

Article

Zero-Copy Messaging: Low-Latency Inter-Task Communication in CHERI-Enabled RTOS

Mina Soltani Siapoush  and Jim Alves-Foss * 

Center for Secure and Dependable Systems (CSDS), University of Idaho, Moscow, ID 83844, USA;
solt5189@vandals.uidaho.edu

* Correspondence: jimaf@uidaho.edu

Abstract

Efficient and secure inter-task communication (ITC) is critical in real-time embedded systems, particularly in security-sensitive architectures. Traditional ITC mechanisms in Real-Time Operating Systems (RTOSs) often incur high latency from kernel trapping, context-switch overhead, and multiple data copies during message passing. This paper introduces a zero-copy, capability-protected ITC framework for CHERI-enabled RTOS environments that achieves both high performance and strong compartmental isolation. The approach integrates mutexes and semaphores encapsulated as sealed capabilities, a shared memory ring buffer for messaging, and compartment-local stubs to eliminate redundant data copies and reduce cross-compartment transitions. Temporal safety is ensured through hardware-backed capability expiration, mitigating use-after-free vulnerabilities. Implemented as a reference application on the CHERI-IoT RTOS, the framework delivers up to 3× lower mutex lock latency and over 70% faster message transfers compared to baseline FreeRTOS, while preserving deterministic real-time behavior. Security evaluation confirms resilience against unauthorized access, capability leakage, and TOCTTO vulnerabilities. These results demonstrate that capability-based zero-copy ITC can be a practical and performance-optimal solution for constrained embedded systems that demand high throughput, low latency, and verifiable isolation guarantees.

Keywords: real-time operating systems (RTOSs); FreeRTOS; capability hardware-enhanced RISC instructions (CHERI); inter-task communication (ITC); zero-copy; synchronization



Academic Editor: Stefano Rinaldi

Received: 15 September 2025

Revised: 13 October 2025

Accepted: 24 October 2025

Published: 4 November 2025

Citation: Soltani Siapoush, M.; Alves-Foss, J. Zero-Copy Messaging: Low-Latency Inter-Task Communication in CHERI-Enabled RTOS. *Future Internet* **2025**, *17*, 506. <https://doi.org/10.3390/fi17110506>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Embedded systems (ESs) are pervasive in modern life, silently controlling critical functions, from traffic lights to financial transactions, and powering countless daily operations. These compact computing units prioritize performance to ensure timely and reliable execution. As embedded systems increasingly handle diverse tasks, such as processing data streams from sensors and cameras and executing complex control algorithms, network connectivity through the Internet of Things (IoT) becomes essential for seamless communication with other devices. This interconnectivity, while enabling new capabilities, also introduces additional security risks.

Designing embedded systems demands a careful balance between security and performance. Systems must resist unauthorized access while maintaining speed and reliability—a challenge that grows as system complexity increases. Traditionally, embedded systems consolidate applications and components within a shared memory space managed by a

Real-Time Operating System (RTOS) [1]. Although efficient, this approach exposes vulnerabilities like data corruption and unauthorized access. Implementing robust security measures is necessary but can increase system complexity and reduce execution predictability. Thus, balancing security and predictability is critical for reliable operation and threat protection (see Table 1).

Table 1. Trade-off between improving security and predictability. (a) Security improvement measures that negatively impact predictability. (b) Measures for improving predictability that negatively impact security.

(a)	
Measures	Predictability Failure
Schedule randomization	Increased Worst-Case Execution Time (WCET) of critical real-time tasks
Monitoring and detection	Frequent task invocations and delays
Cryptographic protection	Delaying execution of critical real-time task
(b)	
Measures	Security failure
Priority-based scheduling	Unauthorized access to high-priority tasks
Lowering task frequency or priority	Denial-of-Service (DoS) attacks by resource starvation
Reducing the complexity of computation	Side-channel attacks exploiting timing variations

A key mechanism in any OS is inter-task communication (ITC), which significantly impacts system response time. Embedded system tasks often interact across domain boundaries, requiring ITC to maintain low latency and high security. ITC is crucial not only in monolithic systems with multiple user-level services but also in architectures such as Android’s Binder [2], where applications communicate with system services for tasks like rendering through the window manager. Unfortunately, current ITC mechanisms still incur high latency in both microkernel and monolithic kernel designs. Previous work has proposed various software and hardware optimizations [3–6], but most software solutions suffer from kernel trapping overhead and multiple data copies during message passing.

2. Motivation

Recognizing key limitations in the current RTOS communication model, this section outlines the requirements for developing a lightweight, secure, and extensible ITC solution. For this work, we use FreeRTOS as a representative RTOS, which is described in detail in Section 4.

2.1. Fast and Secure Inter-Task Communication

Efficient and secure ITC is a critical component in ES. Communication between tasks can be achieved through various methods: message passing, shared memory, and signaling mechanisms, each offering distinct advantages and challenges.

Message passing, often implemented through synchronous RTOS functions such as *MsgReply()* and *MsgError()*, facilitates task coordination but can incur overhead due to message copying between tasks. Zero-copy messaging techniques minimize CPU involvement in data transfers, improving performance and reducing latency; however, they introduce complexities such as the need for robust synchronization to avoid race conditions, careful buffer management to prevent data corruption, and strict memory access controls to secure data. Shared memory allows asynchronous, high-speed access to common memory regions, eliminating message passing overhead and increasing efficiency. However, this approach is vulnerable to security risks like Time-Of-Check-To-Time-Of-Use (TOCTTOU) attacks and adds performance overhead through costly Translation Lookaside Buffer (TLB) management operations during virtual-to-physical memory address translations, which can

degrade system responsiveness. Signaling mechanisms, such as event flags and task flags, provide lightweight communication to notify tasks of specific events, enabling efficient synchronization with minimal overhead. Although effective for signaling occurrences, these methods are limited to simple event notifications rather than complex data, and improper management can lead to synchronization issues such as race conditions. Together, these communication paradigms illustrate the trade-offs between performance, security, and complexity that drive the need for enhanced ITC frameworks for secure and efficient operation in resource-constrained embedded environments.

2.2. Compliance and Support of Memory Isolation

Most RTOS solutions, including FreeRTOS, effectively manage access to large code and data regions. However, protecting smaller, more specific elements often requires explicit checks. FreeRTOS lacks inherent memory protection and strict isolation between tasks. Consequently, tasks may inadvertently or maliciously access and modify each other's memory, risking security breaches and system instability. Developers must manually enforce memory boundaries, increasing complexity and the potential for errors.

To address these challenges, architectures must support fine-grained, cost-effective memory protection that minimizes performance overhead. Such mechanisms should be intuitive to implement, reducing developer burden while maintaining system responsiveness. Without these protections, systems remain vulnerable to data leaks, resource conflicts, and task interference, jeopardizing reliability.

2.3. Compliance and Support for Task Isolation

Task isolation is fundamental to RTOS security, ensuring tasks operate within well-defined domains without mutual interference. Effective isolation protects sensitive data, prevents unauthorized access, and mitigates risks of system-wide failures.

FreeRTOS employs priority-driven scheduling where higher-priority tasks preempt lower-priority ones, enabling prompt responses to critical events. However, FreeRTOS lacks inherent task isolation, allowing tasks to potentially access resources or data belonging to others. This can cause resource contention, data corruption, and unintended interference, especially in complex or dynamic workloads. Robust task isolation demands careful architectural design and prioritization to maintain system stability and protect sensitive operations.

2.4. Scalability/Extensibility

As embedded systems grow more complex, scalability and extensibility become essential. These principles ensure RTOSs can adapt to increasing demands—more tasks, evolving requirements, and expanded functionality. FreeRTOS's lightweight, modular design supports extensions and customization. Yet, traditional scalability approaches, such as static memory partitioning via Memory Protection Units (MPUs), impose limitations. Static partitioning struggles to handle increasing task counts or dynamically shifting memory needs, constraining system scalability and flexibility.

To overcome these limitations, future designs should embrace dynamic memory allocation and flexible task management, enabling RTOSs to evolve with applications and meet the demands of modern interconnected devices.

3. Contribution

Building on the challenges and objectives outlined above, this paper makes the following key contributions:

- We conduct a comprehensive review of recent communication techniques for embedded systems, with a particular focus on limitations in current inter-task communication methods within CHERI-enabled RTOS environments.
- We propose a novel zero-copy communication mechanism that integrates synchronization primitives such as mutexes and semaphores with CHERI capabilities. This design enhances both performance and security by eliminating redundant data copying, maintaining memory isolation, and protecting access to critical synchronization data.
- We implement and evaluate the proposed mechanism on a CHERI-enabled RTOS, benchmarking it against traditional communication methods. Our experimental results demonstrate significant improvements in efficiency and security, validating the practical benefits of our design.

The rest of this paper is organized as follows: Section 4 provides preliminary information on system calls in RTOS, serving as background for understanding the proposed framework. Section 5 reviews related studies in this field. Section 6 introduces our proposed model, outlines the design requirements, and details how we developed a secure solution to address existing challenges. Section 7 presents the implementation and evaluation of the proposed model, including a comprehensive comparison with existing approaches. Section 8 discusses the implications of the results, highlights the strengths and limitations of the proposed framework, and explores potential improvements. Finally, Section 9 concludes the paper by summarizing our findings and suggesting directions for future research.

4. Background

This section discusses fundamental concepts underlying the proposed work. FreeRTOS is used as a representative example, as it exemplifies modern RTOS design principles.

4.1. FreeRTOS

To manage embedded applications with strict timing and reliability requirements, RTOSs are commercially developed to provide a high degree of determinism and predictability for executing tasks in real-time. As an open-source and well-known RTOS, FreeRTOS [7,8] was created by Richard Barry and first released in 2003 under the MIT license. As shown in Figure 1, it employs a layered architecture that separates privileged kernel functions from the unprivileged layer and hardware abstraction (HAL) to provide portability across microcontrollers. The unprivileged layer hosts user applications, libraries, and drivers, interfacing with the kernel via well-defined APIs. The privileged layer forms the kernel core, managing task scheduling, synchronization, inter-task communication, and dynamic memory allocation, with kernel objects like tasks, queues, and semaphores allocated at runtime to optimize resource use. At the lowest level, the HAL abstracts hardware-specific details, enabling portability across various microcontrollers by handling low-level tasks such as interrupt management, context switching, and timer control. FreeRTOS offers a variety of services tailored to the constraints and requirements of embedded systems, including dynamic memory allocation, task management, inter-task communication, and resource sharing. These services ensure efficient memory use, deterministic task execution, seamless communication, and effective resource management among tasks.

4.2. Why FreeRTOS?

We have chosen FreeRTOS as the foundational OS framework to meet our specific requirements and achieve our objectives. The following summarizes the key reasons for this choice.

FreeRTOS is widely favored by developers in the embedded systems domain due to its distinct advantages over other operating systems. First, as an open-source platform, FreeR-

TOS offers unrestricted access to its source code, allowing extensive customization and fine-tuning to suit specific project needs. Its efficiency—characterized by a small footprint and low overhead—enables it to operate on microcontrollers with limited resources, ensuring optimal resource utilization. Furthermore, FreeRTOS’s real-time scheduling guarantees deterministic task execution, satisfying the stringent demands of real-time applications.

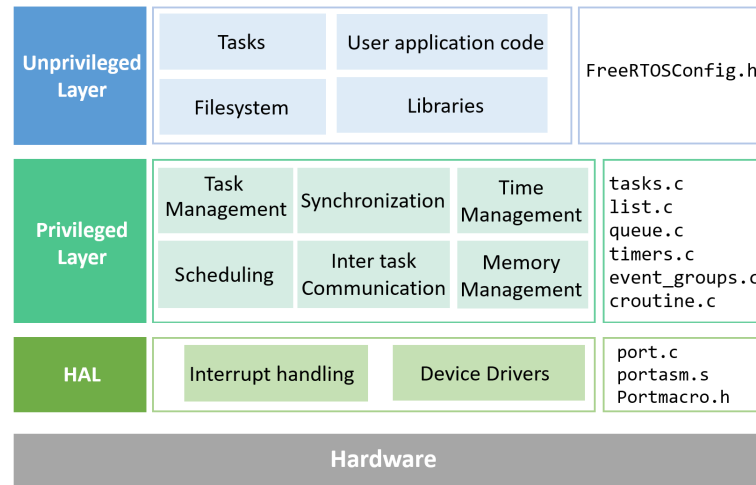


Figure 1. Architecture of FreeRTOS.

FreeRTOS also provides a comprehensive ecosystem of components and libraries for communication, synchronization, and memory management, which simplifies development and system integration. One notable tool is FreeRTOS+Trace [9], developed by Percepio AB, offering trace and debug capabilities for FreeRTOS-based applications. This tool delivers detailed insights into system behavior through an intuitive graphical interface, facilitating performance analysis and aiding in identifying security issues such as unauthorized access and privileged operations.

Additionally, FreeRTOS supports a wide range of hardware platforms, ensuring seamless portability across diverse architectures [10]. For comparison, Table 2 summarizes key features of FreeRTOS alongside other RTOSs, listed alphabetically by OS name.

Table 2. Summary of key features of common Embedded OS.

OS	Architecture	Scheduler	Programming	Language
ChibiOS [11]	Microkernel	Preemptive, cooperative	Threads, mutexes, semaphores	C
Contiki [12]	Modular	Preemptive FIFO	Multithreading	C
EmbOS [2]	Monolithic	Priority-based	Tasks, semaphores, message queues	C, C++
FreeRTOS [7]	Microkernel	Preemptive priority-based	Threads, mutexes, semaphores	C, Assembly
LiteOS [13]	Modular	Preemptive priority-based	Multithreading	LiteC
MbedOS [14]	Modular	Priority-based	Event-driven	C, C++
Mynewt [15]	Modular	Preemptive priority	Multithreading	C, Go, C++
NuttX [16]	Monolithic	Preemptive priority-based	POSIX-like with message passing support	C, C++
RIOT [17]	Microkernel	Preemptive priority-based	Multithreading	C
RTEMS [18]	Monolithic	Preemptive Priority-based	Tasks Periods, and Signals	C, Ada
SeL4 [19]	Microkernel	Deterministic, priority-based	Capability-based	C
ThreadX [20]	Microkernel	Preemptive	Threads, mutexes, semaphores	C
VxWorks [21]	Microkernel	Priority-based	Tasks, semaphores, message queues	C, Ada, Assembly
Zephyr [22]	Monolithic	Preemptive	Threads, mutexes, semaphores	C
μC/OS [23]	Microkernel	Preemptive	Tasks, semaphores, message queues	C
uClinux [24]	Monolithic	Preemptive priority-based	Multithreading	C, Assembly
TinyOS [25]	Monolithic	Non-preemptive FIFO	Event-driven	NesC
Raspbian [26]	Monolithic	Real-time preemptive	Multithreading	Python
QNX [27]	Microkernel	Preemptive	Processes, message passing	C, C++
NucleusRTOS [28]	Microkernel	Priority-based	Tasks, semaphores, message queues	C, C++

5. Related Work

Extensive research has been conducted on ITC in RTOS, focusing on enhancing synchronization and messaging mechanisms to efficiently coordinate and transfer data between concurrent tasks. RTOS designs generally prioritize either high performance through speed or enhanced system security. RIOT [17], an open-source OS, is designed with a minimalist kernel that aims to reduce code duplication. Its architecture emphasizes compliance with open network and system standards, unified APIs across supported hardware, and real-time capabilities. However, RIOT's lightweight design introduces certain security trade-offs. Its prioritization of code efficiency and minimalism may result in limited implementation of security mechanisms such as access control and memory protection, making it a less optimal choice for security-sensitive IoT applications.

To address the needs of regulated domains, WITTENSTEIN, a German technology firm, developed SafeRTOS [29], a safety-certified extension of the FreeRTOS kernel [30]. SafeRTOS incorporates critical safety features such as memory protection, task isolation, and stack overflow detection, and is designed to comply with stringent industry standards, including DO-178C for aerospace and ISO 26262 for automotive applications. It further supports automated code generation, formal verification, and provides certification artifacts to streamline compliance and integration into safety-critical systems.

Similarly, Wang et al. [31] proposed a method for securing RTOS architectures in building automation systems through a policy-driven approach. Their design employs a formal specification language to define a security policy regulating communication between processes. This policy is compiled into an Access Control Table (ACT) stored within the kernel's address space, which the microkernel references to validate ITC permissions. The policy also incorporates process-level metadata, such as worst-case execution times, which the kernel monitors at runtime to ensure compliance. This combination of formal specification, runtime validation, and process-level constraints enhances both reliability and security in distributed embedded environments.

A concept closely aligned with such policy-based controls is the use of capabilities [32,33], unforgeable tokens, or keys that reference resources (e.g., memory or I/O) and specify the associated permissions. Capabilities enforce the principle of least privilege by granting each component only the access necessary for its function. They may be transferred between principals and serve as both identifiers and access controllers, enabling secure delegation and controlled sharing of resources. Capability-based designs have been explored extensively in operating systems, ranging from software-based implementations such as Hydra [34], seL4 [19], KeyKOS [35], Eros [36], and Fiasco [37], to hardware-assisted approaches such as CHERI [38].

In practice, capabilities can restrict access through three primary mechanisms: direct access, delegation, and invocation. Direct access allows a capability to serve as a token authorizing specific operations on a resource. Delegation enables one entity to derive a new capability from an existing one, granting another entity a subset of privileges [39]. Invocation involves presenting a capability to a resource, triggering a verification process that validates the entire chain of transfers before permitting the requested operation.

Recent work by Xia et al. [40] presents a detailed performance study of inter-process communication (IPC) in the seL4 microkernel. While seL4 offers unmatched assurance for critical security-sensitive systems through formal verification and strong security policies, its IPC latency is higher due to the overhead of kernel transitions and meticulous enforcement mechanisms. In contrast, this work aims to reduce such overhead by leveraging a lightweight CHERI-enabled RTOS design that uses hardware-backed temporal safety to balance high performance with strong compartmental isolation.

6. Proposed Model

In multitasking environments, tasks must coordinate activities, share resources, and synchronize execution to achieve correct and timely behavior. This can be realized through mechanisms such as message passing, shared memory, and cross-domain function calls. Two fundamental requirements for effective ITC are: low latency, to meet real-time performance requirements, and strong security, to prevent data corruption or privilege escalation. Traditional RTOS designs often face trade-offs between these objectives, favoring either performance or protection. CHERI-based systems, however, enable secure cross-compartment interactions that combine high performance with fine-grained, hardware-enforced memory safety.

6.1. Synchronization Primitives Protected by Capabilities

The proposed model leverages CHERI capabilities to enforce strict, compartment-level access control over synchronization primitives. We employ mutexes and semaphores as the core synchronization mechanisms for cross-compartment interactions. Mutexes provide exclusive write access to shared memory, ensuring that only one sender can modify data at a time. Moreover, semaphores enable concurrent read access, allowing multiple receivers to retrieve data safely while avoiding race conditions.

Interaction with these primitives requires possession of sealed capabilities that explicitly authorize specific operations. CHERI's shared library abstraction encapsulates mutex and semaphore implementations in a reusable, compartment-shared codebase, avoiding code duplication while maintaining strict isolation. Functions execute in the caller's context without exposing mutable global variables, reducing the attack surface. This capability-protected synchronization framework is integrated into a low-latency communication model that eliminates redundant data copying while enforcing hardware-backed access control. The design consists of three main components: (a) *capability-protected mutexes and semaphores* to enforce least-privilege access and ensure reliable synchronization, (b) *shared memory message passing with temporal bounds* to achieve zero-copy communication while mitigating race conditions, and (c) *compartment-local stubs* to reduce the cost of system calls for inter-compartment communication.

The proposed framework follows the sequence of operations, as shown in Figure 2 and described below:

1. **Capability Creation for Mutexes/Semaphores:** When a communication channel is created, the Channel Manager Compartment (CMC) allocates memory for its associated synchronization primitives (mutexes and semaphores) from a secure heap. These primitives use CheriOT *futexes* (Fast Userspace muTEX) for atomic locking and signaling. For each primitive:
 - A corresponding capability (*MutexCap* or *SemaCap*) is generated.
 - The capability points to the primitive's state structure in memory, with CHERI hardware enforcing spatial bounds.
2. **Sealing Mutex/Semaphore Capabilities:** Before distributing the capabilities to compartments, the CMC seals them using CHERI's *CSeal* instruction with unique types (*MUTEX_SEAL_TYPE* and *SEMAPHORE_SEAL_TYPE*) ensuring that:
 - Only authorized compartments can unseal and use them.
 - Capabilities cannot be forged or modified by untrusted code.
3. **Locking/Unlocking Mutexes:** Compartments invoke mutex operations either through cross-compartment calls (e.g., *CCall*) or compartment-local stubs:
 - The sealed *MutexCap* is passed as an argument.
 - Inside the call, the capability is unsealed using *CUnseal*.

- Atomic futex operations check and update the lock state.
 - Successful lock acquisition records the owning compartment ID; failed attempts trigger `futex_wait` until the mutex becomes available.
4. **Signaling/Waiting on Semaphores:** Semaphores allow controlled concurrent access while maintaining safe limits on resource usage. Futex-semaphore operations are implemented using futexes. The `futex_wait` function is a key part and allows a compartment to block without constantly spinning, reducing CPU usage. The compartment will be woken when the futex value changes. The `futex_wake` function signals to the scheduler that waiting compartments should be woken up. This is more efficient than constantly checking the lock status.

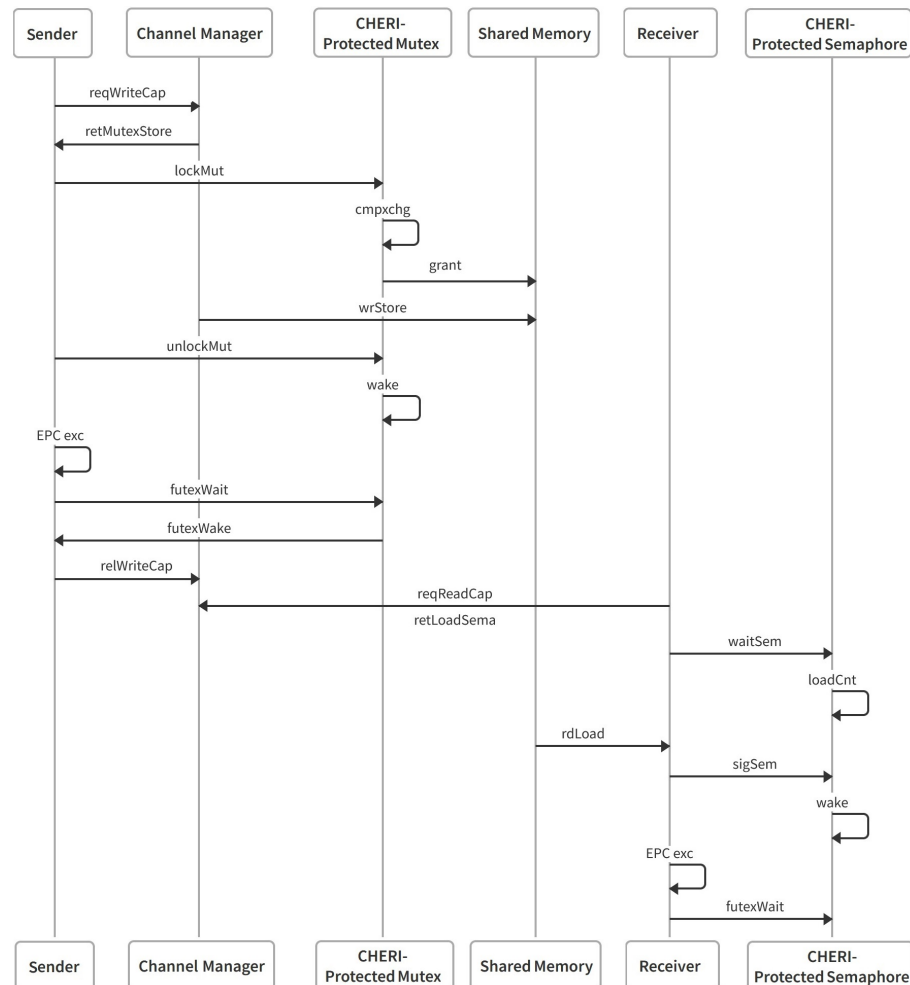


Figure 2. Sequence diagram of the proposed cross-compartment communication.

6.2. Zero-Copy Message Passing Using Shared Memory

The model introduces a zero-copy message-passing mechanism built on CHERIoT's capability-based shared memory architecture. A statically allocated region is partitioned into fixed-size message slots within a ring buffer. *Senders* receive store-only capabilities, granting write access to free slots. *Receivers* receive load-only capabilities, granting read access to occupied slots.

CHERI's temporal safety further enhances security by automatically invalidating capabilities after a configurable timeout, preventing use-after-free attacks. A trusted channel manager oversees channel creation, allocates ring buffer slots, and issues sealed capabilities to participating compartments. Data transfer occurs entirely within the shared memory region, with CHERI hardware enforcing bounds checks, eliminating extra copies, and mini-

mizing latency. A shared library abstraction provides producer–consumer counter management routines, reducing cross-compartment calls and leveraging CHERI’s lightweight synchronization primitives for optimal throughput, as shown in Figure 3.

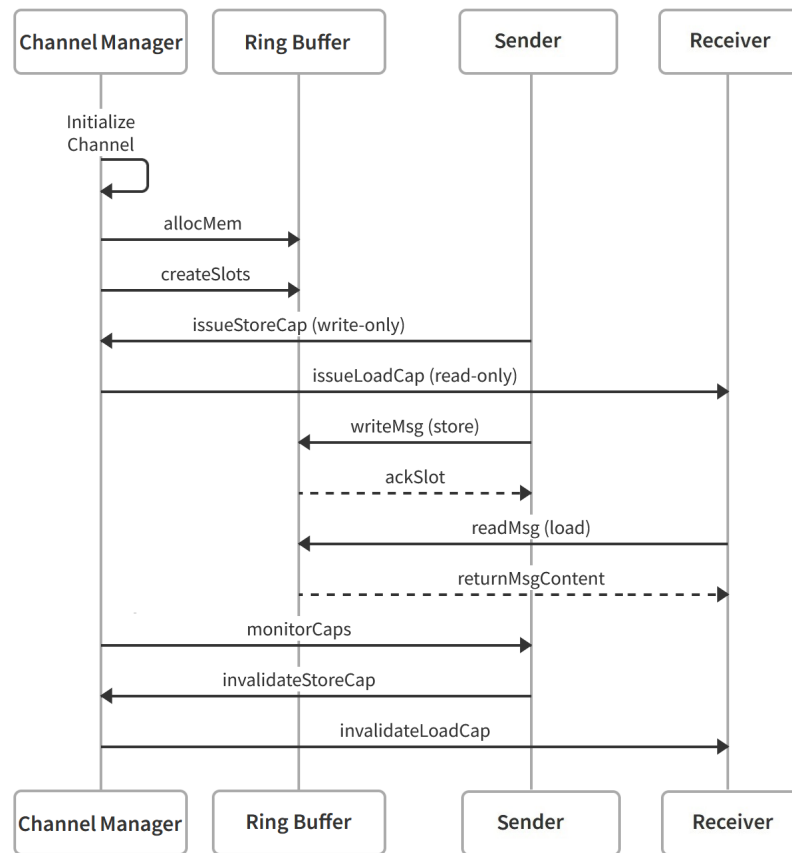


Figure 3. Proposed zero-copy message-passing mechanism.

6.3. Optimized System Call Using Compartment-Local Stubs

Traditional ITC methods in capability-aware systems often depend heavily on cross-compartment calls. These calls incur substantial overhead due to context switches, capability validation, and stack sealing/unsealing operations. To overcome this inefficiency, our approach employs lightweight, verified code segments embedded within each compartment. These stubs mediate access to shared memory through pre-delegated capabilities, allowing compartments to communicate directly without triggering repeated transitions through a trusted intermediary. This design ensures that CHERI’s spatial and temporal memory safety properties are maintained while significantly improving runtime performance.

The system’s operation is divided into two primary phases: (a) initialization and (b) runtime.

- **Initialization:** A trusted compartment allocates shared memory and derives restricted capabilities by applying CHERI-provided operations such as `cheri_bounds_set()` for limiting memory access ranges, `cheri_perms_and()` to enforce least-privilege access by stripping global permissions, and `cheri_seal()` to bind capabilities to specific compartments using unique type tags, as shown in Listing 1. These capabilities are then distributed to each compartment alongside their respective local stubs.
- **Runtime:** Each compartment invokes its local stub to access shared resources. The stub handles unsealing (`cheri_unseal()`), enforces access bounds and permissions, and manages synchronization via lightweight, capability-based locking primitives (`lock_guard_t`), as shown in Listing 2. Because these operations occur entirely within

the compartment, no system calls or context switches are required during routine communication. The runtime flow of the stub-based model can be summarized as follows:

- *Message Invocation*: A sending compartment calls its designated stub instead of directly invoking the target compartment’s function.
- *Stub Mediation*: The stub applies compartment-specific sealing to encapsulate the message and associated capabilities, preventing unauthorized access or capability leakage.
- *Capability Validation*: Minimal capability checks are performed at the stub level, reducing cross-compartment transitions by two to three per communication event.
- *Target Dispatch*: The sealed message is passed to the target compartment, where it is unsealed and processed.
- *Response Handling*: For reply messages, the reverse path is followed, again through the stub, maintaining symmetry and isolation.

Listing 1. One-Time cross-compartment call (initialization).

```

1 void *buffer = heap_allocate(SHARED_HEAP, 4096);
2 void *restricted_cap = cheri_perms_and(buffer, PERM_LOAD |
    PERM_STORE_LOCAL);
3 // Set bounds to 4096 bytes
4 restricted_cap = cheri_bounds_set(restricted_cap, 4096);
5 // Seal the restricted capability
6 void *sealed_cap = cheri_seal(restricted_cap, COMPARTMENT_STUB_TYPE)
    ;

```

This structured flow not only reduces cross-compartment transitions but also maintains fine-grained capability enforcement at each step, ensuring that security and isolation guarantees are preserved while achieving substantial performance gains.

Listing 2. Compartment-local stub phase (runtime).

```

1 void write_packet(const void *data, size_t len) {
2     // In-compartment
3     void *cap = cheri_unseal(sealed_cap, COMPARTMENT_STUB_TYPE);
4     // Acquire atomic lock
5     lock_guard_t guard = lock_acquire(&buffer_lock);
6     // Bounded copy
7     cheri_memcpy(cap, data, len);
8 }

```

7. Implementation and Evaluation

Our proposed model was implemented directly as a reference application within the CHERIoT research platform, building atop the native infrastructure provided by the cheriot-rtos [41]. Unlike traditional porting efforts, this work leverages CHERIoT’s base environment, integrating the design as a new example, under the repository’s *examples* directory. The implementation utilizes three primary software components: the CHERI-LLVM compiler toolchain, the dynamic linking library (*libdl*), and FreeRTOS. Compilation is handled using CHERIoT-clang, which is part of the extended CHERI-LLVM suite and enforces fine-grained capability security. The underlying ISA is *CHERIoT-Ibex*, which is a capability-aware RISC-V core engineered for resource-constrained embedded systems. Firmware artifacts are managed using *xmake*, a lightweight, cross-platform build system, configured for the CHERIoT development workflow, as listed in Table 3.

Table 3. Core software components used in the implementation.

Component	Tool
Compiler	CHERIOT-clang
ISA	CHERIOT-Ibex
Build System	xmake

Experimental Setup

We implemented the proposed model and evaluated its performance against a baseline FreeRTOS configuration. The primary objective of the experiments is to validate the efficiency and security of the communication mechanism between compartments leveraging CHERI capabilities. The core security and performance properties of the design are as follows:

- **Compartment Isolation:** Synchronization primitives are distributed as sealed capabilities, ensuring that only explicitly authorized compartments can access them.
- **Ownership Provenance:** Mutex ownership is cryptographically bound to CHERIOT Compartment IDs, eliminating the risk of unauthorized unlocking of shared resources.
- **Temporal Safety:** Capabilities incorporate expiration metadata and are automatically revoked upon invalidation, mitigating use-after-free and stale reference attacks.
- **Low Latency and Zero-Copy:** A ring buffer mechanism enables direct data transfer between compartments, eliminating redundant memory copies and reducing CPU cycles for message passing.

To quantitatively assess efficiency, we measured execution cycles for each operation using the RISC-V `rdcyc1e64()` instruction, which accesses the processor's 64-bit hardware cycle counter directly. This low-level approach provides deterministic and accurate timing, unaffected by OS scheduling or instrumentation overhead. Measurements were conducted on a CHERIOT-compatible platform running at a fixed clock frequency of 50 MHz. Each operation was executed 1000 times under controlled, uncontended conditions to ensure repeatability and mitigate transient effects such as cache warm-up, instruction prefetching, and branch prediction. The resulting cycle counts were averaged to obtain statistically meaningful results with minimal noise.

Table 4 presents a comparative analysis of execution cycles for key synchronization and communication operations across three configurations: baseline FreeRTOS and the proposed low-latency ITC model. The data reveals significant performance improvements in the proposed system, particularly for latency-sensitive operations like mutex locking and message transfer. These gains stem from CHERIOT's hardware-enforced capability model, which eliminates reliance on software shadow tables, runtime checks, or garbage collection. By enabling deterministic capability access, our system achieves synchronization latencies as low as 8 cycles, facilitating near real-time inter-compartment coordination without compromising memory safety.

Figure 4 further illustrates these performance trends by plotting execution cycle counts for key operations. The proposed ITC model consistently outperforms baseline FreeRTOS, delivering up to a threefold reduction in mutex lock latency and over 70% improvement in message transfer efficiency. Notably, while message transfer latency scales linearly with message size, the proposed model maintains substantially lower per-byte overhead compared to traditional implementations. This linear scaling, combined with a lightweight synchronization design, ensures sustained high performance even for larger messages.

Complementing the performance evaluation, Table 5 summarizes the security properties guaranteed by each core ITC operation. For example, mutex lock operations benefit from hardware-enforced bounds checking, while semaphore synchronization leverages

capability sealing to prevent TOCTTO (Time-Of-Check-To-Time-Of-Use) vulnerabilities. Message transfers employ a zero-copy mechanism secured by temporal safety guarantees, ensuring both performance and memory correctness. Additionally, context switching overhead is minimized through execution confinement, enabling compartment transitions without invoking the global scheduler, thereby enhancing both security isolation and system responsiveness.

Table 4. Execution cycles for key synchronization and communication primitives.

Operation	FreeRTOS	Proposed ITC
Mutex Lock	24	8
Semaphore Wait	35	12
64B Message Transfer	900	250
Context Switches	1200	6

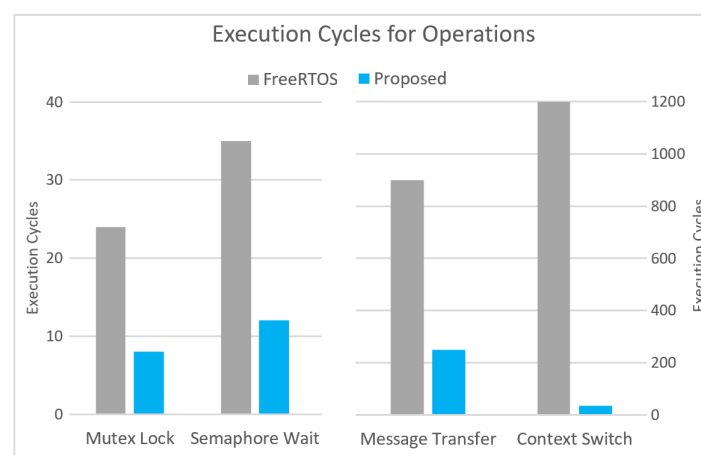


Figure 4. Execution cycles for key synchronization and communication primitives.

Table 5. Security advantages of proposed low-latency ITC operations.

Operation	Security Advantage
Mutex Lock	Hardware-enforced bounds checks
Semaphore Wait	Capability sealing prevents TOCTOU
64B Message Transfer	Zero-copy with temporal safety
Context Switches	Compartment execution avoids switching

Evaluation of the stub-based cross-compartment communication model showed notable performance gains. It significantly reduces latency for small messages (e.g., 64 bytes), as shown in Table 6, while also decreasing the number of required security checks (Figure 5). From a security standpoint, compartment-specific sealing in the stub-based model not only prevents unauthorized access but also minimizes capability leakage. The reduced trusted computing base (TCB) and small stub footprint (200 bytes) make the approach more maintainable and amenable to formal verification compared to auditing entire compartments.

Table 6. Security overhead for naive and stub-based ITC.

Approach	Cycles/Packet	Security Checks
Naive cross-compartment	1300–1500	4 capability checks
Local stub	350–450	2 capability checks

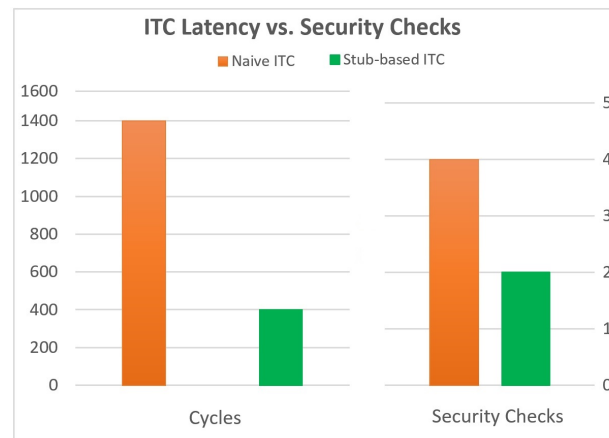


Figure 5. Comparison of ITC latency and security checks.

Beyond quantitative gains, the modularity introduced by stubs supports clear separation of concerns: each stub mediates only the necessary interactions, simplifying protocol updates without affecting unrelated components. Stress testing under varying message sizes and compartment loads further confirmed that the approach scales effectively while preserving low latency, making it suitable for real-world systems requiring both high performance and strong isolation guarantees.

8. Discussion

The proposed communication framework is grounded in the formal architectural features of CHERI capabilities, which provide hardware-backed enforcement of spatial and temporal memory safety. Capability sealing and unsealing operations, along with temporal bounds on capability validity, collectively establish fundamental guarantees of access control and use-after-free prevention. This design rationale rests on low-level hardware primitives rather than abstract mathematical models, ensuring reliable enforcement of synchronization and protection at runtime. Although formal models expressing these properties mathematically could complement this work, the architectural formalization inherent in CHERI’s capability hardware suffices to justify the theoretical soundness of our approach. The tight coupling of hardware-enforced capability semantics with zero-copy communication primitives enables both efficient and secure inter-task communication in embedded systems.

To situate our performance results within the broader landscape of capability-based IPC mechanisms, we consider recent benchmarks from seL4. Prior work by Xia et al. [40] reports that small message transfers on RISC-V platforms using seL4 can require several thousand CPU cycles due to its strong emphasis on formal verification and comprehensive security enforcement, which introduces higher latency from kernel transitions and policy checks. In contrast, our CHERI-enabled RTOS adopts a lightweight zero-copy communication model designed specifically for resource-constrained embedded systems with real-time constraints, enabling significantly lower synchronization and message transfer latency. Given the architectural and execution environment differences, direct benchmarking of our approach on seL4 remains an open challenge. We focus our experimental comparison with FreeRTOS to demonstrate practical performance and security improvements achievable in this widely used baseline RTOS. Future work includes exploring integration and benchmarking with formally verified systems and extending evaluations across diverse hardware platforms.

9. Conclusions

This paper introduced a zero-copy, capability-secured inter-compartment communication model for CHERI-enabled RTOS environments, addressing the long-standing trade-off between performance and security in embedded systems. By leveraging CHERI's hardware-enforced capabilities, the design enforces strict compartmental isolation for synchronization primitives and message buffers, integrates temporal safety controls, and minimizes runtime overhead through compartment-local stubs. Implementation within the CHERIoT RTOS demonstrated that the approach can be realized natively without modifying the base system, making it directly applicable to other CHERI-compatible platforms. Empirical results highlight substantial latency reductions, while sustaining strong protection against common memory safety violations and privilege escalation attempts. The findings confirm that capability-based zero-copy ITC is well-suited for performance-sensitive and security-critical embedded workloads. Future work will explore (a) extending the model to support distributed CHERI nodes in IoT networks, (b) formal verification of the stub logic and ring buffer design, and (c) adaptive flow-control strategies for unpredictable real-time communication patterns. These directions aim to further advance the integration of hardware capability systems into mainstream embedded and IoT deployments, enabling secure, high-speed communication at scale.

10. Future Direction

This research lays a robust foundation for developing resilient and adaptive embedded systems; however, the rapidly evolving landscape of cyber threats and system complexities necessitates continuous innovation. Future directions include advancing from component-level formal methods to end-to-end verification by applying frameworks like VeriCHERI not only to buffers and primitives but also to communication scheduling, protocol logic, and side-channel resilience, thereby enabling exhaustive proofs of safety, liveness, and security properties. Another promising avenue is the integration of TinyML-powered prediction engines for adaptive flow control and scheduling, allowing channel parameters and scheduling to be tuned intelligently in real time to optimize latency, throughput, and energy efficiency in mixed-criticality workloads. Complementing this, the development of zero-trust secure communication protocols tailored for resource-constrained CHERI RTOS environments (such as streamlined DTLS or AES-GCM variants) can provide dynamic, context-aware capability management, continually revoking or restricting access based on real-time threat analysis and anomaly detection across local and distributed tasks. To empower developers, future work should also focus on toolchain integration, including compiler passes, debugging APIs, and visualization tools that automate capability management, support provenance tracking, and even leverage AR-powered insights into inter-task communication and buffer utilization, thereby making security and performance considerations transparent. Finally, engineering self-healing and forensically traceable ITC channels that autonomously detect, localize, and recover from failures, leaks, and attacks through secure hardware-backed provenance tags, live compartment migration, and redundancy will ensure robust and auditable operation in safety-critical embedded networks.

Author Contributions: Conceptualization, M.S.S. and J.A.-F.; methodology, M.S.S. and J.A.-F.; software, M.S.S.; validation, M.S.S. and J.A.-F.; formal analysis, M.S.S. and J.A.-F.; investigation, M.S.S.; resources, M.S.S.; data curation, M.S.S.; writing—original draft preparation, M.S.S.; writing—review and editing, M.S.S. and J.A.-F.; visualization, M.S.S.; supervision, J.A.-F.; project administration, J.A.-F.; funding acquisition, J.A.-F. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by Schweitzer Engineering Laboratories (SEL).

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ACT	Access Control Table
API	Application Programming Interface
CMC	Channel Manager Compartment
DoS	Denial-of-Service
ES	Embedded System
Futex	Fast Userspace Mutex
HAL	Hardware Abstraction Layer
IoT	Internet of Things
ISRs	Interrupt Service Routines
ITC	Inter-Task Communication
MPUs	Memory Protection Units
RTOS	Real-Time Operating System
TCB	Trusted Computing Base
TLB	Translation Lookaside Buffer
TOCTTOU	Time-Of-Check-To-Time-Of-Use
WCET	Worst-Case Execution Time

References

1. Murti, K.C.S. Security in embedded systems. In *Design Principles for Embedded Systems*; Springer: Singapore, 2022; pp. 419–441.
2. SEGGER. embOS RTOS. 2023. Available online: <https://www.segger.com/products/rtos/embos/> (accessed on 15 July 2025).
3. Witchel, E.; Cates, J.; Asanović, K. Mondrian memory protection. In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and OS, San Jose, CA, USA, 5–9 October 2002; pp. 304–316.
4. Navarro, J.; Iyer, S.; Druschel, P.; Cox, A. Practical, transparent operating system support for superpages. *ACM SIGOPS Oper. Syst. Rev.* **2002**, *36*, 89–104. [CrossRef]
5. Gamsa, B.; Krieger, O.; Appavoo, J.; Stumm, M. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In Proceedings of the Symposium on Operating Systems Design and Implementation, New Orleans, LA, USA, 2–25 February 1999; pp. 87–100.
6. Levy, A.; Campbell, B.; Ghena, B.; Giffin, D.B.; Pannuto, P.; Dutta, P.; Levis, P. Multiprogramming a 64kb computer safely and efficiently. In Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, 28 October 2017; pp. 234–251.
7. FreeRTOS. 2023. Available online: <https://github.com/FreeRTOS/FreeRTOS> (accessed on 1 June 2025).
8. FreeRTOS. Version 10. 2023. Available online: <https://www.freertos.org/Documentation/04-Roadmap-and-release-note/02-Release-notes/03-FreeRTOS-V10> (accessed on 10 October 2025).
9. FreeRTOS Plus. FreeRTOS+Trace. 2023. Available online: https://www.freertos.org/Documentation/03-Libraries/02-FreeRTOS-plus/05-FreeRTOS_plus_Trace/00-FreeRTOS_Plus_Trace (accessed on 10 October 2025).
10. Mastering the FreeRTOS Kernel: A Hands-On Tutorial Guide. 2023. Available online: https://www.freertos.org/media/2018/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf (accessed on 10 October 2025).
11. ChibiOS RTOS. Available online: <https://www.chibios.org> (accessed on 15 July 2025).
12. Contiki. Open Source OS for the Internet of Things. Available online: <https://github.com/contiki-os/contiki> (accessed on 15 July 2025).
13. LiteOS. Lightweight OS for IoT Devices. Available online: <https://github.com/LiteOS/LiteOS> (accessed on 15 July 2025).
14. Mbed OS: Embedded Operating System for IoT Devices. 2022. Available online: <https://os.mbed.com/> (accessed on 15 July 2025).
15. Mynewt. RTOS for Embedded IoT Applications. Available online: <https://github.com/apache/mynewt-core> (accessed on 15 July 2025).
16. Apache NuttX. NuttX. Available online: <https://nuttx.apache.org/> (accessed on 15 July 2025).
17. RIOT. An Open Source OS for IoT. 2023. Available online: <https://github.com/RIOT-OS/RIOT> (accessed on 15 July 2025).
18. Real-Time Executive for Multiprocessor Systems. 2023. Available online: <https://www.rtems.org/> (accessed on 15 July 2025).

19. The seL4 Microkernel. Available online: <https://sel4.systems/> (accessed on 15 July 2025).
20. Microsoft Azure. ThreadX RTOS Overview. Available online: <https://learn.microsoft.com/en-us/azure/rtos/threadx/overview-threadx> (accessed on 15 July 2025).
21. Wind River Systems. VxWorks Product Overview. Available online: <https://www.windriver.com/resource/vxworks-product-overview> (accessed on 15 July 2025).
22. The Linux Foundation. Zephyr RTOS. Available online: <https://www.zephyrproject.org/> (accessed on 15 July 2025).
23. Weston Embedded Solutions. μ C/OS. Available online: <https://github.com/weston-embedded> (accessed on 15 July 2025).
24. uClinux. Embedded OS Adapted for the leanXcam. Available online: <https://github.com/scs/uclinux/> (accessed on 15 July 2025).
25. TinyOS. OS for Embedded and Wireless Devices. Available online: <https://github.com/tinyos/tinyos-main/> (accessed on 15 July 2025).
26. Raspbian. Free Operating System Based on Debian. Available online: <https://www.raspberrypi.com/software/operating-systems/> (accessed on 15 July 2025).
27. BlackBerry QNX. RTOS and Software for Embedded Systems. 2023. Available online: <https://blackberry.qnx.com/en> (accessed on 15 July 2025).
28. Nucleus RTOS by Siemens. 2023. Available online: <https://resources.sw.siemens.com/en-US/fact-sheet-nucleus-rtos/> (accessed on 10 October 2025).
29. High Integrity Systems. Building on FreeRTOS for Safety Critical Applications. 2022. Available online: https://www.highintegritysystems.com/wp-content/uploads/2025/10/Building_on_FreeRTOS_Safety_Critical_Applications.pdf (accessed on 3 November 2025).
30. WITTENSTEIN. High-Precision Drive Systems and Mechatronic Solutions. Available online: <https://www.wittenstein.de/en-en/> (accessed on 10 July 2025).
31. Wang, X.; Mizuno, M.; Neilsen, M.; Ou, X.; Rajagopalan, S.R.; Boldwin, W.G.; Phillips, B. Secure RTOS Architecture for Building Automation. In Proceedings of the First ACM Workshop on Cyber-Physical Systems-Security and/or Privacy, Denver, CO, USA, 16 October 2015; CPS-SPC '15; pp. 79–90. [CrossRef]
32. Levy, H.M. *Capability-Based Computer Systems*; Digital Press: Los Angeles, CA, USA, 2014.
33. Miller, M.S.; Yee, K.P.; Shapiro, J. *Capability Myths Demolished*; Technical Report, Technical Report SRL2003-02; Johns Hopkins University Systems Research: Baltimore, Maryland, 2003.
34. Hydra. Hydra: The Kernel of a Multiprocessor Operating System. 1971. Available online: <https://homes.cs.washington.edu/~levy/capabook/Chapter6.pdf> (accessed on 15 July 2025).
35. KeyKOS. The KeyKOS System. Available online: <http://cap-lore.com/CapTheory/upenn/> (accessed on 15 July 2025).
36. Shapiro, J.S.; Smith, J.M.; Farber, D.J. EROS: A fast capability system. In Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles, Charleston, SC, USA, 12–15 December 1999; pp. 170–185.
37. The Fiasco Microkernel. Available online: <https://github.com/kernkonzept/fiasco> (accessed on 15 July 2025).
38. Watson, R.N.; Woodruff, J.; Neumann, P.G.; Moore, S.W.; Anderson, J.; Chisnall, D.; Dave, N.; Davis, B.; Gudka, K.; Laurie, B.; et al. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In Proceedings of the Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 20–37.
39. Rasifard, H.; Gopinath, R.; Backes, M.; Nemati, H. SEAL: Capability-based access control for data-analytic scenarios. In Proceedings of the 28th ACM Symposium on Access Control Models and Technologies, Trento, Italy, 7–9 June 2023; pp. 67–78.
40. Xia, Y.; Du, D.; Hua, Z.; Zang, B.; Chen, H.; Guan, H. Boosting Inter-process Communication with Architectural Support. *ACM Trans. Comput. Syst.* **2022**, *39*, 6. [CrossRef]
41. Microsoft Corporation. CHERIoT RTOS Platform. Available online: <https://github.com/CHERI-IoT-Platform/cheriot-rtos> (accessed on 15 July 2025).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.