

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

University of Modena and Reggio Emilia
Department of Physics, Informatics and Mathematics

Doctor of Philosophy (Ph.D.) in
Computer and Data Science for Technological and Social Innovation
XXXVIII Cycle

Building Safety from Security and Predictability: A CHERI-Centric Framework for Autonomous Systems

Supervisor:
Prof. Andrea Marongiu

Candidate:
Donato Ferraro

Adjunct Supervisors:
Ph.D. Andrea Bastoni
Ph.D. Marco Solieri

Ph.D. Program Coordinator: Prof. Andrea Marongiu

Sommario

La sicurezza (safety) nei sistemi autonomi ed embedded dipende sia dalla predicibilità del comportamento sia da un forte isolamento tra i componenti. La predicibilità garantisce che le elaborazioni e le comunicazioni critiche avvengano entro vincoli temporali noti, mentre l'isolamento impedisce che guasti o compromissioni in un componente influenzino gli altri. Ottenere simultaneamente entrambe queste proprietà rimane una sfida significativa. Sebbene approcci esistenti come i sistemi operativi real-time, la virtualizzazione e il Time-Sensitive Networking (TSN) migliorino la predicibilità e il contenimento dei guasti, essi offrono una protezione limitata contro le vulnerabilità legate alla sicurezza della memoria (memory safety). Tali vulnerabilità rappresentano ancora la principale causa di compromissione del software nei sistemi embedded.

Questa tesi inizia esaminando la relazione tra predicibilità, sicurezza informatica (security) e sicurezza funzionale (safety) attraverso l'analisi della Time-Sensitive Autonomous Architecture (TSAA), un framework che combina virtualizzazione, TSN e sistemi operativi real-time per veicoli autonomi. TSAA dimostra come le tecnologie esistenti possano migliorare la predicibilità e supportare i requisiti di sicurezza. Tuttavia, evidenzia anche una limitazione fondamentale: l'isolamento viene generalmente applicato a livello di macchina virtuale, lasciando i componenti software all'interno di una stessa partizione vulnerabili a guasti e attacchi derivanti da problemi di sicurezza della memoria. Questa osservazione motiva la ricerca di un meccanismo di isolamento (*compartimentazione*) più granulare, adatto ai sistemi autonomi safety-critical.

Per affrontare questa sfida, la tesi studia Capability Hardware Enhanced RISC Instructions (CHERI), un'architettura hardware-software che estende i processori convenzionali con *capabilities* applicate e verificate direttamente dall'hardware. CHERI impedisce accessi non autorizzati alla memoria e mitiga le vulnerabilità più diffuse legate alla gestione della memoria associando ai puntatori limiti e permessi. Ciò consente una compartimentazione efficiente. La tesi presenta innanzitutto una panoramica completa dell'ecosistema CHERI, comprendendo implementazioni hardware, compilatori, sistemi operativi, librerie e applicazioni. Vengono inoltre identificate le opportunità e le sfide associate alla sua adozione nei sistemi safety-

critical.

Su queste basi, CHERI viene applicato a due domini rappresentativi. In primo luogo, l'interazione tra isolamento basato su capability e networking ad alte prestazioni viene valutata compartimentando componenti di rete in spazio utente sulla piattaforma Arm Morello, inclusi il driver Ethernet, lo stack TCP/IP e le applicazioni. In secondo luogo, la compartimentazione basata su CHERI viene introdotta nello stack software PX4/NuttX per droni al fine di isolare sensori, moduli di comunicazione e funzionalità critiche di volo su una piattaforma RISC-V CVA6 abilitata a CHERI e implementata su FPGA. I risultati dimostrano che la compartimentazione imposta dall'hardware è una soluzione praticabile e può fornire un isolamento robusto con un overhead prestazionale minimo.

Nel complesso, questa tesi sostiene che la sicurezza dei sistemi autonomi debba essere affrontata considerando congiuntamente predicibilità e sicurezza informatica. TSAA mette in evidenza i limiti delle architetture attuali e sottolinea la necessità di meccanismi di isolamento più granulari. CHERI fornisce una base hardware concreta per realizzare tale isolamento. Dimostrando la fattibilità della compartimentazione basata su CHERI in contesti applicativi reali, questo lavoro apre la strada a future ricerche su architetture sicure e affidabili per sistemi autonomi ed embedded.

Abstract

Safety in autonomous and embedded systems depends on both predictable behavior and strong isolation. Predictability ensures that critical computations and communications occur within known time constraints, and isolation prevents faults or compromises in one component from affecting others. Achieving both properties simultaneously remains challenging. Although existing approaches such as real-time operating systems, virtualization, and Time-Sensitive Networking (TSN) improve predictability and fault containment, they provide limited protection against memory-safety vulnerabilities. These vulnerabilities remain the dominant source of software compromise in embedded systems.

This thesis begins by examining the relationship between predictability, security, and safety through the analysis of the *Time-Sensitive Autonomous Architecture* (TSAA), a framework that combines virtualization, TSN, and real-time operating systems for autonomous vehicles. TSAA illustrates how existing technologies can enhance predictability and support safety requirements. However, it also reveals a fundamental limitation: isolation is usually enforced at the granularity of partitioned virtual machines, leaving software components within a partition vulnerable to memory-safety faults and attacks. This observation motivates the search for a finer-grained isolation mechanism suitable for safety-critical autonomous systems.

To address this challenge, the thesis investigates *Capability Hardware Enhanced RISC Instructions* (CHERI), a hardware-software architecture that extends conventional processors with hardware-enforced capabilities. CHERI prevents unauthorized memory accesses and mitigates common memory vulnerabilities by associating pointers with bounds and permissions. This enables efficient compartmentalization. First, the thesis presents a comprehensive survey of the CHERI ecosystem, covering hardware implementations, compilers, operating systems, libraries, and applications. It also identifies opportunities and challenges for deployment in safety-critical systems.

Building on this foundation, CHERI is applied to two representative domains. First, the interaction between capability-based isolation and high-performance networking is evaluated by compartmentalizing user-space networking components on the Arm Morello platform, including the Ethernet driver, TCP/IP stack, and

applications. Second, CHERI-based compartmentalization is introduced into the PX4/NuttX drone software stack to isolate sensors, communication modules, and flight-critical functionality on a CHERI-enabled RISC-V CVA6 platform implemented on an FPGA. The results demonstrate that hardware-enforced compartmentalization is practical and can provide robust isolation with minimal performance overhead.

Overall, this thesis argues that safety in autonomous systems must be approached through both predictability and security. TSAA highlights the limitations of current architectures and emphasizes the need for finer-grained isolation. CHERI provides a practical hardware-software foundation for achieving this isolation. By demonstrating the feasibility of CHERI-based compartmentalization in real-world domains, this work lays the groundwork for future research on secure and safe architectures for autonomous and embedded systems.

Acknowledgments

I would like to start by thanking my academic supervisor, *Andrea Marongiu*, for his guidance and support throughout my PhD. His advice, feedback, and, without hiding it, the occasional slap on the wrist, have been an important part of my research journey.

I would also like to thank *Minerva Systems*, and, in particular, my supervisors, *Andrea Bastoni*, *Alexander Zuepke*, and *Marco Solieri*, for giving me the opportunity to carry out this research in such an interesting and stimulating environment. I am very grateful for the trust they placed in me to explore a topic that I found genuinely interesting. It has been a challenging experience, but also one that I have greatly enjoyed. I want to give special thanks to *Andrea Bastoni*, for guiding me along the way and for being a *concrete* example of how to work professionally. As I always say, if I had just 1% of his skills at managing projects, I would already be a good project manager.

I would like to thank all my colleagues with whom I have shared these years, both at work and outside of it. Between coffee breaks, beers after work, and countless table football matches, you made the everyday life during these years much more enjoyable.

Special thanks also goes to *Sandro Pinto*, who welcomed me in Portugal, at the *University of Minho*, and to *Bruno Sá* (my hardware-guy) with whom I have had the pleasure of collaborating throughout these three years. I really enjoyed not only the research we have done together, but also all the discussions, ideas, chats, and time we have shared along the way. Working together has been an important part of my PhD.

Finally, I would like to thank my family for always being there for me and for supporting me throughout these years.

Last, but certainly not least, I want to thank *Lucrezia*, who has been by my side throughout this journey and with whom I have recently started the most important one yet: becoming a family. Thank you for your support, your patience, and for sharing all of this with me. And, of course, thank you for making me a dad.

"With infinite time and money, everything is possible."
Marco Solieri, whenever applicable

Contents

1 Introduction **1**

2 Background **13**
2.1 Autonomous Systems Architectures 13
2.2 Memory Safety 17
2.3 Compartmentalization Techniques 21

3 CHERI: A Comprehensive Survey **27**
3.1 CHERI Protection Model 27
3.2 CHERI Hardware 31
3.3 CHERI Software 39
3.4 CHERI Application Domains & Use-cases 52
3.5 Broader Future Directions for CHERI Research 53
3.6 Summary 57

4 CHERI Compartmentalized Network Stack **59**
4.1 Preliminary concepts 60
4.2 Network Stack Compartmentalization Options 61
4.3 Evaluation 65
4.4 Related Work 69
4.5 Summary 71

5 Security for Drones: CHERI-enabled PX4/NuttX **73**
5.1 The CHERI-UAV Platform 73
5.2 CVA6H-CHERI: Microarchitecture and Design 75
5.3 Problem Statement 76
5.4 Preliminary concepts 77
5.5 Alternative Designs for Secure UAV 80
5.6 CHERI-enhanced PX4/NuttX Design 83
5.7 Implementation 88
5.8 Evaluation & CVE Analysis 91

5.9 Related Work 97
5.10 Summary 97

6 Conclusion **99**
6.1 Limitations 100
6.2 Future Work 100

Publications and Academic Output **103**

A Historical Development of Capability Systems **105**
A.1 Hardware Capability Systems 106
A.2 Software Capabilities 111

Chapter 1

Introduction

Autonomous systems are no longer just a curiosity of researchers. They navigate public roads, coordinate logistics in warehouses, manage industrial processes, and guide unmanned aerial vehicles (UAVs) through contested airspace. An autonomous system is usually organized as a pipeline through with (processed) sensor data flows. This data feeds into perception and localization modules and then into planning and decision-making components. Ultimately, it drives actuation commands that control the physical platform. Consider two representative examples. A UAV executing a beyond-visual-line-of-sight (BVLOS) inspection mission, running an autopilot stack such as PX4/NuttX [245], runs a perception pipeline that fuses camera and LiDAR data, a planning stack that computes waypoints despite GPS uncertainty, and a flight controller—operating in either manual or autonomous mode—that must close its control loop within strict timing deadlines. Similarly, an autonomous vehicle hosts dozens of software components with different levels of criticality. A localization algorithm must provide position estimates within milliseconds, a collision-warning system has safety-critical requirements, a diagnostic logger operates on a best-effort basis, and an infotainment stack must remain isolated so that it cannot interfere with higher-criticality functions. In both scenarios, the software components differ in their criticality, timing requirements, and exposure to external inputs.

These systems are uniquely difficult to engineer because they uniquely possess three requirements that have historically been treated as separate engineering problems, designed by different communities, evaluated against different threat models, and validated by different standards bodies [215]. The first requirement is *predictability*: The system must exhibit deterministic and analyzable timing behavior, ensuring that computations and communications complete within known deadlines despite high system load and resource contention. The second requirement is *security*: The system must be resistant to adversarial manipulation and able to contain the effects of a compromised component. It must also ensure that no part of the system can corrupt or observe another part of the system without explicit

authorization. The third requirement, *safety*, emerges from these two: The system must behave correctly and reliably in the physical world, even in the presence of faults, unexpected inputs, or partial system failures. Therefore, safety is not an isolated property, but rather the result of enforcing both predictable execution and strong isolation across the system.

The need for both predictability and security simultaneously is not just a conceptual challenge. It creates tangible, practical tensions at every level of the system stack. For example, security mechanisms that introduce non-deterministic overhead conflict with the temporal guarantees that safety demands. Isolation boundaries that protect components from one another impose communication latencies that violate real-time schedules. Hardware abstractions that simplify security analysis obscures the timing behavior required for real-time analysis. Research in real-time and secure systems has developed a rich ecosystem for each property individually, including real-time operating systems (RTOS), time-sensitive networking (TSN), hypervisors, memory-safe languages, static analysis, and formal verification. However, existing hardware-software approaches expose fundamental trade-offs between isolation strength, performance overhead, and predictability. These approaches do not fully reconcile these properties in a form suitable for autonomous systems.

This thesis argues that such a primitive exists and identifies it as CHERI [316] (Capability Hardware Enhanced RISC Instructions). It also demonstrates CHERI applicability across the networking stack, the operating system kernel, and the autonomous flight software of real embedded systems.

The Structural Inadequacy of Existing Approaches

The inadequacy of existing approaches becomes clear at the level of a deployed autonomous system architecture. Each layer of the software stack reveals a distinct limitation.

The networking layer presents the first challenge. The software stack components share a physical network. Standard Ethernet provides no guarantees about which traffic reaches which destination and when. A burst of best-effort traffic can delay safety-critical messages indefinitely. Time-Sensitive Networking (TSN) [113] is a set of IEEE standards that extend the Ethernet protocol with mechanisms for traffic prioritization, bandwidth reservation, and bounded-latency delivery. TSN addresses this issue by adding traffic scheduling mechanisms to Ethernet, such as the Credit-Based Shaper (IEEE 802.1Qav) and Time-Aware Traffic Shaping (IEEE 802.1Qbv), which reserve bandwidth and enforce temporal ordering for critical flows.

TSN represents a significant contribution because it makes the network predictable. However, it only addresses the network communication. It says nothing about the trustworthiness of the endpoints communicating over that connection. A node that has been compromised by a memory safety vulnerability will transmit whatever its attacker instructs, with whatever priority its network configuration permits. TSN protects network communications, but does not protect the software that generates or consumes the traffic.

The virtualization layer presents the second challenge. A hypervisor is a software layer that partitions the physical resources of a platform, such as CPU cores, memory regions, and peripheral devices, among multiple isolated execution environments (virtual machines). Each environment hosts an independent operating system or a bare-metal application. The standard engineering response to mixed-criticality coexistence is hardware-enforced partitioning through a hypervisor [238]. A type-1 partitioning hypervisor such as Jailhouse [158], Xen [242], PikeOS [279], distributes physical resources (CPU cores, memory regions, and peripheral access) between isolated virtual machines (VMs). A safety-critical real-time OS and a general-purpose OS can coexist on the same hardware with limited interference if the underlying hypervisor and hardware enforce strong spatial and temporal isolation. In these configurations, if the general-purpose OS experiences memory corruption, it is contained within its partition. Similarly, timing anomalies in one VM do not directly affect another. However, residual interference may remain through shared microarchitectural resources (e.g., cache and bus) [132, 186]. Hypervisor-based isolation is a mature and well-understood technology that is increasingly common in automotive and aerospace embedded systems.

Hypervisor isolation is, however, isolation at the granularity of virtual machines, that is, at the granularity of entire operating systems. Within a VM, isolation is delegated to the guest OS (e.g., process and kernel/user-space separation in Linux). A detection pipeline, a planning module, and a network proxy running in the same Linux VM are, from the hypervisor perspective, indistinguishable. A spatial or temporal memory safety violation in any one of them[1] can reach the data structures of any other component. This is not a hypothetical concern. Memory safety violations are the most common class of security vulnerability in systems software [212], and the software stacks of autonomous systems are written almost entirely in C and C++. These languages provide no spatial or temporal memory safety guarantees. The hypervisor establishes a hard boundary between virtual machines and provides no visibility into their internal operations.

[1]Spatial: a class of defect in which a program reads or writes memory outside the bounds of an allocated object; Temporal: a class of defect in which a program accesses memory after it has been freed



Figure 1.1: The Maserati Quattroporte used for the experiments.

A Case Study on the Limits of Existing Approaches

The abstract tension between the aforementioned requirements is not merely theoretical. It manifests as concrete architectural compromises that real systems must make. In prior work, we designed, implemented, and deployed a Time-Sensitive Autonomous Architecture (TSAA) on a real autonomous vehicle—a Maserati Quattroporte (Figure 1.1) instrumented for Level 4 autonomous operation [114]. TSAA integrates TSN networking with a static partitioning hypervisor (Jailhouse) and a mixed-criticality software stack combining a general-purpose Linux environment with the Erika Enterprise v3 RTOS [111] for safety-critical actuation. We tested the system both in simulation, using the CARLA autonomous driving simulator [99], and on the physical vehicle in a real parking lot environment. This validated the ability of TSN traffic shaping to maintain the predictability of safety-critical communication flows, even under deliberate network interference.

Even under adverse network conditions—specifically, a laptop generating maximum-bandwidth interference traffic directly injected into the vehicle network—the TSN Credit-Based Shaper maintained the integrity of the localization-to-actuation flow, enabling the vehicle to complete its autonomous laps. This is a meaningful result. However, the engineering decisions required to achieve this outcome also expose important limitations of the current systems software. The most instructive example is the actuation component. Vehicle actuation is the function with the highest level of safety criticality; failures here do not degrade gracefully—they result in loss of vehicle control. Therefore, the actuation node runs in a dedicated, single-core virtual machine (VM) that hosts the Erika RTOS. The RTOS provides the scheduling determinism and certifiability properties that Linux cannot. However, the problem is that Erika provides no Ethernet driver. The actuation component needs to receive localization data from the network. Under the VM isolation model, granting Erika direct Ethernet access requires either sharing the network driver across VM boundaries or replicating the driver inside the Erika VM.

Neither option is straightforward. TSAA's solution is to introduce a dedicated intermediary: a Linux VM running a node called ProxyROS2. ProxyROS2's sole function is to receive localization messages from the network, extract the relevant odometry data, and forward it to the Erika VM via IVSHMEM shared memory. The existence of an entire virtual machine is a consequence of the mismatch between the available isolation granularity (VM-level) and the needed isolation granularity (component/driver-level).

Given the available tools, this solution is architecturally reasonable, but it emphasizes a deeper limitations in the system design. The isolation required is between individual software components, not between operating systems. This isolation must be enforced at the level of memory region and data structures. It must also be free of the scheduling overhead, resource partitioning constraints, and communication costs that VM-level separation entails.

A second limitation becomes visible when the security model is examined more carefully. The IVSHMEM device, which connects VMs in Jailhouse, implements a permission table that specifies which VM can read or write to which shared memory region. These permissions prevent, for example, the actuation VM from writing to a memory region owned by the planning VM. However, these permissions are static software configurations enforced by the hypervisor rather than enforced at the hardware level on every memory access. A compromised component within a VM sees the entire address space of that VM. The isolation boundary follows the VM, not the data.

Finally, and most importantly, the entire security model of TSAA assumes that the adversary is external. The TSAA experiments inject interference from a laptop connected to the vehicle network. Hypervisor partitioning protects against faults and deliberate interference crossing VM boundaries. However, neither mechanism addresses the case of a software vulnerability within a single component, which is precisely the class of vulnerability that memory-unsafe languages make endemic [124, 214]. If an attacker exploits a buffer overflow in planning VM network stack, the isolation model provides no defense below the VM boundary. These are not TSAA-specific problems. Instead, they are the results of building safety-critical autonomous systems with tools that are isolated at the hypervisor or OS level.

Taken together, these limitations, define a three-dimensional gap in the existing solutions. TSN makes the network predictable, but says nothing about the trustworthiness of the endpoints on it. VM-based isolation provides strong guarantees but at the wrong granularity. It isolates operating systems, not software components. It forces architectural compromises (e.g., ProxyROS2 in TSAA), demonstrating that its isolation boundary does not align with the trust boundaries that safety-critical autonomous systems actually require. These limitations raise the question of whether the underlying hardware mechanism, that hypervisors

themselves rely upon to enforce isolation, could achieve the same guarantees at reduced cost. The next section asks whether a finer-grained hardware mechanism, the memory management unit (MMU), closes this gap.

MMU-Based Isolation: The Standard Answer and Its Limits

Not every embedded system has even this much. Many RTOS systems for resource-constrained platform, including NuttX in its most common configuration, run in *flat mode*: every task, driver, and kernel component shares a single physical address space, with no hardware boundary between them. Flat mode avoids the overhead of address translation entirely, which is attractive under tight memory and power budgets. But it offers no isolation whatsoever. A bug or a compromise in any single task can corrupt the memory of every other task, including the kernel itself. PX4 running on flat-mode NuttX, the configuration we use as a baseline in Chapter 5, is representative of a large class of deployed autonomous flight software.

The standard hardware response that improves on flat mode is the MMU. The MMU is a hardware component present in virtually all application-class processors that enforces memory isolation between processes. It does so by maintaining per-process page tables that map virtual addresses to physical addresses and attach permission bits to each page. A process running in user space cannot access the physical memory of another process because the MMU will not translate the virtual address to a valid mapping for a page it does not own. This is an important and genuine hardware guarantee, and process-based isolation using the MMU forms the basis of operating system security. However, it is insufficient as the sole mechanism for compartmentalization in safety-critical embedded systems, for three distinct reasons.

The most fundamental limitation of MMU-based isolation is that it operates exclusively at the boundary between address spaces. Within a single address space, the MMU enforces no constraints on spatial memory access. A program that reads or writes beyond the bounds of an allocated buffer, accesses heap memory after it has been freed, or corrupts adjacent stack objects remains entirely undetected by the hardware, because all of these accesses resolve to valid virtual-to-physical mappings within the process's own pages. The practical consequences of this limitation are extensively documented in vulnerability records [89]. The Heartbleed vulnerability (CVE-2014-0160), for example, exfiltrated private cryptographic material from a running OpenSSL process through a single out-of-bounds heap read that never crossed a page boundary visible to the MMU [103]. The attacker requested a heartbeat response larger than the allocated buffer, causing the implementation

to read beyond the buffer boundary and returning adjacent heap contents, including private keys and session tokens. The MMU functioned correctly; it simply has no mechanism for detecting an access that remains within the process's own address space. The `sudo` privilege escalation vulnerability (CVE-2021-3156) exploited a heap-based buffer overflow present in a widely deployed system utility for approximately a decade. It corrupted adjacent heap allocations within the same address space to obtain root privileges [248]. The Looney Tunables vulnerability (CVE-2023-4911) compromised the GNU C Library dynamic linker across essentially all major Linux distributions through a buffer overflow in environment variable processing. Again, this occurred entirely within a single address space and was completely invisible to the MMU [223]. These are not isolated incidents attributable to careless programming. They represent a structural and persistent consequence of deploying software written in memory-unsafe languages within an isolation model whose hardware enforcement does not extend beyond the address space boundary. For an autonomous system in which safety-critical and less-critical components share a virtual machine and therefore an address space, the MMU offers no protection against this class of vulnerability. For example, a rear-facing camera may simultaneously support non-critical rear-view video rendering (e.g., on the infotainment display) and safety-critical collision warning functions on the same platform.

The second limitation concerns the security of the MMU's inter-process isolation, which relies on the correctness of the kernel's page-tables management and assumes the absence of microarchitectural side channels. At the kernel level, vulnerabilities such as Dirty COW (CVE-2016-5195) and Dirty Pipe (CVE-2022-0847) have demonstrated that MMU page permissions can be bypassed through race conditions in the kernel's own memory management code [79, 145]. In both cases, the MMU configuration was correct, but a flaw in the kernel code responsible for maintaining that configuration allowed an unprivileged process to obtain write access to read-only pages. At the microarchitectural level, Meltdown (CVE-2017-5754) and Spectre (CVE-2017-5753, CVE-2017-5715) demonstrated that speculative execution allows processes to observe memory across MMU-enforced privilege boundaries through cache timing side channels. This renders the MMU's privilege separation subject to microarchitectural behavior that contemporary out-of-order processors do not guarantee by construction [167, 187].

The third limitation is one of performance under fine-grained use. Crossing an MMU isolation boundary requires a system call, involving a privilege level transition from user space to kernel space and back again. On contemporary processors, this round-trip costs hundreds to thousands of nanoseconds and might introduce timing variability that is incompatible with hard real-time scheduling [323, 327]. For a coarse-grained isolation model, in which components communicate infrequently,

this cost is acceptable. However, for the fine-grained compartmentalization that a component-level security model demands, where individual driver accesses or inter-module calls cross isolation boundaries, the overhead accumulates rapidly, and its variability destroys the temporal predictability required by safety-critical autonomous systems.

Taken together, these three limitations show that MMU-based isolation is not a viable foundation for fine-grained compartmentalization in autonomous systems. MMU-based isolation provides no spatial safety within an address space. Its inter-process guarantees depend on kernel correctness and microarchitectural behavior. Furthermore, its crossing costs are incompatible with the frequency of boundary crossings that fine-grained compartmentalization requires.

Software Mitigations: Necessary but Insufficient

The natural response to this limitation is to ask whether software-based security techniques can compensate for the absence of MMU's intra-address-space isolation. The existing set of software-based security techniques is robust. Static analysis tools can identify classes of memory safety defects at compile time. Dynamic instrumentation frameworks, such as AddressSanitizer [270] and Valgrind [220], can detect runtime violations. Memory-safe languages, such as Rust, eliminate entire categories of spatial and temporal errors through type-system enforcement. Compiler mitigations, such as stack canaries and control-flow integrity (CFI), and system-level mitigations, e.g., address-space layout randomization (ASLR), increase the cost of exploitation. Each of these mechanisms meaningfully reduces the attack surface of systems software, and none should be dismissed.

However, none of them constitutes a complete solution in the context of embedded autonomous systems. Static analysis is inherently incomplete: it can identify known vulnerability patterns but cannot guarantee the absence of all memory safety defects in software of realistic complexity. Runtime instrumentation, such as AddressSanitizer, imposes memory and timing overhead that is incompatible with hard real-time constraints and therefore unsuitable for deployment in safety-critical real-time embedded systems. Memory-safe languages, such as *Rust* and *Go*, require rewriting existing codebases, which is impractical for mature systems such as PX4/NuttX. They also do not address the security of third-party libraries or hardware drivers written in C. System-level and compiler mitigations are probabilistic or heuristic in nature: ASLR reduces the reliability of exploitation but does not prevent it. CFI protects forward-edge transfers but does not enforce data-plane isolation. The fundamental limitation of all software-based approaches is that they operate at the discretion of the software itself. *A sufficiently capable attacker who achieves arbitrary code execution within a component can, in principle, subvert*

any protection mechanism implemented in the same software system.[2] Hardware enforcement is required precisely because software running at the same privilege level cannot bypass it.

Table 1.1: Comparison of isolation and security mechanisms for autonomous systems.

Mechanism	**Isolation granularity**	**Memory safety**	**RT-friendly**	**HW-enforced**
TSN	Network flow	×	✓	Partial
Hypervisor	VM	×	✓	✓
Flat mode	None	×	✓	×
MMU (process)	Address space	✓	Partial	✓
Memory-safe languages	Object/type	✓	✓	×
CHERI	Object/capability	✓	Partial	✓

RT-friendly = compatible with hard real-time;
HW-enforced = hardware enforcement on every access.

Capability Hardware as the Missing Primitive

The isolation mechanism required for fine-grained compartmentalization in autonomous systems must operate below the level of the operating system. It must be enforced by hardware on every memory access. It must follow individual objects and data structures rather than address space boundaries. It must be composable with existing real-time and networking mechanisms without introducing non-deterministic overhead.

Capability Hardware Enhanced RISC Instructions (CHERI) [297] provides precisely this. Originally developed at the University of Cambridge under the DARPA CRASH (Clean-slate Design of Resilient, Adaptive, Secure Hosts) program—a funded research into more secure computer architectures and OSs—, CHERI extends conventional processor architectures with a *capability* system. In this system, *every memory access is mediated by a hardware-enforced, unforgeable token that encodes the address, bounds, and permissions of the access.* The processor checks these constraints on every load, store, and instruction fetch without software intervention.

[2]For example, compromising the Linux kernel may allow control over all user processes, while compromising a hypervisor may enable control over all guest VMs.

With CHERI, isolation no longer requires a virtual machine boundary. Two software components can share a physical core and even a physical address space while being prevented, by hardware, from accessing each other's data. The isolation primitive is the *capability*, not the process or the VM. For example, a network driver can be compartmentalized from the protocol stack that uses it. Similarly, a PX4 autopilot module can be isolated from the sensor fusion module that feeds it. The compartment boundary follows the software architecture, not the hardware resource budget.

This property directly addresses all three dimensions of the gap identified in the preceding section. CHERI enforces spatial memory safety within address spaces, eliminating the class of intra-process violations that the MMU cannot detect. It does so through hardware mechanisms that cannot be bypassed by software executing at the same privilege level. This addresses the conditional nature of both software mitigations and MMU-based guarantees. Because CHERI compartment crossings operate entirely in user space without a privilege level transition, their overhead is lower than an equivalent MMU-based context switch ($\sim 1.22\times$ faster based on our experiment in Section 5.8). This makes fine-grained compartmentalization compatible with the crossing frequency that safety-critical autonomous systems require.

In the safety-critical context, CHERI provides deterministic memory safety enforcement in the sense that each memory access is either permitted or rejected based on hardware capability checks. on a capability violation the processor raises a synchronous exception and transfers control to a software handler. Since trap handling is delegated to the operating system or runtime environment, the resulting execution time is not controlled by the hardware enforcement mechanism and may introduce jitters. However, this behavior is not different from existing real-time systems that already rely on MMU-based virtual memory, where page faults are also handled via software exceptions. The experiments reported in this thesis demonstrate that the overhead of introducing CHERI is small enough to preserve the timing properties that autonomous systems require. Table 1.1 compares isolation and security mechanisms in terms of their isolation granularity, memory safety guarantees, suitability for real-time systems, and reliance on hardware enforcement.

Thesis Statement and Contributions

This thesis demonstrates that CHERI capability hardware is a unifying primitive for safety-critical autonomous systems. It delivers hardware-enforced compartmentalization with fine granularity, preserves the real-time predictability that safety demands, and provides stronger isolation guarantees than coarser VM-based mechanisms at lower architectural cost. This claim is substantiated through a pro-

gression of concrete system implementations, from the structural limits of existing approaches exposed in prior collaborative work on autonomous vehicle architectures, to a comprehensive study of CHERI, and two original systems that deploy CHERI in the networking stack and in an autonomous flight operating system.

The contributions of this thesis are as follows:

The first contribution is a comprehensive survey of CHERI. It covers the technology's origins, hardware architecture, compiler support, software ecosystem, security properties, and open challenges. This survey establishes the theoretical foundation for the subsequent systems contributions and situates CHERI within the broader landscape of capability-based security research. It is among the first surveys to address CHERI from both a historical and a systems perspectives and explicitly address CHERI deployment challenges in safety-critical embedded contexts.

The second contribution is a CHERI-based compartmentalized network stack [115], implemented on the Arm Morello platform. We ported a framework that allows the execution of the Ethernet driver in user-space (i.e., DPDK [1]) and a TCP/IP library on top (i.e., F-Stack [3]) into capability-enforced compartments within the CAP-VMs Intravisor framework [261]. Using iperf3 [4] as a benchmark application, we demonstrate that compartmentalization of the network stack is achievable without sacrificing line-rate throughput. The system sustains 1 Gbps under normal operation. The only measurable overhead appears when multiple compartments contend for shared Ethernet driver access. This cost is attributable to synchronization and is characterized in our evaluation.

The third contribution is a compartmentalized autonomous flight operating system built on CHERI. We ported NuttX to CVA6-CHERI , implemented a CAP-VM-inspired compartmentalization layer, and executed PX4 autopilot modules in capability-based compartments on an FPGA-based experimental platform. We compare this architecture against flat-mode and MMU-based isolation and demonstrate that CHERI provides stronger security guarantees at lower primitive cost. Specifically, capability-based compartment crossings impose significantly lower overhead than MMU-based context switches for fine-grained isolation in autonomous flight software requires.

We draw on collaborative work on the CVA6-CHERI hardware platform to support these contributions. This platform provides the open-source RISC-V foundation for the NuttX and PX4 contributions.

Thesis Organization

The remainder of this thesis is structured as follows. Chapter 2 establishes the technical background necessary to understand the contributions, which include autonomous system architectures, real-time and predictability requirements, mixed-

criticality systems, and a conceptual introduction to memory safety and compartmentalization. Chapter 3 presents a comprehensive survey of CHERI, covering hardware, software, compiler, and security dimensions in depth. Chapter 4 presents the compartmentalized networking contribution on Arm Morello. Chapter 5 presents the NuttX and PX4 compartmentalization contribution on CVA6-CHERI. Chapter 6 concludes the thesis.

Chapter 2

Background

This chapter lays the technical groundwork necessary to follow the contributions presented in subsequent chapters. It covers the architecture of autonomous systems and their mixed-criticality requirements; the principles of real-time computation and predictability; the mechanisms of virtualization in safety-critical contexts; the taxonomy of memory safety violations and existing mitigations; and the landscape of compartmentalization techniques against which CHERI is positioned.

2.1 Autonomous Systems Architectures

An autonomous system is a cyber-physical system that can perform complex tasks in a real-world environment without continuous human intervention. This category includes autonomous ground vehicles, UAVs, autonomous marine platforms, and industrial robotic systems. Despite their diverse applications, these systems share a common software architecture organized around a sense-plan-act pipeline, represented in Figure 2.1. In this pipeline, raw sensor data is transformed through successive processing stages into physical control commands.

2.1.1 The Sense-Plan-Act Pipeline

The sensing layer is the lowest layer of the autonomous software stack. It interfaces with the physical sensors attached to the platform and transforms raw sensor output into structured data representations consumable by higher layers. Representative sensors include Light Detection and Ranging (LiDAR) units, which emit pulsed infrared light and measure return travel times to construct three-dimensional point clouds of the surrounding environment; inertial measurement units (IMUs), which integrate accelerometer and gyroscope readings to estimate platform velocity and orientation at high frequencies; cameras, which provide dense visual information

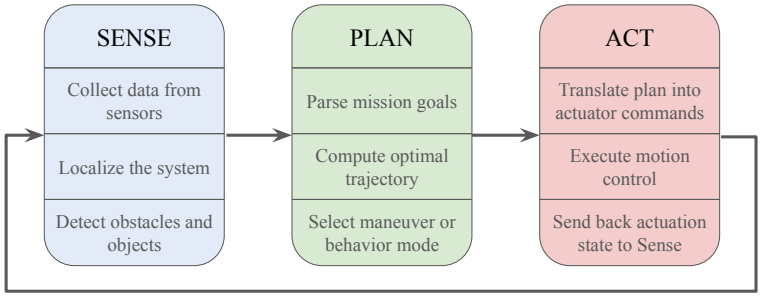


Figure 2.1: The Sense-Plan-Act Pipeline.

about the environment; and global navigation satellite system (GNSS) receivers, which provide absolute position estimates. Each sensor class presents a distinct combination of update frequency, measurement accuracy, and computational demand. LiDAR units typically operate at 10 Hz with high spatial accuracy but limited temporal resolution. IMUs operate at frequencies of 400 Hz or higher with low positional accuracy but high temporal resolution. The localization subsystem is responsible for fusing measurements from multiple sensors to obtain estimates that are simultaneously accurate and high-frequency. This subsystem applies sensor fusion algorithms, typically variants of the Extended Kalman Filter or iterated Extended Kalman Filter [151], to maintain a continuous estimate of the platform's pose within a known or incrementally constructed map of the environment. State-of-the-art localization algorithms such as FAST-LIO2 [324] exploit the complementary characteristics of LiDAR and IMU measurements. The IMU provides high-frequency pose updates that compensate for LiDAR distortion caused by platform motion. Meanwhile, LiDAR provides high-accuracy spatial measurements that bound the drift inherent in IMU integration. The localization subsystem's output is an odometry estimate published at the LiDAR update frequency. This provides the current platform pose to the planning layer. The perception layer operates in parallel with localization and is responsible for detecting and classifying objects in the environment that are relevant to safe navigation. Object detection is typically performed using deep neural networks applied to camera or LiDAR data, or a fusion of both. The perception layer's output is a set of detected obstacles, their estimated positions relative to the platform, and their estimated velocities. These are are consumed by the planning layer. The planning layer uses the platform's estimated pose, the map of the environment, and the set of detected obstacles to generate a trajectory that the platform should follow. Planning is typically decomposed into global planning, which computes a high-level route from the current pose to the goal, and local planning, which adjusts the trajectory dynamically in response to detected obstacles and localization updates. The planning layer outputs

a sequence of waypoints or a continuous reference trajectory and publishes it to the actuation layer. The actuation layer translates the reference trajectory into control commands for the platform's physical actuators. In a ground vehicle, this involves computing steering angle and velocity commands and transmitting them to the low-level vehicle controller via a communication bus such as the Controller Area Network (CAN). The actuation layer typically implements a feedback control algorithm, such as pure pursuit or model predictive control, which continuously adjusts the control output to minimize the deviation between the platform's current pose and the reference trajectory. The actuation layer is the most latency-sensitive component in the stack: delays in receiving localization updates or in transmitting control commands directly affect the platform's ability to safely follow its trajectory.

Although the sense-plan-act architecture is common among autonomous platforms, its implementation varies for aerial vehicles. Due to their highly dynamic motion and reduced inherent stability compared to ground vehicles, drones typically rely more heavily on IMUs. Visual-inertial odometry systems combine camera and IMU measurements to provide high-frequency pose estimates when GNSS is unavailable or unreliable, especially in indoors. Perception systems use cameras, LiDAR, or depth sensors to perform obstacle detection and free-space estimation, supporting collision avoidance. The planning layer must account for six-degree-of-freedom motion and three-dimensional obstacle fields to generate trajectories that satisfy constraints on velocity, acceleration, and vehicle attitude. The actuation layer converts the planned trajectory into thrust and attitude commands that are distributed across the drone's motors through a flight controller. Since aerial vehicles are inherently unstable and operate with higher dynamics than ground vehicles, control loops typically run at significantly higher frequencies—often hundreds of hertz—making low-latency sensing and state estimation critical for safe operation.

2.1.2 Mixed-Criticality Systems

A defining characteristic of autonomous system software is that its components differ substantially in their criticality. Criticality refers to the severity of the consequences resulting from a component's failure to correctly perform its intended function and within its timing requirements [51]. A component is safety-critical if its failure could result in loss of human life, physical injury, or severe environmental damage, as commonly defined in safety engineering standards (e.g., IEC 61508 [153] for industrial systems and ARP4754A [239] in avionics). A component is non-critical if its failure results in degraded functionality or user inconvenience without safety consequences. In a representative autonomous vehicle stack, the actuation layer is unambiguously safety-critical; a failure to execute a correct steering command could result in loss of vehicle control. Similarly, the localization layer is critical because a failure to produce accurate and timely position estimates leaves the

actuation layer unable to compute correct commands. The perception layer is critical in the presence of dynamic obstacles but may degrade more gracefully in controlled environments. The monitoring and logging subsystems, and any infotainment components present on the same hardware platform, are non-critical. The automotive industry formalizes criticality through the Automotive Safety Integrity Level (ASIL), which is defined in ISO 26262 [269]. ISO 26262 is a functional safety standard for road vehicles. ASIL levels range from A (lowest) to D (highest) and are determined by the combination of the severity of potential harm, the probability of exposure to the hazardous situation, and the controllability of the hazard by the driver. ASIL-D components, such as brake control systems and steering systems, must be developed with the strictest engineering processes and are subject to the most rigorous verification requirements. ASIL-A components are subject to reduced requirements. Components that are not safety-relevant are classified as Quality Management (QM). Therefore, an autonomous vehicle software stack hosts components spanning the full range from QM to ASIL-D on shared hardware. Mixed-criticality considerations are equally important in autonomous aerial vehicles. Flight stabilization, attitude estimation, motor control, and emergency recovery mechanisms are safety-critical functions because failures can result in loss of vehicle control and subsequent crash. These functions typically execute within the flight controller under strict timing constraints and may be subject to aviation-oriented safety assurance processes, such as ARP4754A [239] for system-level design and DO-178C [155] for software certification. In contrast, mission-level functions such as mapping, object detection, route planning, video processing, telemetry visualization, and payload management are generally less critical and are often implemented using complex software frameworks executing on general-purpose operating systems. Consequently, autonomous drone platforms frequently integrate software components with substantially different criticality levels on shared computing hardware.

Therefore, autonomous ground and aerial vehicle software stacks commonly host components spanning a wide range of criticality levels on shared computing hardware. This configuration is known as a mixed-criticality system. The challenge of mixed-criticality integration is that the design requirements for different criticality levels often conflict with each other. High-criticality components require strict temporal isolation. For example, a scheduling anomaly or resource contention caused by a low-criticality component must not delay a high-criticality component beyond its deadline [20]. High-criticality components also require spatial isolation. For instance, a memory safety violation in a low-criticality component must not corrupt the data structures of a high-criticality component. High-criticality components require certification: the development process and verification evidence must satisfy the requirements of the applicable safety standard. This standard constrains the

complexity of the software and hardware on which the component executes. In contrast, low-criticality components are often developed using general-purpose frameworks, execute on feature-rich operating systems, and leverage third-party libraries that are neither certified nor certifiable. The central engineering problem of mixed-criticality autonomous systems is integrating these two classes of software on shared hardware without compromising the certification of the high-criticality components.

2.1.3 Representative Autonomous Software Stacks

Two autonomous software stacks are directly relevant to the contributions of this thesis and are briefly described here: Robot Operating System 2 (ROS2) [18] is an open-source middleware framework for robotic and autonomous systems. ROS2 models each software component as a node that publishes and subscribes to typed communication channels called topics using a Data Distribution Service (DDS) middleware layer. ROS2 provides quality-of-service policies that allow users to configure the reliability and durability of topic communication per topic. This enables both safety-critical and non-critical data flows to coexist within the same middleware infrastructure. ROS2 is widely used in autonomous vehicle research and forms the middleware layer on which the TSAA architecture referenced in the introduction is built.

PX4 [204] is an open-source autopilot software stack for unmanned aerial vehicles and is widely used in research and commercial drone platforms. It is structured as a set of modules that communicate through a publish-subscribe bus called uORB. These modules include sensor drivers, state estimators, position controllers, attitude controllers, mixer modules, and output modules that translate controller outputs into motor commands. PX4 is designed to execute on NuttX [35], a real-time operating system for embedded systems that implements a POSIX-compatible interface. The combination of PX4 and NuttX is the system stack on which the SOSP contribution of this thesis is built, and its architecture is described in detail in Chapter 5.

2.2 Memory Safety

Memory safety is the property of a program that ensures all memory accesses are valid. This means that every read accesses memory initialized memory within the bounds of the object, and every write accesses memory within the bounds of the object that has not been freed. Programs that violate memory safety are susceptible to a class of defects that can compromise both their correctness and security. This section provides a taxonomy of memory safety violations, explains

why C and C++ are structurally vulnerable to them, and surveys the landscape of existing mitigations and their limitations.

2.2.1 Taxonomy of Memory Safety Violations

Memory safety violations are classified by two criteria: whether they involve spatial or temporal safety, and whether they involve reads, writes, or executes. Spatial safety violations occur when a program accesses memory outside the bounds of an allocated object. A buffer overflow occurs when a program writes beyond the end of a fixed-size buffer, overwriting adjacent memory that may contain other data structures, return addresses, or function pointers. A buffer over-read is a read beyond the end of a buffer, disclosing the contents of adjacent memory to the program or to an attacker. Both stack-based and heap-based buffers are susceptible to spatial violations. Stack-based buffer overflows that overwrite return addresses are particularly dangerous because they allow an attacker to redirect control flow to arbitrary code. Heap-based overflows that corrupt adjacent heap metadata allow an attacker to manipulate the allocator's data structures in ways that lead to arbitrary write primitives. The most common sources of spatial violations in C and C++ code are out-of-bounds array accesses and incorrect pointer arithmetic. A temporal safety violation occurs when a program accesses memory through a pointer that is no longer valid. A use-after-free violation occurs when a pointer to a freed heap allocation is dereferenced after the allocator has reassigned the underlying memory to a different object. Since the physical memory remains accessible via the stale pointer, the program can read data belonging to a different object, or misinterpret arbitrary heap contents as structured data. A use-after-return violation occurs when a pointer to a stack-allocated variable is retained and dereferenced after the function that allocated it has returned and the stack frame has been overwritten. A double-free violation occurs when a heap allocation is freed more than once, corrupting the allocator's internal state. Format string vulnerabilities arise when user-controlled input is passed directly as the format string argument to a formatted output function. An attacker who can control the format string can use format specifiers to read arbitrary memory locations from the stack or, in some implementations, write to arbitrary memory addresses. An integer overflow or underflow vulnerability arises when arithmetic operations on integer values produce results that exceed the representable range of the integer type, causing the value to wrap around. In contexts where the integer value is subsequently used as a buffer length or an array index, integer overflow can cause a spatial memory safety violation by producing a smaller-than-expected allocation or a larger-than-expected access.

Structural Vulnerability of C and C++ The C and C++ programming languages are structurally susceptible to all of the above violation classes [16, 52]. C does not perform bounds checking on array accesses or pointer arithmetic. The expression `buf[i]` is translated directly to a memory access at the address `buf + i * sizeof(*buf)`, with no runtime verification that `i` is within the bounds of the array. Manual memory management through `malloc` and `free` places the responsibility for allocation lifetime on the programmer. There is no language-level mechanism to prevent the use of freed pointers. Pointer arithmetic allows arbitrary pointer values to be computed from existing pointers, including values that point outside the bounds of any valid object. Type casting allows pointers to be reinterpreted as integers and vice versa, which undermines any type-based reasoning about pointer validity. C++ inherits all of these characteristics from C and introduces additional sources of temporal safety violations through its object model. The lifetime of objects with automatic storage duration is tied to scope exit. Pointers or references to such objects can escape their scope through return values, global variables, or callbacks. Move semantics in modern C++ introduce additional opportunities for use-after-move violations when moved-from objects are accessed after transfer of ownership. These language characteristics result in memory safety violations being endemic in systems software written in C and C++. A study of vulnerabilities in the Microsoft Windows codebase found that approximately 70% of all security vulnerabilities addressed over a twelve-year period were attributable to memory safety issues [212]. Similar proportions have been reported for the Chrome browser and for the Linux kernel [125]. The autonomous systems software stacks most relevant to this thesis, including NuttX, PX4, DPDK, and the F-Stack network stack, are written predominantly in C.

2.2.2 Existing Mitigations and Their Limitations

A substantial body of work addresses the memory safety problem through compiler-based mitigations, runtime instrumentation, and language-level enforcement. Each approach offers meaningful protection against specific vulnerability classes, but leaves others unaddressed or imposes costs that are incompatible with safety-critical embedded deployment.

Address Space Layout Randomisation (ASLR) randomizes the base addresses of the stack, heap, and loaded libraries at program startup. This makes it more difficult for an attacker to predict the memory locations that must be corrupted or targeted to exploit a memory safety violation [198]. ASLR, however, does not prevent memory safety violations from occurring; it merely reduces the reliability of exploitation by introducing uncertainty about target addresses. ASLR can be bypassed through information disclosure vulnerabilities that reveal the base address of a mapped region, and is entirely ineffective against vulnerabilities that do not

require knowledge of absolute addresses, such as heap metadata corruption.

Stack canaries insert a randomly chosen value between a function's local variables and its saved return address at function entry and verify its integrity before returning [85]. A stack-based buffer overflow that overwrites the return address must also overwrite the canary. The verification detects the corruption and terminates the program. However, stack canaries only protect against sequential stack-based buffer overflows that overwrite the return address. They provide no protection against heap overflows, non-sequential writes, overflows that target other saved values such as function pointers, or temporal violations.

Control-flow integrity (CFI) restricts the set of valid targets for indirect branch instructions to a statically computed set of legitimate targets, preventing an attacker who has achieved an arbitrary write primitive from redirecting the control flow to arbitrary code [12, 52]. CFI protects the forward edge of the control-flow graph by restricting indirect calls and the backward edge by restricting returns. Coarse-grained CFI implementations, which allow any function to be called at any indirect call site, provide limited protection against return-oriented programming attacks that chain together legitimate code sequences. Fine-grained CFI implementations, which restrict each indirect call to its specific set of valid targets, are more effective but they impose higher overhead and require complete program analysis. This is difficult in the presence of dynamic linking and separate compilation [52].

AddressSanitizer [270] is a compiler-based runtime instrumentation tool that detects spatial and temporal memory safety violations by inserting checks before each memory access and maintaining shadow memory that records the validity of each byte in the program's address space. AddressSanitizer detects buffer overflows, use-after-free violations, use-after-return violations, and double-free violations with high accuracy and low false-positive rates. A drawback is that it imposes a memory overhead of approximately $1 \sim 2\times$ and a runtime overhead of approximately $1 \sim 2\times$, and requires dedicated shadow memory regions that are incompatible with the memory layout of most embedded systems [67]. AddressSanitizer is a valuable development and testing tool but is not intended or suitable for production deployment in safety-critical systems. Memory-safe languages such as Rust enforce spatial and temporal safety through the type system and a compile-time ownership and borrowing model [163]. Rust's ownership system ensures that each value has exactly one owner at any time, that references (*borrows*) are checked at compile time to prevent concurrent mutable access, and that values are not accessible after they have been moved or dropped. These guarantees eliminate use-after-free, double-free, and data race vulnerabilities without introducing runtime overhead. Nevertheless, Rust does not eliminate all memory safety violations. Unsafe blocks allow programmers to bypass the borrow checker, and compiler does not verify the correctness of unsafe code. OSs rely on these unsafe blocks for different purposes,

including low-level memory management and direct hardware access. Furthermore, adopting Rust for an existing codebase requires a complete rewrite of the existing software, which is impractical for mature systems such as PX4/NuttX. The foreign function interface between Rust and C introduces boundaries across which the Rust safety guarantees do not apply. Moreover, hardware device drivers necessarily involve unsafe operations that interact directly with memory-mapped registers and managed by the OS.

2.3 Compartmentalization Techniques

Compartmentalization is an architectural practice that involves dividing a software system into isolated components. This ensures that a security violation or fault in one component cannot propagate to another. A compartmentalized system limits the consequences of a security breach to the compartment in which it occurs. This prevents an attacker who achieves code execution within one component from accessing the data or subverting the behavior of other components. This section surveys the main compartmentalization techniques in order of increasing hardware involvement. It describes how each mechanism works and the isolation properties it provides. Section 2.3.8 discusses their relative suitability for fine-grained compartmentalization in safety critical autonomous systems.

2.3.1 Software Fault Isolation

Software Fault Isolation (SFI) [266] achieves compartmentalization without hardware support by inserting compiler-generated checks before each memory access and control flow transfer. Each compartment is assigned a dedicated memory region, and all memory accesses are masked or checked at compile time to ensure that they cannot access memory outside the compartment's region. Control flow transfers between compartments are mediated through validated trampolines that verify the target address before transferring control. SFI imposes no context-switch overhead because compartments share a single address space and a single privilege level. However, the security of SFI depends entirely on the correctness of the compiler instrumentation. Code that bypasses or subverts the instrumentation, including arbitrary machine code injected by an attacker, is not constrained by the software checks.

2.3.2 OS-Level Capability Frameworks

Operating system capability frameworks provide compartmentalization by wrapping system resources in *unforgeable tokens that explicitly encode the permissions granted*

to their holder. Unlike hardware capabilities, the operating system kernel maintains and enforces these tokens rather than the processor itself. Capsicum [300] is a lightweight capability framework for FreeBSD that augments the standard POSIX file descriptor model with fine-grained permission control. In Capsicum, a capability is a fat file descriptor that encodes both a reference to an underlying kernel object and a set of rights that specify which operations the holder is permitted to perform on that object. A process enters capability mode through the `cap_enter` system call. After entering capability mode, the process is confined to its own namespace and loses access to the global namespace. New capabilities are derived from existing ones through the `cap_new` system call, which wraps an existing descriptor with a subset of the original rights. Crucially, capabilities can only be derived with equal or lesser authority than the creating capability—they cannot be forged or escalated. This monotonicity property addresses the confused deputy problem [143], in which a privileged program is manipulated by an unprivileged caller into performing operations on resources that the caller is not authorized to access directly. Capsicum requires existing applications to be modified to adopt the capability API. However, the necessary changes for complex applications, such as the Chromium browser, have been shown to be manageable [300].

2.3.3 Microkernel-Based Compartmentalization

Microkernels represent a fundamentally different architectural approach to compartmentalization. Rather than executing operating system services in a privileged kernel context, a microkernel minimizes the trusted computing base to the minimum code necessary for inter-process communication (IPC), scheduling, and memory management. It also implements all other services, including device drivers, filesystems, and network stacks, as unprivileged user-space processes. Each service runs in its own isolated address space and communicates with other services and applications via a capability-mediated IPC mechanism enforced by the microkernel. Therefore, a compromise of any individual service is contained within its address space and cannot directly affect the kernel or other services. seL4 [130] is a microkernel of the L4 family that provides formal guarantees for selected hardware combinations of isolation and functional correctness. Every resource in seL4 is accessed through a capability, which is represented as a structured pointer to a kernel object stored in a Capability Node (CNode). A thread's Capability Space (CSpace) is a directed graph of CNodes that defines the complete set of resources the thread is authorized to access. Capabilities can be copied and delegated to other threads through inter-process communication, thereby transfering authority with equal or lesser rights. The seL4 microkernel has been formally verified for selected architectures and compiler combinations using the Isabelle/HOL theorem prover. This establishes a machine-checked proof that the kernel implementation

correctly implements its formal specification, including the absence of capability leaks or privilege escalation within the kernel itself. CAmkES [268] is a component framework that generates seL4 system configurations from a high-level architectural description, simplifying the construction of compartmentalized systems from individually verified components. Fiasco.OC [175] is an alternative L4 microkernel that provides software capability-based isolation through a user-level thread control block (UTCB) communication model. Each thread's UTCB contains message registers, buffer registers, and thread control registers that are used to pass capabilities and data between threads. Like seL4, Fiasco.OC enforces that kernel objects are accessible only through valid capabilities, which prevents unauthorized inter-thread communication. Fiasco.OC forms the basis of the L4Re operating system environment and supports hardware virtualization through the uvmm virtual machine monitor. This allows Fiasco.OC to host guest operating systems alongside natively running L4Re components. Both microkernel approaches share a structural characteristic that distinguishes them from hardware capability approaches: the isolation mechanism is software-enforced by the microkernel, which must be both correct and trusted itself.

In principle, microarchitectural attacks such as Spectre (CVE-2017-5753, CVE-2017-5715) can leak capability values across isolation boundaries through cache timing side channels. Covert channels exploitable through the memory allocation mechanism have been demonstrated in L4-family kernels. Furthermore, the performance of microkernel-based compartmentalization is limited by the cost of inter-process communication [137]. A service request that crosses a microkernel boundary requires at least two context switches and two privilege level transitions, introducing latency that accumulates with frequent, fine-grained compartment crossings.

2.3.4 Isolation-Agnostic Compartmentalization Frameworks

FlexOS [179] is a library operating system built on the Unikraft [171] unikernel framework. It abstracts the compartmentalization mechanism from the compartmentalized application. The compartment boundaries and their associated isolation mechanisms are determined at build time through a configuration file that specifies which libraries and services belong to each compartment and which backend mechanism enforces the boundary. Supported backends include Intel Memory Protection Keys (MPK) for lightweight intra-process isolation, Extended Page Tables (EPT) for VM-level isolation, and Intel SGX for trusted execution environments. The build toolchain automatically generates cross-compartment calls using semantic patching, and the configuration explictly declares shared variables between compartments. FlexOS shows that a common compartmentalization API can be implemented over heterogeneous hardware backends. However, the build-time commitment to

compartment layout means that compartment boundaries cannot be modified at runtime, and the security guarantee is only as strong as the underlying backend mechanism provides.

2.3.5 Process-Based Isolation

The standard operating system mechanism for compartmentalization is the process model, which uses the MMU to enforce isolation between processes at the level of virtual address spaces. The MMU enforces memory isolation by maintaining per-process page tables that map virtual addresses to physical addresses and attach permission bits to each page. A process running in user space cannot access the physical memory of another process because the MMU will not produce a valid translation for a page it does not own. Crossing a process boundary requires a system call, involving a privilege level transition from user space to kernel space and back again. On contemporary processors, this transition takes hundreds to thousands of nanoseconds and introduces timing variability [211, 275]. For fine-grained compartmentalization in which individual module calls or driver accesses cross isolation boundaries frequently, the accumulated overhead and its variability impose a significant cost on the system's temporal behavior.

2.3.6 Virtualization-Based Isolation

Virtualization extends the process isolation model to entire operating systems. Hypervisors are commonly classified as *type-1*, which execute directly on physical hardware without an underlying host operating system, or *type-2*, which execute on top of a host operating system. Some microkernel-based systems can also provide virtualization (e.g., seL4 and PikeOS). For safety- and mixed-criticality systems, static partitioning hypervisors are of particular interest. They partition resources statically at configuration time rather than managing them dynamically. Each virtual machine receives a statically allocated subset of physical hardware with no resource overcommitment or dynamic reallocation. This deterministic resource assignment simplifies timing analysis, while the small software footprint of partitioning hypervisors facilitates certification.

Jailhouse: a static partitioning hypervisor. Jailhouse [92] is an open-source, static partitioning hypervisor and is used here to illustrate the key concepts and mechanisms. Each isolated execution environment in Jailhouse is called a cell. The root cell hosts the Linux instance from which Jailhouse was loaded. Non-root cells host guest operating systems or bare-metal applications and receive only the hardware resources explicitly allocated to them in their cell configuration. Jailhouse enforces isolation between cells using the stage-2 MMU on ARM processors. The

stage-2 MMU translates the intermediate physical addresses presented by guest operating systems to actual physical addresses, thereby enforcing memory allocation at the hardware level. In a static partitioning hypervisor, communication between cells cannot occur through shared hardware peripherals in the conventional sense since each peripheral is exclusively owned by one cell. Jailhouse provides an Inter-VM Shared Memory (IVSHMEM) device that enables controlled shared memory communication. The IVSHMEM device presents each attached cell with a virtualized PCI device that exposes a set of shared memory regions, each with configurable read and write-permissions per cell. A cell is granted read-write access to its own output region and read-only access to the output regions of other cells. This prevents one cell from overwriting another's data. The IVSHMEM device also provides an interrupt signaling mechanism allowing one cell to notify another of pending data. These properties and their limitations are directly relevant to the discussion of isolation granularity in Chapter 5.

2.3.7 Hardware-Assisted Intra-Process Isolation

Several hardware mechanisms have been introduced in recent processor architectures to support intra-process compartmentalization without the overhead of process boundary crossings. Intel MPK, also known as Protection Keys for Userspace (PKU), add a 4-bit protection key field to the x86 page table in each page table entry [276]. A per-thread register associates each of the 16 possible key values with read and write disable bits, allowing a thread to change its access permissions to groups of pages without a system call, at a cost of approximately 20 nanoseconds. CubicleOS [260] is a library operating system that demonstrates the application of MPK to intra-process compartmentalization. CubicleOS assigns a dedicated set of pages protected by MPK to each software component, referred to as a cubicle. Components interact through three mechanisms: direct calls for synchronous invocations, windows for explicitly shared memory slices between two cubicles, and cross-cubicle calls that switch the MPK register to grant the callee access to its own pages before invoking the target function. MPK-based isolation is limited to 16 distinct protection domains and its permissions can be modified by any code executing in user space. This means that code executing within a compartment can in principle grant itself access to other compartments' memory by modifying the protection key register. ARM Pointer Authentication Codes (PAC) [183] add a cryptographic authentication code to the unused high bits of pointer values, allowing the processor to detect pointers that have been modified by an attacker. PAC protects the integrity of return addresses and function pointers against corruption by memory safety violations, but it does not enforce bounds on memory accesses or prevent the use of legitimate pointers to access memory outside their intended bounds.

2.3.8 The Granularity-Overhead Tradeoff

The compartmentalization techniques surveyed above can be organized along two dimensions: the granularity of the isolation boundary and the cost of crossing it. Microkernel-based approaches provide strong isolation between services, but incur IPC overhead that increases with frequency of crossing. Process-based isolation provides hardware-enforced address space separation, but incurs the cost of system call transitions. Virtualization-based isolation provides the strongest spatial guarantees, but with the coarsest granularity and with the highest communication cost. Hardware-assisted intra-process mechanisms, such as MPK and PMP, substantially reduce crossing overhead but are limited in the number of protection domains they support, the flexibility of domain boundaries, and the privilege required to modify domain configuration. Software-based approaches, such as SFI, and isolation-agnostic frameworks, such as FlexOS, eliminate crossing overhead but either provide no hardware enforcement or inherit the limitations of their backend mechanisms. This trade-off is particularly acute in safety-critical autonomous systems. The software architecture demands compartmentalization at the granularity of individual modules within a real-time operating system. Crossing frequency is high because modules communicate continuously through a publish-subscribe bus. For real-time schedulability analysis to remain valid, the crossing overhead must be bounded and predictable. None of the mechanisms surveyed in this section satisfy all three requirements simultaneously: hardware enforcement at a fine granularity, an unbounded number of protection domains, and crossing overhead compatible that is compatible with high-frequency use under hard real-time constraints. Capability-based isolation, as provided by CHERI, is designed to address this gap. It enforces bounds at the granularity of individual memory allocations. It also supports an arbitrary number of compartments within a single address space, and performs compartment crossings entirely in user space, without a privilege level transition. The technical details are explained in Chapter 3.

Chapter 3

CHERI: A Comprehensive Survey

This chapter provides a comprehensive survey of CHERI, covering its fundamental concepts, hardware implementations, software ecosystem, and applications. Within this thesis, it serves two functions. The first is foundational. Section 3.1 introduces the technical concepts and vocabulary of CHERI capabilities necessary to understand the systems contributions in subsequent chapters. The second is contextual. The remainder of the chapter surveys the CHERI ecosystem, including its hardware and software support, existing applications, and open research challenges. This overview establishes the context for the contributions presented in this thesis and highlights the areas where CHERI remains insufficiently explored.

Capability systems have a long history predating CHERI, with early systems exploring hardware-enforced protection through unforgeable capabilities and protected object references. Subsequent systems explored different designs for capability representation, delegation, and revocation, including both hardware and software-enforced approaches. These systems established the design space from which modern capability architectures emerged. A detailed historical discussion of capability systems and a comparison of their main design dimensions are provided in Chapter A. Here, we focus on CHERI and the mechanisms relevant to the systems studied in this thesis.

3.1 CHERI Protection Model

CHERI enhances ISAs with capability-based primitives to enhance security and prevent vulnerabilities. Its technology introduces two fundamental primitives: (i) fine-grained (byte-level) protection using capabilities that act as fat-pointers, encoding bounds and permissions to restrict memory accesses at the instruction level within a single, shared address space; and (ii) secure code encapsulation based on capabilities that enforce single-address-space software compartmentalization [221].

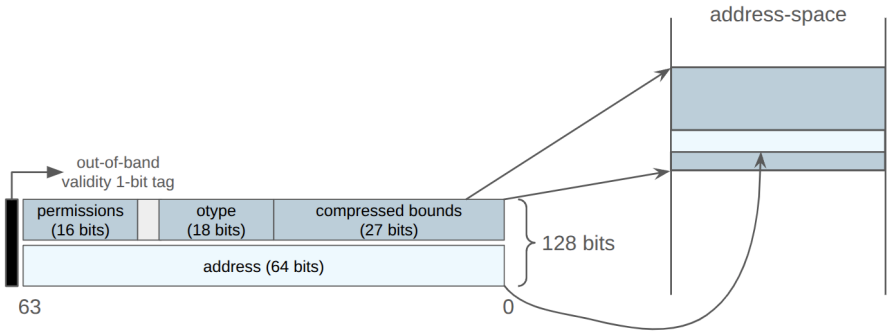


Figure 3.1: CHERI hardware capability 128-bit format representation in memory (simplified).

Table 3.1 outlines the core principles of CHERI, including the composition of the capability, core properties, hybridization and sealing mechanisms, and how potential capability misuses are handled.

3.1.1 Capability Composition, Core Properties, and Misuse Handling

In CHERI, capabilities are unforgeable (*integrity*) tokens of authority through which programs access all memory and services within an address space. Figure 3.1 depicts a possible CHERI capability for a 64-bit architecture. These fundamental ISA-defined hardware types can reside in memory or registers and replace traditional pointers. Memory is represented with robust primitives composed of an address and associated metadata, such as bounds and permissions. Additionally, a dedicated 1-bit validity tag (*validity bit*), stored in the system tagged memory, indicates whether a memory location holds a valid capability (if set to 1), otherwise, the value is interpreted as ordinary data. This ensures that ordinary data cannot be misinterpreted or misused as a capability (see Section 3.2.3). At system boot, only one powerful *primordial capability* with full permissions and memory exists. All other subsequent capabilities are derived from this primordial capability (*valid provenance*), with reduced or equal privileges (*monotonocity*). Consequently, memory accesses are constrained by the bounds (*bound checking*) and permissions (*permissions control*) of the derived capability. *Misuse of capabilities* triggers hardware exceptions, invalidating them or nullifying their tag bit.

3.1.2 Hybridization

CHERI follows a hybrid protection model that complements traditional isolation techniques (like Memory Management Unit (MMU)-based isolation) rather than

Table 3.1: Basic principles of the CHERI ISA, followed by a brief description and some additional details.

Principles	Description	Details
Capability Composition	Address + metadata (bounds, permissions, *otype*) + 1 out-of-bound tag bit for integrity	Size: 2× natural ISA address size + 1 tag bit
Core Properties	*Capability Integrity*: ensures that capabilities cannot be tampered without invalidation	Prevents attacks like capability forgery and unauthorized privilege elevation
	Bound Checking: enforces only in-bound operations	Prevents buffer overflows, stack corruption, and control-flow attacks
	Permissions Control: ensures that operations within a memory region are limited	"perm" field represents a bitmask where each bit provides a specific permission (such as load, store, and execute)
	Valid Provenance: requires all capabilities to be derived only from valid parent capabilities	Prevents the creation of invalid capabilities
	Monotonocity: guarantees that new capabilities inherit privileges $\leq$ to their parent	For example, transforming a read-only capability into a read/write capability is not allowed
Misuse Handling	Hardware Exceptions	Invalidates the capability and/or its tag bit
Hybridization	*Virtual Address*: complements the MMU role providing finer-grained memory protection	MMU for inter-process isolation $\cup$ CHERI in-address-space isolation
	Pointer: supports both legacy and CHERI-aware code	*hybrid-cap* and *pure-cap* modes
	Special registers: introduction of special registers to support hybridization	Such as Default Data Capability (DDC) and Program Counter Capability (PCC)
Sealing Mechanism	Enforces immutability and prevent unauthorized modification, dereferencing, or jumps	Object-based compartments VS *Sentry* capabilities

replacing them. The protection model supports both legacy and CHERI-aware code through special capability registers for memory and execution control. These registers include the Default Data Capability (DDC), which defines the boundaries for memory access in non-capability code, and the Program Counter Capability (PCC), which manages control flow for code execution. The introduction of capabilities and these special registers changes how pointers and function calls are represented, motivating a new application binary interface, the *CheriABI* [91],

which defines the memory layout, calling conventions, and register usage. CheriABI-aware compilers, e.g., Clang/LLVM, allow code to be compiled in (i) *hybrid-cap* mode, where traditional pointers coexist with CHERI capabilities or in (ii) *pure-cap* mode, where all pointers are replaced with capabilities, including implied pointers (e.g., stack pointer and return address). In hybrid-cap mode, programmers annotate in the code which pointer the compiler should treat as a capability (through the keyword `__capability` [304]). This design decision enables incrementally adopting CHERI by progressively adapting and recompiling existing codebases.

3.1.3 Sealing Mechanism

The *otype* field in the CHERI ISA metadata enables the *sealing* capabilities mechanism [297, 302]. This makes a capability *immutable*, i.e., unauthorized modifications, dereferencing, or jumps invalidate the capability and raises a hardware exception. Without this mechanism, CHERI would not be able to support robust software compartmentalization in a single address space. The CHERI ISA supports two categories of sealed capabilities that allow non-monotonic operations: (i) Object Capability, sealed code and data capabilities that enable software compartmentalization within a single-address-space; and (ii) Sentry Capabilities, sealed code capabilities that serve as an entry point for atomic control-flow transition via jump instructions. Object capabilities involve a pair of sealed capabilities (one for code, one for data) with a software-defined *otype*, which can only be atomically unsealed and invoked via specific control-flow instructions (*CInvoke* instructions on RISC-V and *BLRS* on Arm Morello, see Section 3.2.1). Upon invocation, the unsealed code capability is loaded into the PCC register and the unsealed data capability is loaded into the DDC register. This allows compartments within a single-address-space to perform domain transitions without requiring supervisor (e.g., kernel) intervention. Sentry capabilities provide a secure way to jump to a function entry point in another compartment via standard jump instructions. Unlike object capabilities, sentries do not carry an associated data capability and are unsealed automatically during the jump operation. For secure inter-compartment data sharing, compartments can use sealed data capabilities: a capability sealed with a particular *otype* can only be unsealed (*CUnseal*) and dereferenced by a holder of the matching unsealing capability.

The ISA enhancements (such as newer instructions to manage the capabilities) provided by CHERI are the foundation of secure computing stacks. However, for applications to fully benefit from the CHERI approach, intermediate layers such as compilers, OSs, runtime systems, and libraries must be adapted to leverage hardware capabilities and provide enhanced security.

Figure 3.2 depicts a pyramid chart where the position of each component reflects

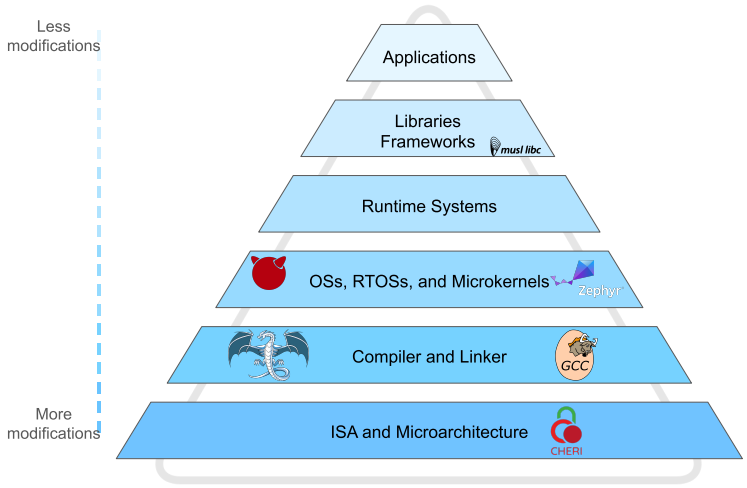


Figure 3.2: Pyramid chart illustrating CHERI impact across the computing stack, from the hardware layer requiring extensive architectural and microarchitectural modifications to the application layer, which requires minimal adjustments.

the extent of modifications required to support CHERI: the hardware, which requires most significant modifications, is placed at the bottom, while applications, which require the fewest changes, are positioned at the top.

The following sections describe the state-of-the-art approaches for each layer, presented in a bottom-up fashion, starting from the CHERI hardware and progressing through the higher layers of the computing stack.

3.2 CHERI Hardware

CHERI is an ISA-agnostic protection model, meaning it can be mapped to multiple ISAs. Different ISAs require specific modifications to support CHERI capability model, including introducing new registers, instructions, and tagged-memory support. Moreover, other components like MMU, cache management, and exception handling require modifications. For instance, the MMU requires newer page-table permission bits for loading or storing capabilities. Currently, the CHERI ISA supports all mainstream ISAs (e.g., RISC-V, Arm, x86, and MIPS).

3.2.1 CHERI-Aware ISAs

CHERI-MIPS was the first instruction set architecture to integrate the CHERI protection model in 2013. This enabled researchers to explore capability semantics and refine the CHERI protection model. For example, the synchronous domain call

CInvoke [306] was implemented in CHERI-MIPS to replace the exception-based mechanism used for cross-domain calls (pair of instructions *CCall* and *CReturn*). *CInvoke* requires no trusted supervisor for cross compartment calls and removes kernel complexity, while reducing the runtime overhead. Such design decisions influenced the development of subsequent extensions to the RISC-V and Arm ISAs, specifically CHERI-RISC-V [306] and *Arm Morello* [184].

Over the last years, MIPS has seen a significant decrease in its presence in research and industry. In CHERI ISA version 9 [306], the University of Cambridge officially replaced CHERI-MIPS with CHERI-RISC-V as the primary reference architecture, reflecting the current industry trends. Several variants of CHERI-RISC-V have contributed to its evolution. For example, CHERIoT [29] is a fork of the CHERI-RISC-V ISA tailored for small embedded devices (32-bit systems). It simplifies the architecture at the cost of backward compatibility by implementing a pure capability system. Notable features include: (i) improved permission encoding, (ii) interrupt-guarded control flow, and (iii) a mechanism for temporal memory safety control.

As of this writing, RISC-V is in the process of standardizing a collection of CHERI extensions [17] (*ZCheri extensions*), which incorporate feedback from various stakeholders, such as academia and industry. This effort includes merging the CHERI special capability registers with the privilege registers in RISC-V.

In the Arm architecture, the *Arm Morello* ISA [184] has laid the basis for the future adoption and growth of CHERI in mainstream commercial processors, potentially influencing broader industry standards. Several features added to the CHERI ISA by the *Arm Morello* ISA include a compartment ID permissions bit and register, aimed at minimizing the side-channel attack surface in speculative execution resources (e.g., branch predictors). This ISA ties capabilities to specific compartments at the microarchitectural level, preventing information leakage between compartments. Lastly, while RISC architecture predominates in the CHERI landscape, a preliminary version of the x86-64 bit CHERI extension has emerged from CHERI ISA version 7 [303]. In this draft, the authors propose extending the general-purpose registers of x86 to serve as capabilities [173].

3.2.2 Capability Compression

Defining an efficient method to encode the capability meta fields in memory and registers, such as the capability memory bounds, without hindering the deployment of CHERI technology has been a key focus in the CHERI literature [161, 317]. Early CHERI implementations, such as the CHERI-BERI CPU [55], encoded the capability metadata (bounds, address, permissions and type) into separate 64-bit words bounds, making a capability 4× the size of the architecture word size (see Figure 3.3). While CHERI-256 offers low-latency bounds checking and

easier pointer arithmetic, it significantly impacts CPU datapaths and memory usage [317].

Inspired by the Low-fat encoding scheme [174], Woodruff et al. [317] proposed *CHERI Concentrate* as an alternative scheme for efficient capability encoding that represents capabilities bounds in a floating-point format relative to the current address. This requires only 2× the architecture word size. The format shown in Figure 3.1 is denoted CHERI-128 for 64-bit systems. This encoding integrates efficiently with RISC single-cycle instruction set architectures and has become the de facto standard for capability compression in 64-bit CHERI processors [60, 66]. To mitigate the performance overhead associated with fully decoding bounds information under this scheme, CPUs typically employ a range of microarchitectural optimizations. CHERI implementations use multiple internal capability representation formats throughout the microarchitectural pipeline and progressively decode capabilities from their compressed in-memory format to a partial or fully decoded capability, as required at different stages of the pipeline. Performance-critical operations (such as pointer arithmetic) operate on only partially decoded state, primarily the cursor and the compressed bounds metadata, combined with single-cycle representability checks that act directly on the encoded capability representation. Full bounds decompression and verification is deferred to later pipeline stages or exception-handling paths, where precise bounds checks are performed off the critical path.

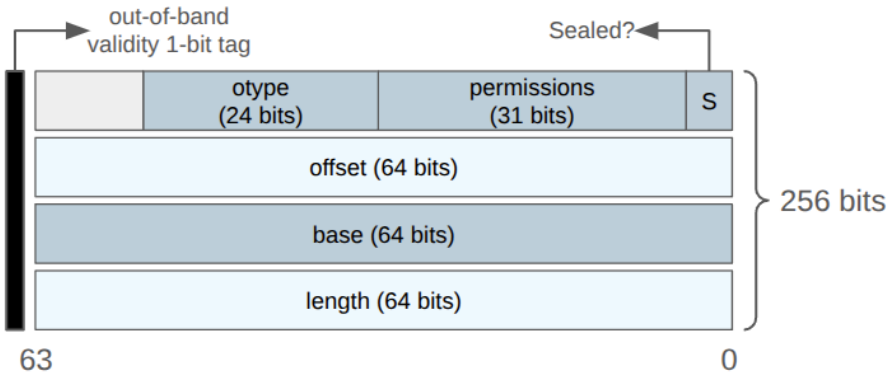


Figure 3.3: CHERI hardware capability 256-bit format representation.

A later study in the CHERIoT [29] identified inefficiencies in CHERI-64 scaled-down version for embedded systems due to precision loss and memory fragmentation issues, reaching up to 12.5% overhead. CHERIoT refined the CHERI-64 scheme by compressing permission bits, reducing object types, and improving precision, minimizing fragmentation to 0.19%. CHERIoT has been used to refine the *CHERI Concentrate* for RISC-V 32-bit CHERI standard specification.

Though most of the research on the CHERI concentrate compression scheme focuses primarily on RISC architectures, there is some work that formalizes the CHERI concentrate in ACL2 [166] for x86 CHERI architecture [173].

3.2.3 Capability Tagging

Efficiently handling and storing tags in memory requires special handling. In CHERI systems, there are two fundamental approaches to storing tags in memory: (i) re-purpose the Error-Correction Code (ECC) bits on DRAMs to store the tag bits, which is expensive and often not supported on low-power RAMs, and (ii) use a shadow bitmap table stored in a reserved area of memory on the DRAM, which requires a table lookup on every memory access and extra logic to update the tag table [161, 162]. Other solutions (not viable for production environments) are available, such as storing the tags in dedicated memory or distributing the cache tags per page [162]. The study conducted by Joannou et al. [162] implemented and characterized an efficient way to store tag bits for capabilities. Specifically, they proposed storing tag capabilities in an in-memory shadow table and tag cache near the memory. Tags are then fetched and updated in parallel to memory operations, ensuring that the validity of capabilities is checked and maintained simultaneously during memory accesses. For these scenarios, results show minimal overhead. Joannou et al. also defined a hierarchy scheme to compress tags based on the assumption that not every page will store capabilities.

Table 3.2: Public Implementations of CHERI CPUs. For each CPU, we provide the following information : (i) name of the CPU and reference to the open-source implementation or other relevant sources; (ii) ISA; (iii) microarchitecture ISA; (iv) virtual memory support (MMU); (v) CHERI hybrid execution modes support; (vi) when available: performance (CoreMark benchmark or SPECInt 2006), area (FPGA LUTs or gates), and power overheads; and (vii) license (Apache version 2 (APv2), MIT, BSD, Solderpad Hardware (SP) or Commercial (C)).

Name	ISA	μArch	MMU	Hyb	Target	Evaluation		Lic.
						Performance	Area/LUTs	
CHERI Microcontrollers								
CHERI-Ibex [61] (2022)	RISC-V (RV32IMCB + M-mode + CHERIv8)	in-order 2- or 3-stage	☐	☑	FPGA	N/A	N/A	APv2
CHERI-Piccolo [62] (2018)	RISC-V (RV32IACDFM + M/U-mode + CHERIv8)	3-stage in-order	☐	☑	FPGA	CoreMark/MHz 4.1(10%) [256]	VCU-118 FPGA +62% [256]	BSD
Proteus [48] (2023)	RISC-V (RV32IM + M-mode + CHERIv8 (no compressed capabilities)	5-stage in-order or superscalar OoO	☐	☑	FPGA	N/A	Zynq Ultra-scale+ XCZU9EG-2FFVB1156 FPGA +2.94 [48]	MIT
CHERIoT-Ibex [209] (2023)	RISC-V (RV32IMCB + M/U-mode + CHERIoT ISA)	2 or 3-stage in-order + HW-assisted revocation	☐	☐	FPGA & ASIC	CoreMark/MHz 1.811(13.18%), worst case. perf 1.624 (21.18%) with load-filter for capability revocation [30]	TSMC 28nm HPC+ technology running at 300 MHz shows 2.15× the gates and 1.79× the power & max 5% in gates and power for HW-assisted revocation [30]	APv2

Continued on the next page

Name	ISA	μArch	MMU	Hyb	Target	Evaluation		Lic.
						Performance	Area/LUTs	
CHERI Application Class Processors								
BERI [55, 301] (2014)	64-bit MIPS + CHERI	5-stage in order + CHERI coprocessor	☑	☑	FPGA	N/A	Altera Stratix IV EP4SGX230 FPGA +27% overhead (CHERI coprocessor + tag controller) [315]	APv2
Toooba [66] (2019)	RISC-V (RV64GC + CHERIv7)	superscalar OoO	☑	☑	FPGA	CoreMark/MHz 5 (-8%) [256]	VCU-118 FPGA +45% [256]	MIT & APv2
CHERI-Flute [60] (2019)	RISC-V (RV64GC + CHERIv8)	5-stage in-order	☑	☑	FPGA	CoreMark/MHz 3.6 (5.2%) [256]	VCU-118 FPGA +50% [256]	APv2
A76 (Arm Morello) [39] (2019)	Armv8-A + CHERIv8	OoO superscalar 13-stage	☑	☑	ASIC	SPECint 2006 benchmark running at 2.5 GHz: 5.70% (microarch. and ABI optimizations) and worst case perf. 28.01% (unmodified) [305]	<6% of the overall SoC [305]	C
A730 CHERI [73] (2023)	RISC-V (RV64G RISC-V Standard (ZCheri))	dual-issue, in-order, 7-stage	☑	☑	FPGA & ASIC	N/A	+4% [73]	C
CVA6 CHERI [334] (2024)	RISC-V (RV64GC + M/**HS**/VS/VU-modes + CHERIv9)	6-stage in-order	☑	☑	FPGA	N/A	+53.9%LUTs and +43.2%FFs compared to the baseline targetting Genesys2 FPGA running at 50MHz [334]	SP

3.2.4 CHERI CPUs, SoC, Platforms

Several actors provide reference implementations of CHERI CPUs targeting different classes of applications. In this respect, the University of Cambridge is particularly active. In Table 3.2, we summarize all publicly available CHERI CPU implementations categorized into two types of processor classes: (i) small-scale microcontrollers (MCUs) for embedded applications with real-time requirements and no virtual memory support and (ii) application-class processors (APUs) capable of running full-fledged OSes with virtual memory support.

CHERI Microcontrollers Several CPU implementations target the MCU space. Piccolo [62] and Ibex [61] are representative examples of 32-bit RISC-V MCUs with CHERI extensions. Both cores support the execution of CHERI hybrid code to maintain legacy support. CHERIoT-Ibex [29, 30, 209] is another 32-bit CHERI-enabled processor that targets IoT devices. As part of the Portmeirion project [210] and based on the Ibex core, the Microsoft team characterized the challenges of porting CHERI technology to an MCU space, specifically for IoT applications. This resulted in a fork (*CHERIoT ISA* [11]) of the original Cambridge CHERI ISA. The CHERIoT-Ibex [29] is a pure capability core with no CHERI integer mode support, i.e., it can only run pure-capability code. Proteus [48] is another highly configurable 32-bit CHERI-RISC-V processor implemented in SpinalHDL.

CHERI Application Processors Many application cores that have been extended to support the CHERI technology. Except for the *Morello*-A76 processor [39], most implementations run as soft cores through FPGA emulation. BERI [55, 301] was the first implementation of CHERI technology in MIPS-like ISA. The processor implements an in-order microarchitecture pipeline and uses a dedicated coprocessor for the CHERI ISA extension. CHERI-Flute [60] was the first RISC-V 64-bit application class processor to support the CHERI-RISC-V ISA version 8 and to be formally verified [222] at the University of Oxford. Toooba [66] is another representative example of a high-performance RISC-V 64-bit out-of-order core with CHERI support. Moreover, Sá et al. [334] have extended the CVA6, a highly configurable processor with hypervisor extension support [281], with the CHERI version 9 extension from Cambridge [306]. Table 3.2 shows a predominance of CHERI-RISC-V implementations with permissive licenses (APv2[1] and SP[2]) stemming from the open-standard nature of its ISA and the strong open-source community.

From an industrial perspective, Arm has contributed to the maturation of CHERI through the *Morello*-A76 processor, which implements CHERI extensions

[1]https://www.apache.org/licenses/LICENSE-2.0
[2]http://solderpad.org/licenses/

on a 64-bit Arm processor with MMU support and is deployed on the *Arm Morello* board [185]. Codasip has also developed CHERI support for RISC-V and launched the A730 [73], a commercially available CHERI CPU for mid-end applications implementing the RISC-V *ZCheri* extensions [17] with hybrid-mode support.

CHERI-enabled Platforms The *Arm Morello* board [134, 184, 185] is the first industrial prototype CHERI-enabled Arm System-on-Chip. It features four Arm application processors (i.e., Cortex-A76), multiple peripherals, and a complete memory subsystem with L1 and L2 caches, accompanied by a full software stack (e.g., toolchains) and a QEMU platform emulator. It aims to evaluate and gather feedback from stakeholders worldwide to promote the adoption of CHERI in real-world applications and to advance the maturity of the CHERI ecosystem.

The *Sonata* FPGA board [190, 192] is an open-source RISC-V CHERI platform for embedded systems based on the Microsoft CHERIoT processor. It includes typical embedded peripherals and an on-chip memory subsystem, together with the CHERIoT software stack, FPGA emulation support, and open-source hardware designs [191]. The Sunburst chip [193, 195] is an open-source MCU chip design based on the Sonata system.

CHERI Overhead on Performance, Area and Power. Measuring performance for CHERI prototypes is challenging because platforms differ in architecture and implementation. Moreover, available evaluations are limited and often rely on synthetic benchmarks executed on FPGA-based prototypes. Such benchmarks may not fully exercise critical microarchitectural components such as caches and TLBs. Table 3.2 therefore summarizes reported performance, area, and power overheads as a baseline for understanding the implications of deploying CHERI in real systems. An analysis of reference CHERI-RISC-V implementations [256], including Piccolo, Flute, and Toooba, reports resource increases of 45%–62% in post-synthesis LUTs and FFs. The authors suggest that the relative hardware impact decreases as core size increases, since unaffected components such as the floating-point unit dilute the additional CHERI logic [256]. Similar trends were observed in CVA6H-CHERI [334]. In more mature cores, such as Morello A76, Codasip A730, and CHERIoT-Ibex [30], the reported impact is less than 6% of the overall gates [29, 73, 305]. From a performance perspective, CHERI introduces overheads particularly in pure-capability mode. CHERI requires additional instructions to manage capabilities, such as setting bounds and permissions, as well as operations required by CheriABI. As shown in Table 3.2, reported slowdowns range from approximately 5.2% to 28%. CHERIoT reports a 13.18% performance overhead, primarily due to the narrower bus used for loading and storing capabilities, which requires two accesses. The worst-case configuration, using hardware-assisted capability revocation based on

load-barriers, increases load-to-use latency and pipeline stalls, resulting in a 21.18% overhead. CHERI-Toooba is an exception, as its CheriABI has fewer jump instructions, reducing branch misprediction rates [256]. For *Arm Morello*, performance overhead depends strongly on both the ABI and microarchitectural configuration. The default FPGA prototype incurs approximately 28% overhead, primarily due to limitations in branch prediction, data-dependent exception-induced pipeline stalls, and increased store-queue pressure. These overheads can be reduced to approximately 5.70% through ABI optimizations, such as mitigating branch stalls, and microarchitectural optimizations, such as enlarging store queues.

3.3 CHERI Software

While it is true that CHERI relies on specialized hardware, it is equally important to emphasize that modifications to the software architecture are required to fully take advantage of its benefits. Over the years, several software efforts have been driven towards achieving complete CHERI-based systems. This section introduces and explores the diverse research fields of interest related to CHERI. We continue to follow the bottom-up approach illustrated in Figure 3.2.

3.3.1 Compilers and Linkers

Since the inception of CHERI technology, CHERI Clang/LLVM and LLD [152, 297] have become the facto compiler and linker for CHERI-aware platforms, including CHERI-MIPS, CHERI-RISC-V, and CHERI-Arm (*Arm Morello*) targets. Key modifications to the original Clang/LLVM implementation required to support CHERI include: (i) the generation of suitable code to ensure capability properties, such as accurate bounds for stack allocations, (ii) the introduction of new C-language qualifiers for capabilities, (iii) the incorporation of capability-aware optimization, and (iv) the support of pure and hybrid capability modes (see Section 3.1 for details). CHERI Clang/LLVM also supports the Rust programming language (see Section 3.3.2). An interactive online resource is also available [285], allowing the users to experiment with CHERI Clang/LLVM by exploring code snippets and viewing the compiled assembly.

CHERI Clang/LLVM served as the foundational toolchain for various research projects within the CHERI literature [15, 283]. For example, Tsampas et al. [283] developed a source-to-source compiler to automate compartmentalization in CHERI C code for CheriBSD, moving beyond manual code annotation. Similarly, El-Korashy et al. [104] created a preprocessing tool [28] to automatically preprocess and annotate CHERI C source codes to define compartments. In addition to Clang/LLVM, other compilers have been adapted for CHERI, notably Arm's extended GCC [40]

part of the GNU toolchain, which targets the Morello platform. This work is still in development. Additionally, Thibault et al. introduced SECOMP [15], an extension of the formal verified CompCert RISC-V [14] compiler for safety-critical applications, incorporating CHERI technology and experimental features for the CHERI ISA version 8.

3.3.2 Programming Languages and Tools

CHERI-C/C++ CHERI-C/C++ is a variant of the traditional C that uses CHERI capabilities to represent pointer types (e.g., pointers, return addresses, variables, and arrays). The adoption of CheriABI invalidates several assumptions commonly made by legacy software and libraries. In CHERI, programmers cannot freely cast between pointers and integers as in conventional C; such operations are restricted to prevent forging capabilities. Specifically, storing a capability as an integer (e.g., using `intptr_t`) results in the loss of its metadata and tag, making it unusable as a capability unless explicitly reconstructed and revalidated. Beyond pointer-integer conversions, CheriABI introduces additional constraints that affect legacy code migration. The increased pointer size (e.g., 128/64 bits for capabilities versus 64/32 bits for traditional pointers) can break data structure layouts, serialization formats (e.g., converting data structures to/from byte streams), and inter-process communication protocols that assume fixed pointers sizes. Capabilities also require strict alignment (typically 16-byte alignment for 64 bits architecture) which can cause issues with packed structures, custom memory layouts, and code that assumes pointers can be stored at arbitrary addresses (e.g., device drivers). CHERI-aware compilers can detect many of these issues at compile time, including invalid pointer-integer casts, alignment issues, and some instances of capability misuse. However, certain issues remain difficult to catch statically and may only manifest at runtime, such as unaligned capability stores in dynamically allocated or externally provided memory and provenance violations in complex pointer manipulation chains. Memory allocators and custom allocation schemes must be capability-aware to properly propagate bounds and permissions while respecting alignment requirements. Furthermore, optimization patterns, such as pointer arithmetic beyond object bounds, become invalid or require significant redesign. System calls and library interfaces that pass pointers through opaque integer types must be refactored to preserve capability provenance throughout the call chain. While these restrictions demand effort during the transition to CheriABI, compiler diagnostics significantly aid in identifying and resolving compatibility issues, with the remaining runtime concerns typically centered around dynamic memory operations and cross-boundary communication [90, 91, 102, 251, 308]. We point readers to an (i) official document provided by University Cambridge with the CHERI-C/C++ programming model [304] and (ii) a set of tutorials targeting

QEMU and CheriBSD to get started with CHERI-C/C++ and CHERI in general [229]. Moreover, the Clang Static Analyzer (CSA) has been extended with checkers that can identify issues such as copying capabilities through under-aligned memory or a tag-stripping copy of capability [101, 102].

Cerberus [56, 205–207] is an open-source tool that provides formal reasoning on the C memory model. Recent works have extended Cerberus to include CHERI-C semantics [331, 332], enabling the formalization of CHERI-C memory models to ensure program correctness and memory safety, while also formalizing the differences between CHERI-C and standard C and bridging the gap between CHERI-C capabilities and standard C usage. Another tool is Efficient SMT-based Context-Bounded Model Checker (**ESBMC**) [81, 107], which is an open-source bounded model checker tool for formally verifying C/C++, Kotlin, Python, and Solidity applications. Support for CHERI-C was recently integrated into the tool [50], but the implementation does not yet include a formally verified CHERI-C model and omits tagged memory support. Recently, Seung et al. [233, 234] formalized the CHERI-C memory model properties and semantics in the formal language Isabelle/HOL for use with verification tools such as ESBMC. They adapted the Gillian verification framework to prove the correctness of CHERI-C programs (*pure-cap* only) and to detect temporal memory issues such as dangling pointers.

CHERI Rust The intersection of the Rust programming language and the CHERI architecture is being actively explored, focusing on compatibility, challenges, and potential optimizations [144, 272]. These works explore the integration of Rust compile-time restrictions with CHERI runtime enforcement, enabling *unsafe* Rust code to run safer. Studies have also evaluated CHERI ability to prevent bugs like out-of-bounds writes in the standard Rust library [272]. Other works have focused on CHERI Rust evaluations on the *Arm Morello* platform. The results show differences based on `usize` interpretations: (i) a 64-bit address integer or (ii) a 128-bit capability integer, highlighting the performance impact and the requirement of additional operations for LLVM intrinsics (like *memcpy* and *inttoptr*) of the latter [144].

3.3.3 OSs, RTOSs and Microkernels

In the context of CHERI, components such as OSs, RTOSs, and microkernels must be adapted or redesigned to benefit from the CHERI memory model, minimizing the performance degradation. Porting these components to the CHERI ecosystem involves addressing the CheriABI compatibility challenges discussed in Section 3.3.2, which are particularly marked at the kernel-level where code frequently interacts with hardware, manages low-level memory layouts, and implements core system

Table 3.3: List of CHERI-aware OSs, RTOSs, and Microkernels. With: [1]Real-Time and [2]Evaluation Platforms

Name	Architectures			CAP-mode		MMU	RT[1]	EP[2]	Ref.
	RISC-V	MIPS	Arm	Hybrid	Pure				
General-Purpose Operating Systems									
CheriBSD	☑	☑	☑	☑	☑	☑	☐	QEMU, FPGA, *Arm Morello*	Davis et al. [91]
CHERI Linux	☑	☐	☑	☑	☑	☑	☐	QEMU, *Arm Morello*	Wang et al. [295]
MMU-less CHERI Linux	☑	☐	☐	☐	☑	☐	☐	QEMU	Almatary et al. [25]
Real-Time Operating Systems									
CheriRTOS	☑	☑	☐	☐	☑	☐	☑	QEMU, FPGA	Xia et al. [320]
CHERI-Zephyr	☑	☐	☐	☐	☑	☑	☑	QEMU, FPGA	Jackson [156]
CHERIoT-RTOS	☑	☐	☐	☐	☑	☐	☑	Sail, FPGA	Amar et al. [31]
Microkernels									
FlexCAP	☐	☐	☑	☑	☑	☑	☑	*Arm Morello*	Pierre et al. [240]
CheriOS	☑	☑	☐	☐	☑	☑	☑	QEMU, FPGA	Esswood [108]
seL4	☑	☐	☑	☑	☑	☑	☑	QEMU, *Arm Morello*	Heiser and Atkins [146]

services. Moreover, OS porting must ensure backward compatibility with legacy applications, optimize performance and security trade-offs for specific use cases, and adapt user-space components alongside kernel modifications. These requirements often necessitate new design strategies and frameworks (see Section 3.3.5).

This section reviews the CHERI community efforts on implementing or porting these stacks into CHERI. We focus on key criteria for comparison such as the types of systems (i.e., general-purpose operating systems (GPOSs), RTOSs, and microkernels), the architectures supported (i.e., MIPS, RISC-V, Arm), and the capability modes adopted. As further criteria, we also focus on the use of MMU-based isolation in addition to CHERI. For each work, identify whether the technology is suitable for real-time targets and report the evaluation platform. Table 3.3 summarizes the analysis.

General-Purpose Operating Systems Adapting GPOSs to the CHERI architecture is essential for demonstrating its security and performance benefits in practical scenarios. Moreover, GPOSs provide a familiar environment for a broader community, making it easier to port applications to the CHERI ecosystem.

CheriBSD [58, 91] is a modified version of FreeBSD with awareness of CHERI capability. As of this writing, CheriBSD is the most mature OS that supports CHERI, making it the primary choice for porting applications to the CHERI ecosystem. CheriBSD supports several architectures, including *Arm Morello*, CHERI-RISC-V, and CHERI-MIPS, and runs on multiple platforms and emulators. CheriBSD kernel supports hybrid and pure-cap compilation. From the library perspective, several libraries, including specific ones for compartmentalization support, have been ported to CHERI (see Section 3.3.5).

CHERI Linux: Linux version 5.18 was ported to CHERI RISC-V through joint efforts by Huawei and the Fraunhofer AISEC [150, 295, 296]. The port includes minimal modifications to the Linux runtime system, drivers, and user-space applications. The kernel is suitable for hybrid or pure-cap compilation. The authors reported that most of the issues were resolved by addressing the problems raised by the compiler during compilation. Moreover, they used the Juliet Test Suite [225] to evaluate the security improvements (the adapted version includes 546 test cases): CHERI Linux mitigates all spatial memory exploits (152 test cases). The authors included a performance evaluation with Dhrystone, Core-Mark, and Mibench (average overhead: 16.3%, 14.4% and 3.7%, respectively).A work-in-progress on *CHERIfying* Linux is also been made by Arm for the *Arm Morello* and by the *CHERI Alliance* and *Codasip* for CHERI RISC-V.

Almatary et al. [25] use CHERI-C and compartmentalization to enhance security of an **MMU-less** version of CHERI Linux by partitioning the kernel into isolated compartments—i.e., kernel modules, user applications, and libraries—reducing the impact of potential vulnerabilities. As the research is still a work in progress, validation and development is being conducted using QEMU. An evaluation on real platforms is reserved for future work.

Real-Time Operating Systems Table 3.3 also reports CHERI applicability to real-time environments. The evaluations focus on properties of real-time systems such as resources utilization, reliability, and predictability.

CheriRTOS [319, 320] was specifically designed to provide a proof-of-concept for embedded systems, executing on a single-address-space physical address model (*MMU-free* system). CheriRTOS is an extension of FreeRTOS that supports CHERI-MIPS and CHERI-RISC-V. The FreeRTOS kernel was modified to manage CHERI capabilities in pure-cap mode, such as PCC, DCC, stack, and context switch. This allows for compartmentalized multitasking and fine-grained memory protection,

while maintaining real-time guarantees. The authors compare the performance of the proposed CHERI system with a baseline system running without CHERI. The evaluation shows that the security benefit of the CHERI system outweigh the low overhead introduced (4.7% to almost no overhead).

CHERI-Zephyr is a port of the Zephyr RTOS to the CHERI-RISC-V architecture, developed through recent efforts by the University of Birmingham and the CAP-TEE project [156, 194]. Zephyr supports MMU and is written in C language, exposing it to common C memory-safety bugs [87, 226]. CHERI-Zephyr demonstrates how CHERI capabilities can mitigate these vulnerabilities, including safely handling known vulnerabilities [88, 227] via CHERI exceptions. The co-existence of CHERI and MMU is not addressed in this work. To the best of our knowledge, the development is still in the early stages and no performance evaluations are available.

CHERIoT-RTOS [31, 70], rather than retrofitting protection into an existing RTOS, decomposes core OS functionality into explicitly trusted and untrusted components from the ground-up. Running on CHERIoT-Ibex core, the system uses a dedicated loader to establish initial capabilities for each compartment, while a switcher handles context switches and cross-compartment calls. Notably, even the scheduler runs in a separate, isolated compartment rather than in the kernel itself. As with other RTOSs discussed, the key challenges are maintaining real-time predictability and low overhead while enforcing fine-grained protection; CHERIoT addresses them by keeping the TCB small and moving policy and high-level abstractions into unprivileged libraries.

Unikernel and Microkernels Pierre et al. [240] argue that the unikernel OS model needs to incorporate intra-unikernel isolation to enhance security. Without such isolation, unikernels are vulnerable to internal threats. By leveraging CHERI, unikernels can enforce strict boundaries between subsystems within the same instance, reducing the impact of potential vulnerabilities without compromising their efficiency.

FlexCap builds on FlexOS and leverages *Arm Morello* hardware features to implement advanced isolation mechanisms [169, 170]. FlexOS [179] is a library OS that emphasizes customizable isolation and security measures. FlexOS integrates with Unikraft [171], a high-performance unikernel for isolated environments (e.g., using the MMU) and single-application operating systems. FlexCap [230] uses a CHERI-aware version of Unikraft to handle capabilities for key primitives such as context switches, exceptions, and device access. FlexCap combines FlexOS and Unikraft to build secure and efficient systems in a compartmentalized environment using CHERI capabilities.

CheriOS [108] is a capability-based microkernel that uses CHERI to mini-

mize the Trusted Computing Base and increase the memory protection. It was implemented on CHERI-MIPS and CHERI-RISC-V in pure-cap mode. CheriOS follows a mutual distrust model, i.e., the OS and user applications do not trust each other. Only two components are considered trusted: the CHERI ISA and the *nanokernel*. The *nanokernel* provides essential primitives and enables the creation of compartments with distinct authority, allowing for fine-grained control over privileges. Additionally, the *nanokernel* uses both MMU and capabilities to manage and secure the state of every physical and virtual page.

seL4 [130], a formally verified microkernel, has explored CHERI integration to enhance userspace safety, as shown at the seL4 Summit 2023 [24, 146]. A recent work [23] introduced an seL4 version for CHERI-RISC-V, supporting pure-cap userspace and a hybrid-cap kernel, passing the *sel4test*.

3.3.4 CHERI Runtime System

The runtime system provides the environment to run an application as specified by the programmer. Runtime systems can implement different functionalities such as memory management (e.g., stack and heap management), threads and processes, OS interactions, exception and error handling, code execution, I/O operations (e.g., network and disk), and dynamic library loading. This section surveys the state-of-the-art literature of enabling CHERI technology in the runtime system, with focus on: (i) efforts to port programming languages runtime systems (e.g., memory allocators) to CHERI and (ii) capability revocation techniques for heap and stack temporal memory safety.

CHERI-aware Memory Allocators Memory allocators are responsible for dynamic memory allocation and deallocation during program execution. Porting memory allocators to CHERI involve careful management of capabilities. Bramley et al. [49] assessed the status of CHERI-compatible memory allocators, uncovering vulnerabilities in both CHERI-aware and non-CHERI-aware compilers. For example, using *jemalloc* [110] can expose hidden memory when resizing capabilities bounds without zeroing out previous contents. There are three notable CHERI-aware forks of well-known memory allocators: the *jemalloc* [110], *snmalloc* [182], and *dlmalloc* [177]. *jemalloc* is the default memory allocator for CheriBSD OS. Microsoft employs a hardened version of *snmalloc* in the CHERIoT [29] project.

Capability Management

Table 3.4: CHERI Capability Revocation Techniques

Technique	Description	Key features
	Heap Memory Safety	
CHERIvoke [321] (2019)	Extends the memory allocator to mark deallocated memory regions in a bitmap shadow table and keeps these regions in a quarantine buffer. Once this buffer is full, a full sweep locates and invalidates capability references to deallocated memory regions recorded in the shadow table.	Implements a special load tag instruction to load multiple tags at once (*CLoadTags*) and adds page-level capabilities tracking bits to indicate if a page holds one or more valid capabilities (capability dirty bits).
Cornucopia [311] (2020)	A sweeping capability revocation system built atop CHERI and the CHERIvoke algorithm [321] implemented on CheriBSD.	Implements the hardware extensions proposed on CHERIvoke [321] in a 64-bit CHERI MIPS-base soft core.
Capability Boehm [157] (2022)	The Boehm-Demers-Weiser Garbage Collector (BDWGC) is a conservative GC widely used in C and C++ programs for automatic heap memory management. This adaptation allows BDWGC-based applications to run on CHERI systems.	It enables automatic heap memory management and safe reclamation of unused objects, enforcing memory safety via capabilities.
Cornucopia Reloaded [117] (2024)	Re-implementation of the Cornucopia [311] on *Arm Morello* with a complete redesign of core algorithm [321]. Reloaded embodies new architectural features to limit the effect of long global stalls and reduce the DRAM traffic, a.k.a stop-world-pauses, on each revocation pass found in previous techniques [311, 321].	Repurposes hardware extensions proposed on CHERIvoke [321] and adds per-PTE bit flag to trap all capability loads (*load barriers*).
	Stack Memory Safety	
Local capabilities [297] (2015)	Control flow technique that limit capability storage in constrained memory regions (typically the stack) and registers.	Adds two flow control stored in the capability metadata (*global capabilities bit* and *local capabilities bit*).
StkTokens [273] (2019)	Formalizes a secure calling convention for cross-domains in CHERI based on linear capabilities to enforce local state encapsulation (LSE) and well-bracket control flow (WBCF) in a single shared stack between two untrusted domains. Enforces revocation upon domain crossing.	Utilizes local capabilities [297] concept.
Tsampas et al. [283] (2019)	Ensures temporal stack memory safety for stack references in an abstract capability-based machine. It adds multiple locality levels to prevent capabilities from being stored in a memory with a longer lifetime cycle than their own.	Extends the capability metadata with n locality permission bits and adds dynamic checks to enforce locality when storing capabilities in memory.
Georges et al. [129] (2021)	New formally proved calling convention written in Coq that expands the idea of local capabilities and unutilized capabilities to share and revoke capabilities.	Adds a new type of architectural capability (unitialized capability) that divides its authority into two memory regions: (i) read-write regions and (ii) write-only regions. System model in CHERI-MIPS-based simulator.

Continued on the next page

Technique	Description	Key features
Georges et al. [128] (2022)	Secure calling convention that ensures LSE, WBCF, and stack temporal and spatial memory safety. It proposes using direct capabilities along with uninitialized capabilities [129] to enforce stack memory safety without requiring expensive full stack clears upon return/calls between two domain boundaries.	Adds new architectural capability type, *direct capabilities*. This capabilities can only be stored in the memory above, where it can read memory.
Borrowed capabilities [293] (2022)	Novel technique to capabilities revocation stems upon the concepts of linear capabilities and the idea of borrowing mutable and immutable objects in RUST language	New capability representation (borrowed capabilities) model in Sail [222] and CHERI-RISC-V.
Heap and Stack Memory Safety		
CHERIoT [30] (2023)	Develop hardware-specific mechanisms for complete temporal heap and stack memory safety tailored for the embedded system space.	Revocation uses 1-bit per capability-size granule, enforced by a load barrier, and is assisted by a MMIO device. While *stack watermarking* enforces stack memory safety.
Conditional Capabilities [208] (2024)	Proposes a CHERI ISA extension to express fine-grained memory policies that prevent undefined behavior or uninitialized memory (e.g., *write-before-read*).	Adds permissions bits to the capability metadata to express conditions. Moreover, it adds new instructions to the set these conditions and operation bounds.

The runtime system plays a crucial role in creating, propagating, and revoking capabilities during application execution. Cross-domain propagation of capabilities requires careful handling to prevent leaks and protect data integrity and confidentiality. The complexity and overhead of capability revocation can limit the adoption of CHERI, prompting research into secure, efficient revocation and propagation techniques. We group the contributions into three categories: (i) heap memory safety, (ii) stack memory safety, and (iii) works that address both of them. Table 3.4 provides a brief description, key features, and references of each contribution.

Heap memory safety ensures the correct management of dynamically allocated memory, avoiding, for instance, memory leaks and buffer overflows [30, 117, 311, 321]. Notably, Cornucopia Reloaded [117] uses a load barrier mechanism to dynamically track loaded capabilities, avoiding costly global operations and improving performance. The CHERI port of the Boehm-Demers-Weiser Garbage Collector [157] is exploring automatic heap management on CheriBSD for CHERI RISC-V and *Arm Morello*, tackling challenges such as pointer shape issue and provenance. In the embedded space, CHERIoT [30] defines a set of mechanisms based on the addition of revocation bits to each heap memory granule, load-barrier filters or a background hardware worker to invalidate the capabilities pointing to freed memory.

Stack memory safety focuses on preventing issues such as stack overflows, unintended overwrite of local variables, and improper usage of function call frames. Several techniques exist to limit capability propagation in CHERI-based systems

and to efficiently revoke ephemeral capabilities shared across domain boundaries [30, 128, 129, 157, 273, 274, 283, 297]. To ensure stack temporal safety when invoking an untrusted domain, the system must uphold two properties: (i) Well-Bracketed Control-Flow (WBCF), which ensures that the callee returns control to the caller and (ii) Local State Encapsulation (LSE), which guarantees that the caller's local state cannot be altered by the callee. The most innovative works include Tsampas et al. [283], which proposed a unique locality-based approach, enhancing CHERI capabilities with locality permission bits and dynamic checks to prevent capabilities from outliving their intended scope. Georges et al. [128, 129] propose two methods to improve stack temporal memory safety. The first method [129] expands the notion of *local* and *uninitialized capabilities* to facilitate capability sharing and revocation using stack *watermarking* (similar to method purposed by CHERIoT [29]). The second method [129] establishes a secure calling convention that combines uninitialized and direct capabilities. Skorstengaard et al. [274] advanced the concept of *linear capabilities*, converting stacks and sealed return capabilities into single-instance entities, ensuring safe delegation and centralized stack sharing. Vercammen et al. [293] proposed *borrowed capabilities*, a flexible mechanism that ties capabilities to lifetimes, allowing temporary breaches of linearity while ensuring safe revocation through lifetime tokens. Recently, Merve et al. [208] propose the idea of using *conditional capabilities* to express memory policies that could improve heap and stack memory safety. For instance, the *conditional capabilities* can be used for a *write-before-read* policy, requiring at least one write operation to the memory location pointed by the capability before granting read access.

Case Studies of CHERI Runtime Systems Numerous studies demonstrated the feasibility and security benefits of porting real-world runtime systems, applications, and operating systems to CHERI [54, 71, 91, 139, 196, 236, 297].

Watson et al. [297] introduced the first CHERI-C userspace runtime, allowing CHERI-C applications to operate in hybrid mode (i.e., supports non-CHERI libraries). CheriABI [91] later extended the runtime system for pure-capability applications, maintaining a hybrid layer for backward compatibility.

Several other runtime systems have also leveraged CHERI technology. For instance, Lowther et al. [196, 197] hardened MicroPython [127] for improved runtime security on the *Arm Morello* and Sonata platform. Gutstein [139] adapted the JavaScriptCore (JSC) interpreter for *pure-cap* code support. CHERI JNI [71] utilized CHERI capability model to optimize buffer sharing (offer direct access) in the Java Virtual Machine (JVM) and prevent type-confusion attacks through sealing mechanisms. Additionally, Butcher et al. [54] explored integrating CHERI primitives with Ada [228] runtime checks to safeguard uncovered parts of code or

unsafe code in *pure-cap* code. The evaluation was performed for the *Arm Morello* platform in *pure-cap* mode. Patel et al. [236] optimized kernel page fault handling using CHERI capabilities to enhance pointer identification and accelerate page prefetching in pointer-intensive applications.

3.3.5 Libraries and Frameworks

A suite of libraries and frameworks has been developed to fully make use of CHERI capability-based security model. These components bridge lower-level system infrastructure and applications-level software, ensuring that the benefits of CHERI principles are implemented across the entire software stack.

musl-libc [75] and **newlib** [74] are lightweight C standard libraries that have been adapted to work within CHERI ecosystems. These libraries provide essential runtime support for creating containerized or baremetal applications. Another set of libraries includes **libreSSL** [9] and **libssh2** [10], which are cryptographic and communication libraries.

CompartOS [22] is a framework and linkage-based compartmentalization model designed to leverage CHERI capabilities into an MMU-free, single-address-space and single-ring-privilege environment for isolating different linkage modules. CompartOS uses an adapted FreeRTOS (CheriFreeRTOS) kernel, extends the *libdl* library (provides functions for runtime loading and symbol resolution of shared libraries) for CHERI-aware dynamic linking, and leverages the LLVM compiler to limit the visibility of global variables. CompartOS has been compared against three different architecture features, i.e., without protection, PMP, and PMP+MMU. The extensive evaluation [22] analyzes the trade-offs of hardware overheads and security benefits, and highlightsthat the flexibility of CompartOS only incurs a slightly increased overhead compared to a full pure-cap solution.

CAP-VMs [261, 262] is a VM-like abstraction that uses CHERI to isolate applications through memory capabilities. Its architecture consists of three main components: the Host OS kernel, the Intravisor, and the capability virtual machine (cVM). CheriBSD serves as the Host OS kernel, providing essential primitives for synchronization, execution contexts, and I/O operations. The Intravisor runs as a process within CheriBSD and handles the isolation and lifecycle management of cVMs. It distributes memory capabilities to cVMs, facilitates communication, and has access to the memory of all cVMs. The cVM, on the other hand, represents an isolated application, e.g., a program binary and its libraries, that runs as an isolated thread of the Intravisor process. At the time of this writing, two extensions of CAP-VMs exists: (i) Object Reuse with Capabilities (ORC) [265] extension, which improves the memory utilization of the overall framework and (ii) Sandbox fork caching mechanism [328], that reduces cold start latency of the cVMs.

3.3.6 Applications

The CHERI ecosystem has demonstrated significant progress in porting and adapting a wide range of real-world software applications to leverage enhanced security and performance benefits. We refer interested readers to the CheriBSD packages for the exhaustive list of applications ported or supported [69].

Furthermore, multiple tools have been developed for emulation, simulation, validation, debugging, and visualization. These tools address the challenges of limited hardware availability and provide robust environments for development and testing. As summarized in Table 3.5, emulator tools like QEMU [299] and Arm *Fixed Virtual Platform* (FVP) [38] simulate the behavior of a complete system, allowing the execution of multiple OSs (see Section 3.3.3). Tools such as Spike [57], Sail [41, 63–65, 222], VeriCHERI [34], and TestRIG [76] focus on the verification process of the ISA. CheriPlot [59, 203] provides graphs of the derivations of capabilities. Finally, the GNU debugger (GDB) [78] has been extended to support CHERI, thereby facilitating debugging of CHERI-adapted software.

Table 3.5: CHERI Emulators, tools, and debugger

Name	Category	Architectures			Description
		RISC-V	MIPS	Arm	
QEMU [299]	System Emulator	☑	☑	☑	A generic and open-source system emulator that supports full system emulation, including some peripherals.
Arm FVP Morello [38]	Virtual Platform Emulator	☐	☐	☑	A software model of *Arm Morello* hardware for prototyping, development, and testing. It allows full system emulation.
Spike [57]	ISA emulator	☑	☐	☐	The official RISC-V ISA simulator, focused on emulating only the RISC-V instruction set.
Sail [41, 222]	ISA Formal Verification Emulator	☑	☑	☑	A formal specification tool for ISA semantics with an executable model. Sail models have been used for ISA development, implementation, and formal verification via automatically generated tests.
VeriCHERI [34]	Formal security verification tool	☑	☐	☐	A tool for formal security verification at the register transfer level (RTL). As a consequence, this tool is ISA-independent and uses SystemVerilog Assertions (SVA) properties to verify the security of the architecture.

Continued on the next page

Name	Category	Architectures			Description
		RISC-V	MIPS	Arm	
TestRIG [76]	Testing Tool	☑	☐	☐	This framework tests RISC-V processors using random instructions generation. Using this tool, together with Sail models, allows automated verification tests.
CheriPlot [59, 203]	Visualization Tool	☐	☑	☐	A graphical tool for visualizing and tracking the capability flow at the user and kernel level.
GNU debugger (GDB) [78]	Debugger	☑	☑	☑	GDB allows to debug CHERI-enabled software. It can inspect capability registers and display their content in a human-readable format.

3.3.7 Lessons Learned from Software Support

The experiences of porting software to CHERI reveal insights that can guide future adoption efforts. CheriABI compatibility issues follow consistent patterns: pointer-integer conversions, assumptions about pointer sizes, capability alignment constraints, and provenance violations. CHERI-aware compilers catch most problems at compile time, with runtime issues primarily in dynamic memory and cross-boundary communication. In practice, porting effort is usually modest; the challenge is locating issues rather than fixing them. In the CheriBSD package ecosystem [69], many applications require only recompilation once core libraries (e.g., libc and libc++) are adapted. Kernel-level components need more targeted changes due to direct hardware interaction and custom memory layouts, but modifications remain localized, as seen in the FreeBSD kernel port. Compilers and interpreters are notable exceptions. Compilers must be adapted to generate CHERI-aware code, ensuring that instruction selection, calling conventions, and memory operations comply with CheriABI, while interpreters face challenges from dynamic memory, code generation, and provenance tracking. This complexity explains why SPECint 2006 benchmarks requiring the compilation of the Perl interpreter (400.perlbench) and GCC compiler (403.gcc) were excluded [305]. For new CHERI adopters, we recommend checking dependencies against CheriBSD ports, monitoring pointer arithmetic and custom allocators, leveraging compiler diagnostics, and, if many dependencies need to be ported, starting with hybrid capability mode.

3.4 CHERI Application Domains & Use-cases

This section explores key application domains where the potential of CHERI is being actively investigated. We focus on two very relevant areas: edge-to-cloud systems and mixed-criticality systems (MCSs).

3.4.1 CHERI for Edge-to-Cloud systems

Edge devices form the backbone of distributed systems and cloud-based infrastructures, yet they face significant challenges in security and resource efficiency.

The IoT sector, in particular, represents a promising direction for CHERI adoption. In fact, these systems are small, cheap and usually lack security hardware or software solutions, e.g., MMU and virtualization. Additionally, the turnover of IoT applications is fast, creating opportunities for incorporating CHERI-based software. Moreover, many IoT applications fall within the domain of Cyber-Physical Systems (CPS), which consist of a distributed network of interconnected components such as sensors, actuators, controllers, and computing devices. CPS systems face multiple challenges related to coordination, synchronization, communication, security, and reliability. For industrial IoT sectors, where security and long product lifetimes are critical, CHERI has emerged as a promising architecture to achieve robust security while keeping almost unmodified source code.

Safronov et al. [259] conducted an analysis of various Common Vulnerabilities and Exposures (CVEs) related to wireless gateways, which are key components on large-scale IoT systems, facilitating device interconnectivity. Their findings indicate that implementing memory protection techniques can eliminate 74% of these CVEs, increasing the overall security. Dodson et al. [98] examine two specific scenarios, a robot and a common Industrial Control System (ICS), where they integrate CHERI capabilities and Macaroon tokens (`libmacaroons`) [46], leveraging their structural similarity. Dodson [97] expands on the exploration of CHERI for CPS, introducing another use-case by integrating `libmodbus` into CheriFreeRTOS from the CompartOS work.

CHERI has also been investigated for cloud and edge-cloud environments, where capabilities can provide fine-grained isolation between processes, containers, or microservices. In particular, CHERI can restrict each component to the resources required for its execution, limiting the impact of a compromised component. CAP-VMs (Section 3.3.5) follows this model by executing application components as isolated capability virtual machines managed by an Intravisor.

These approaches are also relevant to networking systems, where individual components of a network stack can be isolated while retaining efficient communication. This thesis investigates this design space by applying the CAP-VM model to the compartmentalization of a high-performance network stack on the *Arm Morello*

platform. The resulting system isolates DPDK, F-Stack, and network applications using CHERI capabilities, and its design and evaluation are presented in Chapter 4.

3.4.2 CHERI for Mixed Criticality Systems

MCSs are prevalent in domains like automotive, aerospace, and robotics, where safety-critical operations coexist with less critical or non-critical functionalities. Ensuring security, reliability, predictability, and performance in such heterogeneous environments poses significant challenges, including the need to isolate critical components from potential faults or attacks in non-critical components. CHERI technology can facilitate high-level security for mixed-criticality systems, without significantly impacting their performance.

In the automotive domain, the AutoCHERI project [80] (a consortium led by Beam Connectivity) investigates the trade-offs of using CHERI technology in the automotive industry. Specifically, they: (i) study the possible use-cases for CHERI technology (e.g., teleoperation, Over-The-Air (OTA) software updates), (ii) implement a telematics control unit (TCU) for cloud services; and (iii) study the viability of CHERI-enable products for the automotive industry. In a recent work, the *Arm Morello* board was explored to address critical cyber-security risks in automotive systems [148]. Using the ISO/SAE 21434 framework and Threat Analysis and Risk Assessment (TARA), the study identified potential threats and defined security requirements. Key outcomes included secure TCU design, enhanced OTA update protection, and reduced risks for memory-related vulnerabilities. However, challenges such as long technology adoption cycles, complex supply chains, and certification requirements persist. Another study implemented a CHERI-enabled Intrusion Detection System (IDS) for automotive networks, showing CHERI applicability in this field [164].

Aerospace and autonomous systems represent another important application domain. Beyond the automotive use-cases described above, this thesis investigates the application of CHERI to a complete autonomous flight software stack. Chapter 5 presents the integration of CHERI into PX4/NuttX, providing module-level compartmentalization of the flight-control software on a RISC-V platform.

3.5 Broader Future Directions for CHERI Research

Beyond the application domains discussed in the previous section, CHERI is being investigated across several broader research directions, including heterogeneous computing, large-scale applications, I/O protection, virtualization, Trusted Execution Environments (TEEs), and microarchitectural side channels.

3.5.1 CHERI for Heterogeneous Computing.

Heterogeneous computing architectures integrate different processing units, such as CPUs, GPUs, FPGAs, Tensor Processing Units (TPUs), and Neural Processing Units (NPUs), to improve performance and energy efficiency. Such systems increasingly rely on software stacks written in memory-unsafe languages such as C and C++. For example, NVIDIA's CUDA [109] and OpenCL [277] are general-purpose GPU frameworks based on C/C++. Recent studies have also shown that GPUs and other accelerators are susceptible to memory-safety vulnerabilities [232, 244, 253, 326], revealing a significant attack surface in contemporary heterogeneous systems.

CHERI provides a potential approach for extending fine-grained memory protection beyond CPUs. Recent work by Naylor et al. [218] explored CHERI-SIMT, adapting CHERI capabilities to the Single Instruction, Multiple Threads (SIMT) execution model used by GPUs. The approach enforces spatial memory safety across threads while maintaining low performance overhead. Extending CHERI to commodity heterogeneous computing stacks remains an open research direction.

3.5.2 CHERI for Large-Scale Applications.

CHERI is also being investigated for large-scale applications, with early work targeting web services [159] and desktop environments [310]. These studies explore the use of CHERI memory safety and compartmentalization to limit the impact of compromised or untrusted components. For example, desktop applications can use CHERI-backed sandboxing to isolate plugins and extensions, while document editors, multimedia players, and email clients can compartmentalize components that process untrusted data. Mobile systems present additional challenges due to their mature hardware ecosystems and strict power and area constraints. Nevertheless, work is underway to adapt the Android software stack to the *Arm Morello* platform [330]. Broader adoption in such environments requires further investigation of compatibility, performance, and hardware costs.

3.5.3 CHERI for I/O

CPU-level CHERI protection does not automatically protect memory accesses performed by I/O devices using Direct Memory Access (DMA). Such accesses can therefore bypass the protection enforced by CHERI and potentially corrupt protected memory or capabilities. Extending memory protection to I/O devices is consequently important for a complete CHERI-based protection model. However, protection mechanisms for I/O devices in CHERI systems remain relatively unexplored. The CHERI specification currently follows a conservative approach by

stripping the tag from data written through I/O. Markettos et al. [199] investigated capability-based access control for I/O as an alternative or complement to traditional mechanisms such as the RISC-V IOMMU and IOPMP. Capability-based I/O could simplify protection by avoiding explicit mappings between objects and IOPMP registers, while an IOMMU could potentially use capabilities for single-level translation and protection. Other system-level isolation technologies, such as RISC-V WorldGuard [136], could also be integrated with CHERI or potentially replaced by capability-based mechanisms. Further work is needed to understand how these protection mechanisms should compose with CHERI.

3.5.4 CHERI for Hypervisors

Hypervisors provide isolation between multiple operating systems or applications running on shared hardware. Integrating CHERI into hypervisors introduces challenges at the boundary between capability protection and traditional virtual memory. Hypervisors must correctly manage capabilities alongside page tables, while their low-level implementations may require modifications to code relying on unsafe memory accesses. The CHERI-TrEE work implemented a trusted hypervisor on *Arm Morello* to evaluate CHERI-TrEE operations [289]. The implementation runs in EL2 and pauses execution at EL1 to perform the required mechanisms, demonstrating the feasibility of using CHERI capabilities to construct TEE-like isolation. Existing virtualization systems are also being adapted to CHERI. Work is ongoing to port *bhyve* to CHERI [77] on CheriBSD for *Arm Morello*, with the goal of incrementally adopting capability-based protection. In addition, *Bao* [201] has been integrated into the CHERI ecosystem using the RISC-V CVA6 with Hypervisor and CHERI extensions [334]. These efforts demonstrate the feasibility of combining virtualization with capability-based isolation, while leaving open questions regarding performance and the interaction between CHERI and virtual memory.

3.5.5 CHERI for TEE

Trusted Execution Environments (TEEs) isolate sensitive applications from the rest of the system. Common approaches include Intel Software Guard Extensions (SGX) [314], AMD Secure Encrypted Virtualization (SEV) [165], and ARM TrustZone [241]. These mechanisms typically establish isolated regions of code and data that remain protected from other software components. CHERI provides an alternative approach based on fine-grained, capability-based isolation. Van Strydonck et al. [289] demonstrated that minor ISA modifications can be used to construct enclave-like isolation on an embedded system without an MMU. CAP-TEE [194] investigates the use of CHERI to mitigate attacks against TEEs, such as memory

corruption and control-flow hijacking [72]. For example, a CHERI-enabled memory encryption engine has been proposed to extend the CHERI ISA with encryption semantics for enclave creation and invocation [160]. Other work, such as Capstone [329], also evaluates capability-based protection for TEEs. These studies suggest that CHERI can provide fine-grained spatial memory safety and compartmentalization for TEE deployments. This is particularly relevant to RISC-V systems, where existing enclave technologies such as Keystone [178] and CURE [43] rely on the Physical Memory Protection (PMP) unit for isolation. The limited number of PMP entries constrains the number of simultaneously isolated regions. Nevertheless, CHERI-based TEEs remain an active research area, especially for systems combining CHERI with MMUs and for secure enclave communication and management.

3.5.6 CHERI for Side-Channel Attacks

CHERI's memory-safety guarantees do not inherently eliminate microarchitectural side channels. Spectre [168] and Meltdown [188] demonstrated how speculative execution can leave exploitable traces in microarchitectural components through timing channels [126]. While CHERI was not explicitly designed to provide side-channel resilience, it can interact with these attacks by introducing both potential mitigations and new vulnerabilities [122, 123]. Several studies have investigated these interactions. One proposed approach uses a compartment identifier (CID) to tag microarchitectural state with the current protection domain, reducing potential leakage between CHERI compartments [298]. This mechanism has been evaluated on the *Arm Morello* platform. Fuchs et al. [121] developed a bare-metal test suite for transient-execution attacks on RISC-V and CHERI-RISC-V processors, including adaptations of Spectre and Meltdown using CHERI-Toooba. Their results showed that CHERI prevents typical Meltdown attacks but does not inherently prevent Spectre attacks based on speculative execution. Later, Fuchs et al. [123] defined CHERI ISA-level safe-speculation hardware-software contracts [122] to constrain speculation across protection boundaries. The same study identified Meltdown-CF, a new attack targeting the CHERI-Toooba implementation. Henes et al. [147] further investigated Spectre-like attacks on the *Arm Morello* platform and showed that capability speculation can expose the platform to attacks such as Spectre-BTB. Despite this growing body of work, important questions remain open regarding timing side channels through shared microarchitectural resources such as caches, TLBs, and interconnects. Capabilities are typically larger than conventional pointers and therefore increase cache footprint and memory traffic, potentially affecting timing channels. Furthermore, although CIDs can partition microarchitectural state, the extent to which they prevent unintended information flow between compartments remains insufficiently understood. Future work is

therefore needed to determine how CHERI protection boundaries interact with speculative execution, caching, and interconnect behavior, and how additional microarchitectural mechanisms can mitigate residual information leakage.

3.6 Summary

The preceding sections have examined CHERI in depth, covering its protection model, hardware implementations, software ecosystem, application domains, and broader research directions. CHERI has progressed beyond academic prototypes, with growing industrial involvement, hardware implementations, software support, and ongoing standardization efforts. The formation of the CHERI Alliance [8] is a significant development in this direction. Bringing together semiconductor companies, system integrators, software vendors, and academic institutions, the Alliance coordinates standardization efforts around CHERI extensions for mainstream ISAs, most notably the ongoing RISC-V ZCheri standardization process [17]. Arm's investment in the Morello prototype [184] demonstrated the viability of CHERI integration in a high-performance processor microarchitecture. Codasip's A730 [73] and Microsoft's CHERIoT programme [29], which produced both a processor core and a dedicated RTOS, further demonstrate the expansion of CHERI across different classes of systems.

Despite this progress, the survey reveals that CHERI deployment remains in its early stages and that several system-level challenges remain open. Existing RTOS efforts, including CheriRTOS [319], CHERI-Zephyr [156], and CHERIoT-RTOS [31], have demonstrated the feasibility of capability-enforced protection in real-time environments. However, none of these efforts address the compartmentalization of a complete safety-critical autonomous software stack at the module level, nor do they evaluate CHERI-based isolation against MMU-based alternatives in an embedded, autonomous systems context. Similarly, while the application domains section of this chapter identifies the network stack as a promising target for CHERI compartmentalization, the literature lacks a systematic evaluation of the overhead tradeoffs at line rate in a high-performance networking context.

These gaps define the research scope of the remaining contributions of this thesis. Chapter 4 addresses the question of network stack compartmentalization, demonstrating on the Arm Morello platform that CHERI-based compartmentalization of a high-performance networking stack can be achieved without sacrificing line-rate throughput. Chapter 5 addresses the question of safety-critical autonomous flight stacks, demonstrating on the CVA6-CHERI platform that NuttX and PX4 can be compartmentalized at the module level using CHERI capabilities. The evaluation further compares CHERI-based isolation with MMU-based isolation in terms of primitive and end-to-end costs. Together, these contributions investigate the

application of CHERI to two system classes that remain comparatively underexplored: high-performance network stacks and safety-critical autonomous embedded systems.

Chapter 4

CHERI Compartmentalized Network Stack

The survey presented in Chapter 3 established that CHERI has matured significantly as a hardware security architecture; however, its application to the networking stack of resource-constrained embedded systems remains largely uncharted. Chapter 2 highlighted that interconnected edge devices, ranging from UAVs to autonomous vehicles and smart industrial plants, increasingly handle safety-critical data across network interfaces that are directly exposed to untrusted external input. This makes the network stack a primary attack surface. At the same time, these devices often lack hardware memory protection mechanisms, such as MPUs or MMUs, and instead rely on their coarse-grained isolation, which results in performance degradation and timing unpredictability. The CHERI fine-grained, hardware-enforced capability model represents a compelling alternative to address this gap. However, a systematic evaluation of its overhead tradeoffs at line rate in a high-performance networking context had not been previously attempted. This chapter closes the gap by investigating the compartmentalization of a complete user-space network stack comprising an Ethernet driver layer (DPDK), a TCP/IP library (F-Stack), and a network application (iperf3) using CHERI capabilities. The evaluation is conducted on the Arm Morello platform under the CheriBSD operating system, which is the most representative industrial-grade CHERI testbed available. Building on the CAP-VMs compartmentalization model introduced Chapter 3 and analyzed in Section 4.1.1, we design and implement two compartmentalization scenarios with distinct isolation granularity and tradeoffs, and compare them against a non-CHERI baseline. The central question this chapter seeks to answer is whether CHERI-based compartmentalization of a high-performance networking stack can be achieved without sacrificing bandwidth throughput, and at what cost in terms of execution overhead and implementation complexity. The remainder of this chapter is organized as follows. Section 4.1 provides the background on the CAP-VMs

framework and DPDK. Section 4.2 presents the network stack compartmentalization scenarios and their implementation. Section 4.3 evaluates performance and validates the security properties of the proposed designs. Section 4.4 discusses related work, and Section 4.5 concludes with a summary of findings and directions for future work.

4.1 Preliminary concepts

4.1.1 CAP-VMs

CAP-VMs [261], is an example of a dynamic approach to compartmentalization with CHERI. CAP-VMs provide VM-like abstractions to isolate system applications—in a single-address-space environments—by leveraging CHERI memory capabilities. The CAP-VM approach comprises three main elements. (i) A host OS Kernel: a CHERI-aware kernel that provides the primitives for synchronization, execution contexts, and I/O operations. Similarly to CAP-VM [261], we used CheriBSD. (ii) The *Intravisor*: a process responsible to manage configuration, isolation, and lifecycle of capability-VMs (cVMs). It distributes memory capabilities to cVMs and has access to all cVM memory regions. (iii) *capability-VM* (cVM): an isolated application component (*hybrid* or *pure* mode) that run as a thread of the Intravisor.

The Intravisor is in charge of the configuration of system capabilities among the various cVMs, i.e., of the definition of the isolated memory ranges. For cVMs, the Intravisor also acts as a proxy between the cVM capability world and CheriBSD. cVMs do not have direct access to the host OS syscalls, but must use instead a *trampoline* proxy table provided by the Intravisor that correctly handles the capabilities and mediates the access to the OS.

In this work, we leverage the minimal trusted computing base (TCB) of the Intravisor, which makes it practical for integration into embedded systems.

4.1.2 DPDK

The *Data Plane Development Kit* [1] is an Open Source set of libraries and drivers designed to accelerate packet processing in high-performance networking applications. DPDK bypasses the traditional kernel network stack and interacts directly with network hardware. This leads to better performance, but less isolation with the application. DPDK also operates in polling mode to reduce the latency caused by interrupt-triggered context switches. To detach the NIC from the kernel-space, DPDK uses a kernel-module.

Using DPDK, the management of network devices can be encapsulated within a cVM, thus isolating network operations from other system components and

leveraging CHERI's capability-based security to ensure compartmentalization.

DPDK does not provide any TCP/IP protocol implementation, but only focuses on packet I/O and network hardware acceleration. Applications using DPDK must either implement their own protocol stack or integrate an existing user-space networking stack like the Open-Source F-Stack library [3]. In this work, we ported F-Stack on our CHERI architecture and integrated it in our DPDK cVMs. Compared to other TCP/IP libraries built on top of DPDK, the existing integration of F-Stack with the FreeBSD network stack facilitated a more efficient porting process to CheriBSD.

We selected *iperf3* [4] as an application for the evaluation of our compartmentalized network. *iperf3* allows to define a server-client connection to measure the maximum bandwidth achievable.

4.2 Network Stack Compartmentalization Options

Different alternatives exist for the compartmentalization of a network stack using CHERI [26, 32, 322]. Each alternative entails trade-offs regarding its intrusiveness (i.e., the amount of changes required), overheads, and potential scalability. To this end, this work focuses on two possible system designs and compares them with a baseline scenario that does not use CHERI.

While we target resource-constrained systems that lack additional memory protection mechanisms, the trade-offs presented are also applicable to user-space device management with strong inter-thread isolation. This design is particularly efficient when low overhead is essential, leveraging multi-threading solutions to achieve this balance.

An overview of our CHERI-enabled designs—namely, Scenario 1 and Scenario 2—is presented in Figure 4.1 and Figure 4.2. Additionally, we define a *Baseline* scenario for comparison. This serves as a reference point for evaluating the two CHERI-enabled scenarios.

We run our experiments on the Arm Morello on top of the CheriBSD host OS. For the two CHERI-enabled scenarios, the Intravisor is responsible for the configuration of the compartments and mediates the syscalls between compartments and CheriBSD. Since the Morello built-in Ethernet device is not supported by DPDK, we chose a PCI card *Intel 82576 Gigabit Network Connection* with two Ethernet ports.

The two Ethernet ports are connected to a host machine through a point-to-point connections. The host machine runs *iperf3* in server or client mode as appropriate for each experiment.

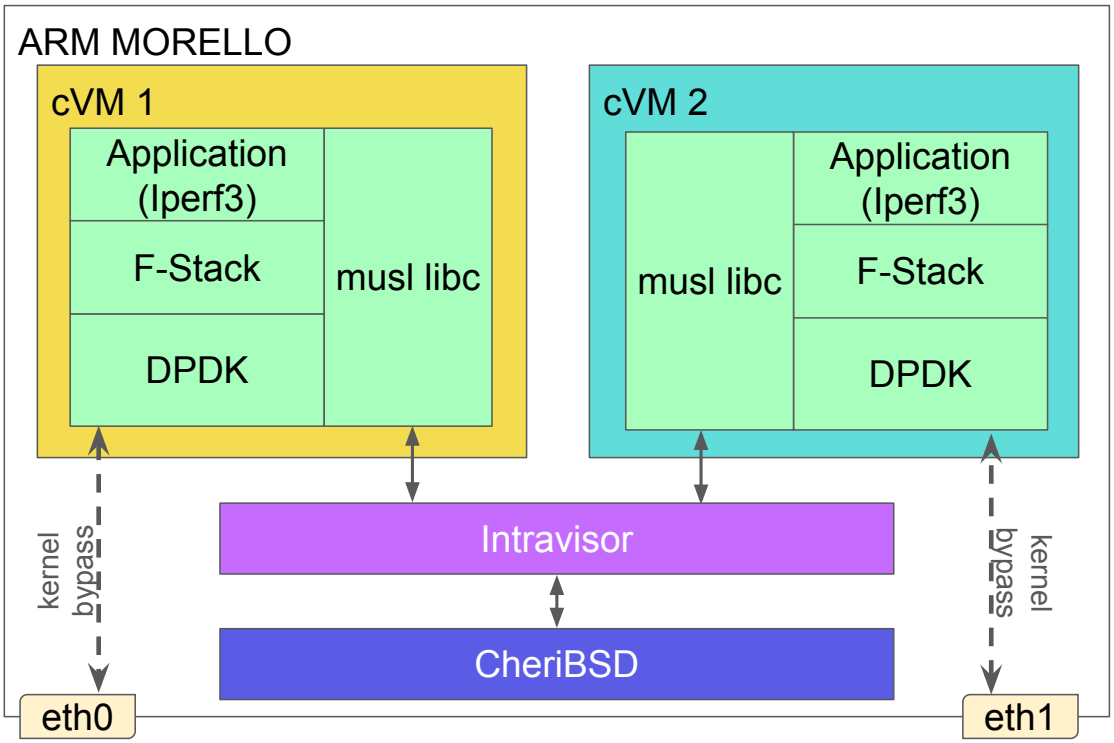


Figure 4.1: Scenario 1: Replication of the entire stack into two different *cVMs*. The arrow shows where the *trampoline* function is executed.

4.2.1 Scenarios

Baseline. *Baseline* consists of a non-CHERI full-network stack running on CheriBSD. For Scenario 1, DPDK, F-Stack, and *iperf3* run as two independent processes, with each process using a dedicated Ethernet port. For Scenario 2, the baseline consists of a single process running the complete DPDK/F-Stack stack and *iperf3*. We could not evaluate a non-CHERI equivalent of the contended Scenario 2 configuration because the DPDK implementation available on CheriBSD did not support the required multi-threaded execution on a shared network port. Therefore, the contended configuration is evaluated only for the CHERI-based design.

Scenario 1. In Scenario 1, each compartment (*cVM1*, *cVM2*) contains one network application (*iperf3*), the F-Stack TCP/IP library, and the DPDK user-space network layer. The components run in *hybrid* mode and are linked against a modified *musl libc* library that provides the *trampoline* wrappers towards the Intravisor. Each *cVM* works as independent network processing unit, with its own dedicated stack and Ethernet interface ("eth0" and "eth1"). This configuration provides a complete compartmentalization approach, ensuring that any security

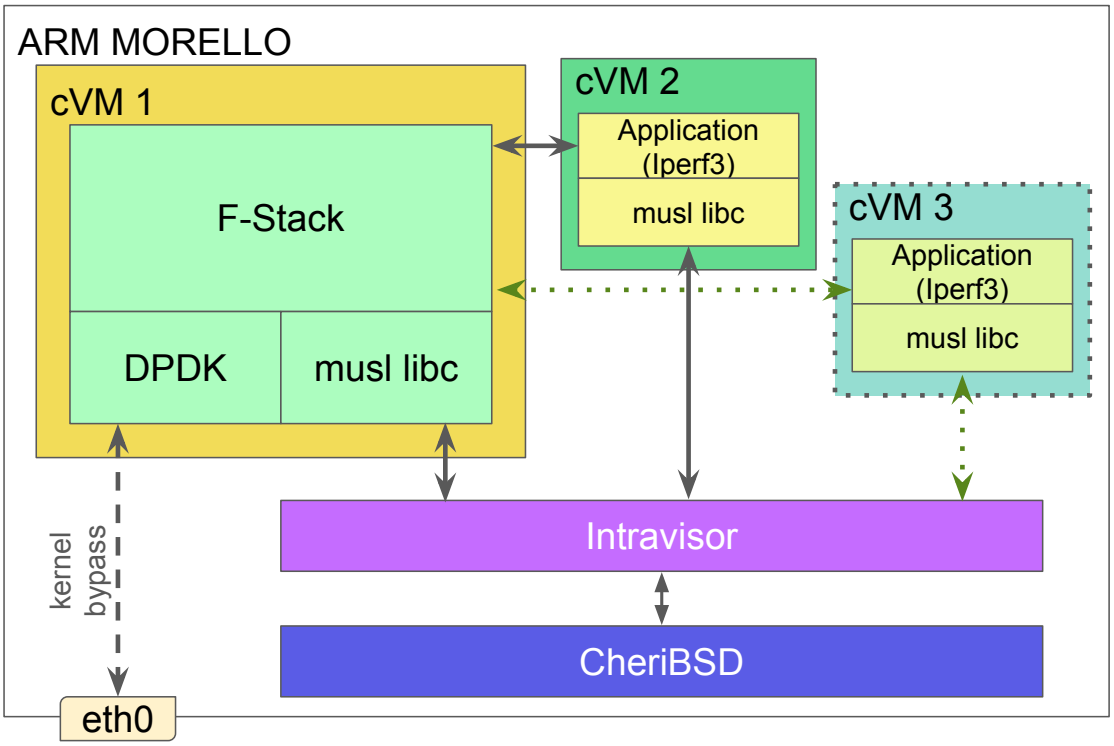


Figure 4.2: Scenario 2: Isolation of two distinct applications *cVMs* from a F-Stack and DPDK cVM in the contended variant.

breach does not affect the other *cVMs*. Moreover, it also serves as a baseline for comparing performance and isolation. The *trampoline* function is only executed when *musl libc* syscalls are executed.

Scenario 2. This scenario separates the application (*iperf3*) from F-Stack and DPDK, thus allowing for potential space optimization in constrained environments. Here, *cVM1* runs the TCP/IP library and DPDK stack, while *cVM2* (and *cVM3*) execute (separate instances of) the application. All compartments link against the modified *musl libc* library to interact with the Intravisor. *cVM1* is responsible to read/write from/to the Ethernet driver queues and sort to the right file descriptor, when requested. This scenario requires a *mutex* to coordinate the execution of the F-Stack API functions and the main-loop execution, which creates a potential contention issue for this shared resource. Therefore, we have evaluated this scenario in two configurations (see Section 4.3) with one application (uncontended) or two compartmentalized applications (contended) respectively. Figure 4.2 depicts the contended configuration, i.e., with two compartmentalized applications.

4.2.2 Implementation

Intravisor. In CAP-VMs [261], *hybrid cVMs* encapsulating applications use a combination of *musl libc* [6] and Linux Kernel Library (LKL) [243]. LKL turns the Linux kernel into a library, providing kernel services—such as file systems and networking—to be run in user-space, making the resulting application highly portable to various environments—e.g., virtualized and bare-metal. However, this may increase the attack surface, and introduces overhead by emulating kernel syscalls in user-space. Conversely, our DPDK and F-Stack are designed to fully execute in user-space and interact with the kernel only at boot time. They can therefore be streamlined within *cVMs*, removing the additional LKL layer. We directly connected *musl libc* in the Intravisor substituting supervisor call instructions (`svc`) with dedicated *trampoline* functions. Specifically, a *trampoline* passes through the syscall ID and arguments, stores register states. It also loads the correct PCC and DDC, and use them to jump into the cVM/Intravisor using CHERI specific instruction (e.g., `blrs` for the Arm Morello). Avoiding LKL helps reducing overhead as well as the overall footprint and attack surface, making the *cVMs* potentially more secure. Similarly to FreeBSD, CheriBSD native libc is not fully compatible with *musl libc*. We adapted the Intravisor proxy function to properly translate *musl libc* calls into CheriBSD libc equivalents. For instance, *musl libc* uses *futex* for thread synchronization, while CheriBSD uses *umtx* [7].

DPDK. DPDK was ported to the CHERI Morello using *hybrid* mode [2]. However, the existing support is very limited and does not initialize nor use any NIC interface. We implemented the module that detaches the NIC from kernel-space and attaches it to user-space, ensuring that the memory allocations it requests are performed with the correct permission flags. Minor adjustments were required to enable execution using *musl libc*.

F-Stack. F-Stack leverages the FreeBSD network stack, but, prior to this work, no available implementations of F-Stack for CHERI existed. The first step in our implementation involved porting F-Stack to work with CheriBSD. This required adaptations on both OS and F-Stack sides. CheriBSD was modified to accommodate the specific requirements of F-Stack, by streamlining the use of the network stack. DPDK and F-Stack operate in polling mode, meaning that the application must explicitly request to read/write from/to the device. After an initialization phase, a main-loop is executed, with the key tasks being: (i) process the ring buffers of the DPDK Ethernet driver; and, (ii) execute a user-defined function where calls to F-Stack API functions can be made. F-Stack also required adaptations to work with capabilities. We extended its data structures to use capabilities and we have extended its Application Programming Interface (API). Applications can integrate with F-Stack through its API, which closely resembles the BSD socket API. For instance, F-Stack exposes `ff_socket()` and `ff_write()` functions that are the

Table 4.1: Number of line of codes added/modified

Library	LoC	global amount in percentage
F-Stack	152	0.99%

equivalent of `socket()` and `write()`. This allows developers to adapt existing network applications for F-Stack with minimal changes. To work with capabilities, the signatures of such APIs has to be modified. For instance, the signature of the `ff_write()` has been modified as follow:

```
-   ssize_t ff_write(int fd, const void * buf, size_t nbytes);
+   ssize_t ff_write(int fd, const void * __capability buf, size_t nbytes);
```

The total amount of modified lines of code (LoC) to achieve our objectives is shown in Table 4.1. Since the FreeBSD version originally supported by F-Stack is v13.0, while we used CheriBSD v23.11 based on FreeBSD v14.0, the modifications in Table 4.1 include not only a transition to a CHERI-enabled library, but also the adjustments needed to adapt the native library to the version of CheriBSD. Overall, this is in line with the usually expected modifications to port a library [309].

Application. To work with our setup, we initially ported *iperf3* to work with the F-Stack API. Next, we replaced the `select` function, with the `epoll` mechanism, which adapts better to F-Stack. For Scenario 2, we also implemented the wrapper functions to the API of F-Stack to do the cross-compartment jump between the running application and the *cVM1*.

4.3 Evaluation

In this section, we evaluate the performance and security benefits of the system architecture and isolation configurations implemented on the Morello platform. In addition to verifying the security achieved through CHERI compartmentalization, the evaluation focuses on two main aspects: performance in terms of TCP bandwidth and the execution-time cost of `ff_write()`, the application-facing F-Stack function used to send data. From `ff_write()`, data enters the F-Stack TCP/IP processing path and is eventually passed to DPDK for transmission. We therefore use `ff_write()` to measure the overhead introduced by cross-compartment communication at the application-to-network-stack boundary. To measure these values, we use `clock_gettime()` with CLOCK_MONOTONIC_RAW. Note that, since DPDK and F-Stack operate in polling mode to minimize latency, CPU utilization is not a meaningful indicator.

Under CHERI, each network-stack compartment is restricted to its DDC capability. We verified the effectiveness of compartmentalization modifying applications

```
WARNING: Adding ifaddrs to all fibs has been turned off by default. Consider tuning net.add_addr_allfibs if neede
d
Timecounter "ff_clock" frequency 100 Hz quality 1
ff_freebsd_init() completed
f-stack-0: No addr6 config found.
f-stack-0: Ethernet address: 98:b7:85:1e:eb:7a
Starting..
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------

Accepted connection from 192.168.1.1, port 40892
/tmp/iperf3.ECNPbd
/tmp/iperf3.ECNPbd
[1027] local 192.168.1.2 port 5201 connected to 192.168.1.1 port 40906
[ ID] Interval           Transfer     Bandwidth
[1027]   0.00-1.00   sec   108 MBytes   906 Mbits/sec
[1027]   1.00-2.00   sec   112 MBytes   941 Mbits/sec
[1027]   2.00-3.00   sec   112 MBytes   941 Mbits/sec
[1027]   3.00-4.00   sec   112 MBytes   941 Mbits/sec
[1027]   4.00-5.00   sec   112 MBytes   941 Mbits/sec
[1027]   5.00-6.00   sec   112 MBytes   941 Mbits/sec
In-address space security exception (core dumped)
cheribsd:~/intravisor/install/case-1 #
```

Figure 4.3: Applications accessing memory outside their boundaries cause exceptions under CHERI.

to access memory ranges outside their valid boundaries. As expected, CHERI triggers a CAP-out-of-bound exceptions, as shown in Figure 4.3. We additionally tested cross-compartment isolation by instantiating two independent cVMs and attempting to access memory belonging to the other cVM. The attempted access produced the same CHERI capability exception observed for invalid memory accesses, and the target memory remained inaccessible. This demonstrates that the capability bounds enforced by our compartmentalization prevent a cVM from directly accessing another cVM's memory.

For the network-related evaluations, we executed our modified *iperf3* to calculate the maximum bandwidth in both server (receiver) and client (sender) modes. We define *efficiency* as the measured bandwidth relative to the *nominal* bandwidth available to each application. For the uncontended configurations, this is 1 Gbit/s, corresponding to the capacity of the Ethernet link. In the contended Scenario 2, two applications share the same 1 Gbit/s link; therefore, we use 500 Mbit/s as the nominal per-application share. In Scenario 1 and the corresponding two-process *Baseline* (P1 and P2), both Ethernet ports achieve approximately 658 Mbit/s. We attribute this limitation to the PCI NIC configuration: the NIC is connected through a single PCIe line, and we observed the same throughput limitation when using the NIC without DPDK. This suggests that the limitation is related to the PCI NIC or its PCIe connection rather than DPDK or CHERI compartmentalization. We therefore consider this limitation a property of the experimental platform rather than an overhead introduced by our approach. In all scenarios, as shown in Table 4.2, we reached the maximum bandwidth possible with our hardware setup.

Table 4.2: Results of TCP benchmarks in the three scenarios, both server and client sides. Values expressed in Mbit/s

Baseline				
Modes	Server	Efficiency	Client	Efficiency
Baseline (P1)	658	65.8%	757	75.7%
Baseline (P2)	658	65.8%	757	75.7%
Scenario 1				
Modes	Server	Efficiency	Client	Efficiency
cVM1	658	65.8%	757	75.7%
cVM2	658	65.8%	757	75.7%
Baseline				
Modes	Server	Efficiency	Client	Efficiency
Baseline (P2)	941	94.1%	941	94.1%
Scenario 2 (uncontended)				
Modes	Server	Efficiency	Client	Efficiency
cVM2	941	94.1%	941	94.1%
Scenario 2 (contended)				
Modes	Server	Efficiency	Client	Efficiency
cVM2	470	94%	531	106.2%
cVM3	470	94%	410	82%

In the contended Scenario 2, cVM2 reaches 106.2% of its nominal share, while cVM3 reaches 82%, showing the lack of fairness between the two applications.. We defer the investigation of Quality-of-Service (QoS) approaches or the integration of DPDK QoS features to future works.

In the evaluation of the execution time of `ff_write()`, each test measures the execution of 1 million iterations, and Figures 4.4-4.6 present averages and standard deviations as box plots. Moreover, in all the experiments, outliers ($\approx$ 10% of the iterations) are removed with a standard IQR strategy. Figure 4.4 compares Scenario 1 with *Baseline*. The first two boxes from the left depict the execution time for *Baseline*, while the other two the execution time of Scenario 1. Note that, in cVMs we can't directly access the timers of the system, the execution time always includes a cross-compartment jump to the Intravisor, the execution of the syscall in CheriBSD, and the return from the Intravisor to access them. As seen in Figure 4.4, the impact of the CHERI compartment is minimal, and amounts to approximately 125 ns, corresponding to the additional indirections required by the musl - Intravisor mechanism. In both *Baseline* (P1 and P2) and *cVM1* for Scenario 1, the 25th and 75th percentiles are identical, leading to the absence of a

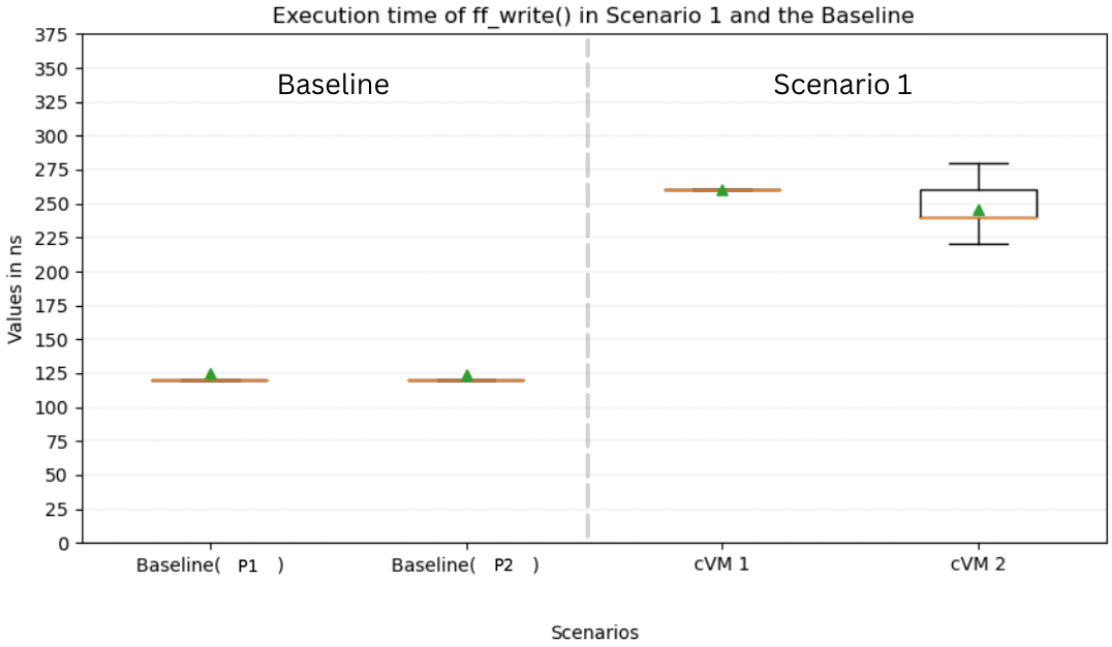


Figure 4.4: Execution time of the `ff_write()` function in Scenario 1, compared with *Baseline*.

visible box in the box plot. This indicates that more than 50% of the results were identical. The experiments were repeated several times, consistently yielding the same outcomes.

In Scenario 2, the cost of the cross-compartment jump and return in the cVM is included in the measurements. Moreover, due to the required synchronization between the main-loop of F-Stack and the execution of the `ff_write()` (see 4.2), the measures also include the time necessary to acquire the mutex. Figure 4.5 shows the execution time measurements of the `ff_write()` of the uncontended setup compared with *Baseline*. In this case, we increased the interval between two consecutive `ff_write()` to reduce the possibility to be blocked for a long time by the mutex. Compared to Scenario 1, despite the additional indirection to jump between the two *cVMs* and the handling of the mutex, the experienced slow down is contained to approximately 200 ns.

The effect of the mutex contention is evident in the contended Scenario 2, where three cVMs can acquire and wait on the mutex. As shown in Figure 4.6, operations on the mutex result in an overhead of around 19.000 ns (152× slowdown). Nonetheless, despite the overhead introduced for handling the mutex, the contended Scenario 2 reaches the maximum bandwidth supported by the hardware in both server (receiver) and client (sender) modes (see Table 4.2). As future work, we plan to investigate in details the impact of different locking strategies to further

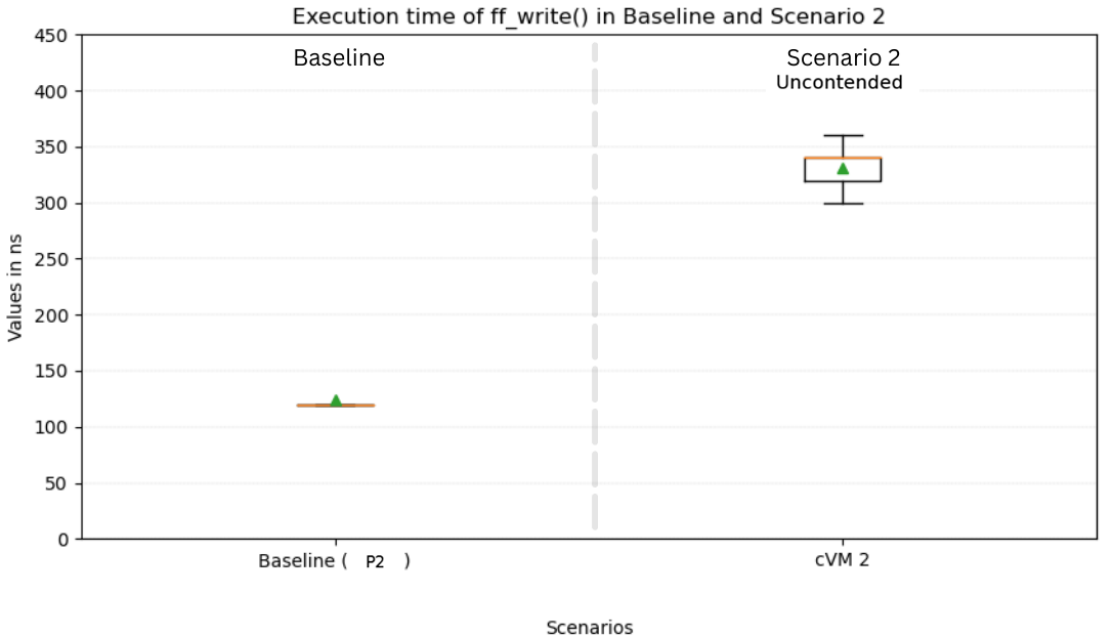


Figure 4.5: Execution time of the `ff_write()` function in Scenario 2, compared with *Baseline*.

reduce the overhead of our designs.

4.4 Related Work

Several works have addressed different aspects of ensuring security in embedded systems [21, 247, 250]. These efforts span the entire embedded system stack, from designing robust and energy-efficient hardware to developing secure, lightweight operating systems[335] and strong encryption protocols. For instance, the OS plays a critical role in securing UAVs, as it must implement countermeasures specific to embedded systems to defend against different security attacks [154]. Prior studies on compartmentalizing network stacks in OSs, like FreeRTOS [138] and RIOT [42], have relied on coarse-grained (task-level) isolation using MPUs or MMUs, still leaving systems vulnerable to memory-related attacks such as buffer overflow [19, 216]. CHERI can mitigate these vulnerabilities by providing fine-grained memory access control, replacing or cooperating with MPU/MMU-based solutions [322]. Other hardware-based solutions also address compartmentalized software design by isolating critical trusted security functions, like authentication, from the main untrusted OS. Nevertheless, they still rely on MPU-like mechanisms for isolation. For instance, Arm TrustZone [37] uses Secure Attribution Unit (SAU)

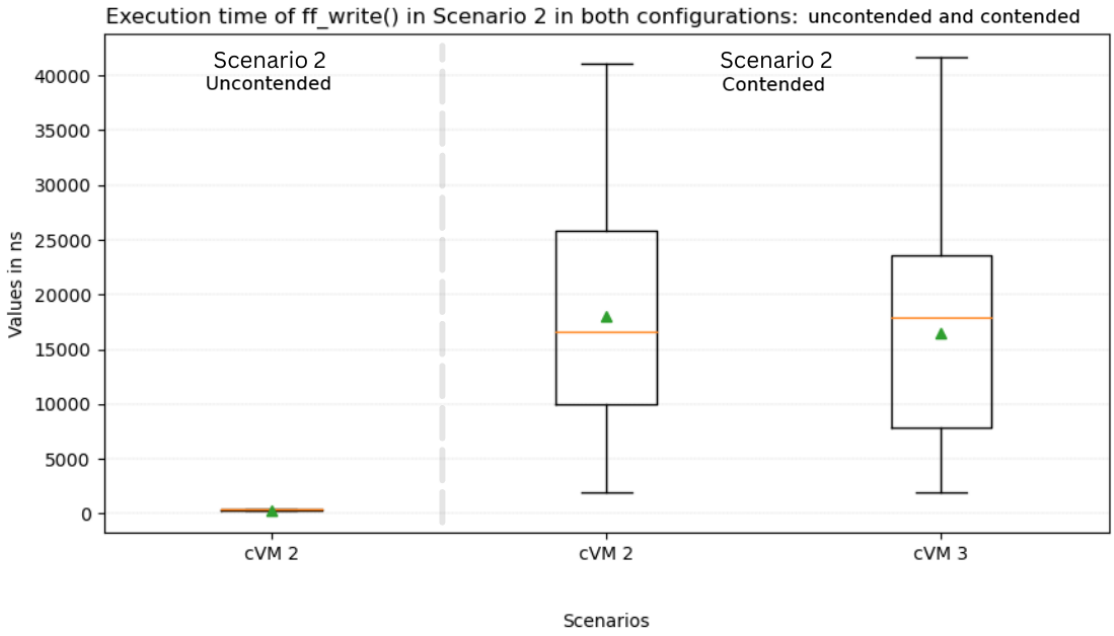


Figure 4.6: Execution time of the `ff_write()` function in Scenario 2 in uncontended and contended configurations.

or Implementation Defined Attribution Unit (IDAU) to separate memory into secure and insecure partitions. As a consequence, CHERI would allow enhanced internal memory protection, mitigating attacks from the untrusted OS [290].

CHERI-enabled architectures, help addressing mixed-criticality environments by isolating vulnerable components. One notable work on CHERI compartmentalization in embedded systems is CompartOS [26]. The authors leverage CHERI capabilities to isolate linkage modules by using a CHERI extension to the library *libdl*. However, as also documented in this work, this approach introduces some overhead compared to static solutions, which are typically more common in the safety-critical embedded systems. CAP-VMs [261] was also extended with a new mechanism to reduce the duplication of the libraries running in the cVMs has been implemented [264]. This mechanism could also be used as a future integration for our work to reduce the memory utilization.

DPDK has been routinely used in all those contexts requiring low network overhead and high bandwidth (e.g., [45, 47, 284]). Among those, F-Stack is the predominant TCP/IP library used, with good scalability and performance [217]. In addition, the high-performance DPDK has been integrated into embedded systems. One example of this integration involves modifying LwIP [5]—a lightweight TCP/IP stack designed for embedded systems—to run on top of DPDK, exploring the use of DPDK also for constrained environments [246].

4.5 Summary

This chapter presented the potential of CHERI to provide strong security properties through compartmentalization of the software components of a network stack. Specifically, we presented different trade-offs for designs that isolate DPDK, F-Stack, and a network benchmark application using Intravisor-based *cVMs*. Our evaluation on the CHERI-enabled Arm Morello platform has validated the effectiveness of CHERI compartmentalization and has shown the low overhead potential of our *cVM*-based design. However, despite achieving full bandwidth, the evaluation also underlined the impact of synchronization among different *cVMs*.

Chapter 5

Security for Drones: CHERI-enabled PX4/NuttX

This chapter presents the design and evaluation of a security-oriented hardware–software platform for UAVs centered on the CHERI ISA extension for RISC-V. The work spans three interrelated contributions that together form the *CHERI-UAV* project.

The chapter is organized as follows. Section 5.1 describes the full CHERI-UAV system architecture: the multi-domain Al Saqr System-on-Chip (SoC) that constitutes the hardware substrate, and the PX4/NuttX-based software stack that runs on top of it. Section 5.2 details the open-source CVA6H-CHERI core—the first RISC-V CPU to simultaneously combine hardware virtualization and CHERI support—covering the microarchitectural extensions required to enforce capability semantics across the full pipeline. The remaining sections (Sections 5.4– 5.10) present the design, implementation, evaluation, and security analysis of a mixed hybrid/pure-capability architecture for PX4/NuttX, evaluated on an FPGA-based CVA6-CHERI prototype.

5.1 The CHERI-UAV Platform

This section describes the hardware and software architecture of the CHERI-UAV platform, which provides the foundation on which the contributions of this chapter rest. The platform is designed to meet the competing demands of drone operation: application-class compute performance, strict power and energy budgets, real-time guarantees, and now—through CHERI—hardware-enforced memory safety and compartmentalization.

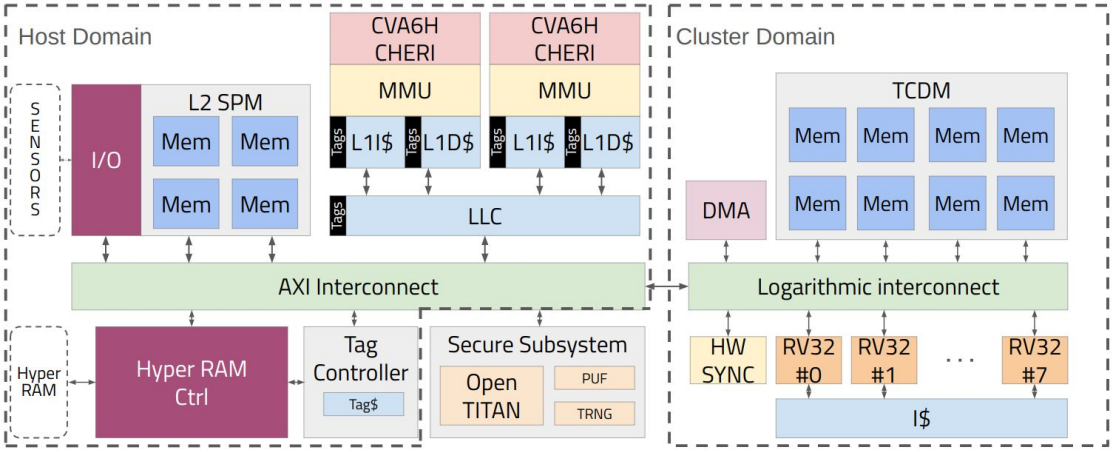


Figure 5.1: The CHERI-UAV hardware platform. The Host Domain (HD) contains two CVA6H-CHERI cores with MMU and private L1 caches, a shared LLC, and a Tag Controller for CHERI-tagged memory. The Cluster Domain (CD) targets QNN inference on a PULP parallel cluster. The Secure Subsystem Domain (SSD) provides a hardware Root of Trust following the OpenTitan design.

5.1.1 Hardware Architecture

The CHERI-UAV hardware platform (Figure 5.1) is built on top of the RISC-V *Al Saqr* SoC—an evolution of the Shaheen platform [288]—extended with CHERI support. Al Saqr is organised into three functionally distinct domains.

Host Domain. The Host Domain (HD) contains two 64-bit CVA6 cores [280] extended with the CHERI ISA for RISC-V, resulting in the CVA6H-CHERI core described in Section 5.2. This makes the platform the first to combine CHERI and RISC-V hardware virtualization in a single open-source core. The CVA6 core is well-established in the embedded and real-time research community: it is open-source, fully RISC-V compliant, and offers a performance envelope appropriate for drone autopilot workloads while keeping power consumption low—a critical property for battery-constrained platforms. The HD further integrates peripheral subsystems for SPI, CAN, Ethernet, and UART, an L2 scratchpad memory, a shared Last-Level Cache (LLC), and HyperRAM connectivity. A dedicated *Tag Controller* sits between the AXI interconnect and main memory: it stores and enforces the one-bit validity tag per aligned 16-byte granule that CHERI requires, propagating tags transparently through the cache hierarchy without software intervention.

Cluster Domain. The Cluster Domain (CD) is based on the PULP parallel ultra-low-power cluster [255] and targets AI-accelerated edge workloads running on board the drone. Concretely, it is used to execute fine-tuned Quantized Neural

Networks (QNNs) for real-time object detection and classification, offloading this compute-intensive task from the flight-control cores while keeping it within the same SoC power envelope. The CD exposes a Tightly Coupled Data Memory (TCDM) fabric connected to an array of RV32 compute cores via a logarithmic interconnect and a DMA engine.

Secure Subsystem Domain. The Secure Subsystem Domain (SSD) provides a hardware Root of Trust (RoT) following the open-source OpenTitan design [189]. It manages cryptographic key material, provides a Physical Unclonable Function (PUF) for device-unique identity, a True Random Number Generator (TRNG), and enforces a secure boot chain through the Open TITAN interface. The RoT ensures that the system boots only authenticated firmware, and that keys used by cryptographic functions are hardware-protected and never exposed to software.

5.2 CVA6H-CHERI: Microarchitecture and Design

The hardware evaluations reported in this chapter rely on the CVA6H-CHERI core: an extension of the open-source 64-bit CVA6 RISC-V processor [280, 333] with full support for the CHERI ISA version 9 [306]. To our knowledge, this is the first publicly available RISC-V CPU to combine hardware virtualization and CHERI support, and it will be released as an open-source artefact to both communities.

5.2.1 Microarchitectural Extensions

The baseline CVA6 is a six-stage, application-class RISC-V core (RV64GC plus Hypervisor extension) with a configurable issue stage. Four central units of the CVA6 pipeline were modified, and a new functional unit was added to the execute stage.

Merged register file. All 31 general-purpose integer registers are extended from 64 to 128 bits to hold either a plain integer or a full CHERI capability (address + metadata), plus one out-of-band tag bit.S This doubles the register file width and requires all data paths between pipeline stages to carry 129-bit values throughout.

CHERI Logic Unit (CLU). A new functional unit added to the execute stage implements all capability-specific instructions: inspection (e.g., `CGetLen` and `CGetPerm`), derivation (e.g., `CSetBounds` and `CAndPerm`), sealing and unsealing (such as `CSeal`, `CUnseal`, and `CInvoke`), and the tag-memory access instruction (`CLT`) for efficient bulk tag queries. The CLU also performs bounds and permission

checks on all memory-referencing instructions and raises a CHERI exception on any violation.

PC Capability and control-flow integrity. The Program Counter is extended into a *Program Counter Capability* (PCC), bounding the range of addresses from which instructions may be fetched. The Branch Unit is adapted to verify that branch targets remain within the current PCC bounds, providing hardware-enforced control-flow integrity without additional software instrumentation.

CHERI Control and Status Registers. A set of CHERI-specific CSRs is added at each privilege level, covering: the *Default Data Capability* (`DDC`), which bounds all legacy integer memory accesses in hybrid mode; the machine-level trap capabilities `mtcc` (Machine Trap Code Capability) and `mtdc` (Machine Trap Data Capability), which the hardware loads automatically on any interrupt or exception to ensure trap handlers execute with precisely defined authority; and the exception-return capabilities `mepcc` and `epcc` for machine and supervisor levels.

Tagged memory and cache integration. The cache subsystem—L1 data cache, shared LLC, and the Tag Controller between the AXI bus and HyperRAM—is extended to store and propagate one tag bit per aligned 16-byte granule throughout the memory hierarchy. Load instructions restore the tag bit into the destination register; store instructions write back the tag (and clear it if the stored value is not a valid capability). This ensures that tag integrity is maintained transparently across all levels of the memory hierarchy with no additional software cost.

Hypervisor extension integration. All capability mechanisms are extended to cover Hypervisor load/store instructions (`HLV.*`, `HSV.*`), enabling a CHERI-aware hypervisor—such as a ported version of the Bao hypervisor [200]—to mediate capability access across virtual machines. This combination provides the foundation for future work that explores flexible combinations of hardware virtualization and CHERI compartmentalization in mixed-criticality drone deployments.

5.3 Problem Statement

PX4/NuttX adopts a flat address space by default [245], allowing every module—including flight-critical control loops, MAVLink communication, and low-level drivers—to address the full process memory without hardware-enforced boundaries. While this design minimizes footprint and inter-module communication latency, it means that a single exploited vulnerability in any module can corrupt shared state

and propagate into safety-critical functionality. The weak security architecture of such autopilot stacks have been repeatedly exploited (e.g., [33, 252]) to inject malicious traffic and alter drone behavior exploiting reported vulnerabilities that include multiple instances of memory corruption (see Section 5.8.2).

MMU-enabled NuttX configurations are rare in the RISC-V flight-controller ecosystem,[1] and even where an MMU is present, protection is confined to task-level boundaries: the tightly-coupled modules within a single PX4 task—*work queues* in particular—share an address space and remain mutually unprotected.

As established in Chapter 3, CHERI provides fine-grained, intra-address-space memory protection and compartmentalization without requiring the decomposition into separate processes that a microkernel-based approach would demand. The central question this chapter addresses is whether CHERI capabilities can enforce module-level compartmentalization across the full PX4/NuttX stack without violating hard real-time constraints, and at what overhead relative to both the unprotected baseline and an MMU-based alternative.

To answer this question, we design and evaluate a mixed hybrid/pure-capability architecture for PX4/NuttX on an FPGA-based CVA6-CHERI prototype. CAP-VMs inspired our compartmentalization and communication model, but its isolation properties were not designed for tightly coupled embedded systems, and whether they could carry over has so far remained an open question. Our work addresses this question through a design specifically tailored to the constraints and requirements of embedded platforms. As discussed in Section 5.6.3, our CHERI-based design integrates fully with the memory, execution, and communication model of PX4/NuttX. It also minimizes application loading latency to meet the boot-time requirements of drone systems and eases the maintenance of CHERI-specific services by matching them with standard NuttX system calls. The contribution includes the compartmentalization design, its implementation, a performance evaluation covering low-level primitives and representative flight-stack workloads, and a security analysis against the identified CVE set.

5.4 Preliminary concepts

5.4.1 PX4/NuttX Autopilot

PX4 [245] is a widely used open-source autopilot framework for UAVs, targeting resource-constrained embedded platforms and supporting a wide range of airframes, sensors, and peripherals. As shown in Figure 5.2, the system is built on top of NuttX [35], a POSIX-compliant real-time operating system (RTOS) providing core

[1] To the best of our knowledge, Section 5.5 compares against the only currently available MMU-based architecture in this space.

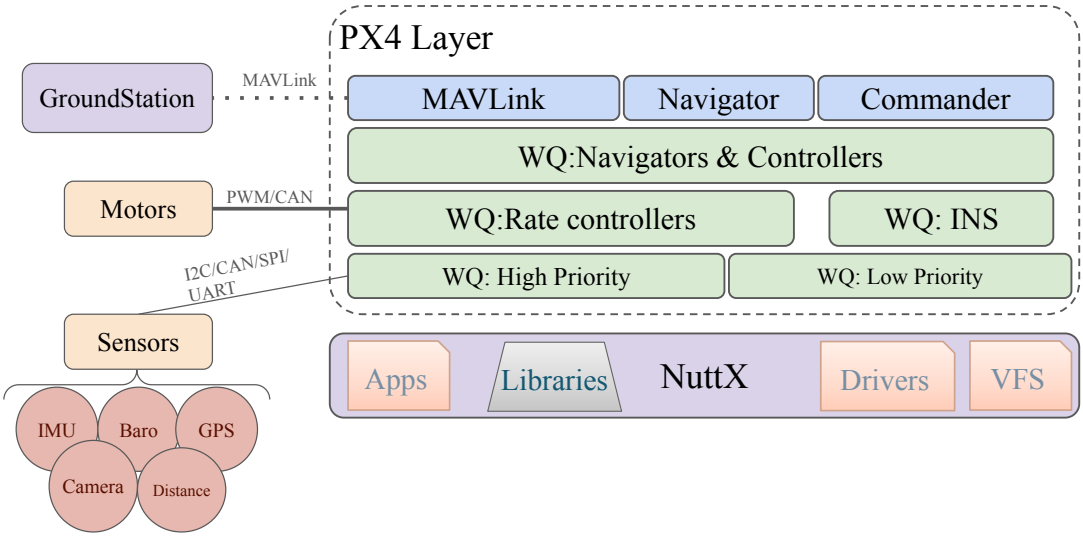


Figure 5.2: PX4/NuttX autopilot software stack. PX4 main modules (blue) runs as dedicated tasks, on-demand modules and access to sensors are multiplexed by work-queues modules (WQ, green). Base services and low-level drivers are provided by the NuttX operating system.

services such as task scheduling and low-level drivers (e.g., SPI, CAN, UART, and I2C) for hardware access, as well as libraries (e.g., *libmm* for memory management, libc, libm, libc++), a Virtual File System (VFS), and utility applications (e.g., the NuttShell (NSH) terminal).

PX4 is organized into modules that are mapped into execution contexts using a combination of dedicated tasks and shared work queues (WQs). Each module implements a functionality of the flight stack. Time-critical modules typically run as dedicated tasks, ensuring predictable scheduling and minimal latency, while to reduce the total number of schedulable tasks, other modules execute as callbacks on WQ tasks.

Figure 5.2 illustrates a representative PX4 configuration. Core modules such as *MAVLink* (Micro Air Vehicle Link), responsible for external communication, *Commander*, which manages the system state machine, and *Navigator*, which is responsible for the autonomous flying mode, execute as dedicated tasks due to their timing and control requirements. In contrast, multiple functional components are grouped into work queues: for example, the *WQ: High Priority* includes sensor-related modules (e.g., camera, manual control input, and LiDAR), the *WQ: Navigator & Controllers* filters-out noisy sensor raw data and exposes it via a common API, the *WQ: INS* (Inertial Navigation/GNSS) fuses this data with localization information using an Extended Kalman Filter 2 (EKF2) to determine

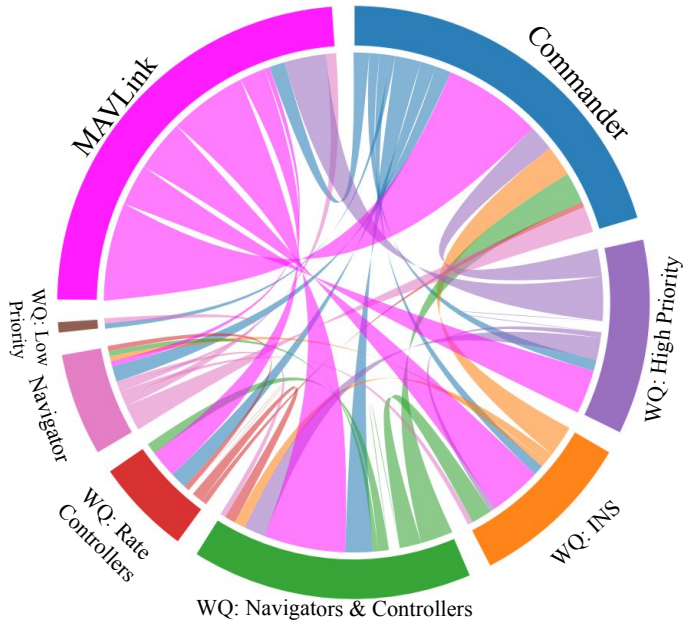


Figure 5.3: Visual representation of the tight uORB-based pub/sub interactions among PX4 modules. The width of the bands relates to the number of topics and links.

position, attitude, rotation, and velocity of the drone. The *WQ: Rate controllers* manages the actuators to control the motors of the drone, while the *WQ: Low Priority* usually includes low-frequency operating modules.

Internally, PX4 modules communicate through the Micro Object Request Broker (uORB), a lightweight publish/ subscribe messaging system. Modules exchange data using named topics, such as sensor readings or control commands. uORB relies on shared memory buffers accessed via a file-like abstraction: producers write data to a buffer, and consumers read from it. Due to the type of performed computations, communication among modules via uORB topics is frequent and highly intertwined. Figure 5.3 illustrates this internal communication structure and shows both its complexity and the large number of topics involved. In a PX4 quadcopter configuration, for instance, we counted 88 distinct topics and 136 publisher–subscriber links.

PX4 achieves low memory footprint and low latency on resource-constrained platforms by leveraging the *flat* NuttX configuration, in which kernel and application code share a single address space. From a security standpoint, however, this design is critical. While it simplifies data exchange (boundaries among PX4 modules and between PX4/NuttX are simple function calls), it also permits one module to directly corrupt memory used by another or by the kernel. The situation is aggravated by

external communication, which enlarges the attack surface of UAV software. PX4 relies on the MAVLink protocol [100] for communication with external systems and a dedicated module handles message transmission and reception (Figure 5.2). Because this path parses untrusted input and interacts with shared memory structures, it is particularly exposed to memory-safety vulnerabilities (Section 5.8.2).

When hardware support is available, NuttX can also be deployed in a *protected* configuration, in which a Memory Protection Unit (MPU) is used to separate kernel and user space. In this mode, NuttX provides a system-call interface (*syscall*) that allows user-space applications to invoke kernel services (e.g., `read`, `write`, etc.) However, even in protected mode, the resulting isolation remains limited because all PX4 modules still share the same user-space address space.

More recently, the NuttX port presented in [282] extends PX4/NuttX to platforms equipped with a MMU, with a particular focus on larger RISC-V cores. In this setup, which enables virtual memory and task-level isolation, uORB buffers are mapped to shared memory (shm) files that provide access to the kernel-managed shared memory. The main drawbacks are an increased latency across address-space boundaries and additional overhead for kernel transitions. System size also increases, since applications must be properly loaded into user space rather than linked directly into a single binary image.

5.5 Alternative Designs for Secure UAV

Building memory-safe UAV flight control software requires balancing safety, real-time performance, and compatibility with existing PX4/NuttX deployments. Current PX4 modules (Section 5.4.1) are tightly coupled, interact frequently through shared buffers, and share the same memory space. The need to minimize porting efforts, while preserving timing and latency requirement, constrains the choice of protection and fault-isolation mechanisms that can be adopted. Secure UAV software stacks should preserve the real-time properties of the underlying operating system (e.g., NuttX), including predictable scheduling and low overhead for interrupt handling and context switching, while ideally satisfying the following memory-safety properties.

P1. Fine-grained Memory Safety: Protect flight-critical modules from common low-level errors (like stack and heap overflows) and invalid pointer references. For instance, a violation in sensor processing module should not corrupt the memory of another module.

P2. Fault Isolation: Ensure that faults or memory violations remain contained within a module or functional domain, preventing cascading failures.

P3. Incremental Integration: Allow existing PX4 functionality to run unmodified alongside hardened, isolated modules, supporting gradual system evolution without requiring a full system redesign or immediate migration of the existing software stack.

While the default PX4/NuttX flat address space and MPU-based protection offer good performance but insufficient flexibility and isolation, MMU- and CHERI-based designs can balance these properties.

5.5.1 MMU-based isolation

MMU-based designs realize task-level separation through virtual memory, preventing one task from directly corrupting another (**P2**). However, in PX4/NuttX-based UAV systems, this approach has limitations. PX4 architecture relies heavily on frequent communication between modules, which makes process boundaries expensive in terms of IPC overhead. This is further worsen by PX4 use of shared work queues, were multiple PX4 modules execute within the same scheduling context. Achieving MMU-based isolation in such a model would require decomposing work queue execution into separate tasks, increasing the number of scheduled entities in the system. This leads not only to additional IPC overhead, but also to increased scheduling and context-switch pressure on NuttX kernel. In addition, address translation and context switching introduce timing variability that may affect high-frequency control loops, particularly under computational load. Concerning isolation (**P1**), the page-level granularity of MMU protection cannot isolate fine-grained components within a single PX4 module, limiting its effectiveness for intra-process faults. Incremental integration (**P3**) also requires explicit support: code written for flat and MPU-based configurations should be refactored to use a syscall-like interface, so that it can be recompiled easily when migrating to MMU support.

Although we compare against a RISC-V MMU-based configuration in Section 5.8, we note that such a configuration have been forked only recently from the NuttX codebase [282], and its integration upstream is still in progress.

5.5.2 CHERI-based Isolation

The fine-granular protection provided by CHERI's capability has the potential of satisfying both **P1** and **P2** properties, but incremental integration (**P3**) and impact on the RTOS are a possible concern.

Pure-capability mode In this mode, all memory accesses are mediated through capabilities, enforcing spatial memory safety. This provides the strongest isolation

(**P1**, **P2**) guarantees among the considered approaches.

Unfortunately, this approach is ill-suited to incremental adoption (**P3**). It requires NuttX to adopt a fully capability-aware execution model, where capabilities replace conventional pointers throughout the kernel, system libraries, and runtime interfaces. All core OS components must be adapted to operate on capability-aware data structures. This fundamentally changes the OS execution model and requires a capability-aware toolchain and runtime environment. Such a design implies a non-trivial migration effort at the OS level, as existing NuttX binaries and libraries (which rely on a traditional pointer-based ABI) cannot execute without recompilation or adaptation. While PX4 modules are also affected, the primary integration burden lies in adapting the NuttX infrastructure to support capability-only execution.

Hybrid-capability mode This mode provides a more incremental approach by allowing conventional pointers and capabilities to coexist. Support at NuttX OS level is still required (e.g., context-switch awareness of capabilities and interrupt managements), however, unlike pure-capability mode, the existing pointer-based execution model is preserved, allowing legacy PX4 and NuttX components (e.g., applications and libraries) to continue operating without modification. This reduces the scope of required changes to selected components, rather than a full redesign of the operating system. As a result, capabilities can be selectively applied to security-critical UAV components (e.g., communication handlers or sensor processing pipelines), while leaving the rest of the system unchanged.

This approach has the potential to provide effective **P1** and **P2** isolation, albeit weaker than pure-capability, while simplifying incremental integration in existing UAVs (**P3**).

5.5.3 MMU + CHERI Isolation

A combined MMU- and CHERI-based design provides fine-grained intra-address-space capability enforcement within task-level isolation. Such an approach may be attractive for systems that already has an MMU-based protection and want to extend it with finer-grained isolation. Capability-based protection could be enabled only for selected address spaces, offering more deployment flexibility than pure-capability designs and comparable isolation (**P1**, **P2**). In the case of PX4/NuttX, however, this approach also accumulates the complexity of both mechanisms. The system must support both MMU-based management and CHERI capabilities within the same RTOS environment. This substantially increases implementation complexity, with limited opportunity to reduce the overheads associated with either mechanism. As a result, integration becomes more complex than using MMU-

or CHERI-based protection alone, and we therefore do not consider this design further.

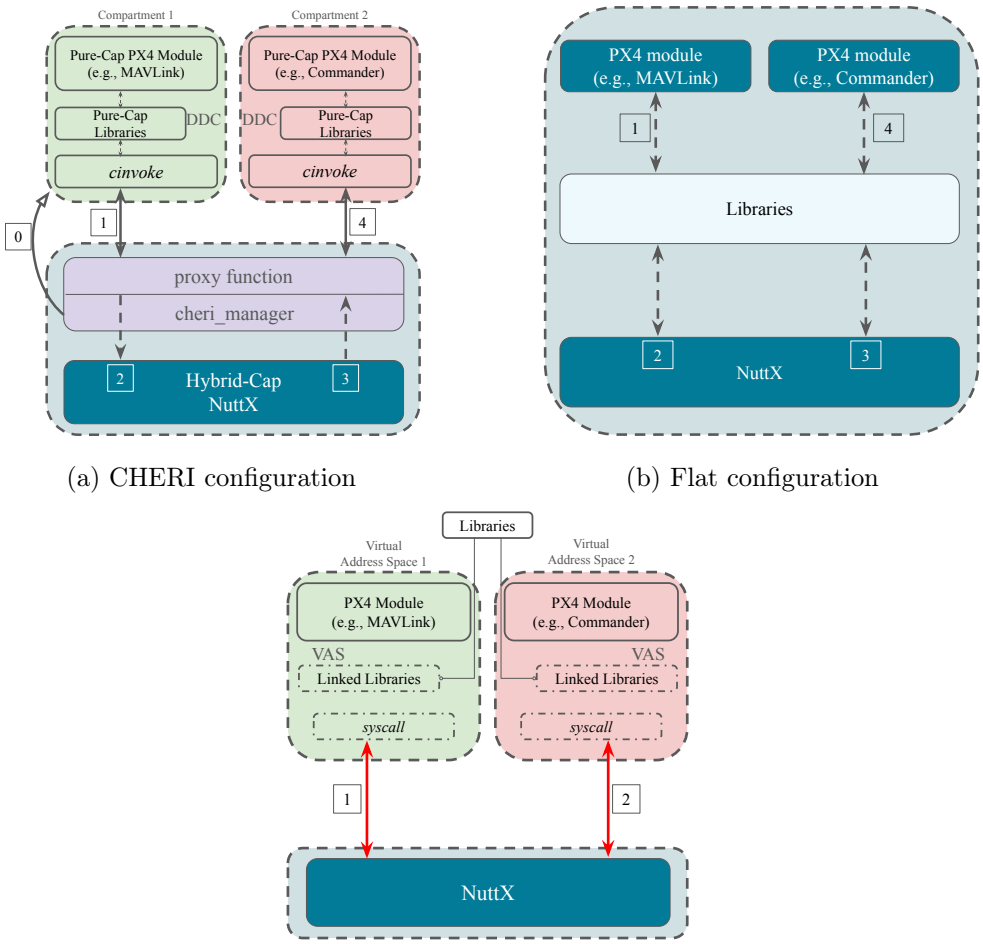
5.6 CHERI-enhanced PX4/NuttX Design

Our design explores how CHERI can enhance the security of UAV architectures and seeks to realize the key properties of a secure UAV software stack through a *mixed* hybrid- and pure-capability strategy. Specifically, we combine a hybrid capability-aware kernel extension with a pure-capability compartment model, balancing therefore incremental-integration requirements (**P3**) and stronger isolation for critical components (**P1**, **P2**). Figure 5.4a illustrates our architecture.

Hybrid-layer. NuttX is extended with hybrid-capability support for its core execution mechanisms, including context switching, interrupt handling, and signal management. NuttX itself runs in an isolated compartment, and the objective of such hybrid-capability support is to preserve capability correctness across kernel-level transitions. In particular, capability metadata must remain consistent when tasks are preempted, interrupted, or resumed. This hybrid model preserves compatibility with existing PX4/NuttX components while introducing only minimal changes to NuttX (**P3**).

Pure-layer. On top of this foundation, selected PX4 modules are executed under a pure capability model. These modules are compiled with capability-only semantics and are used for components that handle external or safety-critical inputs (e.g., MAVLink and Commander in Figure 5.4a). The output of this compilation is a statically linked pure-capability executable that includes both module and its required dependencies (e.g., libc and libm), with all memory accesses expressed through capabilities. By strongly compartmentalizing these high-exposure PX4 modules, we improve overall system security (**P1**, **P2**) and limit the propagation of attacks or unintended misbehavior resulting from unauthorized or erroneous memory operations.

Manager-layer. A trusted runtime component, the *cheri_manager*—deployed within the RTOS compartment, is responsible for loading these executables and instantiating them as bounded-capability compartments. Each compartment is a runtime execution unit that explicitly defines memory and capability boundaries for such executables. At load time, the *cheri_manager* allocates memory regions for the executables, maps their segments, and constructs the set of capabilities that defines each compartment. This includes the default data capability (DDC), stack and heap regions, and the set of authorized entry points (e.g., `main`). All capability bounds are derived from the allocated memory region and enforced both during initialization and execution. Relocation information from each executable is translated into bounded capabilities within this DDC, ensuring that all internal



(a) CHERI configuration

(b) Flat configuration

(c) MMU configuration

Figure 5.4: Interactions among PX4/NuttX modules: (a) CHERI with *cheri_manager*, (b) flat, and (c) MMU. Dashed boxes indicate different isolation boundaries. Dashed arrows indicate function calls within the same address space, solid arrows indicate a cross-compartment calls, solid red arrows are system calls.

references remain confined.

5.6.1 Execution Workflow

Execution follows a controlled entry model. The *cheri_manager* prepares the execution context and transfers control into the compartment through a `cinvoke` invocation with sealed entry capabilities (i.e., sealed DDC and PCC; 0 in Figure 5.4a). This guarantees that execution starts only from validated entry points and within the assigned capability bounds, without requiring capability derivation

at runtime.

Once execution enters a compartment, all interactions between the compartment and the trusted runtime are mediated through a controlled syscall-like mechanism based on capability-protected trampolines. Each compartment includes a lightweight *cinvoke* library that provides a syscall-like abstraction, automatically derived from the standard *syscall* library used in MMU-based NuttX builds. Instead of directly executing a kernel system call, a compartment issues requests in the form `CINVOKE(<syscall number>, args)`, transferring control to the corresponding trusted proxy function exposed by the *cheri_ manager*. The transition from the compartment to the proxy (Figure 5.4a, 1) is implemented through a trampoline using sealed capabilities associated with the proxy entry point. The trampoline saves the current execution context of the compartment and transfers control to the proxy via a `cinvoke` instruction with the sealed capabilities of the proxy (i.e., sealed DDC and PCC) This ensures that execution can later resume transparently.

The proxy function acts as a syscall dispatcher. It reads the syscall identifier and invokes the corresponding NuttX function 2. The system does not rely on a user/kernel separation and these functions are normal calls within the same privilege (compartment) domain. When the function completes 3, control is returned to the compartment through the same trampoline mechanism 1 (but in the proxy to compartment direction), restoring the saved execution context and resuming execution at the point of invocation.

The framework also deals with functions (e.g., `malloc`) that require the proxy to return a valid capability. In these cases, the proxy also performs the allocation within the compartment heap region and constructs a valid capability bounded by the compartment DDC before returning it to the caller. This ensures that dynamically allocated memory remains strictly confined within the compartment capability space and cannot be used to escape its defined boundaries.

5.6.2 Communication Workflow

Given the tight interactions among PX4 modules, minimizing changes to the uORB middleware simplifies the adoption of configurations other than the default flat one (**P3**). Figure 5.4 compares the default flat uORB configuration (b) with our CHERI-based architecture (a) and an MMU-based configuration (c). Across all cases, uORB preserves its shared-memory publish/subscribe model and external APIs, while the underlying implementation shifts from implicit sharing (flat design (b)), to explicit mapping (MMU-based design (c)), and to capability-mediated access (CHERI-based design (a)).

In all configurations, communication is centered on a *DeviceNode*, which is a topic backed by a POSIX shared memory object (`shm_open()`) containing a memory-mapped ring buffer (`mmap()`). When a topic is requested, the *uORB*

Manager either creates or retrieves the corresponding shared memory region and exposes it to the system. Publishers write into this buffer, while subscribers read from it. The configurations differ in: (i) how the shared memory region is mapped into each task address space and (ii) how read/write access to the buffer is enforced or mediated.

Flat Configuration (Figure 5.4b) In the default configuration employed by PX4/NuttX, the uORB infrastructure relies on implicit shared memory, where all tasks directly access the same global *DeviceNode* instances. `shm_open()` and `mmap()` are direct function invocations (1–4) with no transition between privilege domains. In this model, the topic buffer is directly accessible to all components and publish/subscribe operations reduce to direct memory reads and writes synchronized using primitives such as mutexes or semaphores. This yields minimal overhead but provides no spatial isolation between components.

MMU Configuration (Figure 5.4c) Despite the separation of each task in distinct virtual address spaces (VAS), uORB retains its shared-memory design through the use of explicit memory mapping via shared memory objects. During topic initialization or subscription, the *uORB Manager* performs full system calls (i.e., `shm_open()` and `mmap()`) to the NuttX kernel (1 and 2). Each task obtains a virtual mapping of the same physical memory region containing the topic buffer. After this mapping phase, publishers and subscribers access the topic data through direct pointers (within their respective virtual address spaces). The primary source of overhead are system call transitions, page table management, and Translation Lookaside Buffer (TLB) effects during the mapping phase, rather than from message transfer itself.

CHERI Configuration with *cheri_manager* (Figure 5.4a) With our approach, each task executes in an isolated compartment with its own DDC. Nevertheless, the shared-memory design of uORB remains unchanged: each *DeviceNode* is still backed by a shared ring buffer representing a topic. However, access to this buffer is no longer granted by a pointer, instead, tasks must hold valid capabilities that explicitly authorize access to the buffer region. When compartments access a topic 1, 4 and invoke the underlying `shm_open()` and `mmap()` functions, the framework executes the CHERI-based trampoline proxy-functions of the *cheri_manager*. The proxy functions perform the function calls as direct invocation 2. When the *DeviceNode* returns the requested topic information 3, the proxy function returns the execution into the compartmentalized task, following the CHERI-based trampoline mechanism 1, 4. However, the two function calls must be treated differently. While `shm_open()` returns an integer representing a file

descriptor (fd), the cinvoked `mmap()` must return a valid capability. Therefore, after `mmap()` is invoked, the proxy function constructs the corresponding capability by assigning appropriate bounds and permissions. To improve performance, publishers and subscribers operate on this capability reference into shared memory. Upon dereference, the CHERI hardware enforces bounds and permission checks, ensuring spatial safety even within the shared memory region (**P1**, **P2**).

5.6.3 Comparison with CAP-VMs

Although our design draws inspiration from the original CAP-VMs work [263], the two systems differ fundamentally in both their assumptions and their execution models. CAP-VMs target a general-purpose CheriBSD environment, where capabilities are derived from virtual address-space abstractions and compartments are constructed under the control of the Intravisor, a dedicated user-space task. Each cVM executes as a thread within the Intravisor, and all system interactions are mediated through it via a *hostcall* interface. More specifically, the following aspects summarize the principal differences between the two approaches.

Memory model. CAP-VMs rely on virtual memory (initialized by CheriBSD) to derive capability bounds. Instead, *cheri_manager* operates directly on physical memory, constructing and enforcing capability bounds from statically allocated regions at load- and execution-time. This is required by PX4/NuttX, which uses flat physical memory model without virtual address-space isolation.

Execution model. Under CAP-VMs, compartments execute as threads within the Intravisor task. Their lifecycle is therefore owned by the Intravisor rather than directly managed by the OS. In our system, each compartment is a native NuttX task, a first-class entity managed by the RTOS, and preserves the real-time execution semantics required by UAV systems.

Syscall mediation. Both systems use a proxy function to dispatch compartment requests to the appropriate system calls. Within the Intravisor, CAP-VMs uses a dedicated switch-case construct to dispatch hostcall's identifiers to the corresponding CheriBSD syscall. In our design, the majority[2] of the direct dispatching implemented by the *cheri_manager* proxy is automatically derived from PX4/NuttX syscall library, resulting in limited code reworking and good scalability.

Executable loading. While CAP-VMs parse the ELF binary to retrive symbols such as the proxy entry-point return, our design avoids ELF symbol parsing entirely. Instead, we auto-generate a file that places the capabilities of entry functions into a dedicated linker section of the executable. At startup, the *cheri_manager* reads these capabilities directly from that section and derives the runtime capabilities. This simplifies loading and reduces startup time.

[2]Some calls like `printf()` still require dedicated handling.

5.7 Implementation

We implemented our CHERI-enhanced architecture on PX4/ NuttX version 12.11.0. This version also serves as the baseline for the TII PX4/NuttX fork [282], which adds RISC-V MMU support and which we use for comparison in Section 5.8.

On the CHERI side, we rely on the open-source RISC-V prototype CVA6-CHERI [334], which implements the CHERI ISA version 9 [306] and serves as an early reference implementation within the RISC-V CHERI standard [257]. We deploy the core on a Diligent Genesys 2 FPGA [96]. The CVA6-CHERI core [334] extends the baseline CVA6 core [333] with CHERI-based memory protection through pipeline-level support, capability-aware instructions, tagged memory enforcement in load/store paths, and exception handling for capability violations. It supports both pure-capability and hybrid-capability execution modes, as well as operation with and without an MMU.

5.7.1 Enabling PX4/NuttX Compilation with CLANG

Although PX4/NuttX is typically compiled with GCC, support for CHERI modules requires building PX4/NuttX with LLVM/CLANG [286], which supports the CHERI-RISC-V architecture. To this end, we modified the NuttX build system and configuration to ensure that both the RTOS and PX4 modules could compile correctly with CLANG. Only minimal source-code changes were required, primarily to accommodate differences between GCC and CLANG in their assumptions about cast behavior.

To compile PX4 modules in pure-capability mode, we extended the PX4/NuttX CMake build system to support an additional `PURECAP` compilation target. This extension allows modules to be explicitly marked as capability-enabled through the existing `px4_add_module` interface. Under this compilation target, the build system automatically generates the file containing the symbol section that accelerates binary loading by simplifying the resolution of entry points and runtime-exposed functions (Section 5.6.3).

In addition, in line with standard embedded-system practice, the build system compiles all PX4 module dependencies in pure-capability mode and statically links them into the final executable. This includes core NuttX libraries (e.g., libc and libm), the *cinvoke* library, and PX4-specific libraries. As a result, each pure-capability PX4 module is produced as a self-contained statically linked binary with all required capability-aware dependencies embedded.

5.7.2 CHERI-enabled NuttX

We ported NuttX to hybrid-capability mode. In this mode, it is responsibility of the programmer to declare capability instead of pointer by using `__capability` keyword.

We also modified the NuttX task control block (TCB) to load/store capabilities instead of normal registers, and adapted the context-switching mechanisms accordingly to preserve capabilities across task transitions. Moreover, capabilities must be loaded at task creation and restored across context switches occurring during NuttX signal delivery. Note that CHERI requires additional per-task stack space, since each task must store an additional capability (DDC), and each capability uses 128 bits.

Another set of modifications targets NuttX interrupt management. In CHERI, the Machine Trap-Vector Base-Address Register (mtvec) is represented by a capability for the interrupt context, along with its DDC. Specifically, CHERI defines the Machine Trap Code Capability (mtcc) to manage executable access and the Machine Trap Data Capability (mtdc) to manage data access during an interrupt. These capabilities are loaded by the architecture when an interrupt occurs, ensuring that interrupt handlers execute with precisely defined authority and memory bounds. We also include the derivation of CHERI-related fault error, such as "Capability Length Violation" and "Tag Violation."

In our setup, NuttX is considered trusted and its DDC has full privilege on the full memory range.

5.7.3 *cheri_manager*

The *cheri_manager* is responsible for loading PX4 modules and instantiating them as bounded-capability compartments. It is implemented as a PX4 module, which allows it to reuse the existing PX4 infrastructure. Also, as a trusted component, it is deployed directly within the RTOS compartment, similarly to built-in NuttX applications. Its execution is divided into two distinct phases: initialization and interactive.

Initialization phase We use a header file to define the compartment metadata, which specifies the name of the compartment, the path of the executable binary, the amount of memory to allocate, the size of the stack, and scheduling attributes such as policy and priority. Based on this configuration, the *cheri_manager* loads the executable as an ELF binary and extracts its program headers (`PT_LOAD`), symbol metadata (from their dedicated ELF section—see Section 5.7.1), and relocation information (*cap_relocs*). The loader allocates a contiguous memory region for the compartment and copies all loadable segments into it.

From this memory region, the initial set of capabilities is constructed, e.g., the compartment DDC and the stack. Relocation entries are processed and translated into CHERI capabilities, ensuring that all internal references remain constrained within the compartment bounds. The remaining portion of the allocated memory is designated as the compartment heap, used by the *cheri_manager* proxy for dynamic allocation requests originating from the compartment. Since the *cheri_manager* will interact with the compartment when it is executing, we pre-allocate a dedicated stack region for use during the interactive phase. An alternative would be to allocate this memory on demand from the compartment heap using `malloc()`.

After memory and capability setup, the *cheri_manager* instantiates the compartment as a NuttX task using the specified scheduling configuration and stack capability, and starts its execution. Initially, the task still executes in hybrid mode. In this phase, its argument vector (`argv`) is converted into CHERI capabilities and placed on the stack. Afterwards, registers are populated (e.g., `argc` in `a0`, etc.) as well as capability registers holding DDC, stack, entry capabilities, and sealed capabilities for the proxy PCC and proxy DDC. These capabilities are stored into dedicated capability registers (e.g., the sealed proxy PCC in `ct1`). Execution is then transferred to the compartment via a `cinvoke` invocation using sealed entry capabilities (compartment DDC and PCC). This establishes controlled execution within the compartment capability boundaries.

Upon first entry, a minimal runtime stub stores the proxy DDC and PCC sealed capabilities in compartment-local global variables. As a result, later trampoline calls can reuse them directly, without recomputation or hard-coded capability reconstruction.

Interactive phase The *cheri_manager* supports controlled re-entry into an already running compartment. This mechanism is used to carry-out functionalities that should be attributed to the compartment at runtime, e.g., querying module states, or performance stats printing, and that are triggered from the NuttX Shell. Argument data provided by the shell is allocated within the compartment heap and converted into bounded capabilities. Then, *cheri_manager* executes a CHERI trampoline that: (i) saves the current execution context (including register state and stack pointer); (ii) switches its stack pointer with the compartment pre-allocated stack capability; (iii) updates the current DDC; and, (iv) invokes a `cinvoke` instruction to control transfer into the compartment to the target function. Upon completion, execution returns to the *cheri_manager*, which restores the saved context and resumes the original task execution state.

5.7.4 Implementation Complexity

In addition to the changes required to recompile PX4 (Section 5.7.1), and enable CHERI in NuttX (Section 5.7.2), the *cheri_manager* accounts for most of the implementation complexity of our prototype. The *cheri_manager* is implemented in C, C++, and assembly code in 1629 SLOC (as reported by `sloccount` [312]).

For each variant presented in Section 5.6 (flat, MMU, and CHERI), we measured the binary size for the configuration used in Section 5.8. The flat variant has a binary size of 3.2 MiB, MMU variant binary size is of 4.9 MiB, whereas the CHERI variant increases to 9.4 MiB. The larger CHERI binary is mainly due to library replication, as each compartment requires its own pure-capability libraries. Each binary has been compiled with `-O2` and stripped of its debug symbols.

5.8 Evaluation & CVE Analysis

We evaluate the impact of our CHERI-enhanced PX4/NuttX design from two perspectives. First, we assess performance across the three considered configurations (flat, MMU, and CHERI—Section 5.6). Second, we present a systematic analysis of publicly disclosed CVEs affecting PX4/NuttX.

Unfortunately, full-system execution on our FPGA-based evaluation platform, including hardware-in-the-loop simulation, is currently not feasible. The limited operating frequency of the current CVA6-CHERI implementation (50 MHz), together with the computational demands of the PX4 flight stack and its real-time control loops, prevents stable execution of the full flight software even in the lightweight flat configuration.

5.8.1 Performance Evaluation

We evaluate a set of benchmarks that targets low-level NuttX primitives and middleware communication, as well as end-to-end measurements based on the MAVLink PING mechanism. This combination allows us to capture low-level overheads, communication costs (through uORB), and full-stack message latency. In the flat configuration, all benchmarks execute within a single (unprotected) address space. In the MMU configuration, kernel and user space execute separately, and different modules run in distinct address spaces. In the CHERI configuration, benchmarks execute as pure-capability PX4 modules on top of the *cheri_manager*.

Low-level system primitives

We measure the cost of reading high-resolution timer values from the PX4 module layers. Timer information is exposed by NuttX via the `boardctl()` system primitive,

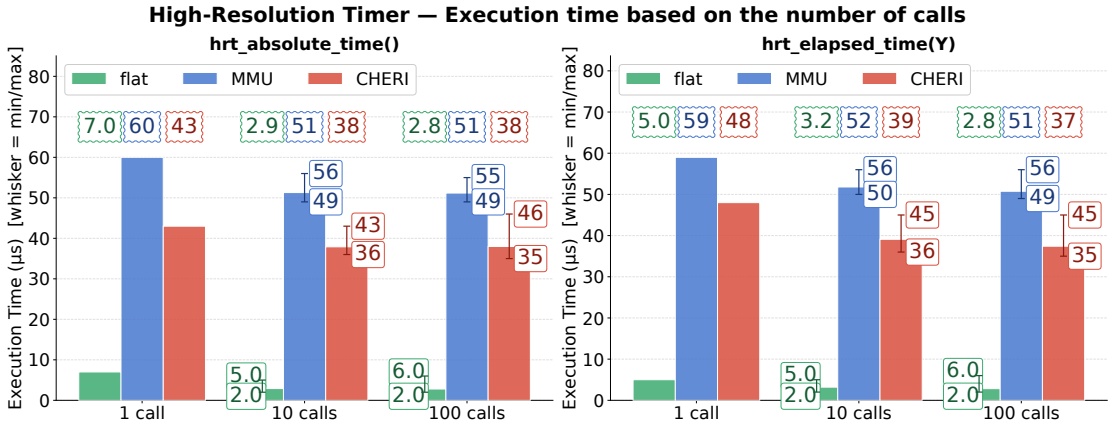


Figure 5.5: Execution time (μs) of the High-Resolution Timer (hrt) microbenchmarks. On top of each box the average value, while whiskers represent minimum and maximum.

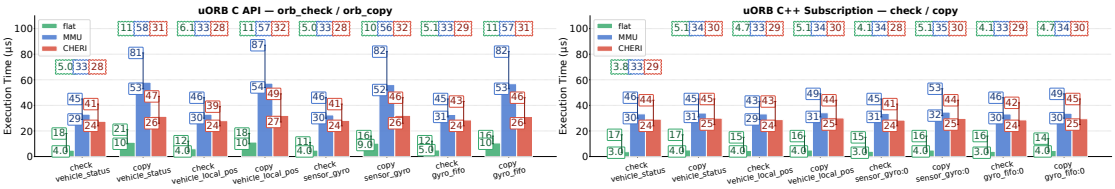


Figure 5.6: Execution time (μs) of the uORB microbenchmarks: C API evaluation on the left side, C++ API on the right. On top of each box the average value, while whiskers represent minimum and maximum.

which provides direct access to board-level functionality, bypassing the traditional `ioctl()` device driver path. The microbenchmark evaluates two timer-related operations: (i) `hrt_absolute_time()`, which retrieves the current hardware timer value via `boardctl()` and (ii) `hrt_elapsed_time(Y)`, which first obtains the current timer value X via `boardctl()` and then computes the elapsed time as $X - Y$. The microbenchmarks are representative of the overheads associated with crossing compartment boundaries in the flat, MMU, and CHERI configurations.

Figure 5.5 shows the average (on top of each boxes), minimum, and maximum of these timer-related operations (in μs). We performed three different measurements for 1, 10, and 100 consecutive calls of each function. As shown, the higher execution observed for a single invocation is consistent with typical cold-start effects in system benchmarks. These one-time overheads are amortized over repeated executions, leading to lower and more stable values for 10 and 100 consecutive events. As expected, the flat configuration achieves the best results due to direct calls. Interestingly, overhead is highest in the MMU configuration, which we attribute to syscall and kernel-transition costs. CHERI avoids part of this cost through a

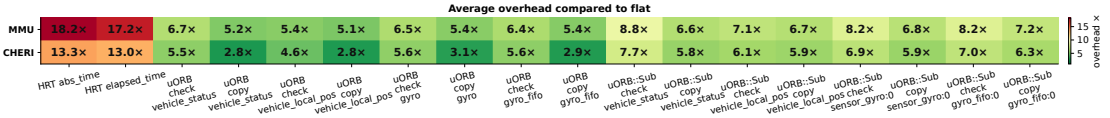


Figure 5.7: Heatmap overview of uORB overheads computed as $avg(MMU)/avg(flat)$ and $avg(CHERI)/avg(flat)$.

lighter trampoline-based mechanism, resulting in consistently lower overhead than MMU.

uORB middleware performance

As benchmarks for middleware (uORB) communication, we measure the execution time of the *check* and *copy* uORB operations that respectively test whether new data is available and transfers the latest topic data to a user buffer. In uORB, two implementations of these operations co-exists. The first one relies on a C API that exposes explicit subscription (e.g., sensors) handles and requires manual interaction with the uORB system, potentially incurring additional lookup- and indirection-overheads. The second one leverages the C++ `uORB::Subscription` wrapper, which encapsulates subscription state and avoids indirections and lookups. This second API is the one typically used by PX4 modules. Figure 5.6 shows the execution times (in μs) for the two operations over 100 repetitions. CHERI consistently incurs lower overhead than the MMU-based configuration, while, as expected, flat mode achieves the best results.

We investigated the consistently higher execution time of the *copy* C operation in MMU mode and attributed it to a `hrt_absolute_time()` system call executed along this path, but absent in the C++ case. As discussed in Section 5.8.1, system calls are more penalized in MMU mode, which is reflected in the consistently higher overhead observed here.

An overall comparison of the overheads of MMU and CHERI configurations relative to flat mode is presented in the heatmap in Figure 5.7, which is based on average values $avg(MMU)/avg(flat)$ and $avg(CHERI)/avg(flat)$.

High-level application

To complement the microbenchmark analysis, we evaluate a more complex system behavior that simulates a Ground Station (see Figure 5.2) communicating with the UAV. In this test, a multi-threaded (sender and receiver) python application runs on a remote host (Ground Station) and periodically sends PING messages to the PX4 MAVLink module on the target platform, which replies with the corresponding response. The host-target connection is realized via UART (115200 baud, 8N1,

no flow control), which is supported by PX4 Autopilot and representative of the bandwidth typically achieved in real-UAV deployments that use serial-based WiFi communication modules.[3] The test measures the round-trip time (RTT) on the host upon reception of the reply. This benchmark captures the latency of a full communication path, including UART transmission, driver processing, MAVLink parsing, and system scheduling on the flight controller. Unlike the uORB microbenchmarks, this evaluation reflects the behavior of a complete application-level communication workload.

Figure 5.8 shows MAVLink PING RTTs (in ms) for increasing sender frequencies. The observed latency is dominated by the UART communication path, as evidenced by the near-identical minimum RTT values across all three configurations and frequencies (∼4.8–5.4 ms). Median latencies and interquartile ranges remain stable, indicating that architectural differences do not significantly affect the end-to-end latency in the current experimental setup. Tail latencies (upper whiskers in the boxplot) show slightly better values for the flat configuration at 10-50 Hz frequencies, indicating the lower overheads in the absence of memory protection mechanisms. In contrast, MMU and CHERI configurations exhibit comparable tail behavior across all tested frequencies. Overall, the absence of a measurable difference between MMU and CHERI in this workload suggests comparable end-to-end performance under UART-bound conditions.

5.8.2 CVE Analysis

To assess the practical security impact of the proposed design, we examine 33 publicly disclosed CVEs affecting PX4 Autopilot and NuttX, collected from CVE tracking databases [89, 131, 325]. These vulnerabilities are categorized in Tab. 5.1 by type and mitigation potential under CHERI compartmentalization and the MMU. It is important to note that CVE reporting for PX4/NuttX is still limited compared to more mature software ecosystems: most entries in the datasets have been disclosed only since 2023, indicating that systematic security analysis of these platforms is still emerging.

The most prevalent vulnerability class is spatial memory corruption (16 CVEs, CWE-120/121/125/787). When a module is compartmentalized, CHERI capability bounds checking triggers a hardware exception on any out-of-bounds access before corruption can occur, thereby mitigating all buffer and stack overflows within affected modules. In the current design, this applies directly to the MAVLink component (CVE-2021-46896, CVE-2024-38951, CVE-2025-15150, CVE-2025-5640, CVE-2026-32743), which has been ported as a CHERI pure-capability compartment. Other vulnerable components, such as drivers (e.g., CVE-2026-32705 through CVE-

[3] An Ethernet driver for NuttX supporting the Ethernet PHY of CVA6 is currently not available.

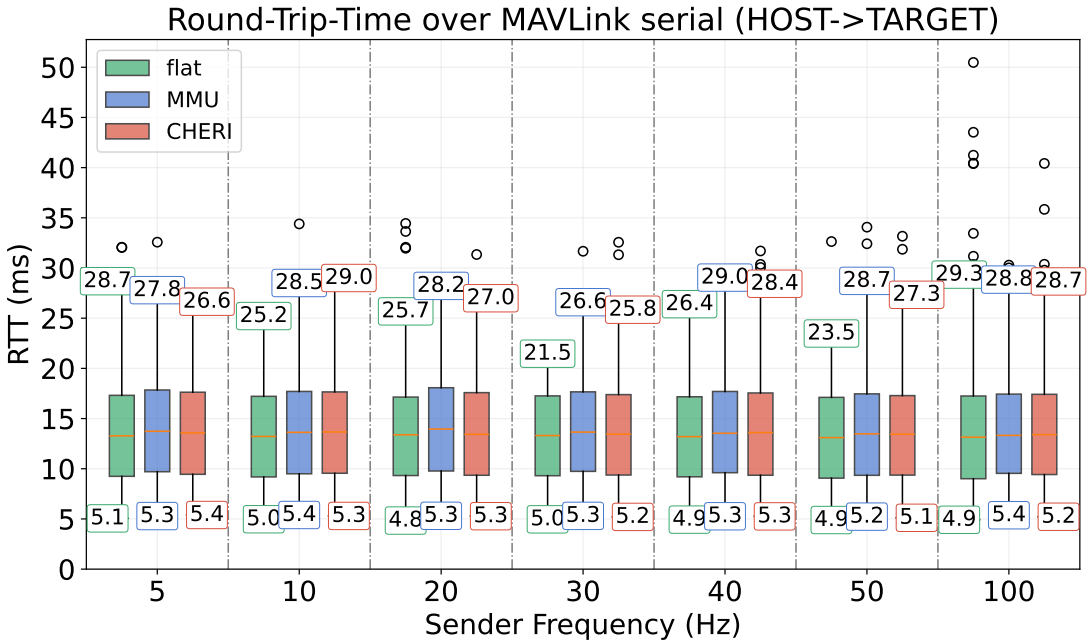


Figure 5.8: Round-Trip Time of MAVLink PING. The x-axis reports the frequencies of the sender, while the y-axis the actual RTT values in ms.

2026-32708), would benefit from the same protection if similarly compartmentalized. The MMU provides only partial mitigation for this class through NX/W^X, which enforces that memory pages are either writable or executable, but not both. This prevents injected shellcode from executing and reduces Remote Code Execution to Denial of Service (DoS), but does not stop the underlying memory corruption.

Four use-after-free CVEs (CVE-2025-9020, CVE-2026-32724, CVE-2026-86096, CVE-2026-86713) cannot be addressed by the MMU: accesses through dangling pointers remain valid after `free()`, since the corresponding page is still mapped. CHERI *revocation* would invalidate such capabilities at the hardware level. However, our current hardware design does not implement such mechanism, and these cases are therefore marked as "protection possible" only (†). CVE-2021-34125 is worth mentioning: the NuttX `mb`/`mh`/`mw` shell commands allow arbitrary physical memory reads, bypassing vendor hardware read protection. As no memory corruption is involved, the MMU offers no protection. In contrast, CHERI can restrict the shell compartment DDC to explicitly authorized regions, causing any access outside these bounds to raise a hardware exception and thereby enforcing access control independently of privilege level. CVE-2026-86097 is the only null pointer dereference: running `param select` without a path argument crashes the autopilot. Neither CHERI nor the MMU mitigates it: CHERI traps the null capability deterministically

Table 5.1: CVE analysis for PX4/NuttX Autopilot. CHERI mitigation marked as ✓, † when possible (requires revocation or proxy handling), and × when not applicable. MMU mitigation is marked as ‡ when NX/W^X reduces impact from code execution to DoS, and × otherwise.

CVE	CVSS v3	CWE	Cat.	Affected Component	C	M
CVE-2020-10281	7.5	CWE-319	P	MAVLink (no encryption)	×	×
CVE-2020-10282	9.8	CWE-319	P	MAVLink (weak encryption)	×	×
CVE-2020-10283	8.1	CWE-288	P	MAVLink (v2→v1 downgrade)	×	×
CVE-2021-34125	7.5	CWE-200	ID	NuttX shell - `mb`, `mh`, and `mw` commands	†	×
CVE-2021-46896	7.5	CWE-120	BO	MAVLink handler (msgid 332)	✓	‡
CVE-2023-46256	9.8	CWE-120/122/787	BO	Lightware sensor parser (heap)	✓	‡
CVE-2023-47625	4.3	CWE-120	BO	CrsfParser (global buffer)	✓	‡
CVE-2024-24254	4.2	CWE-362	RC	geofence.cpp	×	×
CVE-2024-24255	4.2	CWE-362	RC	mission_feasibility_checker.cpp	×	×
CVE-2024-29460	N/A	—	L	mission_block.cpp (home point)	×	×
CVE-2024-30799	4.4	CWE-120	BO	Breach Return Point	✓	‡
CVE-2024-30800	5.6	CWE-229	L	Geofence enforcement	×	×
CVE-2024-38951	6.5	CWE-120	BO	MAVLink (crafted message)	✓	‡
CVE-2024-38952	7.5	CWE-120	BO	Logger (topic_name param)	✓	‡
CVE-2024-40427	7.9	CWE-120	BO	Stack (unspecified module)	✓	‡
CVE-2025-15150	4.8	CWE-119/121/787	BO	MavlinkLogHandler (stack)	✓	‡
CVE-2025-5640	4.8	CWE-121/119	BO	MavlinkReceiver (stack)	✓	‡
CVE-2025-9020	2.0	CWE-416/119	UAF	MavlinkShell (receiver/sender race)	†	×
CVE-2026-26741	8.1	CWE-862	L	Mode switching (Auto→Manual)	×	×
CVE-2026-26742	8.1	CWE-862	L	Re-arm Grace Period logic	×	×
CVE-2026-32705	6.8	CWE-121	BO	BST telemetry driver (stack)	✓	‡
CVE-2026-32706	8.1	CWE-120/787	BO	crsf_rc parser (global buffer)	✓	‡
CVE-2026-32707	6.1	CWE-121	BO	tattu_can (unbounded memcpy)	✓	‡
CVE-2026-32708	8.0	CWE-121	BO	Zenoh uORB (stack VLA)	✓	‡
CVE-2026-32709	6.8	CWE-22	PT	MAVLink FTP	†	×
CVE-2026-32713	6.5	CWE-670	L	MAVLink FTP session validation	×	×
CVE-2026-32724	5.3	CWE-416	UAF	MavlinkShell (heap, telemetry race)	†	×
CVE-2026-32743	6.5	CWE-121	BO	MavlinkLogHandler (sscanf, stack)	✓	‡
CVE-2026-84698	6.5	CWE-787	BO	sd_bench (heap)	✓	‡
CVE-2026-86096	5.9	CWE-416	UAF	TemperatureCalibration::start()	†	×
CVE-2026-86097	6.5	CWE-476	NPD	param_set_{default,backup}_file()	×	×
CVE-2026-86713	7.1	CWE-416	UAF	load_mon	†	×
CVE-2026-86714	5.4	CWE-125	BO	netman	✓	×

Category (Cat.): P: Protocol, ID: Information Disclosure, BO: Buffer Overflow/Over-read, RC: Race Condition, L: Logic, UAF: Use-After-Free, PT: Path Traversal, NPD: Null Pointer Dereference

(an MMU only if page zero is unmapped), but the resulting fail-stop is itself the DoS. The remaining 10 CVEs, including protocol weaknesses (CVE-2020-10281 through CVE-2020-10283), logic and safety flaws (CVE-2026-26741, CVE-2026-26742), and race conditions (CVE-2024-24254, CVE-2024-24255), are outside the scope of hardware-enforced memory protection and must be addressed at the application level.

Overall, CHERI fully or potentially mitigates 22 of the 33 CVEs, whereas the MMU provides 15 partial mitigations. A CHERI violation raises an exception attributable to one compartment. Under the MMU, the same fault is contained but not observable without extra kernel instrumentation.

5.9 Related Work

UAV security has received increasing attention in recent years, and several surveys have systematically characterized the cyber-security threats and vulnerabilities affecting UAV systems across hardware, software, communication, and control layers [27, 82, 172, 259]. ArduPilot [36], another widely used autopilot stack, follows a design similar to PX4/NuttX and typically operates within a shared address space without fine-grained memory isolation, exposing the same class of memory safety risks. Rodday [252] and Andersson [33] demonstrated practical attacks against UAV platforms and showed that weaknesses in communication handling, input validation, and protection mechanisms can enable attackers to inject malicious messages and hijack drone behavior. These works highlight the need for stronger isolation and memory safety mechanisms in embedded flight stacks.

Most CHERI evaluations focus on mixed Capability + MMU architectures, measuring overheads when capability enforcement runs on top of virtual memory [91, 91, 296]. Several studies [278, 307, 308] have evaluated the performance of CHERI and CheriBSD [58] under both real deployments (Arm Morello [135]) and FPGA implementations. These studies do not provide direct MMU-only vs MMU-less CHERI comparisons. Other works have extended CHERI to embedded and real-time environments, among them: (i) CheriRTOS [320] extended FreeRTOS to support CHERI capabilities in pure-capability mode without the MMU; (ii) seL4-CHERI [24] explored CHERI integration into user-space; (iii) FlexCap [170] analyzed benefit of CHERI into unikernel OS (Unikraft [171]); and (iv) CHERIoT [31] co-designed a new CHERI ISA (CHERIoT-Ibex [61]) and RTOS for resource-constrained microcontrollers. However, these works do not specifically target flight-control systems and their RTOSs.

Recent work analyzing 439 CVEs across Linux and FreeBSD kernels found that CHERI can block 35%-61% of kernel vulnerabilities depending on whether temporal safety via capability revocation is enabled [181]. These findings align with our analysis, where CHERI can mitigate 22 of 33 CVEs (67%).

5.10 Summary

This chapter presented the design, implementation, and evaluation of a CHERI-secured PX4/NuttX flight stack on an open-source RISC-V platform. The hardware substrate is the CVA6H-CHERI core, the first publicly available RISC-V processor to combine CHERI and hardware virtualization support, deployed on an FPGA-based prototype of the Al Saqr SoC. The software contribution is a mixed hybrid/pure-capability compartmentalization of PX4/NuttX at the module

level, evaluated against both the default flat memory model and an MMU-based alternative.

The performance evaluation shows that CHERI introduces lower overhead than the MMU-based configuration on low-level system primitives and uORB middleware communication, while providing strictly finer-grained isolation within a shared address space. End-to-end latency measurements show no systematic penalty attributable to CHERI enforcement under representative flight-stack workloads.

The CVE analysis identifies buffer overflows and spatial memory-safety violations as the dominant vulnerability class in PX4/NuttX, and shows that CHERI fully or partially mitigates 22 of the 33 analysed CVEs, compared to 15 partial mitigations under the MMU-based model. Vulnerability classes outside the scope of hardware memory protection—protocol weaknesses, race conditions, and logic errors—require complementary countermeasures at the application level.

Together, these results demonstrate that hardware-enforced capability-based compartmentalization can be integrated into the PX4/NuttX execution model and can provide module-level isolation with lower measured transition overhead than the evaluated MMU-based implementation. The end-to-end measurements further show that, for communication workloads dominated by UART latency, the protection mechanism does not become the dominant source of latency. A complete assessment of UAV schedulability remains outside the capabilities of the current 50 MHz FPGA prototype and requires evaluation on a higher-performance implementation running the full flight-control workload.

Chapter 6

Conclusion

This thesis asked whether a single hardware mechanism could simultaneously address the security and real-time predictability requirements of safety-critical autonomous systems, rather than forcing a trade-off between them and performance. The three contributions reported in the preceding chapters provide a positive answer: CHERI capability hardware can enforce fine-grained memory isolation across the networking stack and the autonomous flight software of real embedded systems, without violating the timing constraints that safety-critical operation demands.

The survey revealed that CHERI has evolved from a research prototype into a deployable architecture with mature hardware, compiler, and OS support, and identified compartmentalization of embedded safety-critical software as the main open challenge for its adoption in the autonomous systems domain. The compartmentalized network stack contribution showed that CHERI enforcement is compatible with bandwidth networking. DPDK and F-Stack execute in capability-enforced compartments on Arm Morello while sustaining 1 Gbps, with overhead localized to synchronization under multi-compartment driver contention. The compartmentalized flight stack showed that the same approach scales to a hard real-time context. Module-level isolation of PX4/NuttX on CVA6-CHERI imposes lower overhead than MMU-based task isolation. It also provides strictly finer-grained protection, and mitigates 22 of 33 CVEs affecting the flight stack, compared to 15 partial mitigations under the MMU-based model.

Together, these results suggest that capability-based hardware can provide a practical basis for security in safety-critical autonomous systems. The findings contradict the prevailing assumption that stronger isolation necessarily conflicts with real-time predictability. Instead, they demonstrate that fine-grained protection can be achieved without sacrificing schedulability. More broadly, this work indicates that capability architectures, such as CHERI, may be a viable alternative to traditional MMU-centric approaches for future embedded and cyber-physical systems.

6.1 Limitations

The evaluation presented in this thesis has several limitations that constrain the scope of its conclusions.

FPGA-only evaluation. The flight-stack evaluation is conducted on an FPGA-based prototype of the CVA6-CHERI core. FPGA implementations run at lower clock frequencies, have different cache hierarchies, and do not reproduce the memory timing of production silicon. The overhead figures reported for the flight stack should therefore be read as indicative of the relative cost of CHERI enforcement, not as absolute performance predictions for a deployed system. The network stack contribution, evaluated on the Arm Morello platform, does not carry this limitation; its results reflect the behavior of a real silicon implementation of CHERI.

Manual compartmentalization policy. The compartmentalization policies for both DPDK/F-Stack and PX4/NuttX were derived and applied manually. Currently, there is no automated mechanism for extracting a least-privilege capability policy from a software architecture description or for verifying the resulting policy's completeness or minimality. A manually constructed policy may be over-permissive in ways that the evaluation does not expose.

Threat model scope. The security analysis covers spatial and temporal memory safety violations, which are the defects that CHERI hardware enforcement directly addresses. However, protocol-level weaknesses, race conditions, logic errors, and microarchitectural side-channel attacks fall outside the threat model and are not addressed by this thesis's contributions.

Workload coverage. Finally, the flight-stack evaluation uses a representative but limited workload. Sensor-rich configurations involving concurrent GPS, IMU, and vision driver activity may reveal overhead patterns not visible in the current benchmark suite.

6.2 Future Work

The most immediate next step is deployment on the taped-out CVA6-CHERI processor. This will allow the FPGA-based results to be revisited under realistic silicon timing and will determine whether the overhead advantage of CHERI over MMU-based isolation holds at production clock frequencies.

Automating compartmentalization policy derivation would address the scalability limitation of the current approach. A tool that takes a module dependency

graph or a software architecture description as input and produces a least-privilege capability configuration would make the methodology applicable to larger stacks without a proportional increase in engineering effort, and would enable policy correctness to be reasoned about formally.

Extending the evaluation to sensor-rich flight workloads is necessary to characterize overhead under conditions closer to operational use, particularly for interrupt-driven paths where compartment crossings occur at high frequency and timing variability directly affects schedulability.

Beyond these immediate extensions, several broader research questions remain open. First, this work demonstrates that effective compartmentalization can be designed for complex systems such as DPDK, F-Stack, and PX4/NuttX, but the process remains largely manual and relies on application-specific knowledge. A key open problem is developing principled methodologies for identifying compartment boundaries that balance security, real-time performance, fault containment, maintainability, and certification requirements. Such methodologies would complement automating capability assignment by addressing a different question: not how capabilities should be assigned once compartment exist, but how the compartments themselves should be identified and evaluated.

Second, the interaction between CHERI compartmentalization and the hardware virtualization support present in CVA6H-CHERI remains an open question. Section 3.5.4 surveyed early efforts to integrate CHERI with hypervisor-level isolation, including the CHERI-TrEE prototype and the ongoing *bhyve* and Bao ports; none of these address how capability-based compartmentalization within a guest composes with hypervisor-enforced partitioning. Whether the two mechanisms compose cleanly, and whether capability-based isolation within a VM provides guarantees complementary to or redundant with hypervisor-level partitioning, has not been systematically studied. Answering these questions will be important for future mixed-criticality systems that combine virtualization with fine-grained capability-based isolation.

Finally, a longer-term challenge is enabling capability-based compartmentalization to be used in certified safety-critical systems. While this work demonstrates that CHERI compartments can provide strong isolation with limited overhead, broader adoption will require design methodologies, verification techniques, more debugging support, and certification evidence that allow capability-based software to be developed and assessed with the same confidence currently expected for conventional safety-critical systems. Formal proof connects those guarantees at the architectural level, but no equivalent proof connects those guarantees to a concrete compartmentalized software stack with real-time constraints. Certification evidence is similarly absent. Standards such as DO-178C and ISO-26262 have no defined path for capability-based isolation, and building one will require mapping

capability invariants onto the failure modes these standards require to be bounded. Closing these gaps is a precondition for capability-based compartmentalization to be adopted in systems that must pass a real safety case, not just a research prototype.

Closing Remarks

Working on CHERI proved to be an unexpectedly broad experience. Engaging with a capability-based architecture required building intuition at a level of abstraction that sits below most systems research: understanding why a bounds violation traps, what the compiler does to capability registers, and how hybrid and pure-capability execution modes interact in practice. The architecture is recent enough that many of these questions did not have settled answers in the literature, which made the work genuinely exploratory. It also meant operating across an unusually wide range of the stack, from hand-written assembly to PX4 module compartmentalization, which is not an experience that most systems projects afford, and one that left me with a deeper appreciation for how security, architecture, operating systems, and applications interact across the computing stack.

Publications and Academic Output

List of Publications

- Ferraro, D., Palazzi, L., Gavioli, F., Guzzinati, M., Bernardi, A., Rouxel, B., Burgio, P. and Solieri, M., **Time-Sensitive Autonomous Architectures**, *Springer Real-Time Systems*, 2023, (**First Author**) [114].

- Ferraro, D., Bastoni, A., Zuepke, A. and A. Marongiu, **Enabling Security on the Edge: A CHERI Compartmentalized Network Stack**, *2025 Design, Automation & Test in Europe Conference (DATE)*, 2025, (**First Author**) [115].

- Sá, B., Ferraro, D., De Matos, E., Psiakis, R., Marongiu, A., Bastoni, A. and Pinto, S., **Advanced Security through Hardware Capabilities: A Comprehensive Survey on CHERI Technology**, *ACM Computing Surveys*, 2026, (**First Author**) [258]

List of Submissions

- Ferraro, D., Sá, B., De Matos, E., Pinto, S., Marongiu, A. and Bastoni, A., **CHERI-Enhanced Security for RISC-V-based UAVs on PX4/NuttX**, *European Conference on Computer Systems (Eurosys)*, 2027, (**First Author**)

- Sá, B., Ferraro, D., De Matos, E., Bastoni, A. and Pinto, S., **CVA6 RISC-V CHERI: Architecture, Microarchitecture, and Evaluation**, *IEEE Transactions on Computers*, 2026

Other Conferences

- Ferraro, D., Sá, B., Martins, J., Pinto, S., De Matos, E., Marongiu, A. and Bastoni, A., **Improving Safety and Security of Embedded Drone Sys-**

tems with CHERI RISC-V, *9th Italian Workshop on Embedded Systems (IWES)*, 2024

- Ferraro, D., Sá, B., Martins, J., Pinto, S., De Matos, E., Marongiu, A. and Bastoni, A., **Security, Safety, and Predictability of CHERI RISC-V for Drone Systems**, *RISC-V Summit Europe 2024*, 2024
- Ferraro, D., Sá, B., Pinto, S., De Matos, E., Marongiu, A. and Bastoni, A., **Capability-Based Isolation for UAV Software Stacks: Porting PX4/ NuttX to CHERI**, *CHERITECH'25*, 2025

Appendix A

Historical Development of Capability Systems

The concept of hardware-enforced capability-based protection predates CHERI by several decades. In the mid-1960s, systems such as the Burroughs B5500 [53] and the Rice Computer [116] established foundational concepts for the formalization of capabilities. These architectures employed partitioned memory into district segments (*segmentation*). Each segment contained either data or code and was accessed via descriptors that encoded the segment's location together with the set of authorized operations. In 1966, Dennis and Van Horn [94] introduced a set of abstract instructions to provide memory protection for multiprogrammed systems, i.e., systems that execute multiple programs concurrently on a single or multiple CPUs. In their model, a program executes within a sphere of protection defined as a collection of capabilities. Each capability is an unforgeable authority token that confers access to a system object with specific permissions, such as read or write. The work of Dennis and Van Horn catalyzed the development of numerous capability-based architectures through the 1980s, as surveyed comprehensively by Levy [180]. For example, Lampson [176] and Fabry [112] formalized a hardware capability-based addressing scheme suitable for object-based programming, which in turn led to the emergence of a variety of other capability systems in subsequent years [142].

Despite their advantages, capability-based systems largely fell out of favor during the 1990s. Several factors contributed to this decline: (i) the complexity of capability management, particularly the operation of invalidating (*revoking*) capabilities to prevent their further use by programs, posed significant implementation challenges; (ii) the widespread adoption of RISC architectures [224, 237] and evolving market demands shifted priorities toward performance and efficiency rather than security; (iii) constraints on available silicon area limited the feasibility of dedicating hardware resources to capability-based designs; and (iv) pervasive

misconceptions about the nature and advantages of capabilities relative to alternative security mechanisms [213], e.g., access control lists (ACLs), which associate each system resource, for example a file, with an explicit list of permissions.

In the contemporary landscape, characterized by pervasive connectivity and large-scale distributed infrastructures, there is renewed interest in capability-based protection as a foundation for constructing trustworthy systems capable of addressing modern security threats. In the following sections, we survey historical implementations of capability systems (purely hardware, purely software, and hardware-software co-designs) [140] that have informed the design of modern capability architectures, such as CHERI.

A.1 Hardware Capability Systems

Figure A.1 provides a timeline of the historical capability systems from their inception till today. Table A.1 gathers the fundamental features of the most notable hardware capability systems.

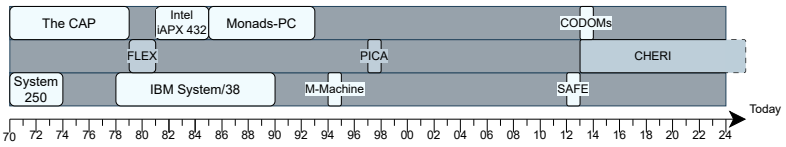


Figure A.1: Timeline of significant hardware capability systems, ranging from the 1970 to today.

Capability Storage There are two principal architectural strategies for preserving the integrity of capabilities: (i) capability lists (*C-lists*) and (ii) tagged memory. A *C-list* is a form of capability storage that physically separates capabilities from ordinary data, e.g., by allocating distinct memory segments for data and for capabilities [112]. The Plessey System 250 [105, 106] and the CAP Computer [219, 313] are canonical instances of *C-list*-based architectures.

In contrast, tagged memory associates one or more metadata bits with each data word, thereby permitting the co-location of data and capabilities within a single segment. While C-lists can complicate object management due to the fragmentation of a single logical object across multiple segments, tagged memory simplifies software design and implementation by enabling the unified storage of object data and its corresponding capabilities within the same segment. Tagged capabilities were first investigated in systems such as Flex [119, 120] and IBM System/38 [44, 149, 287], and subsequently became widespread in capability-based architectures including the M-Machine [118], PICA [291, 292], and SAFE [95].

Table A.1: Hardware Capability-based systems. [1]Capability Storage - Tag-memory (TM), C-Lists (CL) or Access Protection Lists (APL), [2]Capability and Object Representation - Split (S) or Unified (U), [3]Capability Revocation - yes, no or limited, and [4]Capability-Protected Compartmentalization

System	Description	CS[1]	COR[2]	CR[3]	CPC[4]
Plessey System 250 [105, 106] (1972)	Commercial computer for real-time and fault-tolerant communication telephone-switch controller.	CL	S	no	Each protected procedure has a central table with all accessible data, code, and capability (*Central Capability Block (CCB)*) segments. Procedure is accessible through a enter capability to the CCB.
The CAP Computer [219] (1977)	32-bit research capability system developed at the University of Cambridge from '70 to '77. It uses a dedicated 64-entry capability unit that automatically searches and load capabilities to access memory.	CL	S	limited	Requires an entry capability to a process local table entry that changes the addressable data, code and capability segments.
Flex [119, 120] (1979)	System built by UK Ministry of Defence's Royal Signals and Radar Establishment (RSRE) on the principles of capability-based security for highly secure and flexible computer systems.	TM	U	yes	N/A
IBM System/38 [44] (1980)	Mid-range computer designed for business computing applications. The System/38 implements capabilities as a combination of hardware and microcode.	TM	S	yes	One profile for each process with all available addressable objects and access permissions. Specifies mechanism to prevent profile sharing between the caller and callee.
Intel iAPX 432 [83, 84] (1981)	Commercial microprocessor designed to support advanced operating systems with capabilities for fine-grained access control and object-oriented programming. The iAPX 432 uses a combination of hardware and microcode to implement and abstract many operating system operations, like interprocess communication.	CL	S	yes	Supports domain objects as collection of procedures that can access private objects. Consists of a list of capabilities to code and objects. A dedicated instruction is used to create the domain object that returns an entry capability to the protected domain.
Monads-PC [13, 254] (1985)	32-bit research capability-based machine which was built by David Abramson and John Rosenberg at the University of Monash. Monads-PC has the ability to support segments and paging.	TM	U	no	Uses a stack address space to store linkage information (bases), procedure-local data, and temporary data for expression evaluation. During a procedure call, the call mechanism adjusts the bases to establish the called procedure's execution environment.

Continued on the next page

System	Description	CS[1]	COR[2]	CR[3]	CPC[4]
M-Machine [118] (1995)	Proof-of-concept system that explores capability-based in a heterogeneous architecture to improve performance/area ratio. It implements a hybrid system for memory protection: (i) capability-based addressing system using Guarded Pointers for fine-grained memory protection; (ii) paging to support virtual memory and seemly relocate physical memory.	TM	U	limited	Uses entry capabilities that point to the domain code. Requires manual clearing of registers to prevent data and capability leakage.
PICA [291, 292] (1996)	32-bit, superscalar, 7-stage pipeline 32-bit MIPS processor. The PICA architecture adds capability-based control access to regular load-store RISC architecture with controlled hardware complexity and no impact on the processor's critical path.	TM	S	yes	Context for each protect domain are store in a dedicated special stack that prevent capability and data leakage from one domain to another via stack or registers. Call/return sequence is ensured by an instruction tracker in hardware.
SAFE [93, 95] (2011)	Security-oriented processor that focuses on providing fine-grained privilege separation and object-based enforcement semantics at the architectural level.	TM	S	yes	Represents protected domains as a collection of three capabilities, i.e., code, data, and authority. To switch between domains, it uses atomic operations as simple as function calls, a.k.a. gates.
CODOMs [294] (2014)	CODOMs is a *code-centric* capability system that isolates multiple software components in a shared address space while minimizing the impact on existing software protection models.	APL	U	yes	Extends the page table with capability metadata and associates a tag to each memory page (*PT-Caps*). An isolated domain is built by marking multiple pages (code and data) with the same tag. Each domain is isolated from another and communicates or transfers data via capabilities stored in capabilities registers (*CapRegs*) or spills capabilities in dedicated stack capabilities memory.
CHERI [316] (2014)	CHERI is an ISA-agnostic capability-base protection model that provides fine-grained memory protection (pointer-safety) and isolation primitives for application compartmentalization.	TM	U	yes	Defined protected domains via a sealing mechanism, entry points, and atomic domain transitions (see Section 3.1 for more details).

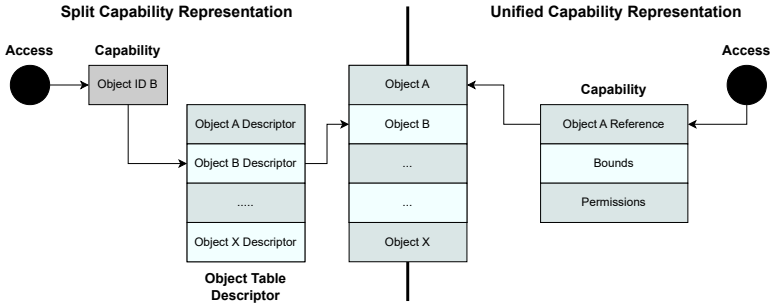


Figure A.2: Comparison of unified and split capability representations schemes.

The Intel iAPX 432 [83, 84] attempted to combine the advantages of both paradigms by allowing data and capabilities to share a single segment that is logically partitioned into two regions. A per-object descriptor specified the sizes of the data and capability regions and thus determined the position of the boundary between them.

Finally, CODOMs [294] adopted an alternative mechanism that avoids both tagged memory and strict physical segregation. In this design, capabilities may only be transferred into pages that explicitly permit load or store operations. A hardware-maintained list, controlled by the trusted computing base (TCB), enforces this policy, thereby regulating where capabilities can be fetched from or stored in memory.

Capability and Object Representation In Table A.1, we categorize capability systems as *split* or *unified* according to their respective capability and object representations. Figure A.2 provides a schematic overview of these two design paradigms. Split architectures (left side of Figure A.2) were prevalent in early capability systems. In such architectures, object descriptors are maintained in centralized mapping tables, while capabilities contain unique identifiers that are used to reference these objects. Representative systems include the Plessey System 250 [105, 106], The CAP [219, 313], PICA [291, 292], and the Intel iAPX 432 [83, 84].

As an illustrative example, the CAP computer employs a multi-level indirection mechanism for capability resolution. Each process maintains a local object table that stores capabilities referencing indices in a global (central) object table. The index held in a process's local storage effectively serves as a pointer to a capability in the parent process's object table, with this chain of indirections continuing recursively until it reaches the root process.

To mitigate the performance overhead introduced by the implicit level of indirection in split capability designs, subsequent research trends focused on unifying capability and object representations by embedding object metadata directly

within the capability itself, thereby obviating the need for a centralized object table (right side of Figure A.2). Representative architectures such as the M-Machine [68], Monads-PC [13, 254], and SAFE [174] adopt a unified capability format that is amenable to storage in processor registers (e.g., the Guarded Pointer abstraction [68]). This representation typically encapsulates access permissions, an object identifier/reference, and spatial bounds describing the region of memory associated with the object. To further reduce storage and propagation costs, these systems frequently employ compression or encoding techniques that enable an efficient representation of capabilities both within the processor pipeline and in memory.

Capability Revocation One approach to revoking capabilities involves using indirection by defining a *weak capability* that points to a *revoker capability* [249]. This *revoker capability* can later be used to invalidate all capabilities that reference it. Examples of this mechanism include the CAP computer *relative capabilities* [219, 313] and IBM System/38 *unauthorized capabilities* [44, 149, 287]. These systems simplify capability revocation, but exert significant pressure on hardware and caches during capability addressing. Instead, tagged architectures such as Plessey System 250 [105, 106] and M-Machine [68], lack efficient mechanisms for capability revocation and require a complete memory sweep through garbage collection to eliminate dangling capabilities (identified by the tag bit) to inactive objects.

Several systems realize capability revocation through coordinated mechanisms spanning both software and hardware. The Intel iAPX 432 [83, 84] supports capability revocation on the stack by disallowing the storage of capabilities in segments whose lifetimes exceed that of the stack frame, thereby ensuring that capabilities cannot outlive their intended scope. In the PICA architecture [291, 292], each object is assigned a unique identifier, which enables capability invalidation by deallocating the corresponding object and subsequently purging both the memory-to-capability mappings and the entries in the TLBs. SAFE [95] employs a dynamic tagging mechanism in which memory locations are tagged, and runtime rules are enforced to regulate and invalidate capabilities as needed. In CODOMs [294], each capability is linked to a revocation counter stored in memory and embeds the counter value observed at the creation time. Incrementing the revocation counter invalidates all capabilities associated with it, providing efficient counter-based revocation.

Capability-protected Compartmentalization All the architectures in Table A.1 can compartmentalize programs into several fine-grained domains protected by capabilities, commonly including sealing and entry mechanisms. For instance, the M-Machine defines new architectural capability types: (i) sealed pointers,

which are unforgeable pointers for building high-level software-defined policies and (ii) entry capabilities, which are opaque, unmodified pointers that can only be used to gate access to protected domains. PICA employs techniques to ensure no data is shared across boundaries, except for implicitly passed registers as arguments. SAFE defines protected domains as three capabilities: code, data, and authority and uses atomic operations, as simple as function calls, to switch between domains. CODOMS follows a hybrid design using a mix of capabilities and access protection lists (APLs). APLs track which domains can access or invoke operations on other domains memory pages, and CODOMS checks these lists at runtime along with page-level permission to enforce cross-domain isolation efficiently.

Discussion The Plessey System 250 [105, 106] is an example of a commercial computing system that effectively integrated capability-based protection mechanisms, extending its model to include I/O devices [180]. In contrast, the System/38 and Intel iAPX 432 [83, 84] faced commercial challenges: high costs and inadequate compiler optimization favored alternatives such as the 8086 and VAX-II/780 [141, 202]. The IBM System/38 [44, 149, 287] complexity in capability revocation led to its replacement by the IBM AS/400, which eliminated capability support [86]. The Flex machine [120] offered robust security features, including compartmentalization and tagged memory, but failed commercially due to high costs and strong competition. The M-Machine [118], while a conceptual pioneer, failed to gain traction due to internal memory fragmentation and incompatibility with standard C idioms [224]. Despite this, its framework influenced modern capability systems like CHERI [316]. The SAFE [174] low-fat scheme for representing capabilities also has notable limitations, including the requirement for all pointers to remain in bounds, complicating pointer arithmetic and being incompatible with common C idioms [224, 317]. Finally, the PICA [291, 292] architecture's lack of adoption stems from insufficient virtual memory support and the absence of physical implementation, inhibiting its validation [224].

A.2 Software Capabilities

Table A.2: Software Capability-based OSs

OS name	Author & Ref	Description	Types	Revoc.	Active
Hydra (1979)	Wulf et al. [318]	One of the first OSs focusing on capability-based security and resource management. It introduced capabilities to control access, influencing later secure OS designs like EROS and seL4.	C-list	Indirect	□
EROS (1999)	Shapiro et al. [271]	A capability-based microkernel OS featuring a single-level storage model and transparent persistence. It manages resources through capability invocations, optimizing memory, address translation, and process handling with virtual copy spaces and space banks, enabling efficient and secure access control.	Hierarchical Persistent object store	Direct	□
seL4 (2006)	Gernot [130]	A formally verified, capability-based microkernel in the L4 family, ensuring system security and compliance across specific architectures and kernel versions. It features three capability types—i.e., kernel objects, abstract resources, and memory allocation—managed in per-thread *CSpaces*. With support for hardware virtualization and paravirtualization, seL4 can also function as type-1 hypervisor.	Hierarchical Explicit capability trees	Direct	☑
Composite (2008)	Parmer and West [235]	An OS featuring hierarchical user-space schedulers, enabling flexible system partitioning through user-defined components. Unlike standard capability systems, it merges *Page Tables* and *Capability Tables* into a unified *Resource Table*, allowing dynamic capability management and operation execution.	Kernel managed Modular	Indirect	☑
Fiasco.OC (2009)	L4Re.org [175]	Fiasco.OC, the microkernel for the L4Re project, is designed as a hypervisor meeting strict safety certifications like ASIL-B. It employs software-defined capabilities for controlled kernel object access and supports source code verification to ensure reliability. With `uvmm`, Fiasco integrates virtual machines, leveraging hardware virtualization and configurable isolation mechanisms for secure operation.	Hierarchical Explicit capability trees	Direct	☑
Google Fuchsia (2021)	Google Inc. [133]	Fuchsia uses the Zircon kernel—a secure, capability-based kernel for embedded systems. Zircon encapsulates system resources as objects (e.g., Channels, Processes, Threads) accessed via *Handles*, enforcing security through sandboxing and the principle of least privilege.	Handles	Indirect	☑

Table A.2 summarizes the most notable capability-based OSs and microkernel architectures. These systems span several decades of research and development, from foundational work in the 1970s with Hydra [318] to modern implementations like seL4 [130] and Google Fuchsia [231]. The table introduces each system with a brief description of their contributions to security and performance in OS design. It also highlights if the project is still active. These systems may present differences on the organization of the capabilities and revocation mechanisms. Moreover, it is also important to consider the trade-offs between static and dynamic configurations

that these systems offer.

Types of capabilities Capabilities may be organized into explicit lists, hierarchical structures, or lightweight references. For instance, capabilities can be managed in a hierarchical derivation tree, as seen in seL4 [130] and Fiasco.OC [175]. In these systems, new capabilities are derived from existing ones with equal or reduced rights, enabling precise delegation and controlled propagation of access rights. Other systems, such as Hydra [318], present capabilities as C-lists, i.e., per-process tables indexing capabilities to system resources, representing an explicit and structured representation of a process authority. Composite [235] employs kernel-managed capabilities designed for modularity and inter-component delegation, i.e., capabilities are transferred and checked during component interactions. Alternatively, Google Fuchsia [133] uses *handles* as pointers to reference kernel-managed resources, and they rely on kernel policies to enforce access control. Persistent object stores, as leveraged by EROS [271], integrate capabilities with long-term storage, ensuring that access rights persist across reboots.

Revocation Revocation ensures that access rights can be rescinded reliably and efficiently. A key distinction is whether revocation is *direct*, i.e., explicitly controlled via system calls, or *indirect*, i.e., automatically handled. Revocation strategies can vary along systems. Direct revocation in seL4 and Fiasco.OC employs a *capability derivation tree* (CDT), which records parent-child relationships between capabilities. Revoking a capability from the tree invalidates all derived capabilities. EROS, as a persistent system, revokes capabilities by modifying or deleting the referenced persistent capability object. In contrast, more dynamic systems like Composite and Google Fuchsia prefer indirect revocation approaches such as *reference counting*—where a capability is revoked when all references to the associated object are closed—, or *revocation lists* for Hydra.

Discussion: Static VS Dynamic capability configuration The balance between static and dynamic configurations marks trade-offs in predictability, security, and adaptability. Static configurations define capabilities at system initialization, simplifying verification processes and eliminating runtime unpredictability. However, they lack flexibility to handle dynamic workloads. Conversely, dynamic configurations are flexible and easily adapt to different scenarios but, at the same time, increase the complexity of formal guarantees. Most of the systems presented use a hybrid approach, supporting both static configurations for predictability and dynamic configurations for runtime responsiveness or a combination of them. For instance, seL4 uses CapDL [267] to describe kernel objects and the distribution of capabilities among the objects; at runtime, capabilities may still be delegated and

manipulated through kernel primitives, allowing the system to remain responsive to dynamic workloads while preserving strong confinement guarantees.

Bibliography

[1] Home – DPDK. https://www.dpdk.org/, .

[2] SPDK-Morello/DPDK: Data plane development kit (morello branch). https://github.com/spdk-morello/dpdk/tree/morello, .

[3] F-Stack: High performance network framework based on DPDK. https://www.f-stack.org/.

[4] iPerf – the TCP, UDP and SCTP network bandwidth measurement tool. https://iperf.fr/.

[5] LwIP – a lightweight TCP/IP stack. https://savannah.nongnu.org/projects/lwip/.

[6] musl libc. https://musl.libc.org/.

[7] _umtx_op(2) – FreeBSD manual. https://man.freebsd.org/cgi/man.cgi?query=_umtx_op&sektion=2&n=1.

[8] Cheri alliance - industry-led security technology, 2024. Retrieved from: https://cheri-alliance.org/.

[9] libreSSL, 2024. Retrieved from: https://www.libressl.org/.

[10] libssh2, 2024. Retrieved from: https://libssh2.org/.

[11] S. A., T. Chen, D. Chisnall, et al. *CHERIoT Architecture specification*. Microsoft, 0.6 (draft) edition, 2025. URL https://cheriot.org/cheriot-sail/cheriot-architecture.pdf. Accessed on: 01-06-2025.

[12] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), Nov. 2009. ISSN 1094-9224. doi: 10.1145/1609956.1609960. URL https://doi.org/10.1145/1609956.1609960.

[13] D. A. Abramson and J. Rosenberg. The microarchitecture of a capability-based computer. *SIGMICRO Newsl.*, 17(4), dec 1986. ISSN 1050-916X. doi: 10.1145/19530.19546. URL https://doi.org/10.1145/19530.19546.

[14] AbsInt. Compcert, (n.d.). https://github.com/AbsInt/CompCert.

[15] AbsInt. Secomp, (n.d.). https://github.com/secure-compilation/CompCert.

[16] P. M. Adamczyk and M. Miłosz. Security vulnerabilities in c++ programs. *Journal of Computer Sciences Institute*, 35:216–223, Jun. 2025. doi: 10.35784/jcsi.7389. URL https://ph.pollub.pl/index.php/jcsi/article/view/7389.

[17] T. Aird, H. Almatary, J. Woodruff, et al. CRISC-V Specification for CHERI Extensions. Technical report, RISC-V International, Feb. 2025. URL https://github.com/riscv/riscv-cheri/releases/download/v0.9.5/riscv-cheri.pdf.

[18] A. Al-Batati, A. Koubaa, K. Gabr, M. Abdelkader, and H. Aloqaily. Ros 2 in a nutshell: A survey. *ACM Comput. Surv.*, May 2026. ISSN 0360-0300. doi: 10.1145/3815113. URL https://doi.org/10.1145/3815113.

[19] A. Al-Boghdady, K. Wassif, and M. El-Ramly. The presence, trends, and causes of security vulnerabilities in operating systems of IoT's low-end devices. *Sensors*, 21(7):2329, 2021. doi: 10.3390/s21072329.

[20] B. Alan and D. Rob. *Mixed Criticality Systems - (A Review: 14th and Final Edition, February 2026)*. 2026.

[21] A. Aldahmani, B. Ouni, T. Lestable, and M. Debbah. Cyber-security of embedded IoTs in smart homes: Challenges, requirements, countermeasures, and trends. *IEEE Open Journal of Vehicular Technology*, 4:281–292, 2023. doi: 10.1109/OJVT.2023.3234069.

[22] H. Almatary. *CHERI Compartmentalisation for Embedded Systems*. PhD thesis, Apollo - University of Cambridge Repository, 2022. URL https://www.repository.cam.ac.uk/handle/1810/341248.

[23] H. Almatary. GitHub - CTSRD-CHERI/seL4: The seL4 microkernel, 2024. Github repository: https://github.com/CTSRD-CHERI/seL4/.

[24] H. Almatary. Cheri-sel4: Enhancing sel4's c/c++ userspace memory safety using cheri. seL4 Summit 2023 presentation: https://sel4.systems//Foundation/Summit/2024/slides/cheri-morello.pdf, 2024.

[25] H. Almatary, A. Mazzinghi, et al. Case Study: Securing MMU-less Linux Using CHERI, 2024.

[26] H. Almatary et al. CompartOS: CHERI compartmentalization for embedded systems. https://arxiv.org/abs/2206.02852, 2022.

[27] A. A. Alquwayzani and A. A. Albuali. A systematic literature review of zero trust architecture for military uav security systems. *IEEE Access*, 12: 176033–176056, 2024. doi: 10.1109/ACCESS.2024.3503587.

[28] S. Amar, D. Chisnall, T. Chen, et al. Capableptrs, (n.d.). https://gitlab.mpi-sws.org/FCS/capabilities-compilation-public.

[29] S. Amar, T. Chen, D. Chisnall, et al. Cheriot: Rethinking security for low-cost embedded systems. Technical Report MSR-TR-2023-6, Microsoft, February 2023. URL https://www.microsoft.com/en-us/research/publication/cheriot-rethinking-security-for-low-cost-embedded-systems/.

[30] S. Amar, D. Chisnall, et al. Cheriot: Complete memory safety for embedded devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23. Association for Computing Machinery, 2023. ISBN 9798400703294. doi: 10.1145/3613424.3614266. URL https://doi.org/10.1145/3613424.3614266.

[31] S. Amar, T. Chen, D. Chisnall, et al. Cheriot rtos: An os for fine-grained memory-safe compartments on low-cost embedded devices. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 67–84, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400718700. doi: 10.1145/3731569.3764844. URL https://doi.org/10.1145/3731569.3764844.

[32] S. Amar et al. CHERIoT: Complete memory safety for embedded devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*, pages 641–653, 2023. doi: 10.1145/3613424.3614266.

[33] J. Andersson. PacSec - Attacking IoT with SDR. https://arstechnica.com/information-technology/2016/10/drone-hijacker-gives-hackers-complete-control-of-aircraft-in-midflight/ Accessed on: 30-03-2026, 2016.

[34] A. L. D. Antón, J. Müller, et al. Vericheri: Exhaustive formal security verification of cheri at the rtl. *arXiv preprint arXiv:2407.18679*, 2024.

[35] Apache Software Foundation. Apache NuttX. https://nuttx.apache.org/, Accessed on: 30-03-2026.

[36] ArduPilot. Ardupilot - versatile, trusted, open. https://ardupilot.org/, Accessed on: 31-03-2026.

[37] ARM. Arm TrustZone technology for the Armv8-M architecture. https://developer.arm.com/documentation/100690/, 2018.

[38] Arm. Morello platform model reference guide revision: Version 2.0, 2021. Retrieved from:https://developer.arm.com/documentation/102225/.

[39] Arm Ltd. Arm® Architecture Reference Manual Supplement Morello for A-profile Architecture, 2022.

[40] Arm Ltd. Morello Development Tools, 2022. https://developer.arm.com/documentation/109399. Accessed on: 08-01-2025.

[41] A. Armstrong, T. Bauereiss, B. Campbell, et al. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi: 10.1145/3290384. URL https://doi.org/10.1145/3290384.

[42] E. Baccelli et al. RIOT: An open source operating system for low-end embedded devices in the IoT. *IEEE Internet of Things Journal*, 5(6):4428–4440, Dec. 2018. doi: 10.1109/JIOT.2018.2815038.

[43] R. Bahmani, F. Brasser, et al. CURE: A security architecture with CUstomizable and resilient enclaves. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/bahmani.

[44] V. Berstis. Security and protection of data in the ibm system/38. In *Proceedings of the 7th Annual Symposium on Computer Architecture*, ISCA '80. Association for Computing Machinery, 1980. ISBN 9781450373906. doi: 10.1145/800053.801932. URL https://doi.org/10.1145/800053.801932.

[45] A. Bhattacharyya, S. Ramanathan, A. Fumagalli, and K. Kondepu. An end-to-end DPDK-integrated open-source 5g standalone radio access network: A proof of concept. *Computer Networks*, 2024. doi: 10.1016/j.comnet.2024.110533.

[46] A. Birgisson, J. G. Politz, et al. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Network and Distributed System Security Symposium*, 2014.

[47] V. Bode, C. Trinitis, M. Schulz, D. Buettner, and T. Preclik. Adopting user-space networking for DDS message-oriented middleware. In *2024 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 36–46, Biarritz, France, 2024. doi: 10.1109/PerCom59722.2024.10494460.

[48] M. Bognar. Proteus: a configurable risc-v core, (n.d.). `https://github.com/cap-tee/proteus`.

[49] J. Bramley, D. Jacob, et al. Picking a cheri allocator: Security and performance considerations. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2023. Association for Computing Machinery, 2023. ISBN 9798400701795. doi: 10.1145/3591195.3595278. URL `https://doi.org/10.1145/3591195.3595278`.

[50] F. Braußе, F. Shmarov, R. Menezes, M. R. Gadelha, K. Korovin, G. Reger, and L. C. Cordeiro. ESBMC-CHERI: Towards Verification of C Programs for CHERI Platforms with ESBMC. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022. Association for Computing Machinery, 2022. ISBN 9781450393799. doi: 10.1145/3533767.3543289. URL `https://doi.org/10.1145/3533767.3543289`.

[51] A. Burns and R. I. Davis. A survey of research into mixed criticality systems. *ACM Comput. Surv.*, 50(6), Nov. 2017. ISSN 0360-0300. doi: 10.1145/3131347. URL `https://doi.org/10.1145/3131347`.

[52] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1), Apr. 2017. ISSN 0360-0300. doi: 10.1145/3054924. URL `https://doi.org/10.1145/3054924`.

[53] *The descriptor–a definition of the B5000 information processing system.* Burroughts Corporation, Detroit, Mich., 1961.

[54] P. Butcher, D. King, et al. Security by Default - CHERI ISA Extensions Coupled with a Security-Enhanced Ada Runtime. Technical report, AdaCore, 2024.

[55] Cambridge University. Beri, (n.d.). `https://github.com/CTSRD-CHERI/beri`.

[56] Cambridge University. Cerberus, (n.d.). `https://github.com/rems-project/cerberus/tree/master`.

[57] Cambridge University. Cheri-spike, (n.d.). https://github.com/CTSRD-CHERI/riscv-isa-sim.

[58] Cambridge University. Cheribsd, (n.d.). https://github.com/CTSRD-CHERI/cheribsd.

[59] Cambridge University. Cheriplot, (n.d.). https://github.com/CTSRD-CHERI/cheriplot.

[60] Cambridge University. Cheri flute, (n.d.). https://github.com/CTSRD-CHERI/Flute.

[61] Cambridge University. Cheri ibex, (n.d.). https://github.com/CTSRD-CHERI/ibex.

[62] Cambridge University. Cheri piccolo, (n.d.). https://github.com/CTSRD-CHERI/Piccolo.

[63] Cambridge University. Sail cheri-mips, (n.d.). https://github.com/CTSRD-CHERI/sail-cheri-mips.

[64] Cambridge University. Sail cheri-risc-v, (n.d.). https://github.com/CTSRD-CHERI/sail-cheri-riscv.

[65] Cambridge University. Sail morello, (n.d.). https://github.com/CTSRD-CHERI/sail-morello.

[66] Cambridge University. Cheri toooba, (n.d.). https://github.com/CTSRD-CHERI/Toooba.

[67] Y. Cao, Y. Feng, H. Li, C. Huang, F. Jian, H. Li, and X. Wang. Tech-asan: Two-stage check for address sanitizer. In *Proceedings of the 16th International Conference on Internetware*, Internetware '25, page 96–107, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400719264. doi: 10.1145/3755881.3755918. URL https://doi.org/10.1145/3755881.3755918.

[68] N. P. Carter, S. W. Keckler, et al. Hardware Support for Fast Capability-Based Addressing. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI. Association for Computing Machinery, 1994. ISBN 0897916603. doi: 10.1145/195473.195579. URL https://doi.org/10.1145/195473.195579.

[69] CheriBSD community. CheriBSD packages, 2024. https://pkg.cheribsd.org/.

[70] D. Chisnall. *CHERIoT Programmers' Guide*. 2025. https://cheriot.org/book/.

[71] D. Chisnall, B. Davis, K. Gudka, et al. Cheri jni: Sinking the java security model into the c. *SIGARCH Comput. Archit. News*, 45(1), apr 2017. ISSN 0163-5964. doi: 10.1145/3093337.3037725. URL https://doi.org/10.1145/3093337.3037725.

[72] T. Cloosters, O. Draissi, et al. Memory corruption at the border of trusted execution. *IEEE Security & Privacy*, 22(4):87–96, 2024. doi: 10.1109/MSEC.2024.3381439.

[73] Codasip. Cheri security technology, (n.d.). https://codasip.com/solutions/riscv-processor-safety-security/cheri/.

[74] A. M. Community. Morello / newlib, 2020. Github repository, url: https://git.morello-project.org/morello/newlib.

[75] A. M. Community. Morello / musl libc, 2021. Retrieved from: https://git.morello-project.org/morello/musl-libc.

[76] C. Community. Github - ctsrd-cheri/testrig: Testing processors with random instruction generation, 2018. Github Repository: https://github.com/CTSRD-CHERI/TestRIG.

[77] C. Community. bhyve hypervisor - getting started with cheribsd 24.05, 2024. Retrieved from:https://ctsrd-cheri.github.io/cheribsd-getting-started/features/bhyve.html.

[78] C. Community. GitHub - CTSRD-CHERI/gdb: The GNU debugger extended to support CHERI, 2025. Github Repository:https://github.com/CTSRD-CHERI/gdb.

[79] D. C. community. Dirty cow (cve-2016-5195). https://dirtycow.ninja/, 2016. Accessed on: 22-04-2026.

[80] B. Connectivity, A. IDIADA, et al. Autocheri, (n.d.). https://autocheri.tech/.

[81] L. Cordeiro, B. Fischer, et al. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 137–148, 2009. doi: 10.1109/ASE.2009.63.

[82] B. Cordill, D. Fang, and S. Xu. A comprehensive survey of security and privacy in uav systems. *IEEE Access*, 13:117843–117866, 2025. doi: 10.1109/ACCESS.2025.3583985.

[83] I. Corporation. iapx 432 general data processor architecture reference manual, 1981.

[84] I. Corporation. Introduction to the iapx 432 architecture, 1981.

[85] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of Buffer-Overflow attacks. In *7th USENIX Security Symposium (USENIX Security 98)*, San Antonio, TX, Jan. 1998. USENIX Association. URL `https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention`.

[86] D. Cox, C. Johnson, et al. Ibm as/400 processor technology. In *[1991 Proceedings] IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 448–452, 1991. doi: 10.1109/ICCD.1991.139944.

[87] CVE. CVE-2020-10064 | CVE, 2021. `https://www.cve.org/CVERecord?id=CVE-2020-10064`.

[88] CVE. CVE-2020-10065 | CVE, 2021. `https://www.cve.org/CVERecord?id=CVE-2020-10065`.

[89] CVE.org. CVE: Common Vulnerabilities and Exposures. `https://www.cve.org/`, Accessed on: 29-03-2026.

[90] B. Davis. Cheriabi: Hardware enforced memory safety for freebsd, 2019. `https://papers.freebsd.org/2019/bsdcan/davis-cheriabi/`.

[91] B. Davis, R. N. Watson, A. Richardson, et al. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment. In *Proceedings of the 24th International Conference on ASPLOS*, ASPLOS '19. Association for Computing Machinery, 2019. ISBN 9781450362405. doi: 10.1145/3297858.3304042. URL `https://doi.org/10.1145/3297858.3304042`.

[92] J. de Bonfils Lavernelle, P.-F. Bonnefoi, B. Gonzalvo, and D. Sauveron. Assessment of spatial isolation in jailhouse: Towards a generic approach. *Computer Networks*, 245:110402, 2024. ISSN 1389-1286. doi: https://doi.org/10.1016/

j.comnet.2024.110402. URL https://www.sciencedirect.com/science/article/pii/S1389128624002342.

[93] A. DeHon, B. Karel, et al. Preliminary design of the safe platform. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, PLOS '11. Association for Computing Machinery, 2011. ISBN 9781450309790. doi: 10.1145/2039239.2039245. URL https://doi.org/10.1145/2039239.2039245.

[94] J. B. Dennis and E. C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Commun. ACM*, 9(3), mar 1966. ISSN 0001-0782. doi: 10.1145/365230.365252. URL https://doi.org/10.1145/365230.365252.

[95] U. Dhawan, A. Kwon, E. Kadric, et al. Hardware support for safety interlocks and introspection. In *2012 IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 1–8, 2012. doi: 10.1109/SASOW.2012.11.

[96] Diligent. Genesys 2 reference manual. https://digilent.com/reference/programmable-logic/genesys-2/reference-manual, Accessed on: 30-03-2026.

[97] M. Dodson. *Capability-based access control for cyber physical systems*. PhD thesis, Apollo - University of Cambridge Repository, 2021. URL https://www.repository.cam.ac.uk/handle/1810/329679.

[98] M. Dodson, A. R. Beresford, et al. Cheri macaroons: Efficient, host-based access control for cyber-physical systems. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 688–693, 2020. doi: 10.1109/EuroSPW51379.2020.00099.

[99] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017. doi: 10.48550/arXiv.1711.03938.

[100] Dronecode Project. Mavlink developer guide. https://mavlink.io/en/, Accessed on: 30-03-2026.

[101] I. Dudina and I. Stark. Static analysis for transitioning to cheri c/c++. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, SOAP 2024. Association for Computing Machinery, 2024. ISBN 9798400706219. doi: 10.1145/3652588.3663323. URL https://doi.org/10.1145/3652588.3663323.

[102] I. Dudina and I. Stark. Static analysis to make the most of cheri c/c++ for existing code: improving memory safety at scale. *International Journal on Software Tools for Technology Transfer*, pages 1433–2787, 2025. doi: 10.1007/s10009-025-00781-6.

[103] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332132. doi: 10.1145/2663716.2663755. URL https://doi.org/10.1145/2663716.2663755.

[104] A. El-Korashy, S. Tsampas, et al. CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle. *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–16, 2020.

[105] D. M. England. Operating system of system 250. Proceedings of International Switching Symposium, 1972.

[106] D. M. England. Capability concept mechanism and structure in system 25. Proceedings of the International Workshop on Protection in Operating Systems, 1974.

[107] ESBMC. The esbmc model checker, (n.d.). https://github.com/esbmc/esbmc.

[108] L. Esswood. *CheriOS: Designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor*. PhD thesis, Apollo - University of Cambridge Repository, 2020. URL https://www.repository.cam.ac.uk/handle/1810/326715.

[109] N. et al. Cuda, release: 10.2.89, 2020. URL https://developer.nvidia.com/cuda-toolkit.

[110] J. Evans. A scalable concurrent malloc(3) implementation for freebsd. *FreeBSD*, 01 2006.

[111] Evidence. Erika enterprise | open source rtos osek/vdx kernel, (n.d.). http://erika.tuxfamily.org/drupal/. Accessed on: 25-08-2026.

[112] R. S. Fabry. Capability-Based Addressing. *Commun. ACM*, 17(7), jul 1974. ISSN 0001-0782. doi: 10.1145/361011.361070. URL https://doi.org/10.1145/361011.361070.

[113] J. Farkas, L. L. Bello, and C. Gunther. Time-sensitive networking standards. *IEEE Communications Standards Magazine*, 2(2):20–21, 2018. doi: 10.1109/MCOMSTD.2018.8412457.

[114] D. Ferraro, L. Palazzi, F. Gavioli, M. Guzzinati, A. Bernardi, B. Rouxel, P. Burgio, and M. Solieri. Time-sensitive autonomous architectures. *Real-Time Systems*, 59:568–608, 2023. ISSN 0922-6443. doi: 10.1007/s11241-023-09404-2.

[115] D. Ferraro, A. Bastoni, A. Zuepke, and A. Marongiu. Enabling security on the edge: A cheri compartmentalized network stack. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–7, 2025. doi: 10.23919/DATE64628.2025.10992964.

[116] E. A. Feustel. The rice research computer: a tagged architecture. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, AFIPS '72 (Spring). Association for Computing Machinery, 1971. ISBN 9781450379090. doi: 10.1145/1478873.1478920. URL `https://doi.org/10.1145/1478873.1478920`.

[117] N. Filardo, B. F. Gutstein, J. Woodruff, et al. Cornucopia reloaded: Load barriers for cheri heap temporal safety. In *Proceedings of the 29th ACM International Conference on ASPLOS, Volume 2*, ASPLOS '24, Apr. 2024. URL `https://www.microsoft.com/en-us/research/publication/cornucopia-reloaded-load-barriers-cheri-heap-temporal-safety/`.

[118] M. Fillo, S. W. Keckler, et al. The M-Machine multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 146–156, 1995. doi: 10.1109/MICRO.1995.476822.

[119] J. M. Foster, C. I. Moir, et al. An introduction to the flex computer system, 1979.

[120] J. M. Foster, I. F. Currie, et al. Flex: A working computer with an architecture based on procedure values, 1982.

[121] F. A. Fuchs, J. Woodruff, et al. Developing a Test Suite for Transient-Execution Attacks on RISC-V and CHERI-RISC-V. 2021.

[122] F. A. Fuchs, J. Woodruff, et al. Architectural Contracts for Safe Speculation . In *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE Computer Society, Nov. 2023. doi: 10.1109/ICCD58817.2023.00093. URL `https://doi.ieeecomputersociety.org/10.1109/ICCD58817.2023.00093`.

[123] F. A. Fuchs, J. Woodruff, et al. Safe speculation for cheri. In *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, pages 364–372, 2024. doi: 10.1109/ICCD63220.2024.00063.

[124] A. Gaynor. What science can tell us about c anc c++'s security, 2020. `https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/`, Accessed on: 29-06-2026.

[125] A. Gaynor. Quantifying memory unsafety and reactions to it, Feb. 2021.

[126] Q. Ge, Y. Yarom, et al. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8, 04 2018. doi: 10.1007/s13389-016-0141-6.

[127] George Robotics Limited. MicroPython - Python for microcontrollers, 2024. Webpage, url: `https://micropython.org/`.

[128] A. Georges, A. Trieu, and L. Birkedal. Le Temps Des Cerises: Efficient Temporal Stack Safety on Capability Machines Using Directed Capabilities. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022. doi: 10.1145/3527318. URL `https://doi.org/10.1145/3527318`.

[129] A. L. Georges, A. Guéneau, T. Van Strydonck, A. Timany, A. Trieu, S. Huyghebaert, D. Devriese, and L. Birkedal. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi: 10.1145/3434287. URL `https://doi.org/10.1145/3434287`.

[130] H. Gernot. The seL4 Microkernel - An Introduction. Technical report, The seL4 Foundation, 2020. Whitepaper. `https://sel4.systems/About/seL4-whitepaper.pdf`.

[131] GitHub. GitHub Advisory Database. `https://github.com/advisories`, Accessed on: 29-03-2026.

[132] E. P. Gomes, A. Jakubiak, G. Gracioli, and T. Kloda. Memguard-rw: Improved real-time memory bandwidth regulation within a hypervisor. In *2025 28th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 1–10, 2025. doi: 10.1109/ISORC65339.2025.00021.

[133] Google Inc. About Fuchsia. `https://fuchsia.dev/fuchsia-src/concepts`, 2022.

[134] R. Grisenthwaite, G. Barnes, R. N. Watson, et al. The arm morello evaluation platform—validating cheri-based security in a high-performance system. *IEEE Micro*, 43(3):50–57, 2023. doi: 10.1109/MM.2023.3264676.

[135] R. Grisenthwaite, G. Barnes, R. N. M. Watson, S. W. Moore, P. Sewell, and J. Woodruff. The arm morello evaluation platform—validating CHERI-based security in a high-performance system. *IEEE Micro*, 43(3):50–57, 2023. doi: 10.1109/MM.2023.3264676.

[136] W. T. Group. Worldguard specification. Technical report, RISC-V Internacional, 2023. `https://lists.riscv.org/g/security/attachment/711/0/worldguard_rvia_spec-v0.4.pdf`.

[137] J. Gu, X. Wu, W. Li, N. Liu, Z. Mi, Y. Xia, and H. Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 401–417. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL `https://www.usenix.org/conference/atc20/presentation/gu`.

[138] F. Guan, L. Peng, L. Perneel, and M. Timmerman. Open source FreeRTOS as a case study in real-time operating system evolution. *Journal of Systems and Software*, 118:19–35, Aug. 2016. doi: 10.1016/j.jss.2016.04.063.

[139] B. Gutstein. Memory safety with CHERI capabilities: security analysis, language interpreters, and heap temporal safety. Technical Report UCAM-CL-TR-975, University of Cambridge, Computer Laboratory, Nov. 2022. URL `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-975.pdf`.

[140] D. Hagimont, F. Saunier, et al. Hidden software capabilities. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE Computer Society, may 1996. doi: 10.1109/ICDCS.1996.507926. URL `https://doi.ieeecomputersociety.org/10.1109/ICDCS.1996.507926`.

[141] P. M. Hansen, M. A. Linton, et al. A performance evaluation of the intel iapx 432. *SIGARCH Comput. Archit. News*, 10(4), jun 1982. ISSN 0163-5964. doi: 10.1145/641542.641545. URL `https://doi.org/10.1145/641542.641545`.

[142] N. Hardy. KeyKOS Architecture. *SIGOPS Oper. Syst. Rev.*, 19(4), oct 1985. ISSN 0163-5980. doi: 10.1145/858336.858337. URL `https://doi.org/10.1145/858336.858337`.

[143] N. Hardy. The Confused Deputy: (Or Why Capabilities Might Have Been Invented). *SIGOPS Oper. Syst. Rev.*, 22(4), oct 1988. ISSN 0163-5980. doi: 10.1145/54289.871709. URL https://doi.org/10.1145/54289.871709.

[144] S. Harris, S. Cooksey, et al. Rust for Morello: Always-On Memory Safety, Even in Unsafe Code. In K. Ali and G. Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 39:1–39:27, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-281-5. doi: 10.4230/LIPIcs.ECOOP.2023.39. URL https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.39.

[145] R. Hat. Cve-2022-0847 - red hat customer portal. https://access.redhat.com/security/cve/cve-2022-0847, 2022. Accessed on: 22-04-2026.

[146] G. Heiser and M. Atkins. seL4 on Arm Morello. seL4 Summit 2023, 2023. URL https://sel4summit2023.sched.com/event/1O3je/sel4-on-arm-morello.

[147] J. Henes, M. Muench, et al. Transient-execution attacks on the cheri morello platform. *Proceedings of the Microarchitecture Security Conference*, 03 2025. doi: 10.46586/uasc.2025.104.

[148] N. Hoang Nga, S. Siraj Ahmed, K. Emirhan, et al. Addressing automotive cybersecurity risks with an arm morello capability-enhanced prototype. Preprint, Authorea, DOI: 10.22541/au.171401582.21766451/v1, April 2024.

[149] M. E. Houdek, F. G. Soltis, et al. Ibm system/38 support for capability-based addressing. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81. IEEE Computer Society Press, 1981.

[150] Huawei. Linux cheri riscv project. https://github.com/cheri-linux, 2022. Accessed on: 10-04-2024.

[151] G. Im. Notes on kalman filter (kf, ekf, eskf, iekf, ieskf), 2024. URL https://arxiv.org/abs/2406.06427.

[152] C. H. E. R. Instructions. LLVM Project, (n.d.). https://github.com/CTSRD-CHERI/llvm-project. Accessed on: 08-01-2025.

[153] International Electrotechnical Commission. Overview of IEC 61508 and functional safety. https://assets.iec.ch/public/acos/IEC%2061508%20&%20Functional%20Safety-2022.pdf?2023040501, 2022. Accessed on: 30-06-2026.

[154] S. Iqbal. A study on UAV operating system security and future research challenges. In *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0759–0765, Las Vegas, NV, USA, 2021. doi: 10.1109/CCWC51732.2021.9376151.

[155] S. Jacklin. Certification of safety-critical software under do-178c and do-278a. 06 2012. ISBN 978-1-60086-939-6. doi: 10.2514/6.2012-2473.

[156] J. Jackson. Cheri-zephyr. In *CHERITech*, 2024. URL `https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/cheritech24/02_01_Jennifer_Jackson_Minmin_Jiang.pdf`.

[157] D. Jacob and J. Singer. Capability Boehm: Challenges and Opportunities for Garbage Collection with Capability Hardware. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2022. Association for Computing Machinery, 2022. ISBN 9781450392518. doi: 10.1145/3516807.3516823. URL `https://doi.org/10.1145/3516807.3516823`.

[158] Jailhouse. Github - jailhouse - linux-based partitioning hypervisor, 2015. URL `https://github.com/siemens/jailhouse`. Accessed on: 22-04-2026.

[159] Jenkinson, G. and Mazzinghi, A. and others. Cheri-based memory protection and compartmentalisation for web services on morello. Technical report, Capabilities Limited, 2024. URL `https://github.com/Capabilities-Limited/Capabilities-Limited.github.io/blob/main/pdfs/20231020-capltd-cheri-server-evaluation-report-public-version1-FINAL.pdf`.

[160] J. Jennifer, J. Minmin, et al. Cheri-crypt: Transparent memory encryption on capability architectures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025. ISSN 2569-2925.

[161] A. Joannou. *High-performance memory safety - Optimizing the CHERI capability machine*. PhD thesis, 09 2017.

[162] A. Joannou, J. Woodruff, R. Kovacsics, et al. Efficient Tagged Memory. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 641–648, 2017. doi: 10.1109/ICCD.2017.112.

[163] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), Dec. 2017. doi: 10.1145/3158154. URL `https://doi.org/10.1145/3158154`.

[164] C. S. Kalutharage, S. Mohan, et al. Enhancing automotive intrusion detection systems with capability hardware enhanced risc instructions-based memory protection. *Electronics*, 14(3), 2025. ISSN 2079-9292. doi: 10.3390/electronics14030474. URL https://www.mdpi.com/2079-9292/14/3/474.

[165] D. Kaplan, J. Powell, et al. Amd memory encryption. Technical report, AMD, 2021.

[166] M. Kaufmann, J. S. Moore, et al. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000. ISBN 0792377443.

[167] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[168] P. Kocher, J. Horn, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019. doi: 10.1109/SP.2019.00002.

[169] J. A. Kressel. Exploring Software Comparmentalisation with Hardware Capabilities, 2023.

[170] J. A. Kressel, H. Lefeuvre, et al. Software compartmentalization tradeoffs with hardware capabilities. In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*, PLOS '23. Association for Computing Machinery, 2023. ISBN 9798400704048. doi: 10.1145/3623759.3624550. URL https://doi.org/10.1145/3623759.3624550.

[171] S. Kuenzer, V. Bădoiu, H. Lefeuvre, et al. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the 16th European Conference on Computer Systems*, EuroSys '21. Association for Computing Machinery, 2021. ISBN 9781450383349. doi: 10.1145/3447786.3456248. URL https://doi.org/10.1145/3447786.3456248.

[172] N. Kumar and A. Chaudhary. Surveying cybersecurity vulnerabilities and countermeasures for enhancing uav security. *Comput. Netw.*, 252(C), Oct. 2024. ISSN 1389-1286. doi: 10.1016/j.comnet.2024.110695. URL https://doi.org/10.1016/j.comnet.2024.110695.

[173] C. Kwan, Y. Xin, et al. Cheri concentrate in acl2: Extended abstract at the 18th international workshop on the acl2 theorem prover and its applications. *Electronic Proceedings in Theoretical Computer Science*, 393, 11 2023. doi: 10.4204/EPTCS.393.2.

[174] A. Kwon, U. Dhawan, et al. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13. Association for Computing Machinery, 2013. ISBN 9781450324779. doi: 10.1145/2508859.2516713. URL https://doi.org/10.1145/2508859.2516713.

[175] L4Re.org. L4Re Runtime Environment - Interface and Usage Documentation. https://l4re.org/doc/l4re_intro.html, 2023.

[176] B. W. Lampson. Dynamic Protection Structures. In *Proceedings of the November 18-20, 1969, Fall Joint Computer Conference*, AFIPS '69 (Fall). Association for Computing Machinery, 1969. ISBN 9781450379069. doi: 10.1145/1478559.1478563. URL https://doi.org/10.1145/1478559.1478563.

[177] D. Lea. A memory allocator, 1996. Retrieved from:https://gee.cs.oswego.edu/dl/html/malloc.html. Accessed on: 01-10-2025.

[178] D. Lee, D. Kohlbrenner, et al. Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the 15th European Conference on Computer Systems*, EuroSys '20. Association for Computing Machinery, 2020. ISBN 9781450368827. doi: 10.1145/3342195.3387532. URL https://doi.org/10.1145/3342195.3387532.

[179] H. Lefeuvre, V. Bădoiu, A. Jung, et al. FlexOS: Towards Flexible OS Isolation. In *Proceedings of the 27th ACM International Conference on ASPLOS*, ASPLOS '22. Association for Computing Machinery, 2022. ISBN 9781450392051. doi: 10.1145/3503222.3507759. URL https://doi.org/10.1145/3503222.3507759.

[180] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, 1984. ISBN 0932376223.

[181] Z. Li, J. Zhang, J. Tlatelpa-Agustin, X. Chen, and A. Burtsev. Understanding the security impact of cheri on the operating system kernel. In *2025 IEEE Annual Computer Security Applications Conference (ACSAC)*, pages 212–227, 2025. doi: 10.1109/ACSAC67867.2025.00031.

[182] P. Liétar, T. Butler, S. Clebsch, et al. snmalloc: a message passing allocator. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2019. Association for Computing Machinery, 2019. ISBN 9781450367226. doi: 10.1145/3315573.3329980. URL https://doi.org/10.1145/3315573.3329980.

[183] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, Santa Clara, CA, Aug. 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand.

[184] A. Limited. Arm architecture reference manual supplement - morello for a-profile architecture. Technical report, Arm Limited, 2022. Retrieved from: https://developer.arm.com/documentation/ddi0606/.

[185] A. Limited. Arm morello system development platform technical reference manual. Technical report, Arm Limited, 2022. Retrieved from: https://developer.arm.com/documentation/102278.

[186] W. Lin, C. Xiong, W. Wu, F. Shi, K. Li, and M. Xu. Performance interference of virtual machines: A survey. *ACM Comput. Surv.*, 55(12), Mar. 2023. ISSN 0360-0300. doi: 10.1145/3573009. URL https://doi.org/10.1145/3573009.

[187] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[188] M. Lipp, M. Schwarz, et al. Meltdown: reading kernel memory from user space. *Commun. ACM*, 63(6), May 2020. ISSN 0001-0782. doi: 10.1145/3357033. URL https://doi.org/10.1145/3357033.

[189] lowRISC. Open source silicon root of trust (rot). https://opentitan.org/, 2023.

[190] lowRISC. Unveiling sonata: Affordable cheri hardware for embedded systems, (n.d.). https://lowrisc.org/news/2024/01/sonata-board-update/.

[191] lowRISC. Sonata pcb, (n.d.). https://github.com/newaetech/sonata-pcb/.

[192] lowRISC. Sonata system, (n.d.). https://github.com/lowRISC/sonata-system.

[193] lowRISC. Sunburst chip, (n.d.). https://github.com/lowRISC/sunburst-chip.

[194] lowRISC CIC, DSbD/UKRI, et al. Cap-tee project, (n.d.). https://cap-tee.org/.

[195] lowRISC CIC, DSbD/UKRI, et al. Sunburst project, (n.d.). https://www.sunburst-project.org/.

[196] D. Lowther, D. Jacob, et al. Morello micropython: A python interpreter for cheri. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2023. Association for Computing Machinery, 2023. ISBN 9798400703805. doi: 10.1145/3617651.3622991. URL https://doi.org/10.1145/3617651.3622991.

[197] D. Lowther, D. Jacob, et al. Secure scripting with cheriot micropython. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, CC '25. Association for Computing Machinery, 2025. ISBN 9798400714078. doi: 10.1145/3708493.3712694. URL https://doi.org/10.1145/3708493.3712694.

[198] H. Marco-Gisbert and I. Ripoll Ripoll. Address space layout randomization next generation. *Applied Sciences*, 9(14), 2019. ISSN 2076-3417. doi: 10.3390/app9142928. URL https://www.mdpi.com/2076-3417/9/14/2928.

[199] A. T. Markettos, J. Baldwin, et al. Position paper:defending direct memory access with cheri capabilities. In *Proceedings of the 9th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '20. Association for Computing Machinery, 2021. ISBN 9781450388986. doi: 10.1145/3458903.3458910. URL https://doi.org/10.1145/3458903.3458910.

[200] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In *Workshop on NG-RES*, volume 77, 2020.

[201] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In M. Bertogna and F. Terraneo, editors, *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, volume 77 of *Open Access Series in Informatics (OASIcs)*, pages 3:1–3:14, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-136-8. doi: 10.4230/OASIcs.NG-RES.2020.3. URL https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.NG-RES.2020.3.

[202] S. Mazor. Intel's 8086. *IEEE Annals of the History of Computing*, 32(1): 75–79, 2010. doi: 10.1109/MAHC.2010.22.

[203] A. Mazzinghi and R. o. Sohan. Pointer Provenance in a Capability Architecture. In *Proceedings of the 10th USENIX Conference on Theory and Practice of Provenance*, TaPP'18. USENIX Association, 2018.

[204] L. Meier, D. Honegger, and M. Pollefeys. Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6235–6240, 2015. doi: 10.1109/ICRA.2015.7140074.

[205] K. Memarian. *The Cerberus C Semantics*. University of Cambridge, 2022. URL `https://books.google.pt/books?id=ohwS0AEACAAJ`.

[206] K. Memarian, J. Matthiesen, J. Lingard, et al. Into the Depths of C: Elaborating the de Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16. Association for Computing Machinery, 2016. ISBN 9781450342612. doi: 10.1145/2908080.2908081. URL `https://doi.org/10.1145/2908080.2908081`.

[207] K. Memarian, V. B. F. Gomes, et al. Exploring C Semantics and Pointer Provenance. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi: 10.1145/3290380. URL `https://doi.org/10.1145/3290380`.

[208] G. Merve, E. Håkan, et al. Mon chèri <3 adapting capability hardware enhanced risc with conditional capabilities, 2024. URL `https://arxiv.org/abs/2407.08663`.

[209] Microsoft. Cheriiot, (n.d.). `https://github.com/microsoft/cheriot-ibex`.

[210] Microsoft. Portmeirion, (n.d.). `https://www.microsoft.com/en-us/research/project/portmeirion/`.

[211] T. Miemietz, M. Planeta, and V. L. Reusch. New mechanism for fast system calls, 2021. URL `https://arxiv.org/abs/2112.10106`.

[212] M. Miller. Trends and challenges in the vulnerability mitigation landscape, Aug. 2019.

[213] M. Miller, K. Yee, et al. Capability myths demolished. 12 2003.

[214] V. E. Moghadam, G. Serra, F. Aromolo, G. Buttazzo, and P. Prinetto. Memory integrity techniques for memory-unsafe languages: A survey. *IEEE Access*, 12:43201–43221, 2024. doi: 10.1109/ACCESS.2024.3380478.

[215] S. Mubeen, E. Lisova, and A. Vulgarakis Feljan. Timing predictability and security in safety-critical industrial cyber-physical systems: A position paper. *Applied Sciences*, 10(9), 2020. ISSN 2076-3417. doi: 10.3390/app10093125. URL https://www.mdpi.com/2076-3417/10/9/3125.

[216] G. Mullen and L. Meany. Assessment of buffer overflow based attacks on an IoT operating system. In *2019 Global IoT Summit (GIoTS)*, pages 1–6, Aarhus, Denmark, 2019. doi: 10.1109/GIOTS.2019.8766434.

[217] A. B. Narappa, F. Parola, S. Qi, and K. K. Ramakrishnan. Z-Stack: A high-performance DPDK-based zero-copy TCP/IP protocol stack. In *2024 IEEE 30th International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 100–105, Boston, MA, USA, 2024. doi: 10.1109/LANMAN61958.2024.10621881.

[218] M. Naylor, A. Joannou, A. T. Markettos, et al. Cheri-simt report: implementing capability memory protection in gpgpus. Technical Report UCAM-CL-TR-997, University of Cambridge, April 2025. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-997.html.

[219] R. M. Needham and R. D. Walker. The cambridge cap computer and its protection system. In *Proceedings of the 6th ACM Symposium on Operating Systems Principles*, SOSP '77. Association for Computing Machinery, 1977. ISBN 9781450378673. doi: 10.1145/800214.806541. URL https://doi.org/10.1145/800214.806541.

[220] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007. ISSN 0362-1340. doi: 10.1145/1273442.1250746. URL https://doi.org/10.1145/1273442.1250746.

[221] P. G. Neumann. *CHAPTER 6 Fundamental Trustworthiness Principles in CHERI*, pages 199–236. MIT Press, 2018.

[222] K. Nienhuis, A. Joannou, T. Bauereiss, et al. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1003–1020, 2020. doi: 10.1109/SP40000.2020.00055.

[223] NIST. Nvd - cve-2023-4911. https://nvd.nist.gov/vuln/detail/cve-2023-4911, 2023. Accessed on: 22-04-2026.

[224] R. M. Norton. *Hardware support for compartmentalisation*. Number 887. 2016. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-887.html.

[225] NSA Center for Assured Software. Juliet c/c++ 1.3 - nist software assurance reference dataset, 2017. url:https://samate.nist.gov/SARD/test-suites/112.

[226] NVD. NVD - CVE-2020-10064, 2021. https://nvd.nist.gov/vuln/detail/CVE-2020-10064.

[227] NVD. NVD - CVE-2020-10065, 2021. https://nvd.nist.gov/vuln/detail/CVE-2020-10065.

[228] Q. Ochem, R. Tice, et al. Ada for Embedded C Developer. Technical report, AdaCore, 2021.

[229] U. of Cambridge. Adversarial cheri exercises and missions, (n.d.). https://github.com/CTSRD-CHERI/cheri-exercises.

[230] P. Olivier. FlexCap: Exploring Hardware Capabilities in Unikernels and Flexible Isolation Oses. CHERITech'24 presentation: https://flexcap-project.github.io/assets/cheritech24-presentation.pdf, 2024.

[231] F. Pagano, L. Verderame, et al. Understanding fuchsia security. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 12, 2021. doi: 10.22667/JOWUA.2021.09.30.047.

[232] S. Park, O. Kwon, et al. Mind control attack: Undermining deep learning with gpu memory exploitation. *Comput. Secur.*, 102(C), Mar. 2021. ISSN 0167-4048. doi: 10.1016/j.cose.2020.102115. URL https://doi.org/10.1016/j.cose.2020.102115.

[233] S. H. Park, R. Pai, et al. A formal cheri-c memory model. *Archive of Formal Proofs*, Nov. 2022. ISSN 2150-914x.

[234] S. H. Park, R. Pai, et al. *A Formal CHERI-C Semantics for Verification*, page 549–568. Springer Nature Switzerland, 2023. ISBN 9783031308239. doi: 10.1007/978-3-031-30823-9_28. URL http://dx.doi.org/10.1007/978-3-031-30823-9_28.

[235] G. Parmer and R. West. Predictable Interrupt Management and Scheduling in the Composite Component-Based System. In *2008 Real-Time Systems Symposium*, pages 232–243, 2008. doi: 10.1109/RTSS.2008.13.

[236] S. Patel, S. Agrawal, et al. Cheri-picking: Leveraging capability hardware for prefetching. In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*, PLOS '23. Association for Computing Machinery,

2023. ISBN 9798400704048. doi: 10.1145/3623759.3624553. URL https://doi.org/10.1145/3623759.3624553.

[237] D. A. Patterson and C. H. Sequin. Risc i: A reduced instruction set vlsi computer. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81. IEEE Computer Society Press, 1981.

[238] J. Perez-Cerrolaza, P. Balbastre, E. Mezzetti, A. Crespo, J. González Eguiluz, M. Martin Acosta, and C.-A. Lefebvre. Virtualization technology for dependable embedded systems: A survey for transportation and industrial domains. *ACM Comput. Surv.*, 58(4), Nov. 2025. ISSN 0360-0300. doi: 10.1145/3765736. URL https://doi.org/10.1145/3765736.

[239] E. M. Peterson. Application of sae arp4754a to flight critical systems, 2015. URL https://ntrs.nasa.gov/citations/20160001634.

[240] O. Pierre, A. Barbalace, et al. The Case for Intra-Unikernel Isolation. mar 2020. The 10th Workshop on Systems for Post-Moore Architectures.

[241] S. Pinto and N. Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Comput. Surv.*, 51(6), Jan. 2019. ISSN 0360-0300. doi: 10.1145/3291047. URL https://doi.org/10.1145/3291047.

[242] X. Project. Hypervisor | xen project. https://xenproject.org/projects/hypervisor/.

[243] O. Purdila, L. A. Grijincu, and N. Tapus. LKL: The linux kernel library. In *9th RoEduNet IEEE International Conference*, pages 328–333, Sibiu, Romania, 2010.

[244] F. D. Pustelnik, X. M. Sass, et al. Whispering Pixels: Exploiting Uninitialized Register Accesses in Modern GPUs . In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*. IEEE Computer Society, July 2024. doi: 10.1109/EuroSP60621.2024.00026. URL https://doi.ieeecomputersociety.org/10.1109/EuroSP60621.2024.00026.

[245] PX4. Open source autopilot for drones - px4 autopilot, 2024. Retrieved from:https://px4.io/.

[246] R. Rajesh, K. B. Ramia, and M. Kulkarni. Integration of LwIP stack over Intel® DPDK for high throughput packet delivery to applications. In *2014 Fifth International Symposium on Electronic System Design*, pages 130–134, Surathkal, India, 2014. doi: 10.1109/ISED.2014.34.

[247] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady. Security in embedded systems: Design challenges. *ACM Transactions on Embedded Computing Systems*, 3(3):461–491, Aug. 2004. doi: 10.1145/1015047.1015049.

[248] Red Hat. Cve-2021-3156 - red hat customer portal. `https://access.redhat.com/security/cve/cve-2021-3156`, 2021. Accessed on: 22-04-2026.

[249] D. D. Redell. Naming and protection in extendable operating systems. Technical report, University of California, 1974.

[250] H. U. Rehman, M. Asif, and M. Ahmad. Future applications and research challenges of IoT. In *2017 International Conference on Information and Communication Technologies (ICICT)*, pages 68–74, Karachi, Pakistan, 2017. doi: 10.1109/ICICT.2017.8320166.

[251] A. Richardson. *Complete spatial safety for C and C++ using CHERI capabilities*. PhD thesis, University of Cambridge, 2020.

[252] N. Rodday. Blackhat Asia - Hacking a Professional Drone. `https://www.blackhat.com/docs/asia-16/materials/asia-16-Rodday-Hacking-A-Professional-Drone.pdf`, 2016.

[253] J. Roels, A. Jacobs, et al. Cuda, woulda, shoulda: Returning exploits in a sass-y world. In *Proceedings of the 18th European Workshop on Systems Security*, EuroSec'25. Association for Computing Machinery, 2025. ISBN 9798400715631. doi: 10.1145/3722041.3723099. URL `https://doi.org/10.1145/3722041.3723099`.

[254] J. Rosenberg and D. Abramson. Monads-pc-a capability-based workstation to support software engineering. In *Proceedings of the Hawaii International Conference on System Science*, pages 222–231. IEEE Computer Society, 1985.

[255] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini. Pulp: A parallel ultra low power platform for next generation iot applications. In *2015 IEEE HCS*, 2015.

[256] P. Rugg, J. Woodruff, et al. A Suite of Processors to Explore CHERI-RISC-V Micro Architecture . In *2024 27th Euromicro Conference on Digital System Design (DSD)*. IEEE Computer Society, Aug. 2024. doi: 10.1109/DSD64264.2024.00054. URL `https://doi.ieeecomputersociety.org/10.1109/DSD64264.2024.00054`.

[257] B. Sá, J. Woodruff, P. Rugg, A. Joannou, S. Pinto, and S. W. Moore. Cva6-cheri—an open-source rv64y implementation for commercialization.

CHERITech '25 — https://cheri-alliance.org/wp-content/uploads/2025/11/2.1-Jonathan-Woodruff-CHERITech25.pdf, 2025.

[258] B. Sá, D. Ferraro, E. de Matos, R. Psiakis, A. Marongiu, A. Bastoni, and S. Pinto. Advanced security through hardware capabilities: A comprehensive survey on cheri technology. *ACM Comput. Surv.*, 58(15), Sept. 2026. ISSN 0360-0300. doi: 10.1145/3833870. URL https://doi.org/10.1145/3833870.

[259] V. Safronov, I. Bostan, et al. How memory-safe is iot? assessing the impact of memory-protection solutions for securing wireless gateways, 2024. URL https://arxiv.org/abs/2411.01377.

[260] V. A. Sartakov, L. Vilanova, and P. Pietzuch. Cubicleos: a library os with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on ASPLOS*, ASPLOS '21. Association for Computing Machinery, 2021. ISBN 9781450383172. doi: 10.1145/3445814.3446731. URL https://doi.org/10.1145/3445814.3446731.

[261] V. A. Sartakov, L. Vilanova, D. Eyers, T. Shinagawa, and P. Pietzuch. CAP-VMs: Capability-Based isolation and sharing in the cloud. In *OSDI 22*, 2022.

[262] V. A. Sartakov, L. Vilanova, et al. CAP-VMs: Capability-Based isolation and sharing in the cloud. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, 07 2022. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/sartakov.

[263] V. A. Sartakov, L. Vilanova, et al. CAP-VMs: Capability-Based Isolation and Sharing in the Cloud. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, July 2022. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/sartakov.

[264] V. A. Sartakov, L. Vilanova, M. Geden, D. Eyers, T. Shinagawa, and P. Pietzuch. ORC: Increasing cloud memory density via object reuse with capabilities. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*, pages 573–587, 2023.

[265] V. A. Sartakov, L. Vilanova, et al. ORC: Increasing cloud memory density via object reuse with capabilities. In *17th USENIX Symposium on*

Operating Systems Design and Implementation (OSDI 23). USENIX Association, July 2023. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/sartakov.

[266] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, page 1, USA, 2010. USENIX Association. ISBN 8887666655554.

[267] seL4 Community. Capdl | sel4 docs, 2020. Retrieved from: https://docs.sel4.systems/projects/capdl/.

[268] seL4 Foundation. *camkes-tool/docs/index.md at master · seL4/camkes-tool · GitHub*, Accessed on: 31-03-2026.

[269] R. Semiconductor. *ISO 26262: Functional Safety Standard for Modern Road Vehicles*, Accessed on: 31-05-2026.

[270] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, June 2012. USENIX Association. ISBN 978-931971-93-5. URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[271] J. S. Shapiro, J. M. Smith, et al. EROS: A Fast Capability System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, SOSP '99. Association for Computing Machinery, 1999. ISBN 1581131402. doi: 10.1145/319151.319163. URL https://doi.org/10.1145/319151.319163.

[272] N. W. S. Sim. Strengthening memory safety in Rust: exploring CHERI capabilities for a safe language. MSc Thesis,https://nw0.github.io/cheri-rust.pdf, 2020.

[273] L. Skorstengaard, D. Devriese, et al. Reasoning about a machine with local capabilities: Provably safe stack and return pointer management. *ACM Trans. Program. Lang. Syst.*, 42(1), dec 2019. ISSN 0164-0925. doi: 10.1145/3363519. URL https://doi.org/10.1145/3363519.

[274] L. Skorstengaard, D. Devriese, et al. Stktokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi: 10.1145/3290332. URL https://doi.org/10.1145/3290332.

[275] L. Soares and M. Stumm. Flexsc: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 33–46, USA, 2010. USENIX Association.

[276] P. Soyeon, L. Sangho, et al. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, July 2019. ISBN 978-1-939133-03-8. URL `https://www.usenix.org/conference/atc19/presentation/park-soyeon`.

[277] J. E. Stone et al. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 2010.

[278] X. Sun, J. Singer, and Z. Wang. Sweet or sour cheri: Performance characterization of the arm morello platform. In *2025 IEEE International Symposium on Workload Characterization (IISWC)*, pages 423–438, 2025. doi: 10.1109/IISWC66894.2025.00042.

[279] SYSGO. Pikeos rtos & hypervisor | sysgo. `https://www.sysgo.com/pikeos`.

[280] B. Sá, L. Valente, J. Martins, D. Rossi, L. Benini, and S. Pinto. Cva6 risc-v virtualization: Architecture, microarchitecture, and design space exploration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 31(11), 2023.

[281] B. Sá, L. Valente, J. Martins, et al. Cva6 risc-v virtualization: Architecture, microarchitecture, and design space exploration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 31(11):1713–1726, 2023. doi: 10.1109/TVLSI.2023.3302837.

[282] Technology Innovation Institute. TII PX4/NuttX with MMU support. `https://github.com/tiiuae/px4-firmware`, Accessed on: 30-03-2026.

[283] S. Tsampas, D. Devriese, et al. Temporal safety for stack allocated memory on capability machines. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 243–24312, 2019. doi: 10.1109/CSF.2019.00024.

[284] J. Umeike, S. Agarwal, N. Lazarev, and M. Alian. Userspace networking in gem5. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 179–191, Indianapolis, IN, USA, 2024. doi: 10.1109/ISPASS61541.2024.00026.

[285] University of Cambridge. Compiler Explorer, 2017. https://cheri-compiler-explorer.cl.cam.ac.uk/noscript/z/1Panf4. Accessed on: 08-01-2025.

[286] University of Cambridge. Cheri clang/llvm and lld. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-llvm.html, Accessed on: 31-03-2026.

[287] B. G. Utley. *IBM System-38: Technical Developments*. IBM Corp, 1980. ISBN 0933186037.

[288] L. Valente, A. Veeran, M. Sinigaglia, Y. Tortorella, A. Nadalini, N. Wistoff, B. Sa, A. Garofalo, R. Psiakis, M. Tolba, A. Kulmala, N. Limaye, O. Sinanoglu, S. Pinto, D. Palossi, L. Benini, B. Mohammad, and D. Rossi. Shaheen: An open, secure, and scalable rv64 soc for autonomous nano-uavs. In *2023 IEEE Hot Chips 35 Symposium (HCS)*, aug 2023.

[289] T. Van Strydonck, J. Noorman, et al. CHERI-TrEE: Flexible enclaves on capability machines. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 1143–1159, 2023. doi: 10.1109/EuroSP57164.2023.00070.

[290] T. Van Strydonck et al. CHERI-TrEE: Flexible enclaves on capability machines. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 1143–1159, Delft, Netherlands, 2023. doi: 10.1109/EuroSP57164.2023.00070.

[291] P. Vasek and K. Ghose. A Fast Capability Extension to a RISC Architecture . In *EUROMICRO Conference*. IEEE Computer Society, Sept. 1996. doi: 10.1109/EURMIC.1996.546488. URL https://doi.ieeecomputersociety.org/10.1109/EURMIC.1996.546488.

[292] P. Vasek and K. Ghose. A comparison of two context allocation approaches for fast protected calls. In *Proceedings Fourth International Conference on High-Performance Computing*, pages 16–21, 1997. doi: 10.1109/HIPC.1997.634463.

[293] Vercammen, T. and Strydonck, T. V. and others. Borrowed capabilities: Flexibly enforcing revocation on a capability architecture. In *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 80–85, 2021. doi: 10.1109/EuroSPW54576.2021.00015.

[294] L. Vilanova, M. Ben-Yehuda, et al. Codoms: Protecting software with code-centric memory domains. In *2014 ACM/IEEE 41st International Symposium*

on Computer Architecture (ISCA), pages 469–480, 2014. doi: 10.1109/ISCA.2014.6853202.

[295] K. Wang, D. Kasatkin, et al. On cherifying linux 5.15, 2022.

[296] K. Wang, D. Kasatkin, et al. Cherifying linux: A practical view on using cheri. In *Proceedings of the 17th EuroSec*, EuroSec '24. Association for Computing Machinery, 2024. ISBN 9798400705427. doi: 10.1145/3642974.3652282. URL https://doi.org/10.1145/3642974.3652282.

[297] R. Watson, J. Woodruff, et al. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, 2015. doi: 10.1109/SP.2015.9.

[298] R. Watson, J. Woodruff, et al. Capability hardware enhanced risc instructions (cheri): Notes on the meltdown and spectre attacks. Technical Report UCAM-CL-TR-916, University of Cambridge, February 2018. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-916.pdf.

[299] R. N. Watson. Department of computer science and technology: Cheri-qemu, 2019. Retrieved from: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-qemu.html.

[300] R. N. Watson, J. Anderson, et al. A Taste of Capsicum: Practical Capabilities for UNIX. *Commun. ACM*, 55(3), mar 2012. ISSN 0001-0782. doi: 10.1145/2093548.2093572. URL https://doi.org/10.1145/2093548.2093572.

[301] R. N. Watson, D. Chisnall, B. Davis, et al. Bluespec extensible risc implementation: Beri hardware reference. 2014. URL https://api.semanticscholar.org/CorpusID:62911458.

[302] R. N. Watson, R. M. Norton, J. Woodruff, et al. Fast protection-domain crossing in the cheri capability-system architecture. *IEEE Micro*, 36(5):38–49, 2016. doi: 10.1109/MM.2016.84.

[303] R. N. Watson, P. G. Neumann, J. Woodruff, et al. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7). Technical Report UCAM-CL-TR-927, University of Cambridge, Computer Laboratory, June 2019. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf.

[304] R. N. Watson, A. Richardson, B. Davis, et al. Cheri c/c++ programming guide. Technical Report UCAM-CL-TR-947, University of Cambridge, Computer Laboratory, 2020. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.html.

[305] R. N. Watson, J. Clarke, P. Sewell, et al. *Early performance results from the prototype Morello microarchitecture*. 2023. doi: 10.48456/TR-986. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-986.html.

[306] R. N. Watson, P. G. Neumann, J. Woodruff, et al. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9). Technical Report UCAM-CL-TR-987, University of Cambridge, Computer Laboratory, Sept. 2023. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.pdf.

[307] R. N. M. Watson, J. Clarke, P. Sewell, J. Woodruff, S. W. Moore, G. Barnes, R. Grisenthwaite, K. Stacer, S. Baranga, and A. Richardson. Early performance results from the prototype Morello microarchitecture. Technical Report UCAM-CL-TR-986, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, September 2023.

[308] R. N. M. Watson, D. Chisnall, J. Clarke, B. Davis, N. W. Filardo, B. Laurie, S. W. Moore, P. G. Neumann, A. Richardson, P. Sewell, K. Witaszczyk, and J. Woodruff. Cheri: Hardware-enabled c/c++ memory protection at scale. *IEEE Security & Privacy*, 22(4):50–61, 2024. doi: 10.1109/MSEC.2024.3396701.

[309] R. N. M. Watson et al. CHERI: Hardware-enabled C/C++ memory protection at scale. *IEEE Security & Privacy*, 22(4):50–61, 2024. doi: 10.1109/MSEC.2024.3396701.

[310] Watson, R. N.M. and Laurie, B. and others. Assessing the viability of an open-source cheri desktop software ecosystem. Technical report, Capabilities Limited, 2021. URL https://github.com/Capabilities-Limited/Capabilities-Limited.github.io/blob/main/pdfs/20210917-capltd-cheri-desktop-report-version1-FINAL.pdf.

[311] N. Wesley Filardo, B. F. Gutstein, J. Woodruff, et al. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 608–625, 2020. doi: 10.1109/SP40000.2020.00098.

[312] D. A. Wheeler. Sloccount. https://dwheeler.com/sloccount/, Accessed on: 30-03-2026.

[313] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and Its Operating System*. Elsevier, January 1979. URL https://www.microsoft.com/en-us/research/publication/the-cambridge-cap-computer-and-its-operating-system/.

[314] N. C. Will and C. A. Maziero. Intel software guard extensions applications: A survey. *ACM Comput. Surv.*, 55(14s), July 2023. ISSN 0360-0300. doi: 10.1145/3593021. URL https://doi.org/10.1145/3593021.

[315] J. Woodruff. CHERI: A RISC capability machine for practical memory safety. Technical Report UCAM-CL-TR-858, University of Cambridge, Computer Laboratory, July 2014. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-858.pdf.

[316] J. Woodruff, R. N. Watson, D. Chisnall, et al. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, June 2014. ISBN 978-1-4799-4394-4 978-1-4799-4396-8. doi: 10.1109/ISCA.2014.6853201.

[317] J. Woodruff, A. Joannou, H. Xia, et al. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Transactions on Computers*, 68(10):1455–1469, 2019. doi: 10.1109/TC.2019.2914037.

[318] W. Wulf, E. Cohen, W. Corwin, et al. HYDRA: The Kernel of a Multiprocessor Operating System. *Commun. ACM*, 17(6), jun 1974. ISSN 0001-0782. doi: 10.1145/355616.364017. URL https://doi.org/10.1145/355616.364017.

[319] H. Xia. *Capability memory protection for embedded systems*. PhD thesis, University of Cambridge, Computer Laboratory, Feb. 2021. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-955.pdf.

[320] H. Xia, J. Woodruff, H. Barral, L. Esswood, et al. CheriRTOS: A Capability Model for Embedded Devices. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 92–99, 2018. doi: 10.1109/ICCD.2018.00023.

[321] H. Xia, J. Woodruff, S. Ainsworth, et al. CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52. Association for Computing Machinery, 2019. ISBN 9781450369381. doi: 10.1145/3352460.3358288. URL https://doi.org/10.1145/3352460.3358288.

[322] H. Xia et al. CheriRTOS: A capability model for embedded devices. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 92–99, Orlando, FL, USA, 2018. doi: 10.1109/ICCD.2018.00023.

[323] Y. Xia, D. Du, Z. Hua, B. Zang, H. Chen, and H. Guan. Boosting interprocess communication with architectural support. *ACM Trans. Comput.*

Syst., 39(1–4), July 2022. ISSN 0734-2071. doi: 10.1145/3532861. URL https://doi.org/10.1145/3532861.

[324] W. Xu, Y. Cai, D. He, J. Lin, and F. Zhang. Fast-lio2: Fast direct lidar-inertial odometry, 2021. URL https://arxiv.org/abs/2107.06829.

[325] Yack. PX4 Drone Autopilot — CVE List. https://cve.yack.one/products/dronecode:px4-drone-autopilot. Accessed on: 29-03-2026.

[326] G. Yanan, Z. Zhenkai, et al. GPU memory exploitation for fun and profit. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Aug. 2024. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/guo-yanan.

[327] H. Yang, H. Zhang, and R. Pan. Enabling deterministic user-level interrupts in real-time processors via hardware extension, 2026. URL https://arxiv.org/abs/2604.04015.

[328] J. You, Y. Chu, et al. Accelerating Cold Start of Thread-level Sandbox Using Snapshot and tfork . In *2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE Computer Society, Oct. 2024. doi: 10.1109/ICPADS63350.2024.00046. URL https://doi.ieeecomputersociety.org/10.1109/ICPADS63350.2024.00046.

[329] J. Z. Yu, C. Watt, et al. Capstone: A capability-based foundation for trustless secure memory access (extended version), 2023. URL https://arxiv.org/abs/2302.13863.

[330] P. Zalewski. Baking Android for ARM Morello without Morello – The Good Penguin, 2022. Retrieved from: https://www.thegoodpenguin.co.uk/blog/baking-android-for-arm-morello-without-morello/.

[331] V. Zaliva, K. Memarian, R. Almeida, et al. *CHERI C semantics as an extension of the ISO C17 standard*. University of Cambridge, 2023. doi: 10.48456/TR-988. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-988.html.

[332] V. Zaliva, K. Memarian, R. Almeida, et al. Formal mechanised semantics of cheri c: Capabilities, undefined behaviour, and provenance. In *Proceedings of the 29th ACM International Conference on ASPLOS, Volume 1*, ASPLOS '24. Association for Computing Machinery, 2024. ISBN 9798400703720. doi: 10.1145/3617232.3624859. URL https://doi.org/10.1145/3617232.3624859.

[333] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, 2019. doi: 10.1109/TVLSI.2019.2926114.

[334] ZeroDay-Labs. Cva6-cheri opensource repository. `https://github.com/zero-day-labs/cheri-cva6`, Accessed on: 30-03-2026.

[335] A. Zuepke et al. For safety and security reasons: The cost of component-isolation in IoT. *Logistics, Supply Chain, Sustainability and Global Challenges*, 7(1):41–50, 2016. doi: 10.1515/jlst-2016-0004.