Deprivileging Low-Level GPU Drivers Efficiently with User-Space Processes and CHERI Compartments

Paul Metzger
University of Cambridge
Cambridge, United Kingdom
paul.metzger@cl.cam.ac.uk

A. Theodore Markettos
University of Cambridge
Cambridge, United Kingdom
theo.markettos@cl.cam.ac.uk

Edward Tomasz Napierała
University of Cambridge
Cambridge, United Kingdom
en322@cl.cam.ac.uk

Matthew Naylor
University of Cambridge
Cambridge, United Kingdom
matthew.naylor@cl.cam.ac.uk

Robert N. M. Watson
University of Cambridge
Cambridge, United Kingdom
robert.watson@cl.cam.ac.uk

Timothy M. Jones
University of Cambridge
Cambridge, United Kingdom
timothy.jones@cl.cam.ac.uk

Abstract

Device drivers are a prominent source of operating system bugs and vulnerabilities, due to market pressures on hardware vendors and access to privileged system resources. OSes increasingly deprivilege drivers by moving them out of the kernel into user space, but this is widely understood to come with significant overhead. The perfect storm concerns GPU drivers, which are very large, complex and yet highly performance-sensitive. For performance reasons, large parts of these drivers run with full kernel privileges on major OSes.

We deprivilege a GPU driver by moving it to user space. To avoid context-switching latency we run interrupt handlers inside eBPF sandboxes. Additionally, we take away the ability of the GPU driver to manage its own page tables and move this into an OS-vendor-vetted component. We create two variants. Firstly, a microkernel-inspired implementation which runs the driver in a standard Unix process. Secondly, we move the driver into a CHERI compartment. CHERI allows isolation of distrusting code in sandboxes without needing MMU-based separation. Compartments safely co-exist within an address space, but efficiently share data by passing CHERI capabilities between each other, and incur reduced context-switching costs. To do this we use 'co-located processes', an existing framework which allows us to run graphics drivers and their applications as separate OS processes in a shared address space.

Microkernel-like user-space drivers are still often believed to have high overheads, yet our Unix process-based implementation increases execution time on average by only 7.9% (geometric mean of the benchmark suite; max. 48.2%, min. 0.1%) for GPGPU and by 5.5% (max. 12.6%, min. -0.2%) for graphics workloads, while providing major security benefits. Despite these low costs, isolating processes with CHERI compartments, instead of address spaces, reduces average overheads to 6% (max. 36.6%, min. -0.2%) and 5% (max. 11.2%, min. 0.01%) respectively.

CCS Concepts

• **Security and privacy → Operating systems security**.

CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765036

Keywords

Security; User-space driver; GPU; CHERI

ACM Reference Format:
Paul Metzger, A. Theodore Markettos, Edward Tomasz Napierała, Matthew Naylor, Robert N. M. Watson, and Timothy M. Jones. 2025. 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 (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3719027.3765036

1 Introduction

In recent years, graphics processor (GPU) vendors have reported a seemingly endless tide of GPU-driver vulnerabilities [6, 31, 48, 68, 72, 89]. GPU drivers consist of a low-level operating system (OS) kernel component and a user-space component, as shown in Figure 1. The low-level kernel components are untrustworthy for multiple reasons. They are known to be prone to vulnerabilities due to their size and complexity. For example, one of the Linux kernel GPU drivers consists of more than three million lines of code [51, 54]. Low-level GPU drivers are often closed source, which means independent security audits are infeasible. Windows applies some basic quality control via static analysis [7] and driver signing [62], which provide a low hurdle to clear for driver attacks [34, 35]. Chipset security updates for some devices, for example smartphones, are provided only for a short timespan and sometimes with considerable delays [1]. At the same time, low-level GPU drivers are deployed on millions of devices, including safety- and security-critical systems like phones, cars and supercomputers [4, 17, 88]. Google indicates that four out of nine Android privilege-escalation zero-days recorded in 2023 were in GPU drivers [86]. Despite this, low-level GPU drivers have OS kernel privileges on all major OSes. It is desirable to deprivilege them to thwart attacks that aim to gain OS kernel privilege. This is a reason microkernels place drivers in user-space processes. Sadly, these are sometimes still believed to have high overheads [14, 38, 59], which is in conflict with the performance-centred ethos of the GPU market, where even a few percent difference is newsworthy [47, 92].

We move a low-level GPU driver from the kernel to user space with low performance costs, and investigate two designs over the baseline kernel driver (see Table 1):

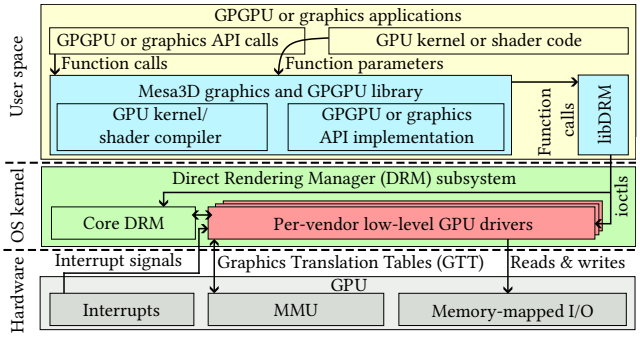


Figure 1: Overview of the Linux and CheriBSD graphics stack. Applications (yellow boxes) link with the Mesa3D library (blue) which implements APIs such as OpenGL or OpenCL, compiles the application's GPU code (*shaders*) and calls the Direct Rendering Manager (DRM) in the OS kernel via *ioctl* system calls. DRM combines a generic part (Core DRM, green) and vendor-provided device-specific low-level drivers (red). The low-level driver includes interrupt handlers, GPU hardware control via memory-mapped I/O (MMIO) and code to manage the GPU's MMU. In this work, we deprivilege the low-level driver (red) by moving it to user space.

Unix Process Driver. With this design GPU drivers run in standard user-space processes, except latency-sensitive interrupt handlers, which run in hardened eBPF sandboxes. Should attackers gain control over such a driver then they must not have direct access to the GPU page tables, as they could map arbitrary memory into GPU address spaces otherwise. Therefore, we move GPU page-table management code into a separate OS-vendor-vetted kernel component, and deny user-space drivers access to the MMIO interface of the GPU's MMU. To support low-level GPU drivers in user space we add a small generic framework for these to the kernel. For example, when GPU interrupts arrive the framework is responsible for waking up device-specific interrupt handlers in user space and the eBPF sandboxes. In a production deployment this framework would be adapted over time to meet the needs of GPU drivers, as is achieved with Linux's generic Direct Rendering Manager GPU subsystem.

CHERI Compartment Driver. This driver variant builds on top of the *Unix process driver*. To further reduce overheads, we let the GPU driver process and its client processes safely share address spaces with CheriBSD's *co-located processes* framework. *CHERI capabilities* are a hardware-enforced bounded pointer type with strong non-forgeability, access permissions and the principle that rights can only ever be reduced, enabling fine-grained memory protection and sandboxing [97]. Co-located processes spawn with capabilities only to their own memory regions, and can efficiently share memory by sending capabilities over inter-process communication (IPC).

Running distrusting co-located processes in the same address space potentially exposes them to transient-execution attacks. Although microarchitectures normally prevent this between different address spaces, co-located processes do not have protection from speculative control-flow attacks. For example, co-located process A could direct speculative execution of co-located process B into A's code by training the branch-target predictor, because the microarchitecture is unaware this is a security boundary. To mitigate this, we devised a minimally invasive tweak to the architecture to prevent speculative attacks between co-located processes.

Since we wanted to evaluate CHERI compartmentalisation, we chose our baseline as a CHERI-enabled system. Our implementation using standard Unix processes could equally be applied to a conventional system, albeit losing CHERI security benefits such as the memory safety and control-flow integrity. The co-located processes implementation builds on this using the sandboxing primitives that are additionally available using CHERI. At present, the only CHERI system with a functional GPU driver is the experimental Arm Morello platform. Our baseline therefore uses the CheriBSD [18, 19] derivative of FreeBSD with the OS kernel compiled to make every pointer a capability, and the Panfrost driver for Arm Mali GPUs, which comes as a kernel module.

We evaluate the overheads of both user-space driver variants over the Panfrost kernel module of CheriBSD with GPGPU and graphics benchmarks. The overheads with the classic microkernel implementation are negligible for some benchmarks and are on average 7.9% (geometric mean of the benchmark suite, with overheads for individual benchmarks ranging from max. 48.2% to min. 0.1%) for GPGPU and 5.5% (max. 12.6%, min. -0.2%) for graphics benchmarks. The impact of our changes on execution time depend on application characteristics such as the number of GPU kernel submissions and the fraction of time spent in computations. While these overheads are low, co-locating the driver process with the benchmark processes still reduces the average overheads to 6% (max. 36.6%, min. -0.2%) for GPGPU benchmarks and to 5% (max. 11.2%, min. 0.01%) for graphics applications.

This paper makes the following contributions:

- We moved a contemporary low-level driver for Arm Mali GPUs from OS kernel space to user space.
- We built a generic framework that allows low-level GPU drivers to run in user space.
- We show that the overheads of a low-level GPU driver in user space with a microkernel-inspired design are low.
- We show that co-locating the driver and its client applications in the same address space reduces these overheads.
- We show that CHERI can be used for fine-grained access control to a GPU's MMIO interfaces by an untrusted driver.
- We outline a mitigation for transient-execution attacks for isolation of co-located processes, including our driver.

2 Approach

2.1 Threat Model

After discussing the threat model we illustrate the possibilities of an attacker who can insert malicious code into a OS kernel-level GPU driver through a supply-chain attack.

We assume an adversary that seeks to execute arbitrary code with OS kernel privileges by attacking GPU drivers in the kernel. Two attack routes are available. Firstly, the adversary can exploit accidental programming mistakes by non-malicious programmers. For example, improper error checks, type confusions, use-after-frees and other memory-corruption vulnerabilities [41, 70, 71, 78].

Security property		Kernel driver non-CHERI ①	**Kernel driver *purecap* ②**	Unix process driver non-CHERI ③	**Unix process driver *purecap* ④**	**Co-located process driver (*purecap*) ⑤**
Object bound checks	ⓐ		✓		✓	✓
Control-flow integrity	ⓑ		✓		✓	✓
Kernel pointer forgery protection	ⓒ		✓	✓	✓	✓
Kernel private API protection	ⓓ			✓	✓	✓
Kernel private struct protection	ⓔ			✓	✓	✓
Driver API call without page table switch	ⓕ	✓	✓			✓

Table 1: Security properties of different approaches, with those we benchmarked in bold. Windows, macOS and Linux run low-level GPU drivers in-kernel ①. Microkernels run them in user-space processes, as ③. We compared different CHERI *purecap* approaches ②,④,⑤, where every pointer is compiled to be a CHERI capability. Our proposed system is ⑤ with GPU driver and applications co-located.

Secondly, the adversary can insert malicious code into a OS kernel GPU driver through a supply-chain attack before the driver is deployed. For example, by compromising build systems [50, 79] or by inserting malicious code into driver sources [9, 33, 74, 79]. We assume that the attacker can insert code only in OS kernel GPU drivers, which are developed by GPU vendors and are either shipped by OS vendors or as separate software packages by GPU vendors.

Assuming the OS kernel is compiled with all pointers as capabilities, an attacker can only use capabilities that are directly available or capabilities that can be transitively reached through those that are available. This means an attacker cannot access arbitrary OS kernel or user data. However, code that implements system calls can reach the credentials of a calling process. On FreeBSD/CheriBSD, either through the global variable `curthread`, which contains a capability to the thread control block of the calling thread, or function parameters that contain such a capability [49], for example functions that implement ioctls or other system calls. An attacker could add the code of Figure 2 to an ioctl() implementation of a GPU driver with checks that execute the malicious code if certain parameters are passed to the ioctl, which are not passed to the ioctl in normal operation. The code elevates the privileges of a calling process to `root` by overwriting the user ID with zero. OS-internal kernel APIs were never designed to mitigate an attacker running inside the OS kernel, and so moving driver code out of the OS kernel blocks these attacks without having to rearchitect the kernel.

2.2 Protection Model and Motivation

CHERI is a promising mitigation against many memory-safety vulnerabilities [45]. A 2025 survey of driver vulnerabilities on Android [55] classified them into out-of-bound (OOB) read and information disclosure; OOB write; use-after-free (UAF) and double-free;

```
struct proc *p = curthread->td_proc;
p->p_ucred->cr_ruid = 0; // real user id := root
```

Figure 2: Sketch of a backdoor that could be inserted into the `ioctl()` code of a FreeBSD/CheriBSD kernel GPU driver.

and others (uninitialised variable, null-pointer dereference, denial of service). Use of CHERI in the kernel would mitigate the first three classes. Hardware bounds checking prevents OOB read/write, while CHERI with revocation would prevent UAF and double-free.

In this work, we anticipate that some of these attack classes have already been blocked by a baseline implementation of CHERI in the kernel, prompting attackers to adapt in response. For example, through a shift towards supply-chain attacks and remaining vulnerability types. Table 1 provides an overview over the different design points and their security properties. Current systems such as Windows, macOS, Linux and FreeBSD have the low-level GPU driver in the kernel, labelled ① in the table. Any use of CHERI would defeat OOB accesses ⓐ, while checks on code pointers and return addresses would defeat many control-flow attacks ⓑ. Additionally CHERI's pointer provenance would prevent drivers generating and accessing pointers to arbitrary data ⓒ. However, malicious kernel drivers are not prevented from calling private kernel APIs ⓓ and maliciously accessing kernel data structures ⓔ. One response is in-kernel CHERI compartmentalisation but this would be highly invasive, retrofitting security boundaries to many kernel APIs and with limited fault recovery. Driver components could instead move to user space where kernel interfaces are already hardened, but this incurs a context-switching penalty due to MMU costs ⓕ.

To explore trade-offs we compared three design points with all pointers compiled as capabilities: the standard kernel driver ②, a driver moved to user space incurring the MMU cost which communicates via traditional OS primitives ④, and a user-space driver which avoids page-table switching costs by using co-located processes and shares data via capability delegation ⑤.

3 Background

3.1 CHERI

CHERI capabilities define a hardware-enforced pointer type. In application-class systems with 64-bit addressing, capabilities contain a 64-bit integer *address* alongside metadata including the 64-bit upper and lower *bounds* of the memory object, and *permissions* (such as read/write/execute). These are stored in-memory in a 128-bit pointer, by encoding the bounds relative to the address in a compressed form [100]. Such pointers flow between registers and memory in the usual way, and capabilities naturally point to data structures containing other capabilities, just as pointers do in conventional software. Dereferencing of capabilities requires that memory accesses obey the bounds and permissions, otherwise a hardware exception is raised. CHERI is expected to add 2-3% run-time overhead to a production CPU [94].

The CHERI architecture supports the *principle of least privilege*. Software is given the minimal rights it needs and nothing more. Its manifestation is that CHERI-enabled CPUs enforce *monotonicity* properties, in which the bounds and permissions of a capability can

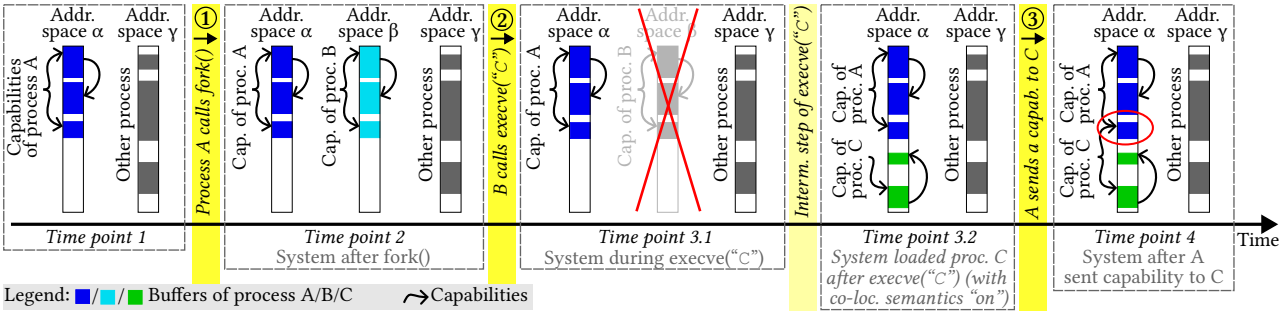


Figure 3: Creation of a co-located process and communication between co-located processes. Co-location semantics of the `execve()` system call was activated before "Time point 1". ① Process B and address space β are created after Process A calls `fork()`. ② After calling `execve("C")` process B and address space β cease to exist. Process C is created and co-located with process A in address space α. The optional co-location semantics of `execve()` always co-locates processes with the parent process. ③ Process A sends a capability to C. Both can now access the encircled memory region. *Co-location does not turn CheriBSD into a single address-space OS.* Other processes and address spaces continue to exist, as indicated by "Address space γ".

only be reduced and never increased. For example, if a capability with certain bounds is passed to a function, there is no way the function can generate a capability with larger bounds in order to access beyond the buffer unless it holds another capability with such rights. To prevent software from arbitrarily modifying capabilities, each capability is protected by an out-of-band validity tag in registers and one bit per 128-bit word of memory: any manipulation of a capability with non-capability instructions will clear the tag bit and it can no longer be dereferenced.

Arm designed the Morello prototype, a modern server CPU with CHERI extensions, and other industrial designs are appearing [2, 3, 27, 37]. CheriBSD and CHERI-LLVM modify the OS and compiler so that the FreeBSD kernel and C/C++ applications can be compiled with capability protection instead of integer memory references.

On current systems, an attacker who has achieved arbitrary code execution in the OS kernel can access any memory address in the kernel's address range since addresses are forgeable. This includes sensitive data such as network plaintext, page tables and memory-mapped I/O interfaces of devices. On a CHERI system with an OS kernel that implements pointers as capabilities instead of integers, such an attacker is more constrained because they can only access memory to which they have capabilities. However, an attacker might still be able to access sensitive data structures. In CheriBSD, system-call code, such as ioctl handlers in GPU drivers, can reach a capability to the user and group ID of the calling process, which can be used to escalate privileges (see Section 2.1). Attackers can also call sensitive OS kernel functions and execute privileged instructions. Therefore, we argue to move untrustworthy and complex drivers, like low-level GPU drivers, to user space from which OS kernel data structures and APIs are not directly reachable. The access rights of such drivers could be further constrained with existing sandboxing frameworks like Capsicum and seccomp [22, 93].

3.2 CHERI Compartmentalisation with Co-located Processes

Capabilities with read/write permissions enforce object bounds, and capabilities with execute permissions constrain the valid targets for branch instructions since the CPU cannot run code for which it does not hold an executable capability. Together this enables development of hardware-enforced compartmentalisation. Software can be divided into compartments which can run mutually distrusting code, with minimal rights such that they cannot access data or execute code outside the set of capabilities they are given [96].

CheriBSD offers two general-purpose CHERI compartmentalisation frameworks: library compartmentalisation [30] and processes *co-located* in a shared address space [64]. We use co-location for our CHERI compartment variant of the driver, because running the driver in a process allows it to be shared by multiple client applications. Co-located processes are Unix processes in all respects, except that they share an address space. They are separated from each other by virtue of each co-located process being delegated only the capabilities appropriate to its own address-space mappings, and not those for other processes in the same address space. CheriBSD's co-location framework is designed to lower the overheads of process-based isolation in general, and is not specific to drivers. If the subsystem is present for applications, we demonstrate that additionally user-space GPU drivers can benefit from it.

Figure 3 illustrates co-location with a simplified example. A new `coexecve()` system call co-locates the new process with a process chosen by the caller [65]. Alternatively, a new behaviour of `execve()` can be activated system-wide by `root` on the command line (Figure 3). If switched on, the `execve()` system call spawns a child process in the parent's address space, instead of their own. This eases prototyping as processes are co-located without source-code modification. `fork()`, `rfork()` and `vfork()` are not changed.

CheriBSD's standard implementation of Unix domain sockets prevents passing capabilities by invalidating tag bits in messages. Co-location modifies this by allowing capabilities to pass between co-located processes via a flag which also confirms that sender and receiver are in the same address space. Passing a capability to a memory object this way delegates access rights to the recipient.

A traditional handicap of capability systems is revocation — making sure that nobody retains a capability to an object that has been freed. In CheriBSD, prior work [98, 101] deals with this by quarantining freed objects until a sweeper has scanned memory to clear

any dangling references to them, or by using an architectural barrier to vet capability loads on a JIT basis [25]. Temporal safety using these techniques builds on top of the inherent spatial safety in the CHERI architecture, and is an optional feature in recent CheriBSD. Unfortunately at the time of our work the branch of CheriBSD that supports co-located processes did not support temporal safety. An additional temporal-safety issue comes up with our work since co-located processes are mutually distrusting. A capability shared from one co-located process to another and later freed must be safely revoked, and we must prevent one co-located process calling munmap() on a buffer shared with another. We posit that these are small changes to the code of the revoker to make it co-located-process aware, and would have minimal performance impact over the baseline temporally safe CheriBSD given the infrequency of extra revocations. However we were not able to measure this due to lack of a temporal-safety subsystem in our baseline OS, which is an important task for future work.

CHERI theoretically also allows in-OS-kernel compartmentalisation, isolating different components within a monolithic kernel. This differs from isolation with user-space processes since all components exist in a single flat address space, instead of manipulating the MMU when switching between components and paying the costs of memory-translation churn. However, adding a framework for such compartments to the OS kernel would increase its size and complexity and, hence, its attack surface, which is the opposite of what we want to achieve. Unix processes are a decades old concept with robust implementations. For this reason we choose to build on top of them instead of adding a new and therefore necessarily less-well-understood concept to the kernel. Additionally, processes have a well-understood fault model. Previous work has shown that faulting driver processes can be restarted in some circumstances [28, 40, 44]. However, we leave investigating to what degree this is possible with user-space GPU drivers for future work.

3.3 The Software Graphics Stack

3.3.1 The `ioctl()` *System Call.* Device-specific functionality is implemented by OS kernel drivers with the general-purpose `ioctl()` system call rather than separate system calls. The parameters of this system call are a file descriptor for a device node, a request number and either an `int` or a pointer to a struct that's defined by the driver. The struct is used for input and output data. The request number determines the functionality that will be invoked. This system call is used extensively by GPU drivers in the OS kernel.

3.3.2 GPU Kernels and Shaders. In the context of the graphics stack, *shaders* are programs that run on GPUs and compute the position of vertices of 3D models, the colour or texture of surfaces, and effects such as lighting. General-purpose computations are implemented with *GPU kernels*, which are also called *Compute Shaders* by OpenGL (see Section 3.3.4). We refer to programs that are executed on the GPU as *GPU kernels* in the remainder of this paper. GPU kernels are typically written in specialised languages such as CUDA or the OpenGL Shading Language (GLSL). OpenGL GPU kernels are either shipped as GLSL code or in an intermediate representation, which are JIT compiled. GPU kernels are submitted by CPU code for execution on the GPU. GPU kernel invocations might then be scheduled by a software scheduler before being inserted into a hardware queue. The GPU raises an interrupt when a GPU kernel terminates, either due to successful completion or an error.

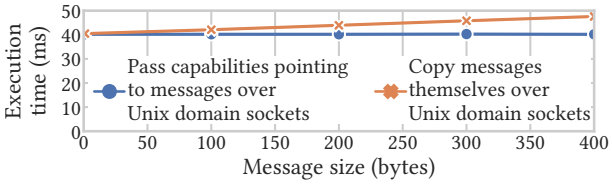


Figure 4: Total execution time of a micro-benchmark consisting of two processes that send a total of 10,000 messages back and forth. Neither variant performs serialisation. The execution time does not increase with the message size if capabilities that point to messages are passed over Unix domain sockets instead of the messages themselves.

3.3.3 GPU Memory Buffers. GPU buffers store compiled GPU kernel programs and input and output data such as matrices or graphics textures. These buffers are either in dedicated GPU memory or in main memory, if GPU kernels can directly access main memory. GPU buffers are mapped into the GPU kernels' address spaces by a GPU's MMU page tables named as *Graphics Translation Tables* (GTT) by Intel [43]. GTTs store translations from graphics virtual addresses issued by GPU kernels on the GPU to physical addresses, or I/O virtual addresses if they are translated again by an IOMMU.

Low-level GPU drivers can separately use the CPU's MMU to map GPU buffers into processes' address spaces to populate GPU buffers with input data or to read output data.

3.3.4 OpenGL. This is a graphics API for interaction with GPUs, used by games and other graphics software [85]. It includes functions to create GPU buffers, compile GPU kernels, pass input data to them, and execute them. While the Vulkan API is increasingly popular, only OpenGL is currently supported by the GPU driver of our evaluation platform. The Mesa3D library is an OpenGL implementation that is commonly used on Linux and on some BSDs.

3.3.5 Direct Rendering Manager (DRM). This OS kernel subsystem consists of a device-specific part that contains all low-level GPU drivers and the so-called *"Core DRM"* part which consists of generic code. DRM was ported from Linux to CheriBSD to support the Panfrost driver [11]. Core DRM includes a scheduler for GPU kernels, synchronisation primitives and GPU buffer-management code.

4 Exploratory Investigation

Moving GPU drivers from the kernel to user space introduces costs of additional context switches, Inter-Process Communication (IPC) and serialisation. Therefore, we carried out small-scale investigations of performance and security of shared address spaces.

4.1 Communication Latency

Additional data copies that are not required with a kernel driver are performed by IPC primitives or during IPC serialisation. If we were to let application and GPU driver processes share an address space and isolate them with CHERI, then we could safely pass a bounded pointer instead. Figure 4 illustrates the performance benefits of

this. One variant uses standard Unix processes with private address spaces, which means that the messages are copied. In the other variant the two processes are co-located in a shared address space and send capabilities that point to the messages without creating copies of the messages. Co-located processes send capabilities to each other over Unix domain sockets, enabling sharing of memory objects and even graphs of memory objects without having to copy them. Due to some limitations with our implementation, we believe there is scope to reduce the 40 ms fixed cost, but the difference due to sending only a small fixed-size capability, instead of potentially much larger messages, would remain.

4.2 Measuring Context-Switching Costs

The costs of context switches have risen because of mitigations against transient-execution attacks. One mitigation technique is to invalidate the branch-predictor state during context switches. This way, malicious processes cannot influence the output of the branch-direction predictor or the branch-target buffer (BTB) during the execution of potential victim processes. The effects of this on Morello can be seen in Figure 5. The number of mispredicted branches and misspeculated instructions is significantly higher if the address-space identifier (ASID) is changed during a context switch. This indicates that Morello's microarchitecture invalidates the branch-predictor state when the ASID changes. The costs are application dependent and here the cycle count is 12% higher with separate address spaces (dark red bars), and so separate ASIDs, than with shared address spaces and shared ASIDs (green bars) (see Figure 5).

4.3 Improved Transient-Execution Mitigation for Co-located Processes

Branch-predictor invalidation prevents transient-execution attacks but is expensive since all the branch-predictor history is lost. We propose an alternative approach via a small addition to the experimental Morello architecture to allow safe speculation more efficiently in co-located address spaces.

Control flow on high-performance processors is divided into two phases. First, the processor *speculatively executes* operations that are not yet confirmed. Later, the operations may be confirmed and *committed* to the CPU state. The former need not follow the architectural model as long as the latter does. We must however not allow speculation to breach security boundaries. For example the branch predictor should not allow speculation into code we do not have rights to execute. Non-CHERI processors do not have bounds for committed execution, only the process model via the MMU. Transient-execution mitigations prevent speculation out of the process, and BTB flushes on ASID changes ensure no aliasing of speculative data from one process to another.

In CHERI architectures, the Program Counter Capability (PCC) augments the PC with *execution bounds* for committed instructions, which must lie within the PCC bounds in order to commit their results. We can conceptualise similarly the *speculation bounds* to describe which instructions may be safely speculatively executed in advance of being committed, and consider how to define them. In Morello's experimental microarchitecture, as we understand, speculation is not bounded by PCC — i.e. it is possible to speculate

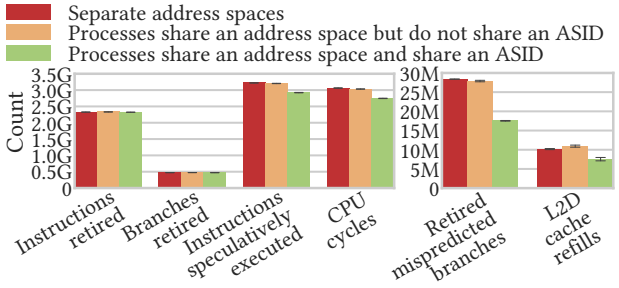


Figure 5: The number of mispredicted branches and misspeculated instructions is higher with per-process ASIDs (red) than with shared ASIDs (green). These measurements indicate that Morello invalidates the branch-predictor state when the ASID changes during a context switch. The measurements are taken with a vector-addition micro-benchmark implemented with OpenGL Compute Shaders. Two processes are launched: our GPU driver and the micro-benchmark. If co-located, they can share an ASID. The number of mispredicted branches and misspeculated instructions is higher with separate ASIDs. This negatively impacts the cycle count and has secondary effects like cache pollution caused by misspeculated instructions. Error bars are 99% confidence intervals.

branches outside the bounds of the current compartment. Therefore when changing between co-located processes we must either change the ASID without modifying the page mapping (paying the BTB flush cost unnecessarily), or not, allowing unsafe speculation.

An obvious option would be to use the PCC bounds as speculation bounds; we are not allowed to speculate outside the current committed execution context. This is a strict policy, but may limit performance. For example, we might launch a leaf function with PCC bounds only for that function, which means we cannot speculate any instructions outside that function.

We propose decoupling the execution bounds and the speculation bounds. An additional privileged control register, the Speculative PC Bounds for EL0 (SPCBEL0) allows the OS to explicitly set the bounds in which user-mode (EL0) speculation is allowed. This provides the microarchitect with metadata of the security context which enables them to implement safer speculation. SPCBEL0 allows the OS context-switching code or similar runtime environment to choose any granularity between the current PCC and the full Unix process. For example, as long as we prevent interleaving of co-located processes in the virtual address space, the runtime may choose the speculative bounds to the whole of a co-located process. Code running there would be unable to speculate into another co-located process, while a domain transition would switch SPCBEL0 to the bounds of the new co-located process.

Allowing control of SPCBEL0 permits the runtime to adjust the speculation security/performance trade-off according to design goals. The architecture may also allow tying SPCBEL0 to PCC for the tightest constraint, to avoid regularly having to manipulate SPCBEL0. The microarchitecture enforces that any branch outside these bounds is never speculated. The OS context-switching code is configured to update the Speculation PC Bounds when switching from one co-located process to another. Additionally, at the

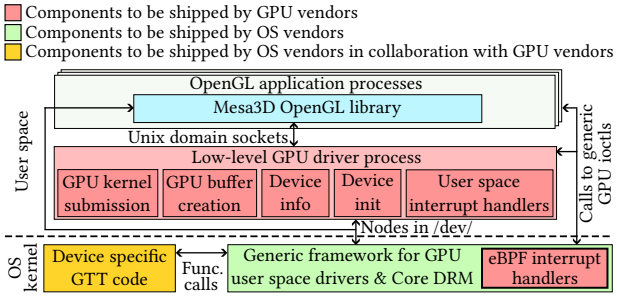


Figure 6: Our implementation and the parties that would ship the components. Our driver can be configured to either run co-located with its client applications in the same address space or as Unix process with a private address space. Each use Unix domain sockets. Driver and clients exchange capabilities pointing to messages in the co-located configuration. The eBPF interpreter is hardened.

point of changing SPCBEL0 the pipeline is flushed to ensure any uncommitted instructions take heed of the new value.

This causes speculative execution to be bounded to the current in-address-space co-located process, rather than the set of MMU mappings denoted by the ASID. Inside the bounds, the common case, speculation proceeds as normal. Execution outside the bounds can proceed without speculation, allowing debuggers and similar tools to still function albeit at reduced speed. Microarchitecturally, when loading from the BTB we need to do an additional bounds check on whether the target is within the SPCBEL0 bounds and if not make the value unavailable. This check can happen in parallel with the PCC bounds check so as not to impact critical-path timing.

We cannot add this proposed register to the Morello silicon design (as it is proprietary to Arm and would cost many millions of dollars to respin), which limits the scope of our evaluation. However our model suggests we can potentially mitigate the cost of BTB flushing with the penalty of just a pipeline flush per context switch. We have simulated it by including an Instruction Synchronisation Barrier (ISB) in our benchmarks to effect a pipeline flush.

5 Design and Implementation

We move the Panfrost GPU driver out of the CheriBSD kernel into a standard Unix process and, building on that, we create a second driver variant with CheriBSD's framework for co-located processes. Both drivers have the same design and implementation, the only difference being that the co-located driver and applications that use it securely share a common virtual address space (i.e. a shared page-table map) by being isolated from each other with CHERI, instead of separate address spaces (see Section 3.2). Co-located processes, including the driver and applications, retain the attributes of distinct Unix processes (such as metadata, user/group, etc), but are launched so that they share an address space and page-table map.

With both driver variants, applications and benchmarks run unmodified, and use the existing user-space graphics stack (Mesa3D). They utilise an `LD_PRELOAD` shim to intercept system calls from Mesa3D and redirect them to the driver. The user-space GPU driver contains low-level GPU functions including buffer and GPU kernel management, except for GPU MMU code, which we handle separately. To provide the necessary support for such drivers we design and build a generic OS kernel framework for GPU user-space drivers. Figure 6 provides an overview over the graphics stack with our driver. The following sections describe the implementation in detail, including the components shown in Figure 6.

5.1 GPU Driver Process

The code moved to user space consists of a *GPU interrupt handler*, *device initialisation* code, ioctls for *GPU kernel submission*, *GPU buffer creation* and *device-information queries* (see Figure 6). Functions that implement the ioctls in the original driver are invoked through messages sent over Unix domain sockets. The messages contain the ioctl parameters as in the original implementation; specifically the ioctl request number that indicates the function to be executed and the ioctl argument struct which contains ioctl-specific input and output data. Our `LD_PRELOAD` shim intercepts these ioctls and forwards them to functions that handle the communication with the driver process instead. A further message is used to pass any output data, including return values, to the client application. The OpenGL library calls Core DRM ioctls directly with nodes in `/dev/dri/`, as with the original driver.

The contents of GPU buffers are not transferred over the ioctl interface of the Panfrost kernel module or, in the case of our driver, over messages. With the kernel module and our driver, GPU buffers can be mapped into the address space of the application where the OpenGL library can directly access them.

The *GPU kernel submission* function enqueues GPU kernel jobs into a generic software scheduler (see Section 5.3). The scheduler enqueues a GPU kernel into the GPU's hardware queue through a function that has knowledge of the GPU's memory-mapped I/O (MMIO) interface. The *GPU buffer-creation* code computes buffer alignments and rounds buffer sizes to Panfrost's preferred 2MiB regions. Examples for *device-information* queries are product number and shader-core count. The *GPU kernel interrupt handler* is invoked when a GPU kernel completes execution or if a fault occurs during GPU kernel execution. The interrupt handler in user space is woken up by our generic framework, discussed in Section 5.3. *Device-initialisation* code is executed once when the driver is started. For example, it checks if the GPU model has known issues and configures the hardware accordingly.

5.2 Graphics Translation Tables

GTTs are managed by low-level GPU drivers. However we cannot allow an untrustworthy driver free access to them, as it could make arbitrary GPU mappings of sensitive data in other processes and use a GPU kernel to copy it to an exfiltration process. Therefore, low-level user-space GPU drivers must not have direct control over the GPU page tables. We propose that OS vendors ship and vet this code and that this code is provided to them by the GPU vendors. Since page-table manipulation is a small part of a GPU driver, this amounts to a limited vetted TCB in the OS kernel.

A malicious driver must not be able to map arbitrary parts of main memory into a GPU address space through calls to such a GTT component. This type of attack is not possible with ours. Only two functions create new entries in GPU page tables: the page-fault

interrupt handler and a function that creates all GPU page-table mappings for a GPU buffer ahead of time so that no page faults occur when a program on the GPU uses the buffer. Both functions create page-table entries only for page frames allocated to back GPU buffers and can therefore not be misused to map arbitrary main memory into a GPU address space. Additionally, the GTT component cannot be directly called into by user-space code. Only our generic framework calls functions in the GTT component.

An alternative might use an IOMMU to allow the OS kernel to enforce memory-access protection downstream of an attacker-controlled GTT. However not all systems have IOMMUs, or the IOMMU may be controlled by a hypervisor that prevents the guest OS kernel using it. Our GTT management operates independently from any IOMMU layer that may or may not be available.

5.3 Generic Framework for GPU User-Space Drivers

This OS kernel subsystem manages interrupt handlers and allows reuse of generic OS kernel code for GPU drivers. While this adds a small amount of additional security-critical code to the kernel, it is independently auditable and shared between drivers so allows removal of much more. We added ioctls so that user-space GPU drivers can use Core DRM functionality that would otherwise not be available through already-existing ioctls.

Upcalls to User Space. Core DRM components call low-level GPU drivers via device-specific function pointers. This is not possible with user-space drivers because OS kernel code cannot directly call functions in user space. Therefore, we built an upcall mechanism with per-function `/dev` device nodes. User-space functions that, on a conceptual level, can be called from the OS kernel, run in their own threads, and block in the `read` system call on their `/dev` node until OS kernel code invokes their execution by waking them up. We use the `read` system call to pass input data to the function in user space. Outputs are returned to the OS kernel through per-function ioctls. The same mechanism is used for user-space interrupt handlers.

Interrupt Handlers. Interrupts are high priority and need timely handling. Current practice executes interrupt handlers in the OS kernel because it is too slow to dispatch them to user space. Therefore, we split interrupt handlers and make use of eBPF, a sandboxing technique popularised by the Linux kernel that lets users attach their own code to OS kernel hooks [21]. eBPF only expresses a limited set of operations and is not Turing complete. eBPF programs are analysed statically for security issues by a verifier before they are either just-in-time compiled or executed by a bytecode interpreter. For example, eBPF programs must not use uninitialised variables. Our Generic Framework does initial acknowledgment of the interrupt in an eBPF sandbox and records the nature of the interrupt and then dispatches the remaining handling to user space via upcalls. The GPU kernel interrupt handler of our user-space driver is implemented this way. We add an eBPF bytecode interpreter, heavily based on uBPF [8], which is also used by Microsoft in the eBPF-for-Windows project [60]. Our interpreter is considerably more constrained than the Linux eBPF implementation to harden it against attacks. We use CHERI to bound the memory area that an eBPF bytecode interpreter can access to the memory-mapped I/O interface of the device. eBPF *Maps* and eBPF *Helper Functions* are not supported. Maps are buffers that are shared by user-space code and eBPF programs and helper functions are a fixed set of compiled functions. Our interpreter allows bytecode programs to return an integer to the code that started them. The only exception are interrupt handlers that are associated with the GTT. These are part of the GTT component and are executed natively.

ioctl System Calls. Low-level GPU drivers in the OS kernel call Core DRM code through function calls. This is not possible with low-level user-space drivers because user-space code cannot directly call OS kernel functions. Therefore, our Generic Framework offers DRM functionality to user space through ioctls. We hardened this ioctl interface against attacks. We compiled a subset of the Common Weakness Enumeration (CWE) with weaknesses that are relevant to our API. We then designed the ioctl interface to avoid the weaknesses in this list. For example, some Core DRM functions that are called by the original Panfrost driver return pointers. Our ioctl interface never exposes OS kernel addresses to user space because this could be used to circumvent Kernel Address-Space Layout Randomisation (KASLR), if enabled. Instead our interface passes unique per-process integer identifiers to user space to refer to OS kernel objects. The ioctls fall into four categories:

Initialisation. This ioctl sets up interrupt handlers. The driver in user space configures which interrupts have a user-space handler in addition to the eBPF handler and sets the eBPF interrupt-handler byte code. It also sets the names of the `/dev` device nodes that are created for upcalls to user-space interrupt handlers.

Synchronisation. DMA fences are a synchronisation primitive to coordinate accesses to buffers that are shared between multiple devices or between a device and the CPU. DMA fences can be one of *signalled*, *not signalled* or in an error state. In the context of DRM, they are, for example, attached to GPU buffer objects and so-called *sync objects*. Sync objects let user-space code wait on DMA fences and sync object handles can be sent to other processes. For example, they can be used to wait for a GPU kernel to run to completion. Using code from Core DRM we added ioctls to create new DMA fences, signal fences, acquire references to DMA fences that are attached to DRM objects and to attach DMA fences to DRM objects. DRM objects for which we have added these ioctls include GPU buffers and sync objects.

GPU Kernel Submission. DRM includes a GPU kernel scheduler including run queues, dependency tracking and priority management. Some modern GPUs use a firmware scheduler with which the user-space driver could submit jobs directly over the MMIO interface. The GPU of our evaluation platform does not have a firmware scheduler. Therefore, we need to do an extra roundtrip to the DRM scheduler in the OS kernel and back to user space. We added ioctls to submit GPU kernels to the DRM scheduler.

GPU Buffers. Multiple DRM components are concerned with GPU buffers. For example, DRM has abstractions for GPU buffers, a facility to create user-space handles for GPU buffers, and GPU address-space management components. We offer ioctls to: create a GPU buffer and return a user-space handle; map GPU buffers into the address space of the calling process so that programs on the CPU can write to and read from these buffers; fill a GPU buffer with zeroes with the CPU; reserve buffers for exclusive access and to release them; wait for exclusive access to be released.

5.4 GPU MMIO Interface

Memory-Mapped I/O (MMIO) serves as the control interface for the hardware. CPU reads and writes into the GPU's MMIO space serve to configure and control the GPU. Often MMIO interfaces are built for the silicon designer's convenience rather than in a security-conscious way. For example, on early NVIDIA GPUs there were multiple 'indirect' ways to access registers and video RAM through different apertures to allow access from legacy operating systems [23] — these could allow an attacker a backdoor even if the direct route was blocked. Similarly attackers found an exposed debug MMIO interface in Apple GPUs [52]. Hence sensitive registers of the MMIO interface cannot be exposed to the user-space driver.

Our design approaches this as follows. First, we identify which regions of MMIO registers are 'safe' for the driver and which are unsafe. Ideally this would be done by the silicon vendor and stored in the Device Tree as an authoritative record. Next, we map the GPU's MMIO space into the user-space driver process. However, this driver requires capabilities to be able to access this region. The Generic Framework gives it a list of capabilities to regions of safe registers which it needs to do its work. Other registers are completely walled off from the driver. For example, there would be no need to access NVIDIA's legacy apertures. If there are any registers that are needed but aren't safe to be delegated to the driver in this way, they require an additional function in the privileged GTT-management component to offer an API by which the driver can request the privileged component make changes on its behalf. The byte-granularity of capabilities allows fine subdivision of MMIO registers, minimising such points of conflict, but this mechanism allows safe handling of the remainder. A roundtrip cost would be incurred similar to the cost we incur when setting up GTT entries.

With our specific prototype, making this division is difficult because we do not have documentation for the Mali GPU. The Panfrost driver authors reverse engineered the GPU and it only has a cryptic header file of register names with no descriptions of their functions. Thus we have no information about side-effects or indirect access. We can however broadly separate registers into setup, feature identification (e.g. product ID), performance counters, power management, GPU kernel management and GTT management. Since the driver has sole use of the GPU, the first five categories are those by which the driver can control the GPU but does not gain any privilege to attack the rest of the system. However the GTT page-table base registers are examples of registers that the driver cannot have access to, and these lie at the top of the GPU's MMIO space. To restrict access, at startup the Generic Framework maps the MMIO into the driver address space and generates a capability whose bounds include most of the MMIO registers but excludes the GTT registers. This is then passed to the user-space driver. During execution, the requests we make to set up GTT entries are examples of asking the privileged component to intermediate on the driver's behalf, and these are captured in our benchmarks.

5.5 Multiple-Client Applications

Both driver variants support dozens of client applications running concurrently. We run the Xorg display server, on top of which runs the KDE desktop environment including graphical widgets, then the user can launch multitasking graphical applications as they would in a standard desktop session. In the co-location case all of these applications that depend on Xorg are co-located inside a single shared address space. Non-Xorg applications (e.g. SSH sessions) run in separate address spaces as normal. To achieve this, dependants of Xorg are started as processes co-located with Xorg and the driver.

In the baseline setup with the kernel module, only a graphics application itself calls into the kernel (i.e. into DRM or the kernel module) for its graphics operations. This allows kernel-side metadata to be conveniently associated with the calling process. However, to perform the graphics operations of a single client with the user-space driver either the client itself or the user-space driver call into the kernel (i.e. into DRM or the Generic Framework). This means that kernel-side metadata for this client needs to be associated with two processes instead of one: the graphics application and the user-space driver. Regardless of whether the kernel module or our user-space driver is used, graphics applications open a DRM device file in `/dev/dri/`. This file descriptor is used when calling into the kernel for graphics operations and is there associated with the metadata. The first time a graphics application invokes the user-space driver, the file descriptor is sent by the `LD_PRELOAD` shim to the driver process with Unix domain socket control messages. From then on, this descriptor is used by the graphics application and by the driver when it serves requests of this application. The driver receives multiple such file descriptors from different clients and, therefore, associates descriptors with process IDs (PIDs). For this, we added a flag to Unix domain sockets that causes the OS kernel to insert the PID of the sending process inline into messages.

The driver could be extended in future so that multiple instances of it can run simultaneously in different address spaces. Access to MMIO interfaces and shared memory would need synchronisation primitives. We leave this for future work.

5.6 Generality

All of Windows, macOS/iOS, Linux, Android and FreeBSD use a user-space/kernel split for their GPU drivers and, while platforms differ, the work done by these drivers is similar. We expect our Unix process-based design to be portable to other OSes as it does not use CheriBSD-specific features like co-location. CheriBSD's Panfrost and Core DRM were ported from Linux, and both systems use Mesa, which should simplify a Linux port. Our high-level architecture accords with the current GPU driver architecture on Linux/Android which already provides a generic driver framework as Core DRM, albeit for in-kernel drivers. We have less visibility for closed-source OSes but expect a similar subsystem may be constructed.

Safe co-location requires a CHERI CPU. While currently not widely deployed, a commercial offering of a Linux-capable CHERI CPU has recently emerged [27]. Our design using Unix processes without co-location would apply to non-CHERI systems, but would lack CHERI's memory protection within the driver (e.g. protection from buffer overflows) and fine-grained access control to MMIO registers. Safe MMIO registers that do not share a page with unsafe ones could be mapped into the driver's address space. The mechanism described in Section 5.4 for unsafe registers needed by the driver could be used for the remaining ones. Run-time checks within the in-kernel interrupt handler sandboxes could limit access to specific parts of the GPU's MMIO interface.

Our approach does not depend on the Panfrost driver or Mali-GPU-specific features. While we cannot inspect closed-source drivers, based on open-source Linux ones we expect our approach to be applicable to other GPUs and their potentially larger drivers. Additional code comes from more complex hardware (especially large numbers of register definitions) and more subsystems to manage it, but the shape of the drivers are similar to our evaluation system. They typically also have an MMU, buffer and job-management ioctls, interrupts and an MMIO interface. GPU drivers are typically developed by the hardware vendor who has knowledge of the necessary internals, such as MMIO functionalities.

6 Evaluation

6.1 Experimental Setup

Arm Morello is currently the only CHERI system with a GPU. It is a research prototype with four cores that are based on the Arm Neoverse N1 microarchitecture and the Armv8.2-A architecture but extended with 128-bit capabilities according to the CHERI architectural model [37]. It has an Arm Mali G76 GPU that shares main memory with the CPU. The GPU is unaware of capabilities and clears their tags when writing to memory. The CPU does not support SMT and DVFS is deactivated. A side effect of our upcall mechanism is that the user-space driver makes better use of multi-core CPUs than the original Panfrost driver, because it is split up into multiple threads. Therefore, only a single core is enabled for a fair comparison. We use CheriBSD (22.11) as it is the only stable OS for Morello with a working graphics stack. We use version 13.0.0 of the Arm CHERI Clang compiler [63] and Mesa3D 21.3.8. For our user-space drivers we use the same compiler flags that CheriBSD uses for the Panfrost kernel module. CheriBSD's *'co-located processes'* branch reuses Unix domain-socket control messages to send and receive capabilities — the standard data-transfer path would be slightly more efficient and could have been reused with more engineering. For a fair comparison we also use control messages to transfer data with the 'classic Unix process' variant of the driver.

To collect data with the CHERI-compartment variant of our driver we activate the co-location semantics of `execve()` (discussed in Section 3.2), which, in our experience, does not require application changes. For example, if a shell starts our driver and a graphics application then all three processes are co-located in one address space without modifications. To let graphics applications send capabilities to the co-located graphics driver we only modify the LD_PRELOAD shim (see start of Section 5) that intercepts ioctls.

We use the Phoronix Desktop Graphics benchmark suite and a subset of the widely used Rodinia GPGPU benchmarks [13, 73]. We hand converted the GPGPU benchmarks to OpenGL Compute Shaders because Morello's GPU has no working open-source OpenCL stack. We chose which benchmarks to port based on their submission rates with OpenCL on a system with an Intel i7-5775C CPU and a Nvidia RTX 3060 Ti GPU. These rates are shown in Figure 7. With our user-space port of the driver, we modified code paths for GPU kernel submission by applications to the driver, as well as GPU kernel scheduling and GPU buffer creation. Therefore, the lower the GPU kernel submission rate the lower the expected overheads because the modified code paths are executed less often. We ported the benchmarks in the order shown in Figure 7 starting

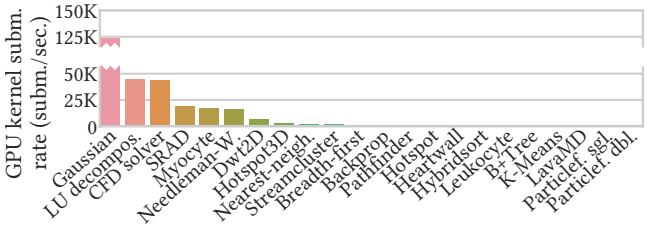


Figure 7: GPU kernel submission rates with the OpenCL Rodinia benchmarks. This data informed which benchmarks we ported to OpenGL Compute Shaders. Some bars are too small to be visible. Each data point is the mean of 20 samples.

with Gaussian until the submission rates of multiple successive benchmarks were so low that the overheads with our user-space drivers over the OS kernel module are negligible. This point is reached with Backprop and Pathfinder (see Section 6.2). To test our conjecture we also ported LavaMD, which is on the far end of the tail in Figure 7. We compiled the GPGPU benchmarks with the `-O3` compiler flag. The Phoronix Desktop Graphics suite contains the Glmark2 benchmarks, three games (OpenArena, Tesseract and Xonotic) in demo mode without user input, the scientific application ParaView and two Unigine game-engine demos.

Some graphics benchmarks were however unavailable. The Unigine benchmarks require an OpenGL extension not supported by Panfrost. Traces of Xonotic at high graphics settings and ParaView Manysphere crash the baseline Panfrost kernel module. Glmark2's Terrain benchmark crashes our driver with its standard settings and to avoid this we turned off its "tilt-shift" post-processing.

Precompiled aarch64 binaries are not suitable because secure co-location requires that all pointers are compiled to capabilities (*purecap binaries*, which may need code modification [95]). Compiling the relatively simple GPGPU benchmarks and Glmark2 to purecap binaries is fine. However, as purecap versions of the many dependencies of the more complex games and the scientific application in the Desktop Graphics suite are not available, we chose a different approach. We prerecorded OpenGL traces of the Phoronix graphics benchmarks except Glmark2 with the widely used Apitrace tool on an Intel i7-1265U machine with integrated GPU and replayed them on Morello and CheriBSD [16, 26, 39]. We configured the recording machine so that the applications use OpenGL, GLSL versions and OpenGL extensions that are supported by Panfrost as if they were run directly with Panfrost and Morello's GPU. We did not use Xorg during data collection with the graphics benchmarks to reduce noise in the measurements. Apitrace rendered to an off-screen buffer and we compiled Glmark2 with its "drm-gl" build flag that allows it to render on-screen without Xorg.

Morello has a mobile GPU and if necessary we set the resolution of the graphics benchmarks so that they run with at least 30 frames per second (FPS). Some modern games target this frame rate and animations are fluid [58]. The Glmark2 benchmark suite (v2023.01), OpenArena (v0.8.8), ParaView, Tesseract (2014-05-12) and Xonotic (v0.8.6), ran with the resolutions 1920x1080, 800x600, 640x480, 640x480, 800x600, respectively. We could not adjust the settings of ParaView Wavelet Contour in this way and it ran with 5.46FPS with the Panfrost kernel module. We used v5.4.1 of ParaView because

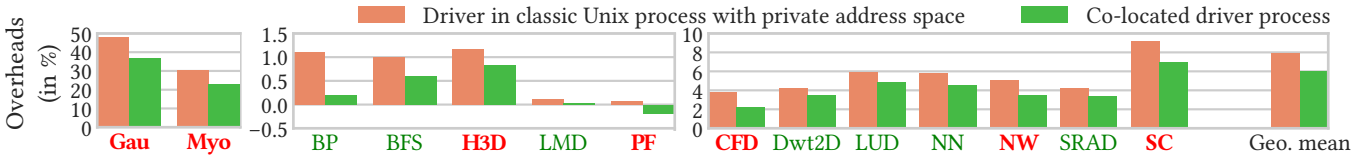


Figure 8: Overheads with our user-space variants of the low-level driver over the OS kernel module with the Rodinia GPGPU benchmarks. The co-located driver process shares an address space with the benchmark applications. Benchmarks with bold red names execute GPU kernels synchronously and benchmarks with green names asynchronously (see Section 6.2). The benchmarks are Gaussian (Gau), Myocyte (Myo), Backprop (BP), Breadth-first search (BFS), Hotspot3D (H3D), LavaMD (LMD), Pathfinder (PF), CFD solver (CFD), 2D discrete wavelet transform (Dwt2D), LU decomposition (LUD), Nearest Neighbour (NN), Needleman-Wunsch (NW) and Streamcluster (SC). Each data point is the mean of 50 samples.

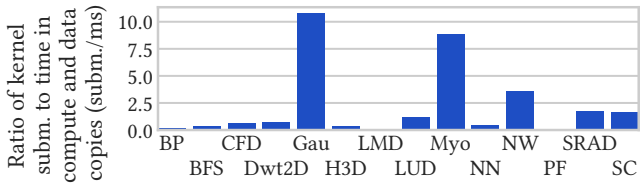


Figure 9: Ratio of GPU kernel submissions to time spent in GPU and CPU computations and copies from and to GPU data buffers. GPU kernel submission counts are measured with OpenCL as described in Section 6.1. This should be the same with our ports as each OpenCL kernel submission is instead an OpenGL submission. Measurements are subject to limitations of the OpenGL driver; GPU kernel execution time was measured by instrumenting the benchmark code. Ratios of Pathfinder and LavaMD are so small that they are not visible. Each data point is the average of ten samples.

the newest version requires a higher OpenGL version than Panfrost supports. When we compiled Apitrace and Glmark2 we used the "release" build type of the CMake and Meson build systems.

We introduce overheads in the GPU buffer-creation code path of the low-level driver, among other places. However, the Mesa3D OpenGL library tries to avoid executing this code path. The library's GPU buffer allocator has a list of unused GPU buffers and tries to serve requests for new buffers with this list first and if that is not possible falls back to the buffer-creation code. Buffer objects that are longer than two seconds in this list are destroyed by the allocator so that the OS kernel can reuse the page frames. We increased this threshold to ten seconds with the graphics benchmarks, to reduce the impact of our changes in the buffer-creation code path. This increased their memory footprint by no more than 2.1%. For a fair comparison, we increased the threshold also when we collected data with the OS kernel module and the graphics benchmarks.

Some benchmarks benefit from the slower GPU kernel submission paths with the user-space driver. We attribute this to a quirk in the GPU buffer allocator mentioned above. Buffers are allocated to GPU kernels when they are submitted by the application and before they are enqueued into the GPU scheduler's work queue. The overheads in the GPU kernel submission path slightly reduce the GPU kernel submission rate, while the rate with which the GPU executes kernels remains the same. This in turn means that fewer GPU buffers are allocated because fewer GPU kernels wait in the GPU scheduler queue. This should decrease the pressure on the GPU's TLBs. Sadly, CheriBSD's Panfrost port does not include performance counters to measure this. This accidental interaction masks the overheads that we would like to measure. Therefore, we avoid this benefiting the user-space drivers by changing the affected GPGPU benchmarks to wait after each GPU kernel submission for it to run to completion, so the number of GPU buffers created does not depend on the length of the submission code paths.

6.2 Overheads With GPGPU Benchmarks

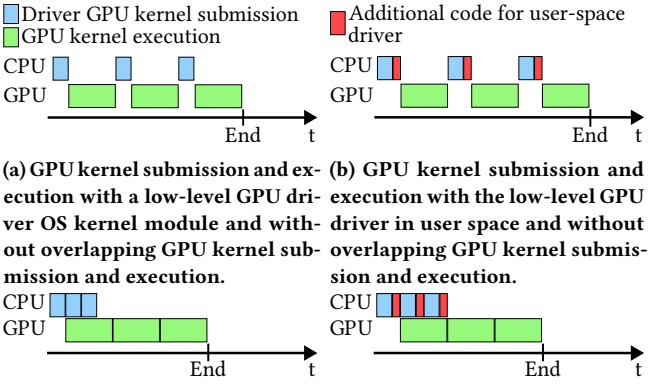
Figure 8 shows overheads with the driver in a classic Unix process and in a process co-located with the benchmark in a shared address space. The geometric mean overheads are 7.9% with the Unix process and 6% with the co-located process. Gaussian and Myocyte are affected the most with over 20% overheads but co-location reduces this significantly. The other benchmarks have overheads below 9% and some less than 2%. The overheads with co-location are consistently lower.

We use simplified models for the execution times t_{KM} and t_{US} of the benchmarks with the Panfrost OS **k**ernel **m**odule and the **u**ser-**s**pace driver to explain why some are more affected than others.

$$t_{KM} \approx s \times t_{SCD} + t_{CGPU} + t_{CCPU} + t_{CGB}$$

$$t_{US} \approx s \times (t_{SCD} + t_{OUS}) + t_{CGPU} + t_{CCPU} + t_{CGB}$$

The execution time with the OS kernel module is approximately the number of GPU kernel **s**ubmissions s times the execution time t_{SCD} spent in the GPU kernel **s**ubmission **c**ode of the **d**river plus the execution time t_{CGPU} spent in **c**omputations on the **GPU**, the execution time t_{CCPU} spent in **c**omputations on the **CPU** and the execution time t_{CGB} spent in **c**opies from and into **G**PU **b**uffers. The formula for the execution time with the user-space drivers captures the **o**verheads of moving the low-level driver to **u**ser **s**pace with the term t_{OUS}. It is evident that the impact t_{OUS} has on the execution time depends on the other terms. The lower t_{CGPU}, t_{CCPU} and t_{CGB} are and the higher s is the higher is the impact of t_{OUS}. This suggests that the higher the ratio $\frac{s}{t_{CCPU}+t_{CGPU}+t_{CGB}}$ of GPU kernel submissions to time spent in computations on the GPU and CPU and in GPU buffer data copies the higher are the overheads in Figure 8. Figure 9 shows these ratios for each benchmark. Gaussian and Myocyte have the highest ratios and are the benchmarks with the highest overheads (see Figure 8). Backprop, Hotspot3D, LavaMD have the lowest ratios and also have the lowest overheads. This



(a) GPU kernel submission and execution with a low-level GPU driver OS kernel module and without overlapping GPU kernel submission and execution.

(b) GPU kernel submission and execution with the low-level GPU driver in user space and without overlapping GPU kernel submission and execution.

(c) Same as in Figure 10a but with execution and submission overlapped. Only the first submission contributes to the execution time.

(d) Same as in Figure 10b but with execution and submission overlapped. Only the first submission contributes to the execution time.

Figure 10: Moving GPU drivers from the OS kernel to user space adds code (red boxes) on the GPU kernel submission code paths (blue boxes), e.g. for IPC. This figure illustrates how this code and GPU kernel execution (green boxes) can be overlapped so that it contributes less to execution time.

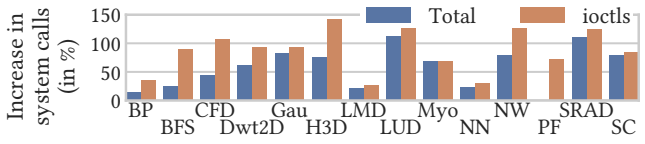


Figure 11: Increase in total system calls and ioctl system calls with the classic Unix process driver over the Panfrost OS kernel module. The increase in system calls with Pathfinder is marginal and thus not visible. We measured the system calls performed by the benchmark application and the user-space driver. Each data point is the average of 20 samples.

suggests that the more time a benchmark spends in the GPU kernel submission code of the GPU driver the higher the overheads of moving the driver to user space.

Some applications can hide the execution time costs of GPU kernel submission code by overlapping kernel execution with it — see Figure 10. The submission of a kernel and the execution of a previously submitted kernel can be overlapped if the output of the previous kernel is only used as input for another GPU kernel and not for computations on the CPU after the kernel submission. For example, an iterative algorithm that uses the output of one iteration as the input of the next iteration can overlap kernel execution and submission. Figure 8 colour codes which benchmarks execute kernels asynchronously. The benchmarks with the highest overheads execute kernels synchronously and do not hide the overheads of kernel submission code and the additional code introduced on the kernel submission path.

Some of our overheads are caused by additional system calls. We moved device-specific low-level GPU driver code out of the OS kernel into user space but left dependencies, like Core DRM, in the kernel (see Section 5.3). The user-space driver interacts with these dependencies through ioctl and other system calls, whereas the kernel module calls into them directly with function calls, which have lower overheads. Figure 11 shows the increase in ioctl and system calls. The overheads in Figure 8 have no clear correlation with by how much the number of system calls made increases. For example, Gaussian has the highest overhead but only the third highest increase in system calls, and Hotspot3D has a negligible overhead but the fifth highest increase in system calls and the highest increase in ioctl calls. As explained above, how our changes in the GPU kernel submission code path (represented by t_{OUS}) impact the overall execution time depends on the ratio between GPU kernel submissions and time spent in data transfers and computations.

6.3 Overheads with Graphics Applications

Figure 12 shows the overheads with the Phoronix Desktop Graphics benchmark suite [73]. The geometric mean overhead with the classic Unix process variant of the user-space driver is 5.5% and with the co-located driver 5%. The overheads are lower with the co-located driver than with the classic Unix process variant for 36 out of 38 benchmarks. All benchmarks run with at least 35 frames per second (FPS), except Paraview wavelet contour, which runs with 5.5 FPS and could not be adjusted to run with higher FPS (see Section 6.1). Most benchmarks run with more than a hundred FPS.

6.4 Code Size

The Panfrost kernel driver in CheriBSD consists of 3209 lines of code (LoC). Our Generic Framework has 2669 LoC, the eBPF interpreter has 1020 LoC, our user-space driver has 2790 LoC and the GTT component has 822 LoC. The co-location code requires 1017 additional or modified LoC. Co-location code and the Generic Framework would be shared by all GPU drivers in a deployment. We applied a uniform code-style (the LLVM style) with `clang-format` to eBPF, our Generic Framework, the GTT component, our user-space driver and Panfrost before counting LoC. Blank lines and comments are not included in the LoC. While other GPU drivers can be more complex they have similar components (see Section 5.6).

7 Related Work

Windows NT 3.1 to 3.51 implemented graphics drivers in user space and its performance problems are well known, not helped by needing up to six changes of CPU mode for every system call [15]. After this salutory experience NT 4 moved graphics drivers into the kernel [75], and popular wisdom has suggested that GPU drivers are too performance critical to run entirely in user space ever since.

Running less critical drivers in user space has been a recurrent theme since NT [12, 29, 53, 84]. No mainstream OS considered GPU drivers due to their complexity and performance requirements. macOS runs some non-GPU drivers in user space using DriverKit [5]. However a hardware vendor complained about lower performance [81]. Inspecting an M1 Mac Mini running macOS 15.0 we found only USB and networking drivers using DriverKit. Windows [61] and Linux run higher-level GPU functions in user space, but low-level drivers are not included. This does not address our threat model as an attacker can still insert malicious code into the remaining kernel component or exploit vulnerabilities in it.

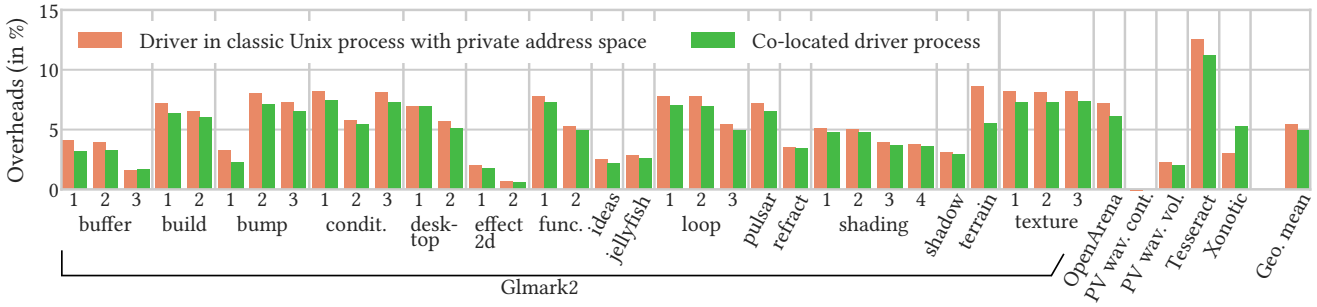


Figure 12: Overheads with both user-space driver variants over the OS kernel module with the Phoronix Desktop Graphics benchmark suite. Glmark2, in its standard configuration (i.e. when no parameters are passed to it), uses some benchmarks multiple times with different parameters. These benchmark instances are numbered in the order in which they are executed by Glmark2. For example, the "buffer" benchmark is executed with three different sets of parameters. The slowdowns with ParaView wavelet contour are so low that they are not visible. Each data point is the mean of at least 15 samples.

User-space drivers are a primary feature of microkernels. Zircon and Genode have user-space GPU drivers [24, 36]. However, both isolate drivers with private address spaces and do not consider CHERI. Neither has publicised overheads over in-kernel designs. Interrupt handlers run in user space and the kernel has no sandboxes for them as we do. Memory-mapped I/O (MMIO) registers are mapped into driver processes, which limits access control to page granularity. In contrast, CHERI allows for finer-grained control.

The Glider technical report [83] reduces the kernel-level GPU driver by moving resource management, such as work submission and buffer management, into a user-space library using User-Mode Linux (UML). The GPU hardware is assigned to only a single application at a time and timesliced between processes, with a context switcher switching MMIO control registers, GPU cache states and potentially MMU/IOMMU configurations. To enforce scheduling decisions and prevent applications from interfering with each others' private GPU buffers, resource isolation must remain in the kernel. We similarly manage GPU page-table isolation in the kernel but compute resource isolation is handled in user space. Since no application has direct access to the GPU, our shared user-space driver allows applications on multiple CPUs to concurrently submit work, which is not possible in Glider without GPU support for hardware virtualisation. For interrupt delivery, we avoid Glider's polling mechanism and instead use upcalls and in-kernel sandboxes.

Other work has moved non-GPU drivers completely or partially into user-space processes. Sawmill Linux runs a Linux file system and part of the network stack in the user space of a L4 microkernel [32]. SUD and Qiang et al. run unmodified drivers in processes by using UML, which runs a second kernel in user space [10, 77]. NetBSD's rump kernels can run drivers and their dependencies in user space [46]. In contrast to works that move an entire driver to user space, Microdrivers and Schwahn et al. leave performance sensitive code in the kernel [29, 84]. With these approaches, drivers are from the kernel's perspective user-space processes, but other isolation techniques exist. For example, LXD, LVD and Ksplit build on top of each other, using Intel VT-x virtualisation extensions to isolate kernel logic [42, 66, 67]. Qubes and VirtuOS isolate kernel components with Xen VMs [69, 82]. Nexus OS monitors interactions between user-space drivers and devices, with the ability to block operations [99]. Nooks uses among others the MMU and function-call interposition to create in-kernel isolation domains [87].

IPC overhead is a key concern for some designs. A number of strategies [59, 90], have considered MMU-based techniques to reduce IPC overhead, or with custom hardware [20]. While others [20, 90] consider networking, they do not cover graphics drivers.

For dedicated use cases, drivers can be subsumed into the application [56, 57, 80] with toolkits such as SPDK [102]. This benefits performance by removing boundaries, but is no good if the device needs to be shared between multiple clients. It is also possible to delegate hardware virtualisation features (e.g. virtual network cards), to specific clients, giving the illusion of full control [76, 91]. Sugar isolates the browser WebGL stack from the kernel, by giving browser processes-exclusive access to a vGPU and running the driver in a library with UML [103]. Such approaches aren't scalable beyond a handful of clients. In our case, we can co-locate the driver with as many clients as needed, including the windowing system.

8 Conclusion

GPU drivers are untrustworthy yet they have OS kernel privileges on a myriad of systems, some security and safety critical. Deprivileging low-level GPU drivers with a microkernel-inspired design prevents attackers from gaining OS kernel privileges. We move a GPU driver from the kernel to user space with classic Unix processes and co-located processes. In lieu of separate address spaces, CHERI can be used to isolate co-located processes from each other, reducing context switching and communication costs.

Some systems of the 90s have given microkernels and microkernel-like designs sadly a reputation for having high overheads. We contribute to a body of evidence that paints a more positive and nuanced picture as our Unix process-based GPU driver introduces low overheads on average, which can be further reduced on CHERI systems, by co-locating the driver with its client applications.

Acknowledgments and Resources

An artifact related to this publication is available in the repository at https://doi.org/10.5281/zenodo.16987522.

This work was supported by the UK EPSRC under the CAPcelerate (EP/V000381/1) and Chrompartments (EP/X015963/1) projects, both part of the Innovate UK project Digital Security by Design (DSbD) and the DSbDtech initiative. This work was supported in part by the DSbD Technology Platform Prototype (105694), the EPSRC CHaOS Grant (EP/V000292/1) and UKRI3001: CHERI Research Centre. This work was also supported in part by Arm and Google.

Distribution Statement A: Approved for public release. Distribution is unlimited. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under contracts HR0011-22-C-0110 ("ETC") and FA8750-24-C-B047 ("DEC"). The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

References

[1] Abbas Acar, Güliz Seray Tuncay, Esteban Luques, Harun Oz, Ahmet Aris, and Selcuk Uluagac. 2024. 50 Shades of Support: A Device-Centric Analysis of Android Security Updates. In *NDSS*.

[2] Thomas Aird, Hesham Almatary, Andres Amaya Garcia, et al. 2025. RISC-V Specification for CHERI Extensions (Draft). Retrieved April 09, 2025 from https://riscv.github.io/riscv-cheri/

[3] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2023. CHERIoT: Complete Memory Safety for Embedded Devices. In *MICRO*. ACM.

[4] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. 2017. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *RTSS*. IEEE.

[5] Apple, Inc. 2025. *DriverKit: Develop device drivers that run in user space.* Retrieved April 9, 2025 from https://developer.apple.com/documentation/driverkit

[6] Arm Developer. 2024. Mali GPU Driver Vulnerabilities. (2024). Retrieved July 17, 2024 from https://developer.arm.com/Arm%20Security%20Center/Mali%20GPU%20Driver%20Vulnerabilities

[7] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. 2006. Thorough Static Analysis of Device Drivers. In *EuroSys*. ACM.

[8] Big Switch Networks, Inc. 2024. *uBPF*. Retrieved April 29, 2024 from https://github.com/iovisor/ubpf

[9] Nicholas Boucher and Ross Anderson. 2023. Trojan Source: Invisible Vulnerabilities. In *USENIX Security*. USENIX.

[10] Silas Boyd-Wickizer and Nickolai Zeldovich. 2010. Tolerating Malicious Device Drivers in Linux. In *ATC*. USENIX.

[11] Ruslan Bukin. 2021. The Panfrost Driver. *FreeBSD Journal* (July/August 2021), 21–26. https://freebsdfoundation.org/wp-content/uploads/2021/08/The-Panfrost-Driver.pdf

[12] Shakeel Butt, Vinod Ganapathy, Michael M. Swift, and Chih-Cheng Chang. 2009. Protecting Commodity Operating System Kernels from Vulnerable Device Drivers. In *ACSAC*. ACM.

[13] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*. IEEE.

[14] Haibo Chen, Xie Miao, Ning Jia, et al. 2024. Microkernel Goes General: Performance and Compatibility in the HongMeng Production Microkernel. In *OSDI*. USENIX.

[15] J. Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazières, Antonio Dias, Margo Seltzer, and Michael D. Smith. 1995. The Measured Performance of Personal Computer Operating Systems. In *SOSP*. ACM.

[16] Lewis Crawford and Michael O'Boyle. 2019. Specialization Opportunities in Graphical Workloads. In *PACT*. IEEE.

[17] William J. Dally, Stephen W. Keckler, and David B. Kirk. 2021. Evolution of the Graphics Processing Unit (GPU). *IEEE Micro* 41, 6 (2021).

[18] Brooks Davis, Robert N. M. Watson, Alexander Richardson, et al. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. In *ASPLOS*. ACM.

[19] CheriBSD Developers. 2024. *CheriBSD*. Retrieved June 20, 2024 from https://www.cheribsd.org/

[20] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. 2019. XPC: Architectural Support for Secure and Efficient Cross Process Call. In *ISCA*. ACM.

[21] eBPF project. 2024. *What is eBPF?* Retrieved April 29, 2024 from https://ebpf.io/what-is-ebpf/

[22] Jake Edge. 2015. A seccomp overview. (Sept. 2015). Retrieved June 20, 2024 from https://lwn.net/Articles/656307/

[23] Envytools. [n. d.]. *PCI BARs and other means of accessing the GPU.* Retrieved October 28, 2024 from https://envytools.readthedocs.io/en/latest/hw/bus/bars.html

[24] Norman Feske. 2025. *Genode Operating System Framework 25.05 Foundations.* Retrieved August 7, 2025 from https://genode.org/documentation/genode-foundations-25-05.pdf Section 3.4.7 discusses of IRQs and MMIO.

[25] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, et al. 2024. Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety. In *ASPLOS*. ACM.

[26] Jose Fonseca. 2025. *apitrace.* Retrieved October 15, 2024 from https://apitrace.github.io

[27] Tora Fridholm. 2025. *Codasip launches complete exploration platform to accelerate CHERI adoption.* Retrieved August 20, 2025 from https://codasip.com/press-release/2025/04/29/codasip-prime-launch/

[28] Vinod Ganapathy, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. 2007. Microdrivers: A New Architecture for Device Drivers. In *HotOS*. USENIX.

[29] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. 2008. The Design and Implementation of Microdrivers. In *ASPLOS*. ACM.

[30] Dapeng Gao and Robert N. M. Watson. 2023. Library-based Compartmentalisation on CHERI. In *PLARCH Workshop*.

[31] Sergiu Gatlan. 2021. AMD fixes dozens of Windows 10 graphics driver security bugs. *Bleeping Computer* (Nov. 2021). Retrieved April 09, 2025 from https://www.bleepingcomputer.com/news/security/amd-fixes-dozens-of-windows-10-graphics-driver-security-bugs/

[32] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. 2000. The SawMill multiserver approach. In *ACM SIGOPS European Workshop*. ACM.

[33] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schäfer. 2021. Anomalicious: Automated Detection of Anomalous and Potentially Malicious Commits on GitHub. In *ICSE-SEIP*. IEEE.

[34] Dan Goodin. 2022. How a Microsoft blunder opened millions of PCs to potent malware attacks. *Ars Technica* (Oct. 2022). Retrieved April 09, 2025 from https://arstechnica.com/information-technology/2022/10/how-a-microsoft-blunder-opened-millions-of-pcs-to-potent-malware-attacks/

[35] Dan Goodin. 2023. Microsoft signing keys keep getting hijacked, to the delight of Chinese threat actors. *Ars Technica* (Aug. 2023). Retrieved April 09, 2025 from https://arstechnica.com/security/2023/08/facing-failure-after-failure-microsofts-driver-signing-program-fails-yet-again/

[36] Google. 2025. *Zircon.* Retrieved August 7, 2025 from https://fuchsia.dev/fuchsia-src/concepts/kernel See https://fuchsia.dev/fuchsia-src/concepts/drivers/mapping-a-devices-memory-in-a-driver and https://fuchsia.dev/fuchsia-src/reference/kernel_objects/interrupts for a discussion of IRQs and MMIO.

[37] Richard Grisenthwaite, Graeme Barnes, Robert N.M. Watson, Simon W. Moore, Peter Sewell, and Jonathan Woodruff. 2023. The Arm Morello Evaluation Platform — Validating CHERI-based Security in a High-Performance System. *IEEE Micro* 43, 3 (2023).

[38] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. 2020. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-Kernel Isolation and Communication. In *ATC*. USENIX.

[39] Ayub A. Gubran and Tor M. Aamodt. 2019. Emerald: Graphics Modeling for SoC Systems. In *ISCA*. ACM.

[40] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. 2006. MINIX 3: A Highly Reliable, Self-Repairing Operating System. *ACM SIGOPS Operating Systems Review* 40, 3 (July 2006).

[41] Jann Horn. 2023. *Qualcomm Adreno/KGSL: unchecked cast of vma->vm_file->private_data in kgsl_setup_dmabuf_useraddr().* Retrieved October 23, 2024 from https://project-zero.issues.chromium.org/issues/42451565 CVE-2023-21665.

[42] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2022. KSplit: Automating device driver isolation. In *OSDI*. USENIX.

[43] Intel. 2023. *Intel Iris Xe and UHD Graphics Open Source Programmer's Reference Manual For 2020-2021 11th Generation Intel Xeon, Core, Celeron, Pentium Gold Processors based on the "Tiger Lake" Platform Volume 6: Memory Views.* See p. 4.

[44] Hiroo Ishikawa, Alexandre Courbot, and Tatsuo Nakajima. 2008. A Framework for Self-Healing Device Drivers. In *SASO (SASO)*. IEEE.

[45] Nicolas Joly, Saif ElSherei, and Saar Amar. 2020. Security Analysis of CHERI ISA. Retrieved July 9, 2025 from https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf

[46] Antti Kantee. 2010. Rump Device Drivers: Shine On You Kernel Diamond. In *AsiaBSDCon*.

[47] Aaron Klotz. 2022. New Intel Driver Delivers Up To 8 Percent Performance Uplift On Arc GPUs. *Tom's Hardware* (Nov. 2022). Retrieved April 09, 2025

from https://www.tomshardware.com/news/new-intel-driver-delivers-up-to-8-percent-performance-uplift-on-arc-gpus
[48] Aaron Klotz. 2024. Nvidia publishes eight security flaws patched by new drivers — update to fix the issues. *Tom's Hardware* (Feb. 2024). Retrieved April 09, 2025 from https://www.tomshardware.com/pc-components/gpus/nvidia-publishes-eight-security-flaws-patched-by-new-drivers-update-to-fix-the-issues
[49] Joseph Kong. 2012. *FreeBSD Device Drivers: A Guide for the Intrepid.* No Starch Press.
[50] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2023. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *S&P*. IEEE.
[51] Michael Larabel. 2022. AMD Graphics Driver Surpassing 4 Million Lines Of Code In Linux 5.19, NVIDIA Opens Up At 1 Million. (May 2022). Retrieved April 11, 2024 from https://www.phoronix.com/news/AMDGPU-4-Million
[52] Boris Larin. 2023. Operation Triangulation: The last (hardware) mystery. (Dec. 2023). Retrieved April 09, 2025 from https://securelist.com/operation-triangulation-the-last-hardware-mystery/111669/
[53] Ben Leslie, Peter Chubb, Nicholas Fitzroy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yue-Ting Shen, Kevin Elphinstone, and Gernot Heiser. 2005. User-Level Device Drivers: Achieved Performance. *Journal of Computer Science and Technology* 20, 5 (Sept. 2005).
[54] Linux Foundation. [n. d.]. *The Linux Kernel Archives.* https://www.kernel.org/
[55] Lukas Maar, Florian Draschbacher, Lorenz Schumm, Ernesto Martinez Garcia, and Stefan Mangard. 2025. The Doom of Device Drivers: Your Android Device (Most Likely) has N-Day Kernel Vulnerabilities. In *USENIX Security*. USENIX.
[56] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network stack specialization for performance. In *SIGCOMM*. ACM.
[57] Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. 2017. Disk|Crypt|Net: rethinking the stack for high-performance video streaming. In *SIGCOMM*. ACM.
[58] Steve Marschner and Peter Shirley. 2022. *Fundamentals of Computer Graphics* (fifth ed.). CRC Press. See Section 22.4 regarding frame rates.
[59] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *EuroSys*. ACM.
[60] Microsoft. 2021. Making eBPF work on Windows. (May 2021). Retrieved April 29, 2024 from https://cloudblogs.microsoft.com/opensource/2021/05/10/making-ebpf-work-on-windows/
[61] Microsoft. 2024. WDDM overview. (Aug. 2024). Retrieved April 09, 2025 from https://learn.microsoft.com/en-us/windows-hardware/drivers/display/windows-vista-display-driver-model-design-guide
[62] Microsoft. 2025. Driver code signing requirements. (March 2025). Retrieved April 09, 2025 from https://learn.microsoft.com/en-us/windows-hardware/drivers/dashboard/code-signing-reqs
[63] Morello Project. [n. d.]. *The CHERI LLVM Compiler Infrastructure.* Retrieved April 09, 2025 from https://git.morello-project.org/morello/llvm-project
[64] Edward Tomasz Napierała. 2024. *"coexecve()" System Call Code.* Retrieved March 13, 2025 from https://github.com/CTSRD-CHERI/cheribsd/blob/cocalls/sys/kern/kern_exec.c#L264-L312
[65] Edward Tomasz Napierała and The FreeBSD Project. 2024. *"execve()" Man Page.* Retrieved March 17, 2025 from https://github.com/CTSRD-CHERI/cheribsd/blob/cocalls/lib/libc/sys/execve.2 Also explains the 'coexecve()' system call.
[66] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, et al. 2019. LXDs: Towards Isolation of Kernel Subsystems. In *ATC*. USENIX.
[67] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2020. Lightweight kernel isolation with virtualization and VM functions. In *VEE*. ACM.
[68] Lily Hay Newman. 2024. Google Researchers Found Nearly a Dozen Flaws in Popular Qualcomm Software for Mobile GPUs. *Wired* (Aug. 2024). Retrieved April 09, 2025 from https://www.wired.com/story/google-android-red-team-qualcomm-gpu-flaws/
[69] Ruslan Nikolaev and Godmar Back. 2013. VirtuOS: an operating system with kernel virtualization. In *SOSP*. ACM.
[70] NIST. 2021. *CVE-2021-28663 Detail.* Retrieved October 23, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2021-28663
[71] NIST. 2021. *CVE-2021-28664 Detail.* Retrieved October 23, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2021-28664
[72] Lindsey O'Donnell. 2019. Intel Windows 10 Graphics Drivers Riddled With Flaws. *Threat Post* (March 2019). Retrieved April 09, 2025 from https://threatpost.com/intel-windows-10-graphics-drivers/142778/
[73] OpenBenchmarking.org and Larabel, Michael. 2020. *Phoronix Desktop Graphics Test Suite.* Retrieved October 14, 2024 from https://openbenchmarking.org/suite/pts/desktop-graphics
[74] Sean Peisert, Bruce Schneier, Hamed Okhravi, et al. 2021. Perspectives on the SolarWinds Incident. *IEEE Security & Privacy* 19, 2 (2021).
[75] Keith Pleas. 1996. Windows NT 4.0. *Windows IT Pro* (April 1996).
[76] Ian Pratt and Keir Fraser. 2001. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *INFOCOM*. IEEE.
[77] Weizhong Qiang, Kang Zhang, and Hai Jin. 2016. Reducing TCB of Linux Kernel Using User-Space Device Driver. In *Algorithms and Architectures for Parallel Processing*. Springer.
[78] Qualcomm. 2021. *Qualcomm January 2021 Security Bulletin.* Retrieved October 23, 2024 from https://docs.qualcomm.com/product/publicresources/securitybulletin/january-2021-bulletin.html See subsection on CVE-2020-11261.
[79] Red Hat, Inc. 2024. *CVE-2024-3094 Detail.* Retrieved September 04, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2024-3094 Backdoor in XZ library.
[80] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *ATC*. USENIX.
[81] RME Audio. [n. d.]. *RME Drivers explained - DriverKit vs. Kernel Extension.* Retrieved October 22, 2024 from https://rme-audio.de/driverkit-vs-kernel-extension.html
[82] Joanna Rutkowska and Rafal Wojtczuk. 2010. *Qubes OS Architecture.* https://qubes-os.org/attachment/doc/arch-spec-0.3.pdf
[83] Ardalan Amiri Sani, Lin Zhong, and Dan S. Wallach. 2014. Glider: A GPU Library Driver for Improved System Security. *Technical Report 2014-11-14, Rice University* (2014).
[84] Oliver Schwahn, Stefan Winter, Nicolas Coppik, and Neeraj Suri. 2018. How to Fillet a Penguin: Runtime Data Driven Partitioning of Linux Code. *IEEE Transactions on Dependable and Secure Computing* 15, 6 (2018).
[85] Mark Segal and Kurt Akeley. 2022. The OpenGL® Graphics System: A Specification (Version 4.6 (Core Profile)-May 5, 2022).
[86] Maddie Stone, Jared Semrau, and James Sadowski. 2024. We're All in this Together: A Year in Review of Zero-Days Exploited In-the-Wild in 2023. *Google Project Zero* (March 2024). https://storage.googleapis.com/gweb-uniblog-publish-prod/documents/Year_in_Review_of_ZeroDays.pdf
[87] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. 2003. Improving the reliability of commodity operating systems. In *SOSP*. ACM.
[88] Devesh Tiwari, Saurabh Gupta, George Gallarno, Jim Rogers, and Don Maxwell. 2015. Reliability Lessons Learned From GPU Experience With The Titan Supercomputer at Oak Ridge Leadership Computing Facility. In *SC*. ACM.
[89] Bill Toulas. 2022. NVIDIA releases GPU driver update to fix 29 security flaws. *Bleeping Computer* (Nov. 2022). Retrieved April 09, 2025 from https://www.bleepingcomputer.com/news/security/nvidia-releases-gpu-driver-update-to-fix-29-security-flaws/
[90] Lluís Vilanova, Marc Jordà, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2017. Direct Inter-Process Communication (dIPC): Repurposing the CODOMs Architecture to Accelerate IPC. In *EuroSys*. ACM.
[91] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. 1995. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *SOSP*. ACM.
[92] Tom Warren. 2023. Nvidia boosts Starfield performance with GPU driver update. *The Verge* (Sept. 2023). Retrieved April 09, 2025 from https://www.theverge.com/2023/9/12/23870123/nvidia-starfield-performance-resizable-bar-new-gpu-drivers
[93] Robert N.M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: practical capabilities for UNIX. In *USENIX Security*. USENIX.
[94] Robert N.M. Watson, Jessica Clarke, Peter Sewell, et al. 2023. *Early performance results from the prototype Morello microarchitecture.* Technical Report UCAM-CL-TR-986. University of Cambridge, Computer Laboratory. doi:10.48456/tr-986
[95] Robert N.M. Watson, Ben Laurie, and Alex Richardson. 2021. Assessing the Viability of an Open-Source CHERI Desktop Software Ecosystem. https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/20210917-capltd-cheri-desktop-report-version1-FINAL.pdf See page 9, among others, regarding adapting software to CHERI.
[96] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, et al. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *S&P*. IEEE.
[97] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, et al. 2020. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8).* Technical Report UCAM-CL-TR-951. University of Cambridge, Computer Laboratory. doi:10.48456/tr-951
[98] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, et al. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *S&P*. IEEE.
[99] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. 2008. Device Driver Safety Through a Reference Validation Mechanism. In *OSDI*. USENIX.
[100] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, et al. 2019. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Trans. Comput.* 68, 10 (2019).
[101] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, et al. 2019. CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety. In *MICRO*. ACM.
[102] Ziye Yang, James R. Harris, Benjamin Walker, et al. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *CloudCom*. IEEE.
[103] Zhihao Yao, Zongheng Ma, Yingtong Liu, Ardalan Amiri Sani, and Aparna Chandramowlishwaran. 2018. Sugar: Secure GPU Acceleration in Web Browsers. In *ASPLOS*. ACM.