIO `ioctl` interface and utilizes the Slicer to map device memory into userspace. Its lifecycle closely mirrors a typical kernel driver: on transmit, it copies the packet to the next free DMA buffer, updates the tail descriptor, and writes to the tail register. For reception, a dedicated thread polls the "done" bit of the receive descriptors. Upon detecting a new packet, it allocates an LwIP buffer, consumes the packet via NETIF's input function, and updates the tail register to point to the last processed descriptor.

Solarflare Driver: For high-performance, low-latency networking, we also implemented a minimal Solarflare userspace driver. In only 405 lines of C, this driver mimics the essential functionality of the native `ef_vi` API. The transmit path copies the frame into a DMA buffer slot, constructs an 8-byte descriptor using a kernel-provided physical address, issues a `dsb sy` memory barrier, and rings the hardware doorbell.

Critically, this doorbell ring is performed through a CAPIO capability whose bounds are restricted *exclusively* to that specific doorbell register. The receive path busy-polls the userspace event queue, reads the filled slot indicated by the receive head pointer, reposts the descriptor, and rings the doorbell again. Because all hot-path functions (poll, receive, transmit) access the hardware solely through finely bounded CAPIO capabilities, a stray write or pointer manipulation error in the driver is architecturally prevented from corrupting other device states.

Component	Lines of Code
CheriBSD	50
Slicer	416
CAPIO Interface	657
CAPIO changes (Solarflare port)	62
e1000e Userspace Driver	601
e1000e Kernel Stub	1,200
Solarflare Kernel Stub	2,721
Solarflare Userspace Driver	405
Memcached	120
LwIP	829
Total	**7,061**

Fig. 3. Code changes by component reflecting the overall porting effort for CAPIO integration.

G. Applications

To demonstrate real-world viability, we ported Memcached [26] to run on top of our LwIP stack in userspace. The main porting cost was CHERI's 128-bit capability alignment: Memcached's slab allocator occasionally produced 8-byte aligned pointers, and loading a 128-bit capability from an 8-byte aligned address faults. We addressed this with structure padding (see Figure 3). Because our architecture bypasses the kernel network stack, we also replaced `libevent`'s `epoll` backend with `select`; this is specific to our implementation, not fundamental to CAPIO (e.g., F-Stack [27] supplies its own `epoll`-compatible interface).

H. Userspace Programming of DMA

Our current implementation delegates DMA physical-address management to the kernel: writing raw physical addresses from userspace would give unbounded memory access. This is not fundamental to CAPIO. A SUD-style approach [15], where the device is restricted to an IOMMU region and userspace freely programs addresses within it, would extend the model to devices that reconfigure DMA at runtime, such as NVMe.

V. Evaluation

We now evaluate the effectiveness of the CAPIO system we have implemented in §IV. Our goal in this evaluation is to show that the systems enabled by CAPIO can take an off-the-shelf commodity device and enable safe, performant, and clean userspace drivers to be built. **With this evaluation we seek to answer the following questions:**

- Does the interface defined by CAPIO effectively control access to protected device registers while making it easy to implement device drivers in userspace using those registers? (§V-B)
- Does the enforcement of fine-grained capability protections introduce significant overhead in the I/O data path? (§V-C)
- Does CHERI capability slicing itself add measurable overhead to individual device register accesses? (§V-G)
- What is the raw latency improvement of a simple network card userspace driver built on CAPIO? (§V-D)
- How does the CAPIO-based kernel-bypass networking approach improve large networked applications and their throughput? (§V-E)
- Is the interface exposed by CAPIO extensible and general, and could it be used in general purpose device driver access? (§V-H)

A. Experimental Setup

Our evaluation was performed on an ARM Morello Development Board running four CHERI-enabled Neoverse N1 ARMv8.2-A processors clocked at 2.5GHz in a single NUMA domain. The CPU features 64KB L1, 1MB Private L2, and 1MB shared L3 caches and 16GB of DDR4 registered DRAM. We installed an Intel 82574L (e1000e) Gigabit Ethernet card in the first PCIe Gen3 x16 slot on the board and drove it with three configurations:

- **Kernel**: Socket-based interface through the CheriBSD kernel.
- **CAPIO-raw**: Raw UDP send/receive operations on the e1000e in userspace through CAPIO capabilities.
- **CAPIO-LwIP**: Driving the e1000e in userspace with CAPIO through the LwIP embedded network stack (§IV-F).

We connected the Morello's e1000e network card *directly* to a Fedora-based Dell Workstation running an identical network card driven through the kernel's network stack. The workstation is equipped with a 3.2GHz Intel i5 quad-core processor and 16 GB of RAM. The workstation can generate workloads that saturate the 1Gbps link. All e1000e latencies and bandwidths are measured as round-trip times from this Fedora workstation.

For the Solarflare evaluation, we installed a single Solarflare SFN7322F-R2 dual-port 10 GbE adapter running the low-latency datapath firmware variant [24]. The two physical functions PF0 and PF1 were connected port-to-port by an SFP+ direct-attach cable, so every frame in the measurement crosses a real 10 GbE link. To keep the measurement self-contained on the Morello, we ran a single test harness that attaches to both PFs and bounces raw Ethernet frames between them, transmitting on PF0, reflecting at PF1, and timestamping the return at PF0 with a single monotonic clock. We compared two configurations on the same NIC: a **Kernel** baseline that performs transmit and receive through ioctl calls into the kernel stub, and a **CAPIO-Direct** configuration in which the userspace driver drives the doorbell, descriptor ring, and event queue through per-region capabilities. We collected 20,000 round-trip samples after 1,000 warm-up round trips at frame sizes of 64, 1024, and 1518 bytes.

B. CAPIO *Correctly Enforces Protections*

By construction, CHERI capabilities are unforgeable [7], [28]: in a pure-capability system a process cannot dereference memory without a capability, cannot exceed a capability's bounds, and cannot widen a derived capability. In CAPIO,

Register	Offset	Size	Perms
CTRL	0x0000	4	RW
STATUS	0x0008	4	RO
IMS	0x00D0	4	Kernel
TDT	0x3818	4	RW

Fig. 4. An example manifest for the e1000e.

the kernel is the sole authority for capability derivation, and userspace drivers receive only the precise capabilities required for their operation. Although the physical page containing both safe and sensitive registers is mapped into the process's address space, the hardware blocks any access not covered by a valid capability.

To validate the *Slicer* against the policy in the *Device Manifest*, we inspect the capabilities it grants for four e1000e registers: `CTRL` (RW), `STATUS` (RO), `IMS` (kernel-only, interrupt mask), and `TDT` (RW). See Figure 4.

Listing 1. Accessing the slices from the manifest in userspace is easy. They are directly mapped into capabilities.

```
/* Userspace Driver */
slices = map_mmio_slices()
enum Regs { REG_CTRL, REG_STATUS, ... };
print_capability("CTRL",   slices[REG_CTRL])
print_capability("STATUS", slices[REG_STATUS])
print_capability("TDT",    slices[REG_TDT])
```

The resulting output confirms the granularity of the protection mechanism: `CTRL` and `STATUS` at offsets `0x00` and `0x08` yield distinct, non-overlapping capabilities.

Listing 2. The capabilities loaded from the slices table directly encode the manifest, and the hardware enforces this protection with unforgeable guarantees.

```
CTRL:   0x40add000, len=4, Read+Write
STATUS: 0x40add008, len=4, Read Only
TDT:    0x40ae0818, len=4, Read+Write
```

Despite residing on the same 4 KB page, each register is bounded and permissioned as the manifest specifies, and `IMS` is absent from the slice list entirely, making it unreachable from userspace. Attempting to reach it by offsetting from `CTRL` to offset `0xD0` also fails:

Listing 3. Attempting to reach the kernel-only `IMS` register by offsetting from the write-enabled `CTRL` capability is invalid.

```
u32 *ctrl = slices[REG_CTRL]; // Offset 0x0
u32 *ims = (u32*)((u8*)ctrl + 0xD0);
*ims = 42; // Invalid Capability (bounds)
// [SIGNAL CAUGHT] SIGPROT (Capability Fault)
```

The hardware guards make cross-register access impossible even though both registers reside on the same page. Despite these strict protections, the engineering cost of a CAPIO userspace driver is not much higher than a traditional in-kernel driver: capabilities behave as pointers, so the driver logic is unchanged; the only substantive difference is indexing into the capability array instead of adding offsets to a single base address.

C. Low-Latency I/O on Commodity Hardware

One of the main reasons to bring a device driver into userspace, bypassing the kernel, is to reduce end-to-end latencies. Many systems built on top of kernel-bypassed network cards operate without a network stack, and often construct raw

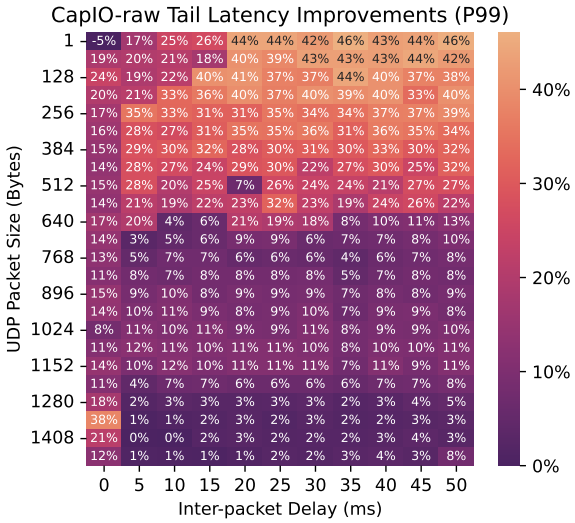


Fig. 5. When a CAPIO-enabled kernel-bypass network driver is driven directly with UDP packets, tail (here, the 99th percentile tail) round-trip latencies are reduced substantially. We see a speedup of up to 46% with packet sizes below 512 bytes. Higher numbers are better (lower latencies). These results are in line with what is expected from kernel-bypass networking. This latency speedup is a reduction of $73\mu s$ ($235\mu s \rightarrow 162\mu s$).

Ethernet frames in the buffers before sending them. To that end, we evaluate the "raw" performance of our commodity e1000e network card using the CAPIO-enabled userspace driver, comparing against the CheriBSD/FreeBSD network stack driven over the POSIX socket interface.

To measure the raw performance we ran a simple "echo server" on the ARM Morello using both CAPIO-raw and kernel sockets. The Dell workstation then sends UDP packets through the Linux network stack, varying two dimensions: packet size and inter-packet delay, measuring the time it takes for the Morello to echo that packet back. CAPIO-raw means the e1000e network card is driven by writing Ethernet+IP+UDP[3] frames directly into the transmit buffers and by writing the descriptor index into the transmit queue. Packets are then received by *polling* on the tail descriptor ready bit on the e1000e card, and packets are read directly out of the transmit buffers. This approach to using the network card should approach the lower bound in latencies achievable by the NIC, and this level of control is typical of latency-critical kernel bypass networking.

The results of this are shown in Figure 5. This figure, and others like it, constitute a heat map. The vertical axis is the packet/write size (between 1 byte and the MTU of 1472), the horizontal axis is the interval between packets/writes, and the heat dimension (color) is the improvement of the metric (99th percentile tail latency) of the CAPIO version over the baseline version (higher percentages mean lower latencies).

The main takeaway here is that, as expected, bypassing the kernel improves latency considerably, by up to 46% in the extremes, and the safety guarantees made by CAPIO enable this latency gain on commodity hardware never designed for

[3]We use UDP here as it has the lowest protocol overhead while still allowing the Dell workstation to receive the packets in userspace.

this kind of interface. The latency gains seen by safe kernel-bypass are most apparent in situations where the fixed costs of the kernel dominate the overall time. Those fixed kernel costs are, briefly: entering and exiting the kernel at least twice for the `recvfrom` and `sendto` system calls (roughly 450 cycles each), the kernel handling an interrupt from the NIC when packets arrive (and depart), and the cost of potentially scheduling and context switching back to the application using the network. When there is no inter-packet delay, these costs dominate, and are easily combated with a driver whose operation can be contained entirely in userspace. As soon as there is a delay between packets (5ms and above in Figure 5), this advantage is diminished as now the majority of the time spent handling a packet is simply waiting for it to arrive.

D. Safe Userspace Networking is Practical

While the raw device access results in §V-C demonstrate CAPIO's ability to eliminate kernel overhead, real-world applications rarely interact with network hardware at such a low level. Most networked applications require a TCP/IP stack to handle protocol processing, connection management, and data segmentation. The critical question is whether CAPIO's safety guarantees impose prohibitive overhead when integrated with a full network stack, and whether the performance benefits of kernel bypass survive the addition of userspace protocol processing.

To answer this question, we evaluated CAPIO using LwIP, a lightweight IP stack originally designed for embedded systems. LwIP prioritizes portability, low memory footprint, and implementation simplicity over raw performance. Nothing about CAPIO limits it to LwIP, and many fast userspace network stacks are available [16], [27], [29]. LwIP contrasts sharply with our baseline, FreeBSD's network stack, which is known for its high performance [30] and serves as the foundation for systems like F-Stack [27]. It has been tuned over decades for server workloads and includes sophisticated optimizations including zero-copy socket buffers. By comparison, LwIP lacks many of the optimizations, and processes packets with significantly more memory copies and less efficient buffer management.

We evaluate this LwIP implementation on top of a safe CAPIO-enabled driver in an identical manner to §V-C. Figure 6 demonstrates that CAPIO-enabled kernel-bypass with LwIP consistently reduces 99th percentile tail latencies compared to FreeBSD's kernel-mediated socket interface. The improvements are most pronounced for small packets (16-300 bytes) at low inter-packet delays where we observe 15-25% reduction in P99 latency. This regime is particularly relevant for latency-sensitive applications like microservices, HPC, and real-time communication where small control messages dominate.

The magnitude of improvement is lower than the raw device access case (§V-C) which is expected. Having LwIP in the data path introduces additional overheads: packet copies between the network stack and application buffers, protocol header processing, checksum computation, and TCP state management.

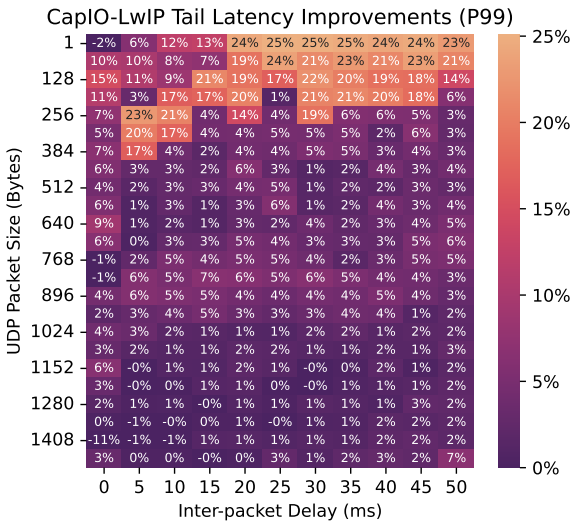


Fig. 6. With a full network stack (LwIP), CAPIO-based kernel bypass networking still outperforms the FreeBSD kernel's stack, but the margin is less. This is an improvement of $44\mu s$ over the kernel in the top right configuration in the plot, implying the LwIP network stack has roughly $30\mu s$ overhead in that configuration.

As packet sizes increase beyond 300 bytes and inter-packet delays grow, the performance gap narrows. This convergence occurs because larger packets amortize fixed per-packet costs (such as system call overhead) over more bytes, and longer delays mean that waiting for packet arrival dominates processing time. In these scenarios, the absolute time spent in the kernel becomes less significant relative to the overall transaction time.

An important observation from these results is that CAPIO decouples safety from performance. The consistent improvements across packet sizes and delay configurations demonstrate that CAPIO's architectural approach to safe user-level access to byte-granular privileged memory does not adversely affect round-trip packet/write latencies. We believe that integrating an alternative network stack such as the one found in F-Stack would improve these results even further, but is outside the scope of this paper.

E. Applications Using CAPIO-based Kernel-Bypass Networking Benefit Similarly

To demonstrate the real-world performance of CAPIO, we ported Memcached [26] to use the CAPIO-based LwIP network stack from §V-D.

Memcached is a widely deployed in-memory key-value store that serves as a critical component in web infrastructure, caching database query results, session state, and rendered page fragments. It often handles millions of requests per second with strict latency requirements. Its performance characteristics are well studied, and it serves as a representative workload for latency-sensitive services.

We evaluated this modified version of memcached using the Yahoo Cloud Serving Benchmark suite (YCSB) [31], an open-source suite designed to evaluate the performance of key-value and cloud data serving systems under realistic workloads. It provides a standardized set of workload patterns ranging from

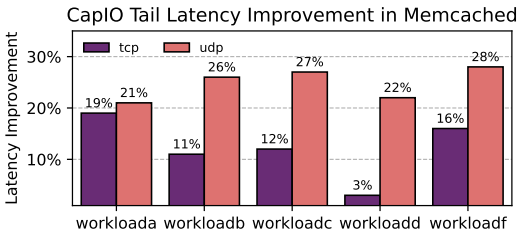


Fig. 7. End-to-end latencies in Memcached driven by the YCSB workload generator have improvements across the board in TCP and UDP.

read-heavy (workloadb) and write-heavy (workloada) to read-modify-write scenarios (workloadf). We used YCSB to generate workloads and evaluated these against memcached using both TCP and UDP protocols[4]. The latency improvements are shown in Figure 7 for these workloads. These results demonstrate that CAPIO's architectural benefits of sub-page device isolation through hardware capabilities translate into measurable, practical improvements for real-world networked applications.

F. Kernel-Bypass on a Modern 10 GbE NIC

The e1000e results demonstrate CAPIO on hardware that was never designed for kernel bypass, but the more interesting question is what CAPIO looks like on a NIC that *was*. The Solarflare SFN7322F-R2 already exposes doorbells, descriptor rings, and event queues to userspace through ef_vi. On such hardware the design question is not whether kernel bypass is possible, but whether host-side CHERI capabilities can replace the NIC's own page-based isolation without giving back its latency benefits.

Figure 8 answers this by comparing the CAPIO-Direct configuration to the Kernel baseline described in §V-A. At 64-byte frames the CAPIO-Direct path completes a port-to-port round trip in 7.16 μs compared to 12.55 μs for the kernel path, a 43% reduction. At 1024 bytes the improvement is 32%, and at 1518 bytes it is 27%. Note also the tightness of the distributions: because both configurations use synchronous busy polling and no interrupts, p99 tail latencies sit within a few percent of the observed minimums, so the observed speedup is not an artifact of a heavier tail on one side.

To confirm the speedup does not come at the cost of the safety property, we also ran memory-safety tests against the Solarflare driver covering the three classes of capability granted to userspace: an out-of-bounds store on the doorbell, a one-byte overflow past a 1 MB packet-buffer capability, and an off-by-one write past the 512-entry transmit descriptor ring. Each violation ran in a forked child, and all three raised `SIGPROT` before any device state or DMA memory could be corrupted, confirming that the sub-page isolation demonstrated on the e1000e (§V-B) extends unchanged to the substantially more complex Solarflare interface.

One caveat: the driver runs the NIC in Physical Addressing Mode, in which a driver that could write arbitrary values into the physical-address field of a descriptor could in principle DMA anywhere. CAPIO's bounds prevent the driver from writing that field to a value the kernel did not approve, so the attack surface is closed under our threat model (§III-A); we discuss Buffer Table Mode as a future extension in §VII.

[4]Note that the UDP protocol for memcached is not fully supported by the *libmemcached* library, so some light engineering was needed to get it working.

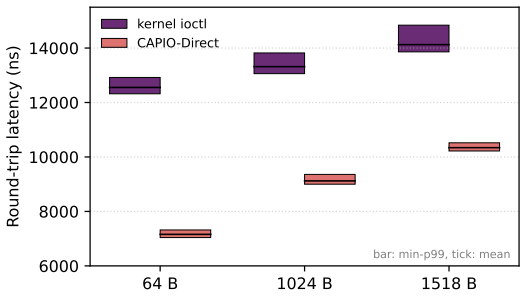


Fig. 8. Port-to-port round-trip latency of the Solarflare SFN7322F-R2 driven through the kernel ioctl path (purple) and through the CAPIO-Direct userspace driver (salmon). Bars span minimum to p99 across 20,000 samples; the tick marks the mean. Kernel bypass through CAPIO reduces round-trip latency by 43%, 32%, and 27% for 64 B, 1024 B, and 1518 B frames respectively.

Hierarchy	Baseline (ns)	CHERI Overhead Δ (ns)	Δ %
Cached RAM	0.78	+0.00	+0.5
DRAM	125.90	+2.10	+1.6
e1000e MMIO	918.70	+0.03	+0.0
mlx5 MMIO	2589.00	-0.05	-0.0

Fig. 9. Adding capabilities across various memory types has a negligible effect on I/O latency.

G. Zero-Cost Safety on the I/O Path

A central claim of CAPIO is that fine-grained access control preserves I/O performance. Because CHERI bounds checking is implemented as hardware combinational logic evaluated in a single cycle [32], capability overhead on the critical path is minimal. We test this via an ablation across four memory tiers (Figure 9): cached RAM, DRAM (a 256 MB shuffled pointer-chase), and MMIO on the e1000e and ConnectX-4.

Using a hybrid-mode binary, we compare a no-CHERI baseline (`ldr w0,[x1]`) against capability-checked loads (`ldr w0,[c1]`) over 500,000 iterations. For MMIO, we compare raw BAR0 access to CAPIO's per-register **sliced** capabilities. As shown, CHERI hardware overhead is negligible on cached RAM (+0.5%) and small on DRAM (+1.6%). Crucially, CAPIO's per-register slicing overhead is within measurement noise on both NICs. Because baseline MMIO latencies vastly exceed capability-check times, CAPIO achieves sub-page safety while maintaining full kernel-bypass speed.

H. The CAPIO *Interface is Generalizable*

Beyond the e1000e and 10 GbE Solarflare, we demonstrate generalizability on the 100 GbE Mellanox ConnectX-4. Its register layout highlights the necessity of sub-page protection: an initialization segment (firmware, status) and a User Access Region (doorbells, BlueFlame write-combining area) co-locate on the same 4 KB BAR0 pages, rendering standard MMU-level isolation insufficient.

Register	Offset	Size	Perms
fw_rev	+0x000	4 B	RO
cmdif_rev	+0x004	4 B	RO
cmd_dbell	+0x010	4 B	RW
initializing	+0x1FC	4 B	RO
sq_dbell	+0x1000	4 B	RW
cq_dbell	+0x1008	4 B	RW
bf_area	+0x1010	128 B	RW

Fig. 10. ConnectX-4 CAPIO manifest: seven slices across two 4 KB BAR0 pages.

Our manifest exposes seven precise slices across these regions (Figure 10), explicitly omitting hardware-internal control registers. We implemented a userspace driver capable of packet transmission (deferring receive, as the native Linux driver exceeds 30,000 lines [33]). Hardware verification confirms that valid accesses succeed, while writes to RO registers or dereferences past a slice boundary deterministically raise CHERI capability faults.

VI. Limitations

Safe userspace DMA. CAPIO delegates DMA address management to the kernel. This suits static descriptor rings but excludes runtime DMA reconfiguration (e.g., NVMe). Combining capability slicing with IOMMU restrictions is a promising path forward. **Application overhead.** Our evaluation isolates the I/O path; full purecap applications incur up to 1.65× overhead [32]. Adopting split hybrid-purecap runtime execution paths could decouple CAPIO's benefits from this global cost. **Hardware requirements.** CAPIO requires a CHERI kernel and Morello-class hardware, currently unavailable on commodity x86 or standard AArch64. We present CAPIO primarily as an architectural roadmap for future capability systems.

VII. Conclusion and Future Work

We present CAPIO, the first system to leverage hardware capabilities for fine-grained MMIO control, enabling safe, latency-critical userspace access while protecting privileged registers. Evaluating across three NICs of varying complexity, CAPIO reduces UDP round-trips by up to 46% on the e1000e (1 GbE), port-to-port latency by 43% on the Solarflare (10 GbE), and Memcached TCP latency by 19%, entirely hiding capability checks within PCIe latency. The framework is highly extensible—supporting the Solarflare required only a 62-line change to the CAPIO core, and our Mellanox ConnectX-4 (100 GbE) port confirms it scales to complex register layouts.

Future work will extend CAPIO beyond networking to storage architecture, specifically targeting NVMe. Because commodity NVMe drives typically lack SR-IOV support, frameworks relying on standard VFIO must assign the entire physical device to a single userspace process. CAPIO provides a mechanism to safely multiplex a single commodity drive by granting multiple applications finely bounded access to their specific submission and completion queue doorbells. We also plan to explore Solarflare's Buffer Table Mode to complement CAPIO's CPU-side capabilities, as well as extending capability protections to privileged kernel data.

References

[1] S. Yadalam, C. Alverti, V. Karakostas, J. Gandhi, and M. Swift, "Bypassd: Enabling fast userspace access to shared ssds," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2024, pp. 35–51.

[2] D. S. Ritchie and G. W. Neufeld, "User level IPC and device management in the raven kernel," in *USENIX Microkernels and Other Architectures Symposium (USENIX Microkernels and Other Architectures Symposium)*. San Diego, CA: USENIX Association, Sep. 1993. [Online]. Available: https://www.usenix.org/conference/usenix-microkernels-and-other-architectures-symposium/user-level-ipc-and-device

[3] D. R. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel: an operating system architecture for application-level resource management," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, p. 251–266, Dec. 1995. [Online]. Available: https://doi.org/10.1145/224057.224076

[4] M. Jasny, M. El-Hindi, T. Ziegler, and C. Binnig, "A wake-up call for kernel-bypass on modern hardware," in *Proceedings of the 21st International Workshop on Data Management on New Hardware*, ser. DaMoN '25. New York, NY, USA: Association for Computing Machinery, 2025. [Online]. Available: https://doi.org/10.1145/3736227.3736235

[5] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, pp. 1471–1480, 2012.

[6] I. Xilinx, *ef_vi User Guide*, issue 10 ed., https://www.amd.com/content/dam/amd/en/support/downloads/solarflare/onload/enterprise-onload/SF-114063-CD-10_ef_vi_User_Guide.pdf, Xilinx, Inc., Sep. 2019, last Revised: September 2019; Accessed: 2025-12-10. [Online]. Available: https://www.amd.com/content/dam/amd/en/support/downloads/solarflare/onload/enterprise-onload/SF-114063-CD-10_ef_vi_User_Guide.pdf

[7] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 20–37.

[8] I. NVM Express, "Nvme base specification," 2019, submission and completion queues use descriptor rings for command processing, analogous to NIC TX/RX rings. [Online]. Available: https://nvmexpress.org/specifications/

[9] Intel, "Spdk nvme driver," Intel SPDK Project, 2024, nVMe poll-mode driver uses submission/completion queue pairs with descriptor rings, extending DPDK NIC patterns to storage. [Online]. Available: https://spdk.io/doc/nvme.html

[10] *VFIO - "Virtual Function I/O"*, The Linux Kernel Documentation, 2023, pCIe devices (NICs, storage, GPUs) expose uniform doorbells and descriptor rings for user-space DMA via mediated device model. [Online]. Available: https://docs.kernel.org/driver-api/vfio.html

[11] "Amd sev-tio: Trusted i/o for secure encrypted virtualization," Advanced Micro Devices, Inc., Tech. Rep., 2023, whitepaper. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/developer/sev-tio-whitepaper.pdf

[12] L. Guo and F. X. Lin, "Minimum viable device drivers for arm trustzone," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 300–316.

[13] H. Liang, M. Li, Y. Chen, L. Jiang, Z. Xie, and T. Yang, "Establishing trusted i/o paths for sgx client systems with aurora," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1589–1600, 2020.

[14] S. Weiser and M. Werner, "Sgxio: Generic trusted i/o path for intel sgx," in *Proceedings of the seventh ACM on conference on data and application security and privacy*, 2017, pp. 261–268.

[15] S. Boyd-Wickizer and N. Zeldovich, "Tolerating malicious device drivers in linux," in *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[16] DPDK Project, "Dpdk – the open source data plane development kit," https://www.dpdk.org, 2025, accessed: 2025-12-06.

[17] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "Spdk: A development kit to build

high performance storage applications," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017, pp. 154–161.

[18] L. Rizzo, "netmap: a novel framework for fast packet i/o," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 101–112.

[19] S. M. Seefried, P. G. Neumann, R. N. M. Watson, J. Anderson, A. Richardson, M. D. Richardson, D. Chisnall, M. Roe, A. W. Appel, and R. N. M. Watson, "Cheribsd: A research fork of freebsd," in *AsiaBSDCon 2016*, 2016, conference paper discussing CHERI-enabled FreeBSD extensions and challenges. [Online]. Available: https://papers.freebsd.org/2016/asiabsdcon/brooks-cheri/

[20] R. Grisenthwaite, G. Barnes, R. N. M. Watson, S. W. Moore, P. Sewell, and J. Woodruff, "The arm morello evaluation platform—validating cheri-based security in a high-performance system," *IEEE Micro*, vol. 43, no. 3, pp. 50–57, 2023.

[21] D. Ferraro, A. Bastoni, A. Zuepke, and A. Marongiu, "Enabling security on the edge: A cheri compartmentalized network stack," in *2025 Design, Automation & Test in Europe Conference (DATE)*, 2025, pp. 1–7.

[22] P. Metzger, A. T. Markettos, E. T. Napierała, M. Naylor, R. N. M. Watson, and T. M. Jones, "Deprivileging low-level gpu drivers efficiently with user-space processes and cheri compartments," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 2699–2713. [Online]. Available: https://doi.org/10.1145/3719027.3765036

[23] N. Gordon, "Secure i/o on trusted platforms with lightweight kernels." January 2025. [Online]. Available: https://d-scholarship.pitt.edu/46961/

[24] Advanced Micro Devices, Inc., *Solarflare Server Adapter User Guide*, https://www.amd.com/content/dam/amd/en/support/downloads/solarflare/drivers-software/SF-103837-CD-28_Solarflare_Server_Adapter_User_Guide.pdf, AMD, sF-103837-CD-28. Accessed 2026-07-20.

[25] A. Dunkels, "Design and implementation of the lwip tcp/ip stack," *Swedish Institute of Computer Science*, vol. 2, no. 77, 2001.

[26] B. Fitzpatrick, "Memcached: A distributed memory object caching system," https://memcached.org/, 2003, originally developed at Danga Interactive.

[27] Tencent, "F-stack: Userspace network development kit," https://github.com/F-Stack/f-stack, accessed: 2025-12-08.

[28] K. Nienhuis, A. Joannou, T. Bauereiss, A. Fox, M. Roe, B. Campbell, M. Naylor, R. M. Norton, S. W. Moore, P. G. Neumann *et al.*, "Rigorous engineering for hardware security: Formal modelling and proof in the cheri design and implementation process," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1003–1020.

[29] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: a highly scalable user-level tcp stack for multicore systems," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. USA: USENIX Association, 2014, p. 489–502.

[30] H.-y. Kim and S. Rixner, "Performance characterization of the freebsd network stack," *Computer Science Department, Rice University*, 2005.

[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[32] X. Sun, J. Singer, and Z. Wang, "Sweet or sour CHERI: Performance characterization of the Arm Morello platform," in *Proceedings of the 2025 ACM/IEEE International Symposium on Computer Architecture*, 2025. [Online]. Available: https://eprints.gla.ac.uk/362548/

[33] The Linux Kernel Organization, "mlx4 driver directory," https://github.com/torvalds/linux/tree/master/drivers/net/ethernet/mellanox/mlx4, accessed 2026-05-12.