CHERIoT: Rethinking security for low-cost embedded systems

Microsoft Technical Report MSR-TR-2023-6

Saar Amar, Tony Chen, David Chisnall, Felix Domke,
Nathaniel Wesley Filardo, Kunyan Liu, Robert M. Norton, Yucong Tao,
Robert N. M. Watson, Hongyan Xia

6th February 2023

Abstract

Small embedded cores have little area to spare for security features and yet must often run code written in unsafe languages and, increasingly, are exposed to the hostile Internet. CHERIoT (Capability Hardware Extension to RISC-V for Internet of Things) builds on top of CHERI and RISC-V to provide an ISA and software model that lets software depend on object-granularity spatial memory safety, deterministic use-after-free protection, and lightweight compartmentalization exposed directly to the C/C++ language model. This can run existing embedded software components on a clean-slate RTOS that scales up to large numbers of isolated (yet securely communicating) compartments, even on systems with under 256 KiB of SRAM.

Acknowledgments

This document contains some elements from the CHERI ISA Specification[1], which is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit:

http://creativecommons.org/licenses/by/4.0/

We acknowledge all the authors of that report:

Robert N. M. Watson	Peter G. Neumann	Jonathan Woodruff	Michael Roe
Hesham Almatary	Jonathan Anderson	John Baldwin	Graeme Barnes
David Chisnall	Jessica Clarke	Brooks Davis	Lee Eisen
Nathaniel Wesley Filardo	Richard Grisenthwaite	Alexandre Joannou	Ben Laurie
A. Theodore Markettos	Simon W. Moore	Steven J. Murdoch	Kyndylan Nienhuis
Robert Norton	Alexander Richardson	Peter Rugg	Peter Sewell
Stacey Son	Hongyan Xia		

as well as the many other contributors to the CHERI project:

Sam Ainsworth	Ross J. Anderson	Ruben Ayrapetyan	Hadrien Barral
Thomas Bauereiss	Stuart Biles	Andrew Bivin	Peter Blandford-Baker
Matthias Boettcher	David Brazdil	Reuben Broadfoot	Kevin Brodsky
Ruslan Bukin	Brian Campbell	Gregory Chadwick	Serban Constantinescu
Chris Dalton	Nirav Dave	Dominique Devriese	Mike Dodson
Lawrence Esswood	Jonas Fiala	Wedson Filho	Anthony Fox
Paul J. Fox	Franz Fuchs	Ivan Gomes Ribeiro	Paul Gotch
Tom Grocutt	Khilan Gudka	Brett Gutstein	Jong Hun Han
Andy Hopper	Alex Horsman	Timothy Jones	Asif Khan
Myron King	Joe Kiniry	Chris Kitching	Wojciech Koszek
Robert Kovacsics	Karthik Muthusamy	Patrick Lincoln	Marno van der Maas
Anil Madhavapeddy	Ilias Marinos	Tim Marsland	Ed Maste
Alfredo Mazzinghi	Kayvan Memarian	Dejan Milojicic	Andrew W. Moore
Will Morland	Alan Mujumdar	Prashanth Mundkur	Edward Napierala
Philip Paeps	Lucian Paul-Trifu	Austin Roach	Colin Rothwell
John Rushby	Hassen Saidi	Hans Petter Selasky	Andrew Scull
Muhammad Shahbaz	Bradley Smith	Lee Smith	Ian Stark
Ramy Tadros	Andrew Turner	Richard Uhler	Munraj Vadera
Jacques Vidrine	Hugo Vincent	Philip Withnall	Bjoern A. Zeeb

[1] https://github.com/CTSRD-CHERI/cheri-specification

The CHERI-RISC-V pseudocode is derived from the Sail CHERI-RISC-V model[2], which has the following license:

[2] https://github.com/CTSRD-CHERI/sail-cheri-riscv

The RISC-V pseudocode is derived from the Sail RISC-V model[3], which has the following license:

[3] https://github.com/riscv/sail-riscv

Contents

Part I

Software model and toolchain

Chapter 1

Introduction

The CHERIoT (Capability Hardware Extension to RISC-V for Internet of Things, pronounced like 'chariot') design is heavily based on prior work by the CHERI project. Our RISC-V extension is based on the CHERI ISAv8[22] and would not have been possible without this work.

This document describes the current status of the CHERIoT ISA. This ISA is not intended to be a final CHERI specification for embedded RISC-V devices but is a work-in-progress that is sufficiently close to final that feedback is valuable. In particular, the current C extension to RISC-V makes a number of optimization decisions that are not useful in a CHERI context and so we would likely benefit from an alternative (as yet unspecified) 16-bit instruction extension for embedded CHERI targets.

The CHERIoT project would not have been possible without the existing "big CHERI" research [6, 22], exploration of green-field CHERI-aware operating systems [8], and work to adapt CHERI software models to embedded systems [1, 2, 24, 25]. This ISA attempts to scale CHERI down yet further, into smaller embedded systems than previously considered; we have taken the opportunity to simultaneously design our ISA, compartment model, programmer model, compiler, and RTOS [16]. The capability encoding and instruction set have been tuned to enable this use and validated by running existing embedded software in compartments. Curious readers are invited to see our study of related works in Appendix E.

This document describes:

- The RISC-V ISA extension (Part II).
- The compartment model that the ISA is intended to support (Chapter 2).
- The RTOS implementation used to enforce the model (Chapter 3)
- The language extensions used to expose this model to developers (Chapter 4).
- The ABI used to implement the compartment model (Chapter 5).

Performance, power, and area costs for the implementation are not part of this and will be presented in a follow-up publication.

1.1 The CHERIoT RTOS Model

Because the ISA given here is the result of simultaneous design with software, we will often refer to software concepts for motivation or justification of design choices. Particularly central to the discussion are the notions of memory safety, compartments, and threads. We provide coarse definitions here and will refine them as we continue.

A **thread** is a schedulable entity associated with a general-purpose register file and a designated region of memory for use as a call stack. Threads are either running, in which case their register file is the CPU's, or suspended, in which case the register file is saved to memory for later use.

A system is said to be **memory safe** if its references to memory are:

Unforgeable A reference to memory (in particular, the authority to access memory) can be constructed only from other references.

Monotonic A constructed reference will have no more authority than its progenitor reference(s) (and may have less).

Spatially Safe References to memory authorize access to a set of memory locations determined when the reference is constructed.

Temporally Safe References to a region of memory will not remain usable across *reuse* of memory for a different allocation.

CHERIoT is based on CHERI and so language-level references are expected to compile down to CHERI capabilities. We may gloss over subtle distinctions and conflate the terms "capability," "pointer," and "reference."

A **compartment** is a collection of code, data, and capabilities that serves as an invocable security context. Compartments statically export *entry-points*, which may be statically imported by other compartments or passed as opaque *cross-compartment function pointers*. A compartment they (statically or dynamically) imports an entry point may then invoke it to perform a *cross-compartment call*. Such calls are synchronous and transition the calling thread from one compartment to another. The thread's stack is used, with appropriate bounds adjustment, in both the caller and callee compartment.

1.2 Security goals

We aim to provide a minimal TCB that can run user software (including third-party code) in compartments. These compartments are not part of the TCB and are assumed to have a mutual distrust relationship with each other.

We define a set of security guarantees that apply to all untrusted compartments. These guarantees are enforced by three system components, which form the TCB for confidentiality and integrity. These components are:

The loader, which is responsible for setting up all of the initial capabilities for everything in the system. This never accesses any data that was not part of the initial firmware image.

The switcher, which is responsible for transitions between compartments and between threads. This is around 300 instructions in hand-written assembly.

The memory allocator, which is responsible for providing the hardware with the information required to enforce heap memory safety.

The code in these components must be carefully audited.

An embedded system may have user-provided components that run at the highest priority level, with interrupts disabled. Any code that runs with interrupts disabled is part of the TCB for availability.

1.2.1 Heap memory safety

CHERIoT RTOS provides a **heap**, a system-provided compartment with dedicated memory to which it grants other compartments dynamic, temporary access.

The allocator ensures that the capabilities it gives out have bounds that do not overlap any other allocation, and so the CHERI bounds checks enforce spatial memory safety. Heap memory is also temporally safe: no heap memory will be reused until the system has ensured that all dangling pointers to it are deallocated. The hardware provides a guarantee that no capability can be loaded if the memory allocator has marked the memory it points to as deallocated. This mechanism is described in detail in Section 7.8. A revocation service (which can be implemented in hardware or software) periodically scans all memory and deletes capabilities that point to revoked memory.

The memory allocator is part of the TCB as it could violate confidentiality and integrity in a number of ways. It holds a capability to the entire heap and so, in principle, can violate confidentiality of heap contents either directly or by failing to clear memory before issuing an allocation in reused space. Similarly, it can violate integrity by directly using or improperly revealing capabilities held within heap memory. It can violate spatial safety by

not correctly bounding capabilities it returns. Finally, it could violate temporal safety by either not marking freed objects as deallocated or by un-marking the memory and reusing it before revocation.

However, despite all that, the memory allocator does not hold capabilities to normal compartment memory, only to region(s) reserved for the heap it manages. As such, even a completely compromised memory allocator can violate safety properties of the heap only; it cannot directly violate memory safety for non-heap memory.

1.2.2 Local stack memory safety

Allocations on a thread's stack are bounded by CHERI capabilities – the compiler generates instructions to derive these capabilities from the stack capability – but are not guaranteed to be temporally safe.

However, CHERIoT ISA and CHERIoT RTOS have mechanisms to ensure that capabilities to stack allocations can not be stored anywhere other than on the stack to which they point (which may include address-taken allocations on the caller's portion of the stack). In practice, this means that violating stack temporal safety is very difficult: stack-derived capabilities cannot be stored onto the heap or into a global (it will deliver a trap). Therefore, the only way of having a stack pointer outlive the allocation is to store it through another stack-derived pointer that points to a higher frame or return it directly in a register.

1.2.3 Cross-compartment stack memory safety

During a cross-compartment call, the thread and its stack transition security contexts. The stack bounds are restricted so that the callee has no access to the caller's stack, except via capabilities that are explicitly passed as arguments. This suffix of stack memory accessible to the callee is zeroed both on call and on return, which prevents any information leak from uninitialized memory. The implicit stack pointer and stack-derived arguments provided by the caller are the only pointers that are both held by a compartment and capable of storing other stack-derived pointers.

Lexically-Scoped Delegation

CHERIoT ISA's capability permission scheme allows software to derive variants of capabilities that can be stored only to stack memory. Additionally, CHERIoT ISA can impose this derivation *transitively* per capability: any capability loaded via such a capability becomes another such and, so, will impose the same on those loaded through it, and so on.

These two mechanisms allow for **lexically-scoped delegation**: a calling compartment may, for any capability it holds, construct a variant that can only be stored to the stack and can load only capabilities that can be stored only to the stack. Passing this derived capability to the callee ensures that the callee compartment cannot capture either the passed capability or any loaded through it in memory not visible to (and mutable by) the caller after return.

1.2.4 Global memory safety

All objects with static storage duration are compartment-local and are visible to any thread executing within a compartment. Memory safety for globals is therefore advisory (untrusted code can simply access the global capability directly). The compiler will insert bound for any address-taken global. Immutable objects with static storage duration are derived from **PCC** (with the execute permission removed) and so cannot be written to.

1.2.5 Higher-level security properties

The local security properties outlined above are used to build isolation between threads and compartments. This leads to the following high-level security goals:

- No compartment should be able to access another compartment's data, except where explicitly shared.
- No thread should be able to access another thread's data, except where explicitly shared.

Compartments that have not been explicitly granted the rights to run with interrupts disabled should also not be able to impact availability.

1.2.6 Threat model

We assume that code running in a compartment is untrusted. As such, an attacker is assumed to have the ability to execute arbitrary code within a compartment. It is not possible to prevent programmers from introducing bugs but we aim to provide a set of tools that will make it easy to write code in a compartment that is able to protect itself from another attacker-controlled compartment that can invoke its entry points.

Chapter 2

Compartment model

CHERI is designed to support fine-grained compartmentalization. A compartment, in the CHERI sense, is defined by the memory that is transitively reachable[1] from the capability registers in the running code. The mechanism for transitioning between compartments is key to any CHERI compartmentalization strategy.

The original CHERI/MIPS prototype had an instruction that raised a synchronous abort, providing a transition into an in-kernel compartment switcher. Morello and newer CHERI/RISC-V implementations for large systems have instructions that perform atomic unsealing and domain transition. This provides a rich set of tools for building compartmentalization models but leaves concerns such as stack management to the software stack. The CHERIoT model relies on a mixture of hardware and software to enforce compartment isolation.

The threat model for this work assumes that compartments all exist in a mutual distrust relationship with each other. Compartments should not be able to see or tamper with other compartments' data unless they are explicitly granted access to it via capabilities passed across an exposed interface.

Compartments, in isolation, are not automatically trusted (or untrusted) with respect to availability. Each compartment explicitly lists the entry points that it exposes or may invoke that run with interrupts disabled and it is the responsibility of the firmware integrator to determine whether this is acceptable. For a compartment to run code with interrupts disabled, the linker and loader must have explicitly granted it these rights when initializing capabilities, and so it is possible for the firmware integrator to audit the compartment graph.

This is intended to give flexibility for system integrators with different levels of real-time requirements. At one extreme, a hard-realtime control loop can run in a realtime-

[1] i.e. including memory reachable from capabilities loadable from memory by any number of indirections

priority thread, with interrupts disabled except at explicit yield points. Other threads in such a system would not be allowed to call any compartment entry points that can invoke functions that run with interrupts disabled and so the realtime-priority thread can always resume in the context-switch time. A somewhat softer realtime system may allow a small number of functions to be invoked from compartments that are exposed to lower-priority threads. These functions would be audited to ensure that their worst-case execution time didn't cause realtime components to miss their guarantees. At the weakest extreme, global forward progress is purely a best-effort objective and any compartment may be allowed to call functions that have no guarantees on bounded execution time and run with interrupts disabled.

We expect that compartments may be provided by untrusted third parties and so it is important that every cross-compartment interaction is amenable to auditing. In particular, the linker can see everything that the loader will set up and the loader is required to explicitly grant access to a compartment for every:

- MMIO region that a compartment has access to.
- Cross-compartment entry point that a compartment exposes (and its interrupt status on entry).
- Internal function that a compartment may run with interrupts disabled.
- Cross-compartment call that a compartment may perform to another compartment.
- Shared library routine that a compartment may invoke.

This is sufficient to retrieve a complete graph of cross-compartment communication, including which compartments may be running with interrupts disabled. This provides tools for firmware integrators to write policies such as:

- Only the specific code that the regulator approved may communicate directly with this device.
- Any code may run on the device but only the TLS compartment may talk to the network stack and only a compartment that exposes a small set of well-defined APIs may call the TLS stack.
- There must be no interaction between any of the compartments managing service A and the compartments managing service B on the device, except yielding via the scheduler.

2.1 Compartments define spatial ownership

At its most reductionist, a CHERIoT RTOS compartment is defined by two registers:

PCC is the program-counter capability, which is used to reach code and read-only globals.
CGP is the capability global pointer, which is used to reach read-write globals.

These define a set of code and data that represents a compartment. A compartment is a single security context. While running in a compartment, any code in the memory reachable by **PCC** may be executed, any data in that memory may be read, and any data in the globals reachable from **CGP** may be read or written.

Note, in particular, that compartments are responsible for enforcing an object abstraction on top of their global memory. The C/C++ compiler will automatically insert bounds when the address of a global is taken but an assembly programmer in a compartment is able to reach any globals. Our security model assumes that all code within a compartment trusts all other code within that compartment.

2.2 Threads define temporal ownership

A CHERIoT RTOS thread is a schedulable entity that owns a stack, a trusted stack, and a register set. When a thread is scheduled, it owns the microcontroller's register file. When it is suspended, the register file is stored in a register save area.

Each thread is isolated from other threads. The CHERIoT ISA provides a simple 2-bit information-flow enforcement mechanism in the form of the global bit and the store-local permission. Capabilities without the global bit can be stored only via capabilities that have the store-local permission. In CHERIoT RTOS, only three types of memory have the store-local permission:

Stacks, reachable from a running thread's **CSP** and any capabilities derived from this (address-taken stack allocations).
Register save areas, reachable only from a special capability register (SCR) that are used to store a thread's state on context switch.
Trusted stacks, reachable only from a SCR, which are used to save and restore the stack pointer on compartment switch (more on this later).

Of these, normal compartment code has access only to the stack. The latter two are a single allocation that is reached via a SCR. The switcher is the only code that runs (after the loader has exited) with the rights to access this SCR. Threads' register files and stacks dynamically define a set of reachable objects.

2.3 Execution at the intersection of threads and compartments

Threads do not own code and compartments do not own a register file. Execution requires (at least) both of these and happens when a thread is scheduled to run within a specific compartment. Each thread starts at an entry point within a compartment and execution continues within that compartment until either the thread calls another compartment or a context switch invokes another thread.

This means that running code always has access to the code for the current compartment, the globals for the current compartment, the part of a thread's stack and register state associated with the current compartment invocation.. Two threads might be in the same compartment at the same time (one of them preempted or yielded, the other running), if the compartment permits this. If two threads enter the same compartment (either at the same time or sequentially) then they will see the same set of globals and can use them to communicate.

Globals (more specifically, capabilities derived from the value in the **CGP** register) do not have store-local and so it is not possible to construct a capability that is reachable from a global and which points to a stack allocation. This gives strong cross-thread isolation. If a thread enters a compartment that is compromised, a thread running compromised code within that compartment cannot tamper with the victim thread's stack or register file and must use data-oriented attacks from data reachable from globals.

2.4 Compartment switches enforce compartment isolation

Cross-compartment calls require that a thread loses access to one compartment and gains access to another. CHERI provides a *sealing* mechanism to build this kind of model. We use this with an explicit compartment switcher to build a robust compartment invocation mechanism for embedded systems.

When a thread wishes to invoke another compartment, it loads two capabilities from its import table (see Chapter 5). The first is a sealed capability to a structure describing the entry point in the callee. The second is a *sentry* capability to the compartment switcher. The sealed capability is passed in a register when the sentry capability is called.

A CHERI sentry is a capability that can be jumped to but cannot be used for any other operations. The CHERIoT ISA extends this by allowing different kinds of sentry to control interrupt state. The sentry for the compartment switcher implicitly disables interrupts on entry to the switcher, which makes it easier to reason about the execution flow within the switcher.

The compartment switch routine unseals the target capability and uses it to find the **PCC** and **CGP** of the target compartment, and the offset within the **PCC**. It can then construct a target to invoke. In addition, it reads the number of registers that the callee expects to have passed (which it uses to zero unused argument registers) and the interrupt status for the callee (which it uses to reenable interrupts immediately prior to invocation, if required). The RV32E ABI defines only two callee-save registers. The switcher saves these onto the trusted stack and then zeros all non-argument registers except for **CGP** and **CSP**, which have special handling.

In addition to these steps, the compartment switcher is responsible for preventing the stack from being used to leak data between compartments (other than via explicit arguments). This requires several steps. First, the stack passed in the **CSP** register must be shrunk to allow CHERI's spatial bounds protection to prevent any access by the callee to the caller's portion of the stack. Second, both before a call and before completing the return transition, the compartment switcher zeroes the portion of the stack that is made available to the callee. Zeroing the stack seems expensive but recall that in embedded systems a 2 KiB stack is considered *very* large. Our stacks are typically 1 KiB. With a 33-bit memory bus, we need 256 stores (in the worst case) to zero the whole thing. That's more expensive than a function call, but not vastly so.

At the end of a compartment transition, the new compartment has access to:

- Its own code (**PCC**)
- Its own globals (**CGP**)
- A portion of the thread's stack, excluding any frames owned by the caller, and full of zeroes.
- Any memory pointed to by argument capability registers, passed explicitly from the caller.

On return, any temporary state is cleared and the caller has access only to explicit return capabilities.

This does not prevent one compartment from having access to another compartment's globals, but there are legitimate reasons for wanting this. For example, a compartment may derive a read-only (no store permission) capability to one of its globals and use that to cheaply broadcast state updates to subscribers.

2.5 Context switches enforce thread isolation

Context switches happen as a result of an interrupt (including synchronous aborts / exceptions). The context switcher code saves the register file into a save area pointed to by a

SCR. The register save area and the trusted stack are both reached by the same SCR and the two switchers (thread and compartment) are the only code in the system that runs with permission to access this register after the loader has finished.

The context switch routine (part of the switcher's approximately 300 instructions) is the only code that is able to violate thread isolation. It has access to two threads simultaneously:

- The stack pointed to by **CSP** on entry to the interrupt handler.
- The stack that the scheduler will use, loaded from a read-only global in the switcher's **PCC**.

Before invoking the scheduler, the switcher will seal the capability to the register save area (from which the stashed **CSP** is reachable) and pass it as an argument into the scheduler. The scheduler is therefore in the TCB for availability but, crucially, not for confidentiality or integrity.

The scheduler runs with interrupts disabled and selects the next thread to run, returning a (sealed) capability to the register save area to the switcher. This must be sealed with the object type that the switcher expects. The loader guarantees that nothing except the switcher has a permit-seal capability for that type and so the scheduler is able only to provide register save areas that were previously provided by the loader or the switcher.

The current CHERIoT RTOS scheduler is a very simple priority scheduler that does round-robin scheduling within a priority level. A more complex one could be added for use cases that need something more complex without changing the security model. Conversely, an even simpler scheduler that exposes a less rich set of inter-thread communication primitives could be used for safety-critical systems.

The scheduler is a compartment just like any other and so can expose more complex scheduling operations such as message queues as cross-compartment calls that then explicitly yield.

2.6 Adding shared libraries

In a compartmentalized system it is very common to have routines that are required from many different compartments. This is trivial to support by duplicating the code into all compartments that use it. On large systems with a memory-management unit it's possible to logically duplicate the code in the virtual address space without duplicating it in the physical address space. This is not possible on a system such as ours, without any virtual memory support.

Instead, CHERIoT RTOS provides a shared-library abstraction that is designed to work in concert with our compartmentalization model. A shared library is much like half of a compartment: it may contain code and read-only data (**PCC**) but may not contain read-write globals and so runs with the **CGP** of the caller. A function in a shared library runs with the context of the caller and so invoking a shared-library function does not need to go via the compartment switcher.

Cross-library calls, as with cross-compartment calls, must change **PCC** to a specific location in another block of code. This is enforced by the loader providing callers with a sentry capability to the jump target. This prevents the caller from being able to jump to arbitrary points in a shared library. It also allows shared libraries to expose routines that run with interrupts disabled. For example, on a core that doesn't provide native atomics, we can expose atomic-increment functions that perform a simple read-modify-write with interrupts disabled, without having to go via the compartment switcher.

Chapter 3

RTOS implementation

The CHERIoT RTOS is intended to provide a minimal TCB. The core of the RTOS comprises:

A loader, which runs before any untrusted data is encountered and sets up the capabilities for the rest of the system.
Switch routines, which add up to around 300 instructions in hand-written assembly, for switching between thread and compartments.
A heap allocator, which allocates memory from a shared heap for use by compartments.
A scheduler, which selects the next thread to run.

Of these, only the loader and the switch routines run with access to the trusted stack and register save areas (that is, with the access-system-registers permission on their **PCC**). This means that these are the only two that can completely compromise all of the security properties on which the rest of the system is built. The loader runs once on system start and then erases itself. The loader is not needed on systems where the persistent storage (e.g. flash) can store tag bits.

The switch routines (currently) add up to a total of around 300 instructions, with no memory allocation and very little control flow. For comparison, the trusted (unverified) part of seL4 is 340 instructions [12], so CHERIoT RTOS contains less code in its TCB than seL4 contains unverified code in its TCB.

The heap allocator holds capabilities to the shared heap and the revocation bitmap for the shared heap and so is able to violate heap memory safety. The scheduler is more or less an untrusted compartment, though with a private stack. Importantly, the scheduler does not have access to the stacks, trusted stacks, or register-save areas of the threads that it manages. It may choose the next thread to run, but it cannot tamper with a thread's state.

3.1 Per-thread state

Each thread has a stack (reachable from **CSP**) and a *trusted stack*. The trusted stack maintains the state required for cross-domain calls. The same data structure also contains the register save area, where the contents of the register file will be saved when an interrupt is delivered.

A capability to the trusted stack and register save area for the running stack is stored in the **MScratchC** register, which can be accessed only by code whose **PCC** has the permission to access system registers.

3.2 The loader

The loader starts with access to the root capabilities and so has complete access to everything on start. The majority of the loader is written in C++ with rich types conveying intentionality, including templates that provide capability permission sets as compile-time constants that can be statically checked

The loader splits the architectural roots into four software-defined roots:

Executable capabilities are used for deriving capabilities that will end up installed in **PCC**.

Global capabilities do not have permit-store-local and are used for deriving capabilities for globals, heap memory, and so on.

Local capabilities do not have the global permission and are used only for stacks, trusted stacks, and register save areas.

Sealing capabilities have no memory-related permissions and are used only for sealing and unsealing.

Each capability that is derived from a root is derived via a mechanism that validates (at compile time) that the requested permissions are less than the permissions of the root.

The C++ portion of the loader is stored in a portion of memory that will eventually become the heap. Once it returns to a small assembly stub, this stub zeroes all of the memory used by the loader (stack, code, and globals) and almost all of the register file, ensuring that no capabilities are leaked. It then yields to the scheduler (via an `ecall` instruction) and becomes a stackless idle thread.

The loader is responsible for initializing import tables (the capabilities that may point outside of the compartment, see Chapter 5 for details), preparing each thread's initial state, and applying caprelocs (dynamically initialized capabilities stored in globals). For each capreloc, the loader finds the compartment that it refers to and attempts to derive the target

from the compartment's **PCC** and **CGP**. If this fails, then the boot image is corrupted and the loader resets (allowing a first-stage loader to perform A/B installations).

3.3 Interrupt handling

The interrupt handling code is part of the switcher and runs with permission to access **MScratchC**. It first saves the register file in the current thread's register save area and seals the capability to this area. Next, it loads the other special-register values that describe the interrupt and prepares a context invoking the scheduler.

The scheduler is always invoked with the same stack, with arguments containing a sealed capability to the register-save area of the interrupted and yielding thread. It then returns a sealed capability to a register-save area for a thread to resume. The scheduler runs with interrupts disabled and provides a simple static priority scheduler with round-robin scheduling within a priority level. Threads are run when no thread with a higher priority is runnable. If two threads at the same priority level are runnable, one runs until it either yields or an interrupt is delivered, then the next one runs.

3.4 Synchronization and scheduling primitives

The scheduler is a compartment and so can expose entry points that can be invoked via the switcher. These include semaphore and message queue interfaces that will park a thread (mark it as not runnable) until some event happens.

In addition, an `MCall` instruction will deliver an interrupt, causing the running thread to immediately yield. This is used for an explicit yield operation that transfers control immediately to the switcher and then to the scheduler. This mechanism can be used from inside the scheduler itself to allow a thread to yield after the scheduler has updated some data structures related to it.

In combination, these can be used to build synchronization primitives with timeouts. A thread that wishes to block waiting for an event calls the scheduler, which then records the conditions that will wake the thread (including the timeout) and yields via an `MCall`. One resume, the scheduler can check how long it slept for and return from the cross-compartment call.

3.5 The memory allocator

The memory allocator currently provides a simple `malloc`-like API. Future versions will add explicit memory pinning (no concurrent deallocation of an object during a compartment invocation) and explicit permission to deallocate objects.

In addition, the allocator provides a mechanism for allocating sealable objects. The CHERIoT ISA has only a handful of sealing types and so it is not feasible for every compartment to be able to seal capabilities using the hardware mechanism. Instead, the memory allocator reserves one hardware sealing type for allocating sealable objects. A sealable object has a header describing the capability that is used to unseal it. The allocator will provide a sealed capability to the whole object (including the header) and, if invoked with the correct unsealing capability) will return an unsealed capability to all of the object *except* for the header.

This mechanism allows compartments to provide opaque data types to software running outside of the compartment.

3.6 Error handling

One effect of the CHERI architecture is to turn errors that could result in memory corruption vulnerabilities into traps. To mitigate the availability concerns this could create, the CHERIoT RTOS provides a mechanism for recovering from faults in a controlled way. Compartments can define an error handler that will be called if a fault occurs during execution of that compartment, or if a fault in a called compartment results an 'unwind'. The error handler can inspect the saved register context from the time of the fault and can choose either to resume execution with an amended register context or to unwind the trusted stack, returning an error to the calling compartment.

Careful use of this mechanism can allow an application to continue even after it encounters an unexpected fault. For example, a compartment error handler might reset the compartment state before returning an error to the calling compartment. For more details of this mechanism and special security considerations please refer to the CHERIoT RTOS documentation.

3.7 Other components

All other components, such as a network stack (taken from FreeRTOS), a TLS stack (from mBedTLS), and so on are untrusted. They are treated no differently from any other user-provided code and are packaged only for convenience. The bottom of the network stack

talks to a device driver, which is simply another compartment whose import table is used to grant access to the MMIO region containing the network device's control registers.

Chapter 4

C/C++ language and toolchain extensions

In addition to the existing CHERI C/C++ extensions, we define a small number of additional extensions that are specific to CHERIoT. CHERI C is already able to compile most existing embedded code that we have tried with no modifications. Embedded code has to tolerate targets with Harvard architectures, different pointer sizes for different types of data, different memory banks, and so on. In comparison, a CHERI target is far more like a conventional ISA.

We have not had to change the CHERI C model at all for code running within a compartment. Our extensions are focused on supporting the compartmentalization model.

4.1 Specifying compartments

We have added an attribute to specify the compartment that implements a given function. This is used in conjunction with the `-cheri-compartment=` flag passed to the compiler, which specifies the compartment in which the current compilation unit will end up. The compiler will raise an error if the compartment name for the current compilation unit does not match the name of a function that has an implementation. For example, if a header file contains the following declarations:

```
1 __attribute__((cheri_compartment("example"))) void foo(int);
2 __attribute__((cheri_compartment("other"))) void bar(void);
```

If `foo` is implemented then the compiler must be invoked with `-cheri-compartment=example`. Any call to `bar` will then be treated as a cross-domain call.

This mechanism allows lightweight annotations on functions that are exposed across compilation units. Software that already supports a DLL-style linkage model may have macros on public functions for using this. Other software can easily maintain these annotations for CHERI targets with a macro that expands to nothing for non-CHERI targets.

Adding `__attribute__((cheri_ccallback))` to a function marks it as a cross-compartment callback. Taking the address of such a function will give a pointer that can be passed across compartments and allow the recipient to recursively invoke this compartment.

4.2 Exposing library calls

Adding `__attribute__((cheri_libcall))` to a function marks it as a library call. The compiler will always generate an indirect call (via a sealing capability from the import table) for all library functions, unless the callee and caller are in the same compilation unit and have compatible interrupt states.

All of the functions that the compiler may insert calls to (such as `memcpy`,`__cxa_guard_acquire` , and so on) are assumed to implicitly carry this attribute. The compiler must be able to insert calls to these without knowing the name of the library that provides them and so this attribute provides a single flat namespace for all such functions.

4.3 Controlling interrupt state

The `cheri_interrupt_state` attribute controls whether, during execution of the function, interrupts are enabled, disabled, or in whatever state there were for the caller. It takes a single argument, which must be either `enabled`, `disabled`, or `inherited`. The attribute can also be written as a C+11 / C2x-style attribute. For example:

```
1 // This function runs with interrupts disabled
2 [[cheri::interrupt_state(disabled)]]
3 int disabled(void);
4 // This function runs with interrupts enabled
5 __attribute__((cheri_interrupt_state(enabled)))
6 int enabled(void);
7 // This function runs with whatever interrupt state the caller has.
8 __attribute__((cheri_interrupt_state(inherit)))
9 int inherit(void);
```

All functions that are not exposed across compartment boundaries (including library calls) default to `inherit`. Cross-compartment calls default to `enabled` and must be set to either `enabled` or `disabled`.

4.4 Linking compartments

The version of LLD used with CHERIoT provides a `-compartment` flag for linking compartments. This is somewhat similar to `-r`, which creates a relocatable object file. Linking in this mode marks all symbols except for those in the export table as local. Unlike `-r`, COMDATs are merged when linking a compartment. In other respects, this is identical to `-r`: relocations are not processed and will be handled in the final link step.

Chapter 5

ABI

CHERIoT is a hardware-software co-design project, where the ISA and ABI have been carefully designed together to provide the desired compartment model and security guarantees.

5.1 Compartment layout

Each compartment has two reachable regions, bounded by **PCC** and **CGP**. The **PCC** region contains the compartment's code, read-only data, and *import table*. Read-only data includes relocation read-only data, which is initialized by the loader at boot time. A compartment's import table is a read-only table containing capabilities that are used for cross-compartment and cross-library calls, as well as any imported data.

A compartment also has an export table associated with it. The export table defines the set of functions that are exported from the compartment (callable by others). This is read by the compartment switcher (see Section 5.5).

Shared libraries are identical to compartments in structure, except that they lack a **CGP**.

5.2 Access to globals

Read-only globals are accessed using PCC-relative addressing. CHERI RISC-V extends the RISC-V `auipc` (add upper immediate to program counter) instruction to `auipcc` (add upper immediate to program counter *capability*). This adds a 20-bit immediate, left shifted by 12, to the current **PCC** value, giving an address that is within the immediate range of a RISC-V load or store instruction of the target.

Unfortunately, the result of the auipcc instruction may be out of the bounds of **PCC**. This does not matter on CHERI systems with 128-bit capabilities because the encoding guarantees that capabilities remain valid 4096 bytes out of bounds. However, this is not the case with our 64-bit capability encoding that has much tighter 'representable bounds'. The tag bit is cleared if the capability is too far out of bounds, we must therefore modify the standard CHERI-RISC-V relocation scheme to avoid taking capabilities out of bounds in the middle of computing an address.

We are able to solve this problem by having at least a 1-bit overlap between the immediate field of the loads and stores (or cincoffset instructions) and the aupicc. The auipcc instruction and the second instruction must both displace the **PCC** in the same direction, keeping the intermediate capability in bounds and hence representable. If the target address is after the current instruction then both values must be positive, otherwise both values must be negative, which is a simple property for the linker to ensure when applying the relocations. To make this possible we reduce the shift of auipcc by one, meaning auipcc can always produce a value within the 2 KiB range required.[1] This does limit the maximum offset for a relocation to less than 2^{31} but this is not a problem in practice due to the limited size of compartments. Any system that needs more than 2 GiB compartments would likely benefit from a 64-bit address space.

Accesses to read-write globals is very similar. The **CGP** register is biased by half the size of the combined globals section (.data, .bss, and so on). This means that the full immediate range is accessible for displacements. With a 12-bit immediate, a single compartment can access 4 KiB of globals in a single load or store (or take their address with a cincoffset instruction). We define a new instruction, auicgp, and a relocation type that uses it to mirror the **PCC**-relative addressing mode.

We rely on linker relaxation to optimize both **PCC** relative and **CGP** relative relocations. This means that relocations within ± 2 KiB of **PC** or **CGP** require only one instruction. Given the security incentive to keep compartments small we expect relaxation to work well in the common case. In particular, if a compartment has more than 4 KiB of mutable global state it may be advisable to split it into multiple compartments or use dynamic allocation.

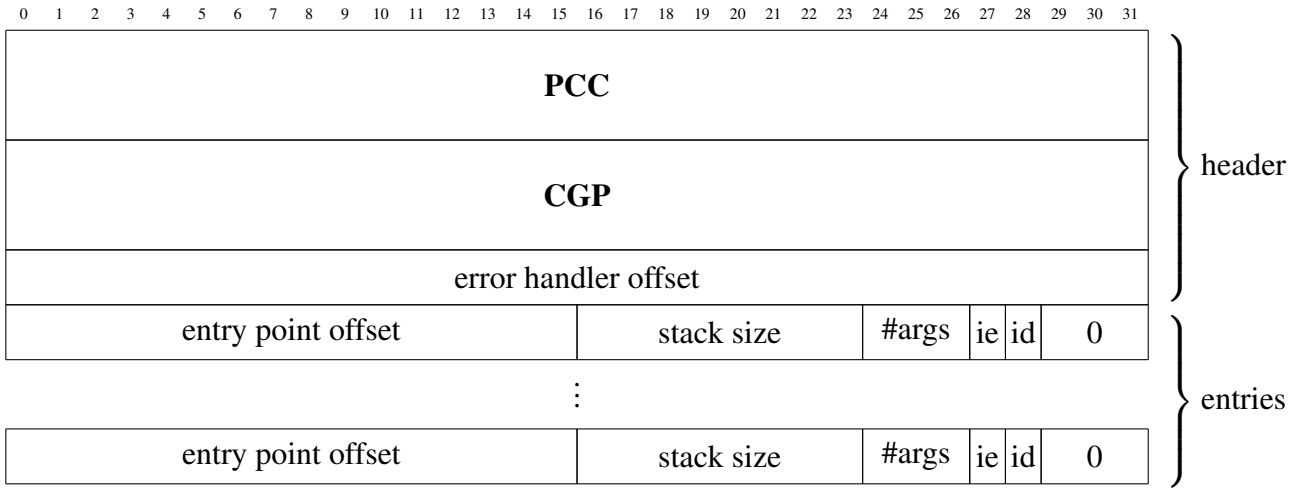


Figure 5.1: Compartment export table layout

5.3 Export table layout

Figure 5.1 shows the layout of the export table for a compartment. Each export table starts with a copy of the **PCC** and **CGP** for the target compartment. The next 32-bits is the offset of the compartment's error handler realtive to **PCC**.**base**, or -1 if the compartment does not have an error handler. If an error occurs the switcher may jump to this as described in Section 3.6. After the header, the export table is comprised of one 32-bit entry per exported function. The first 16 bits of each entry provide the displacement from the start of the compartment's **PCC** to the entry point. This limits a compartment to exporting functions in the first 64 KiB of its code section. Most compartments have significantly under 64 KiB of code, the few that are larger can sort their internal layout to ensure that the exported functions all fit within the start.

The next 8 bits are the minimum amount of stack space that the function requires. This allows compartments to be defensive against callers that try to invoke them without enough stack space for their prologues. If a function requires more than 256 bytes of stack space then it can add a dynamic check on the size of **CSP** after the compartment switch.

The final 8 bits are reserved for flags, described in the following table:

[1] An alternative solution would be to increase the size of the immediate on loads and stores. On RV32E this could be achieved using the register selection bits that are freed by moving from 32 to 16 registers. Our first prototypes did this but we choose to remain compatible with RV32I by modifying `auipcc`.

Bits	Meaning
0-2	Number of argument registers used.
3	Interrupts enabled
4	Interrupts disabled

The compartment switcher is responsible for clearing all registers except for the used argument registers and so must know how many are used. The compiler fills in this set. This provides a value from 0 (no arguments) to 7 (all six argument registers used, plus **C5** carrying stack arguments.

Exports from compartments must set either the interrupts-enabled or interrupts-disabled bit. Code running in a different security context always runs with an explicit interrupt status, to make it easier to reason about compartment behavior. Functions exposed from shared libraries may set neither, in which case the function will be invoked with the caller's interrupt status.

Each export table entry from a compartment is exposed as a symbol of the form `__export_{compartment name}_{function name}`. Each export table entry from a library is exposed as a symbol of the form `__library_export_{library name}_{function name}`. Libraries all use the same name in their export symbols because moving a function from one library to another does not involve running the target in a different security context. The existence of multiple libraries is purely to improve auditing: libraries (their entry points, called functions, and the contents of their code sections) can be individually tracked, allowing code-signing rules to be driven by specific implementations of individual libraries. For example, code signing might require a specific FIPS-certified binary of a crypto library, but allow the shared library providing `memcpy` to be replaced with a more optimised version.

The function name in the export symbol is mangled according to the Itanium C++ ABI rules. This provides some defense against accidental (non-malicious) type mismatches in the caller and callee.

5.4 Import table layout

The import table is similar to a captable in structure. It is the only piece of state reachable from a compartment that is allowed to contain capabilities that point outside of the compartment's **PCC** and **CGP** on system start. This makes it a single place to audit the compartment graph. The import table is mutable only by the loader. After the loader finishes it is reachable only by the read-execute **PCC** for the compartment, not by any capabilities with store permission. Import table entries, at run time, are one of three things:

- Sealed capabilities to export table entries, used for cross-compartment calls.

- Sentry capabilities to library functions.
- Capabilities to memory-mapped I/O (MMIO) regions.

The loader is responsible for initializing these, based on information provided by the compiler and linker. Prior to the loader running, import table entries for the first two categories contain addresses of the corresponding export table entry. Import table entries for MMIO regions contain the start address and the size of the region. This allows a compartment to be granted a subset of an MMIO region, down to access to a single byte (for example, allowing a compartment to poll the 'ready' status of a UART but requiring that it performs a call to the compartment that owns the UART to read or write data with it). A future version will allow read- or write-only access to MMIO regions.

The loader will populate the import table with capabilities. Each import table entry that is used for cross-compartment calls will contain a sealed capability that has the bounds of the target compartment's export table and whose address points to the correct entry. This allows the switcher to load the **PCC** and **CGP** values from the start and to jump to the correct address.

The first entry in the import table has the (local) symbol name `.compartment_switcher`. It is initialized to 0 at static link time and will be initialized by the loader with a sentry capability for jumping to the compartment switcher.

5.5 Cross-compartment calls

Cross-compartment calls pass their arguments in the same registers as the RV32E ABI (**C10**–**C15**). In addition, any stack arguments are passed via **C5** (**Ct0**). The callee does not have access to the caller's stack other than via these arguments and so cannot use **CSP**-relative addressing for on-stack arguments.

The capability loaded from the import table is passed to the switcher in **C6** (**Ct1**). The last step on the caller side is to jump to the sentry pointed to by the `.compartment_switcher` symbol.

If a compartment calls a function that it also exports, and that function has the same interrupt status as the caller, then the compiler may insert a direct call and skip the switcher.

5.6 Cross-library calls

Cross-library calls are simple indirect calls via a capability provided in the import table. The import table entry contains a sentry capability to the target function. The CHERIoT

ISA has sentries that enable, disable, or inherit the current interrupt status and so cross-library calls can toggle or preserve the interrupt state. This makes it easy to reason about the current interrupt state using structured programming idioms.

If a function explicitly changes interrupt state within a compartment then it will be handled as if it were a library function exported from and consumed by the function. In this case, the symbol in the export table will be local.

5.7 Callbacks

In some situations, one compartment wishes to provide a callback that another compartment can invoke. In the CHERIoT ABI, this callback is represented as the same form of sealed capability that would be loaded from the import table. Functions used as cross-compartment callbacks are both exported and imported by the compartment that wishes to take their address. Taking the address of such a function is simply a load from the import table.

As with non-exported functions that change the interrupt status, the symbol in the export table will be local if the function is not also exported as a directly callable function.

5.8 Relocations

The relocations added to RISC-V for CHERIoT ABI are listed in Table 5.1. As with existing RISC-V, some of these are in two forms because RISC-V loads and stores place their immediate operands in different locations. The relocation numbers here are the ones used in the current prototype and are expected to change prior to standardization.

PCC or **CGP** relative relocations consist of a pair of either `auipcc` or `auicgp` plus a 12-bit immediate instruction. In most cases (when the offset is within ± 2 KiB) linker relaxation can reduce this to a single instruction. The `AUICGP` instruction uses an entire major opcode and is rarely needed because it is uncommon for a compartment to have more than 4 KiB of read-write global data (arguably a large globals section is an indication that a compartment should be split or refactored). Therefore, in future we could consider alternative relocations that don't require `auicgp`, such as a three instruction sequence consisting of `lui`, `addi` and `cincaddr`. This would require more complex linker relaxations to retain good code size and efficiency and we have not yet attempted it.

Relocation	Value	Meaning
`CHERI_COMPARTMENT_CGPREL_HI`	220	20-bit, 11-bit shifted **CGP**-relative displacement for use in `auicgp`.
`CHERI_COMPARTMENT_CGPREL_LO_I`	221	12-bit **CGP**-relative displacement for use in I-type instructions.
`CHERI_COMPARTMENT_CGPREL_LO_S`	222	12-bit **CGP**-relative displacement for use in S-type instructions.
`CHERI_COMPARTMENT_PCCREL_HI`	223	20-bit, 11-bit shifted **PCC**-relative displacement for use in `auipcc`.
`CHERI_COMPARTMENT_PCCREL_LO`	224	12-bit displacement for use in I-type instructions. The displacement is relative to the `auipcc` instruction.
`CHERI_COMPARTMENT_SIZE`	225	The size of the referenced symbol, applied to a `CSetBounds` instruction.

Table 5.1: The relocations defined for the CHERIoT ABI

Chapter 6

Known caveats

There are a small number of known caveats for developers attempting to secure a compartment. Future iterations may have compiler mitigations for some of these.

6.1 Shared stacks

Compartments invoked by the same thread all use the same stack. This is important for embedded systems. Embedded stacks are often on the order of 1 KiB in size and so requiring a separate stack segment for every thread and compartment pair would quickly exhaust memory. This sharing provides one useful tool for an attacker: The caller has access to all of the callee's stack prior to a call.

A malicious compartment may construct a capability to a currently-unused part of the stack, which will become the callee's stack. It cannot stash these on the heap or in globals but it can then pass these as arguments. The compartment switcher will check that these are not passed as direct arguments but a compartment may load a capability and corrupt its own state. For example, consider the following interface:

```
1 struct A
2 {
3     char *outBuffer;
4     size_t length;
5 };
6 __attribute__((cheri_compartment("victim")))
7 void copyOutSomething(const struct A *out);
```

This API takes a structure containing a pointer to a buffer and a length and is expected to write something via this pointer. If the attacker sets up the `outBuffer` field to point to

something on the victim's stack, then the victim may corrupt its own stack. The victim must check this. The `check_pointer` function from cheri.hh validates that pointers do not do this. This function takes a set of permissions that the capability must have and a size (optionally: if unspecified it assumes that the pointer must be big enough for one instance of pointee type) and returns true only if the pointer is not on the current compartment's stack, is tagged, unsealed, and has sufficient permissions.

The following snippet is taken from the implementation of one of the scheduler functions.

```
1 if (!check_pointer<PermissionSet{Permission::Store,
2                                  Permission::LoadStoreCapability}>(ret))
3 {
4     return -EINVAL;
5 }
```

This ensures that the `void **ret` argument can be used to store a capability. If this is not the case, the function returns early.

6.2 Explicit leakage

Passing a pointer to a cross-compartment function call at the C level grants the callee access to the pointee data. Similarly, returning a pointer from a cross-compartment function call grants the caller access to the pointee. The ISA provides several tools for restricting this:

Sealing allows pointers to be made unusable by anyone who lacks the matching permit-unseal capability and unforgeable by anyone who lacks the matching permit-seal capability.

Deep immutability (the permit-load-mutable permission) provides a mechanism for passing data structures between compartments that prevents the recipient from mutating any objects reached via the initial pointer.

Local capabilities (shallowly local, or deeply local capabilities that lack the permit-load-global permission) provide a mechanism to prevent the callee (including any indirect callee) from capturing a pointer.

These tools provide a security benefit only if they are used. It is good practice for any compartment-owned data that is returned to a caller to be sealed. For example, a network stack returning a connection context should return a sealed capability. As a rule of thumb, if you are not writing code that is correct in the presence of every possible bit pattern for a

data structure then pointers to that data structure that are shared outside of a compartment should be sealed.

Any pointer that is `const` should be marked as deeply immutable. This is not done automatically because C and C++ both allow the `const` qualifier to be cast away. Any pointer-typed function argument pointer that is not expected to be captured by the callee should be marked as local.

Future versions of the compiler will provide declarative annotations that implicitly drop these permissions in the caller.

6.3 Availability

In some embedded systems (most control systems), availability is a critical part of the TCB. In such systems, an attacker who can prevent the system from responding can do real-world damage. For example, preventing the brake-control system from engaging the brakes in response to a signal or preventing an emergency cut-out from being delivered can cause injury or loss of life.

In such systems, the scheduler becomes part of the TCB. The scheduler is not trusted for confidentiality or integrity and has a limited interface to the rest of the TCB and so can potentially be replaced by something simpler for these use cases (or something formally verified to ensure that the highest-priority thread will be scheduled in a timely fashion).

Similarly, any lower-priority thread that can reach a compartment that can invoke an interrupts-disabled function is part of the TCB for availability. It is important to carefully audit all such paths to ensure that they will not prevent interrupt delivery for long enough to violate hard realtime guarantees.

Part II

Architecture specification

Chapter 7

The CHERIoT ISA

The CHERIoT ISA extends RV32 [20] with CHERI [22] memory safety features. It is designed to be a very minimal subset of RISC-V and CHERI that supports strong spatial and temporal memory safety, and compartmentalization. This is intended to be a concise description of the architecture and currently assumes some familiarity with both CHERI and RISC-V.

7.1 Starting subset of RV32

The CHERIoT ISA is based on a minimal subset of RV32 supporting machine mode only (i.e. no virtual memory). The C instruction compression extension is required (with minor modifications) and always enabled in the `misa` CSR. PMP support is optional, although we anticipate that it will not add value owing to the strong protections already offered by the CHERIoT ISA and RTOS combination. Floating point is optional and may conflict with some encoding choices for compressed instructions.

7.2 Omitted CHERI features

Those familiar with CHERI-RISC-V will note that some features are omitted to simplify the architecture at the cost of a little flexibility and backwards compatibility. In particular, CHERI hybrid mode is not supported, so there is no need for **DDC** or a cap mode bit in **PCC**. Instead all code runs in pure-cap mode, where existing instructions use capabilities for address operands. Offset addressing is also eliminated, so the `CGetOffset` and `CSetOffset` instructions are not present and special capabilities registers (including **PCC**) are always interpreted simply as their address, rather than as an offset relative to the base

as in CHERI-RISC-V. This allows us to drop checks for the alignment of the base of executable capabilities as there is no possibility of confusion arising from an unaligned base, as there is in hybrid mode CHERI-RISC-V.

7.3 Changes to register file

The 16 32-bit integer registers from RV32 are extended into 65-bit *capabilities* (64-bits + tag). Abstractly, capabilities have the following fields:

address A 32-bit address or integer value.
base The 32-bit inclusive lower bound.
top The 32-bit exclusive upper bound.
perms The capability permissions (Section 7.13.1).
otype Used for sealing (Section 7.13.2).
tag A single bit indicating valid or invalid. Capabilities with this bit set are called *tagged*.

The actual capability encoding is compressed as described in Section 7.13. Instructions that read integer operands use only the lower 32-bits (the capability **address**). Instructions that produce integers write a NULL capability (Section 7.13.6) with the **address** set to the integer result. In assembly the capability registers are referred to as `$c0..$c15`, with `$x0..$x15` referring to their address parts. At reset the registers are initialized to the NULL capability.

7.4 Instruction encodings

As described in Chapter 8 the new capability instructions use major opcode 0x5b and the standard I-type and R-type formats. Additionally `AUICGP` uses major opcode 0x7b.

7.5 Changes to instruction fetch / control flow

The program counter is extended to a capability, **PCC**, with **PCC.address** assuming the role of the **PC**. If **PCC** is untagged, sealed, lacks PERMIT_EXECUTE, or the instruction is not entirely within the bounds of **PCC**, then an exception is raised. Note that the bounds check must take account of whether the fetched instruction is 2 or 4 bytes. Checks on jumps and branches mean the only way for **PCC** to be untagged, sealed or non-executable is after an `MRET` with an invalid `MEPCC` or a trap with an invalid `MTCC`.

CJAL and taken branch instructions check the destination address against the bounds of **PCC**: if the bounds do not permit at least one 2-byte instruction to be loaded from the destination then a capability length violation exception is raised on the jump / branch instruction.

CJALR replaces the JALR instruction and uses capabilities for the target and link register. It checks that the target capability is tagged, unsealed, executable and the address is in bounds before it is installed in **PCC**. If the target is sealed with the reserved 'sentry' type then it is unsealed before jumping to it. The link register, including the current **PCC**, is sealed as a sentry.

7.6 Changes to memory accesses

All existing load and store instructions are modified to take a capability for the base address. The address of the capability is used as the base address for the memory operation and an exception is raised if the capability:

- has the tag unset
- is sealed
- does not have the appropriate memory permissions (Section 7.13.1)
- has bounds that do not cover the entire region being accessed

7.7 Tagged memory

Memory is extended with a single tag bit for each capability sized and aligned memory location. The new capability load and store instructions, CLC and CSC, transfer a capability-sized-and-aligned region of memory to or from a register, including the tag bit. The tag bits are not accessible directly and may be set to one only by a store of a tagged capability using CSC. To prevent tampering with valid capabilities in memory, non-capability stores (e.g. CSW) clear the tag bits for the capability-aligned region(s) they touch. If an unaligned store crosses a capability alignment boundary then two tag bits need to be cleared.

The platform must define which regions of memory support capability tags. In memory without tag support capability stores will silently drop the tag, and capability loads will always return untagged values. The value of the tag bits is undefined at reset, so software should take care to zero all memory on start up unless running on a platform that defines them to be zero.

7.8 Temporal safety

In addition to the capability tag bits there is a revocation bit for each *revocation granule* (currently the same size as a capability, 8 bytes). After loading a capability with the tag set, the `CLC` instruction loads the revocation bit corresponding to the **base** address of the capability (N.B. not the **address**). If the granule's revocation bit is set then the capability's tag is cleared before writing it to the destination register. The revocation bits are memory mapped so that they can be manipulated by the allocator. The platform defines the location of revocation bits in the address space and their mapping to addresses. Not all mapped addresses need have corresponding revocation bits: capabilities whose **base** points to a region of address space without corresponding revocation bits will not be revoked by `CLC`. It is the responsibility of software to allocate capabilities in regions with revocation bits when support for revocation is desired.

A typical implementation is expected to exclude all of the MMIO space from the revocation bitmap. In addition, an implementation with multiple SRAM banks may support revocation only for some granules. Memory used for code and globals will never be revoked (in the software model) and so may be excluded. An implementation may also provide a configuration interface that allows software to specify the range of heap memory and avoid the cost of the load barrier on globals.

The **base** of sealing capabilities (see Figure 7.4) refers to a distinct namespace to that of memory capabilities, therefore they are not revoked using this mechanism. Software should take care not to reallocate sealing capabilities unless there is some other mechanism to revoke previously issued ones. Given the 2^{32} sized space for sealing capabilities we expect most applications will never have to reallocate them.

Like the capability tag bits, the value of the revocation bits in memory is undefined on reset unless defined by the platform.

7.9 Controlling access to system registers

In the absence of supervisor or user modes, it is useful to be able to restrict access to sensitive control and status registers (CSRs) and special capability registers (SCRs). The PERMIT_ACCESS_SYSTEM_REGISTERS permission on executable capabilities can be used to enable or disable access to certain special registers. If PERMIT_ACCESS_SYSTEM_REGISTERS is set on the current **PCC** then access to all registers is permitted, otherwise attempting to access restricted registers or execute `MRET` will cause a PERMIT_ACCESS_SYSTEM_REGISTERS exception. Table 7.1 shows the allowlist of CSRs that can be accessed without PERMIT_ACCESS_SYSTEM_REGISTERS. Similarly Table 7.2 lists the access requirements for special capability registers.

CSR	Read/Write
`cycle(h)`	Read-Only
`time(h)`	Read-Only
`instret(h)`	Read-Only
`hmpcounter(h)`	Read-Only
`fflags`	Read-Write
`frm`	Read-Write
`fcsr`	Read-Write

Table 7.1: CSR allowlist. The accesses shown are the only CSR accesses that are permitted when the installed PCC does not have the PERMIT_ACCESS_SYSTEM_REGISTERS permission bit set.

7.10 Special capability registers

Special Capability Registers (SCRs) are similar to CSRs in that they affect special functions such as exception delivery, except that they contain capabilities rather than integers. SCRs are accessed via a new instruction, `CSpecialRW`, which behaves similarly to the RISC-V `CSRRW` instruction. `CSpecialRW` requires that **PCC** has PERMIT_ACCESS_SYSTEM_REGISTERS, otherwise it will raise an exception.

Some SCRs replace existing RISC-V CSRs. Attempting to access the legacy RISC-V CSR via the `CSR*` instructions results in a Reserved Instruction exception. Any special meaning or behavior associated with the CSR applies to the SCR's address field. For example, the lower two bits of **MTCC.address** select the trap mode, and the remaining bits (including the capability metadata) form the trap base address in the same way as `mtvec`. Some RISC-V CSRs have write-any read-legal (WARL) bits that implicitly modify the written value to restrict the CSR to legal values. This legalization must be applied to the SCR's address when reading or writing an SCR. If this results in the capability becoming unrepresentable then the tag is cleared, as per `CSetAddr`. If a sealed capability is written to an SCR with WARL bits then the tag is cleared, even if the bits would be unchanged by legalization.

Table 7.2 lists the SCRs and their properties: **Reset** indicates the reset value as one of the capability roots defined in Section 7.13.

	Register	Reset	Replaces
28	Machine trap code capability (**MTCC**)	$\top_X$	`mtvec`
29	Machine trap data capability (**MTDC**)	$\top_M$	-
30	Machine scratch capability (**MScratchC**)	$\top_S$	-
31	Machine exception PC capability (**MEPCC**)	$\top_X$	`mepc`

Table 7.2: Special Capability Registers (SCRs)

7.11 Changes to exception handling

Exception handling is the same as RISC-V except that `mtvec` and `mepc` are replaced by their equivalent SCRs. When taking an exception the current **PCC**, with the address set to that of the trapping instruction, is placed in **MEPCC**. **MTCC** is then installed in **PCC** and execution proceeds from the configured trap address according to the usual rules for `mtvec`. When executing an `MRET` instruction **MEPCC** is moved to **PCC** and execution proceeds from **MEPCC.address**.

A new RISC-V exception code, 0x1C, is used for all CHERI specific exceptions, with a more detailed CHERI cause placed in `mtval` as shown in Figure 7.1.

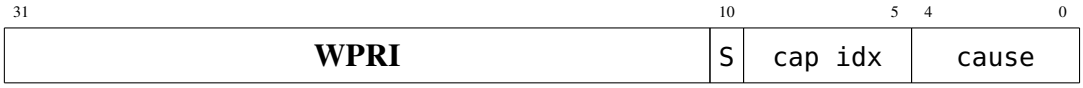


Figure 7.1: `mtval` register format for Capability Exception

cause The `cause` field reports the capability exception code as described in Table 7.3.

cap idx The `cap idx` field reports the index of the capability register that caused the last exception. When the `S` bit is zero, it is the number of the general purpose register that caused the capability fault. Otherwise, it is the number of a special purpose capability register given in Table 7.2 or zero if the fault was caused by **PCC**.

7.12 The AUIPC and AUICGP instructions

The RISC-V `AUIPC` instruction becomes `AUIPCC`, which generates a capability derived from **PCC** by incrementing the address by the 20-bit signed immediate left shifted by 11. Note

Value	Description
0x00	None
0x01	Bounds Violation
0x02	Tag Violation
0x03	Seal Violation
0x11	PERMIT_EXECUTE Violation
0x12	PERMIT_LOAD Violation
0x13	PERMIT_STORE Violation
0x15	PERMIT_STORE_CAPABILITY Violation
0x16	PERMIT_STORE_LOCAL_CAPABILITY Violation
0x18	PERMIT_ACCESS_SYSTEM_REGISTERS Violation

Table 7.3: Capability Exception Codes. All unused codes are *reserved*.

that this shift is reduced by one compared to the `AUIPC` as this allows relocations that combine `AUIPCC` with a 12-bit immediate instruction to always have immediates with matching signs. This is necessary to ensure any intermediate capabilities created are in-bounds otherwise there is a risk they could be unrepresentable. This does limit the maximum range of such relocations, but given our compartmentalization model and expected memory limitations this is not a problem in practice.

Additionally, we use major opcode 0x7b to encode `AUICGP`, which is similar to `AUIPCC` except that the immediate is added to capability register `$c3` (the global pointer in the ABI).

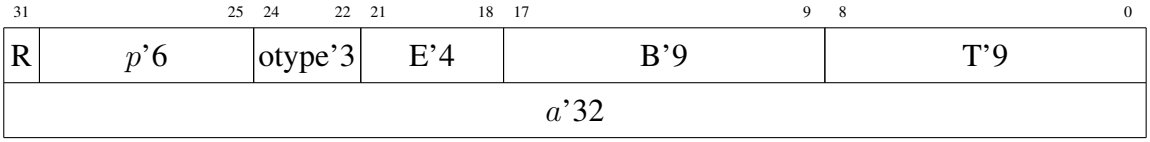
7.13 Capability encoding

Figure 7.2 shows the 64-bit encoding of capabilities which is described in detail in the following sections.

7.13.1 Capability permissions

Figure 7.3 shows the architectural permissions as used by `CGetPerm` and `CAndPerm`. They have the following meanings:

EX If PERMIT_EXECUTE is set then this capability is executable and can be used as the target of `CJALR` and in other contexts requiring an executable capability, such as `TCC`.



R a reserved bit, which is zero in the root capabilities (and hence all tagged capabilities), but may be set if untagged data is loaded into a register. In this case its value must be preserved. This is very important because memory copies are performed with capability load a store instructions in order to preserve the tag on any capabilities present, meaning these instructions must also faithfully copy arbitrary untagged data.

p a 6-bit compressed permissions field (see Section 7.13.1)

otype a 3-bit 'object type' used for sealing capabilities (see Section 7.13.2)

E a 4-bit exponent used for the bounds encoding (see Section 7.13.3)

B a 9-bit base used for the bounds encoding (see Section 7.13.3)

T a 9-bit top used in the bounds encoding (see Section 7.13.3)

a the 32-bit address of the capability

Figure 7.2: Capability format

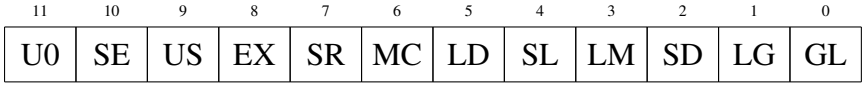


Figure 7.3: Architectural permissions

SR PERMIT_ACCESS_SYSTEM_REGISTERS may be set on executable capabilities. When set in **PCC** access to all CSRs and SCRs is permitted, otherwise attempts to access restricted registers or execute an `MRET` results in an exception (See Section 7.9).

SE If PERMIT_SEAL is set then this capability may be used as the authority for `CSeal`.

US If PERMIT_UNSEAL is set then this capability may be used as the authority for `CUnseal`.

U0 USER_PERM0 is a user permission on capabilities with the sealing format. It has no special meaning to hardware but behaves like other permissions in that it may be cleared by `CAndPerm` and cannot be set after being cleared. It is intended to be used as a software defined permission.

GL If GLOBAL is set then this capability is global and can be stored anywhere, otherwise it is local and may be stored only via capabilities with the PERMIT_STORE_LOCAL_CAPABILITY permission.

SL If PERMIT_STORE_LOCAL_CAPABILITY is set (along with PERMIT_STORE and PERMIT_LOAD_STORE_CAPABILITY) then any capability may be stored via this capability, otherwise attempting to store a capability with GL cleared will result in an exception.

LM If PERMIT_LOAD_MUTABLE is not set then any tagged capabilities loaded via this capability will have SD and LM cleared. Thus, if SD and LM are cleared on a capability then it, and any capability loaded via it (including via indirection), will be read-only. This is useful for delegating a read-only pointer to a data structure, for example to enforce a language level transitive `const`. Untagged or sealed capabilities that are loaded are unaffected and retain their existing SD and LM bits.

LG If PERMIT_LOAD_GLOBAL is not set then any tagged capabilities loaded via this capability will have LG and GL cleared. Thus, if LG and GL are cleared before delegating a capability then it, and any capability loaded via it (including via indirection), may be stored only via capabilities with PERMIT_STORE_LOCAL_CAPABILITY. This limits the ability of the delegee to retain capabilities to a delegated data structure or part thereof, making it easier to later revoke access to the delegated data structure. Note that GL and LG are cleared even on sealed capabilities that are loaded, making this an exception to the immutability of sealed capabilities. This differs from the behavior of LM on sealed capabilities. Untagged capabilities are unaffected.

MC If PERMIT_LOAD_STORE_CAPABILITY is set then the load and store permissions for this capability are modified to enable capability loads (PERMIT_LOAD_CAPABILITY) and / or stores (PERMIT_STORE_CAPABILITY). The `CLC` instruction logically ANDs the tag of the loaded capability with MC from the capability base operand, so only capabilities with MC and LD set can be used to load tagged capabilities. The `CSC` instruction raises an exception if the stored capability has the tag set and the capability base operand lacks either MC or SD permission, so only capabilities with MC

and SD can be used to store tagged capabilities.

SD If PERMIT_STORE is set then this capability can be used as the base operand for stores, otherwise an exception is thrown.

LD If PERMIT_LOAD is set then this capability can be used as the base operand for loads, otherwise an exception is thrown.

Some combinations of permissions are not very useful (e.g. PERMIT_ACCESS_SYSTEM_REGISTERS but not PERMIT_EXECUTE), so permissions are stored in a compressed format that restricts the available combinations. Figure 7.4 shows the different formats of the compressed permission field. Each format has some fixed bits (shown as 0s or 1s) that unambiguously identify the format. A given format unconditionally grants some number of 'implicit' permissions and the non-fixed bits encode the presence or absence of the permissions indicated by the two-letter abbreviation.

For example the 'cap-read-write' format has bits 3 and 4 of the permissions field set to one. Capabilities with this format implicitly have PERMIT_LOAD, PERMIT_LOAD_STORE_CAPABILITY and PERMIT_STORE while bits 0, 1, 2 and 5 encode PERMIT_LOAD_GLOBAL, PERMIT_LOAD_MUTABLE, PERMIT_STORE_LOCAL_CAPABILITY and GLOBAL respectively (the permission is granted if the bit set to one). The logic of this is that each format need only encode permissions that make sense given the set of implicitly present permissions, giving a dense encoding of useful permission encodings. The format used to represent a capability may change if permissions are cleared by `CAndPerm` or `CLC`. Figure 7.5 shows a graphical representation of the possible permissions combinations and possible transitions between them.

One consequence of this encoding is that is not possible to have a single capability with all permissions. Instead there are three *capability roots* corresponding to the three nodes with no edges leading to them in Figure 7.5. We label these as follows:

$\top_M$ The memory root, with GLOBAL, PERMIT_LOAD, PERMIT_STORE, PERMIT_LOAD_STORE_CAPABILITY, PERMIT_STORE_LOCAL_CAPABILITY, PERMIT_LOAD_GLOBAL and PERMIT_LOAD_MUTABLE. The bounds are the entire address space.

$\top_X$ The executable root, with GLOBAL, PERMIT_EXECUTE, PERMIT_LOAD, PERMIT_LOAD_CAPABILITY, PERMIT_LOAD_GLOBAL, PERMIT_LOAD_MUTABLE and PERMIT_ACCESS_SYSTEM_REGISTERS. The bounds are the entire address space.

$\top_S$ the sealing root, with GLOBAL, PERMIT_SEAL, PERMIT_UNSEAL, and USER_PERM0. The bounds are the entire address space even though only a limited set of **otype** values can be used with `CSeal`. This allows sealed, sealing-format capabilities with an address outside the range of valid **otype**s to be used as unforgeable tokens by software.

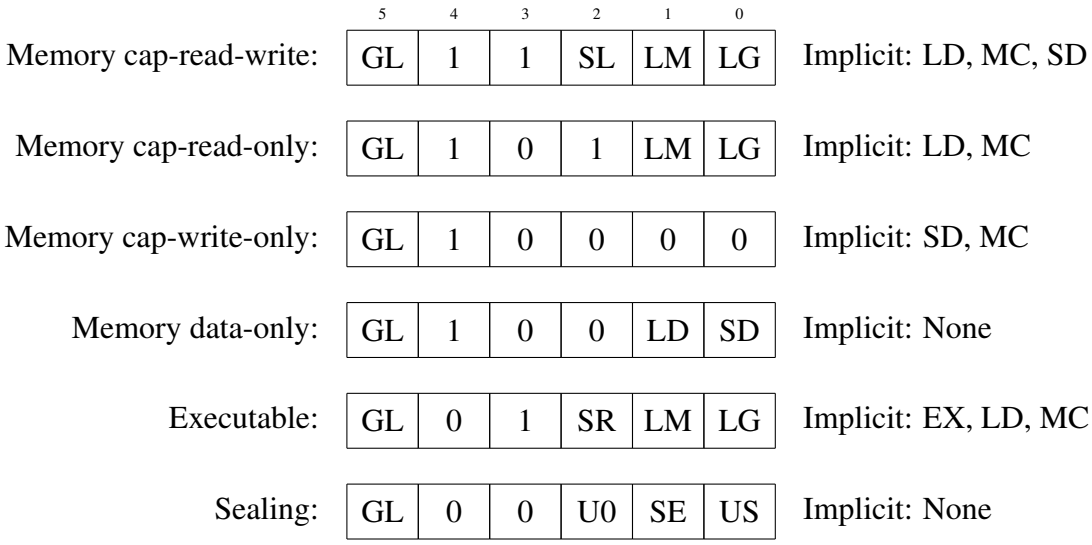


Figure 7.4: Compressed permission formats

On reset the SCRs are initialized to the different capability roots as shown in Table 7.2. **PCC** is initialized to $\top_X$.

See Appendix B for a description of the constraints on useful permission combinations that led to the encoding scheme.

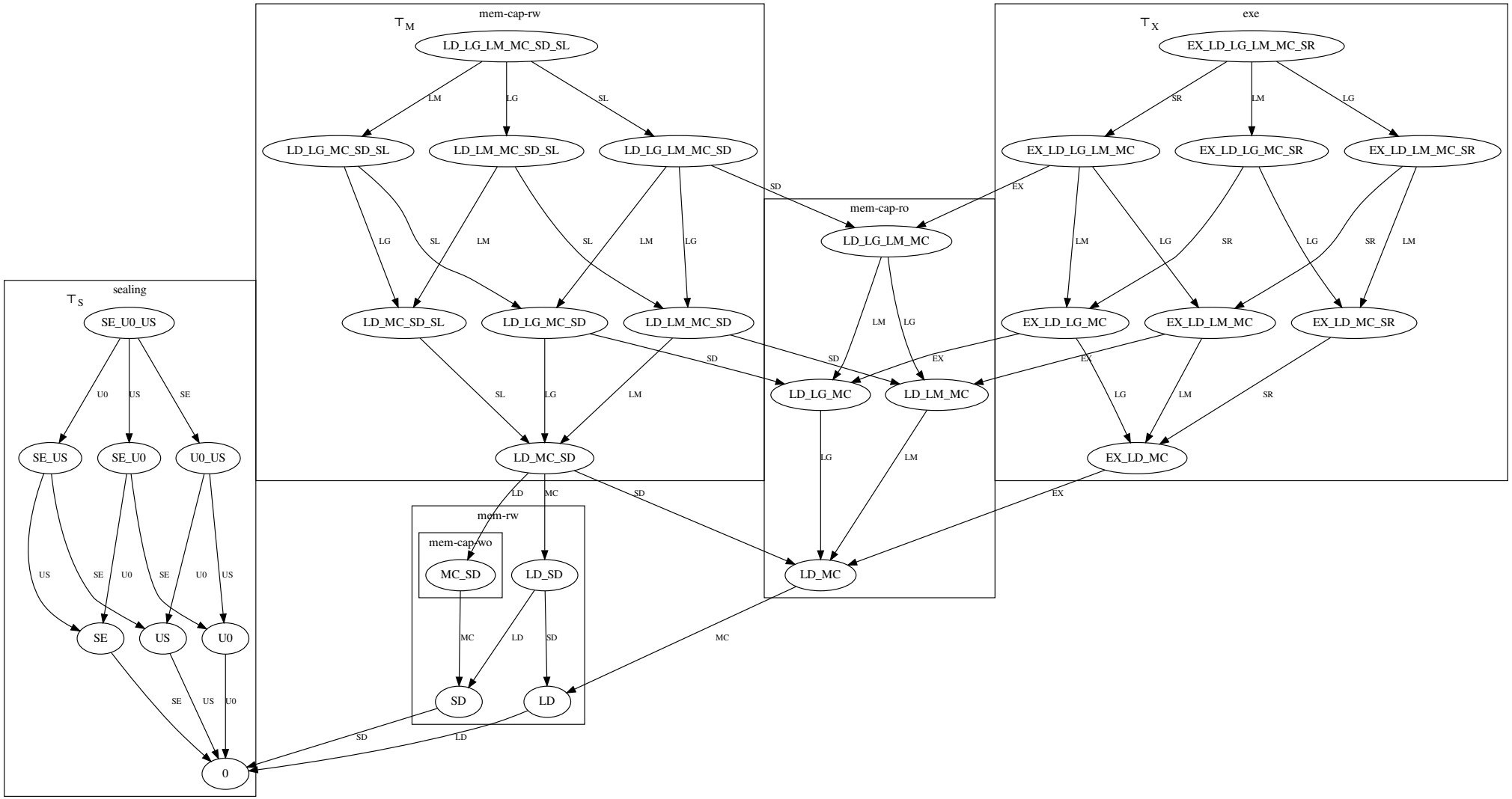


Figure 7.5: Graph of allowed permission combinations, grouped by encoding format and ordered by inclusion. Edges are labelled with the permission that is dropped by that transition. Edges implied by transitivity are omitted. GLOBAL is omitted because it is entirely orthogonal.

CAndPerm operates on the decompressed permissions so it is possible to request combinations that cannot be represented in the compressed encoding (for example PERMIT_EXECUTE but not PERMIT_LOAD). In that case the resulting capability will have a (possibly empty) subset of the requested permissions. The following procedure is used to encode a given set of requested permissions:

1. If the permissions include EX, LD and MC then encode SR, LM and LG using the executable format.
2. Otherwise, if the permissions include LD, MC and SD then encode SL, LM and LG using the cap-read-write format.
3. Otherwise, if the permissions include LD and MC then encode LM and LG using the cap-read-only format.
4. Otherwise, if the permissions include SD and MC then encode using the cap-write-only format.
5. Otherwise, if the permissions include LD *or* SD then encode using the data-only format.
6. Otherwise, encode U0, SE and US using the sealing format.

Any permissions that cannot be encoded using the chosen format are dropped. The possible clearing of GL and LG or SD and LM during capability loads can be quite easily performed on the compressed format although note that clearing SD may require switching format and that SL may be cleared as a side-effect.

Note that this legalization of permissions must happen at all points where permissions can change (CAndPerm and CLC). For example, the result of CAndPerm followed by CGetPerm should be consistent regardless of whether the register file stores the permissions in the compressed or decompressed form. Similarly, storing then loading a capability should not change the permissions except possibly GL, LG, LM and SD as specified by CLC.

7.13.2 Sealed capabilities

The **otype** field is used for *sealing* capabilities. A sealed capability cannot be modified or used as authority for operations except special unsealing ones, but can be passed as a token and later unsealed. Two kinds of sealing are supported:

Using otypes: CSeal allows the creation of sealed capabilities with a given value of **otype** given a capability to seal and an authorizing capability with PERMIT_SEAL and the address set to the desired **otype**. CUnseal permits unsealing a sealed capability if provided with a capability with PERMIT_UNSEAL and bounds that contain the **otype** of the capability to be unsealed.

Sealed entry capabilities (sentries): Executable capabilities sealed with the special *sentry* **otype**s can be used with `CJALR`. The capability is unsealed before jumping to it, creating a form of call gate. Three kinds of sentry are defined that affect `mstatus.MIE` in different ways: either leaving it unchanged, enabling interrupts or disabling interrupts. Jumping to an interrupt enabling or disabling sentry will set or clear `mstatus.MIE` accordingly. Additionally, the link register stored by `CJAL` and `CJALR` is sealed as a sentry with the current interrupt status: if MIE is set it will produce an interrupt enabling sentry and if it is cleared it will produce an interrupt disabling sentry.

The **otype** field uses the following values:

0 unsealed
1 sealed as sentry
2 sealed as interrupt disabling sentry
3 sealed as interrupt enabling sentry
4-5 reserved for use as return sentries in future
6-7 executable capability sealed with given **otype**
8 reserved (due to encoding)
9-15 non-executable capability sealed with given **otype**

The **otype**s $1-7$ can only be applied to executable capabilities, while memory and sealing format capabilities can only be sealed with **otype**s $9-15$. If the **otype** field of a memory or sealing format capability is non-zero then bit 3 is implicitly set i.e. **otype**s $9-15$ are encoded using values $1-7$. An attempt to use `CSeal` or `CUnseal` with a reserved **otype**, or with an **otype** not applicable to the capability format, will clear the capability tag.

7.13.3 Capability bounds

The capability bounds (**base** and **top**) are stored in a compressed format relative to the **address**, similar to CHERI Concentrate [23]. The floating point encoding stores 2^e-aligned bounds, where e is the exponent. An exponent of zero can express bounds with byte precision but limits the maximum length of the range to 511 bytes. Larger exponent values can represent larger ranges, but require more aligned bounds.

To form the **base** and **top** the 9-bit B and T fields from the encoding are inserted into the address at bit e as follows:

address, $a =$	$a_{\text{top}} = a[31 : e+9]$	$a_{\text{mid}} = a[e+8 : e]$	$a_{\text{low}} = a[e-1 : 0]$
base, $b =$	$a_{\text{top}} + c_{\text{b}}$	B	0
top, $t =$	$a_{\text{top}} + c_{\text{t}}$	T	0

where the top bits of the address are 'corrected' according to the following formulae:

$$c_b = \begin{cases} -1, & \text{if } a_{\text{mid}} < B \\ 0, & \text{otherwise} \end{cases} \qquad c_t = \begin{cases} -1, & \text{if } a_{\text{mid}} < B \text{ and } T \geq B \\ 1, & \text{if } a_{\text{mid}} \geq B \text{ and } T < B \\ 0, & \text{otherwise} \end{cases}$$

These corrections ensure that the decoded bounds remain the same provided the address is in $[b, b + 2^{e+9})$, the so-called *representable range*. They work by testing for conditions that indicate whether the **top** and **address** are in the same 2^{e+9} aligned region as **base**. The representable range spans two such regions (one if $B = 0$), which we will call the lower and upper regions, with b always lying in the lower region. The ISA is constructed to ensure that, for valid capabilities, a and t are in the representable range and $b \leq t$. Therefore if $a_{mid} < B$ then a must be in the upper region, where the a_{top} bits are one greater than those bits in b. Similarly if $T < B$ then t must lie in the upper region and we can compute the necessary correction based on whether a also lies in the upper region. To maintain the necessary invariant for this to work `CSetAddr` and `CIncAddr` clear the **tag** if the **address** to goes outside the representable range (see Section 7.13.5).

In order to permit the format to represent a range covering the entire address space using only a 4-bit exponent there is a special case when E has its maximum value. The effective exponent, e, is defined as:

$$e = \begin{cases} 24, & \text{if } E = 15 \\ E, & \text{otherwise} \end{cases}$$

Thus the root memory capability has $B = 0$, $T =$ 0x100, $E = 15$ and decodes to the range $[0, 2^{32})$. Note that the decoded top is actually a 33-bit value to accommodate this.

7.13.4 Set bounds operation

The `CSetBounds` instruction must select values for E, B, and T that encode a requested range defined by a given base, b, and length, l. In the case that the requested range is not precisely representable **base** is rounded down and **top** up to multiples of 2^e, where e is the chosen exponent. For maximum bounds precision, we desire the the smallest e that can represent the requested region. From the encoding we can observe that the largest encodable length for a given e is given by $2^e \times (2^9 - 1)$. Therefore we require a solution

e	alignment, 2^e	maximum length, 511×2^e
0	1	511
1	2	1,022
2	4	2,044
3	8	4,088
4	16	8,176
5	32	16,352
6	64	32,704
7	128	65,408
8	256	130,816
9	512	261,632
10	1,024	523,264
11	2,048	1,046,528
12	4,096	2,093,056
13	8,192	4,186,112
14	16,384	8,372,224
24	16,777,216	8,573,157,376

Table 7.4: Capability bounds alignment and maximum length by exponent value. Note that for $e = 24$ the maximum length exceeds the size of the address space. The length of the root capabilities is $2^{32} = 4,294,967,296$ so no valid capability will ever exceed this length.

to the inequality:

$$\begin{array}{rcll} l & \leq & 2^e \times (2^9 - 1) & \\ l & \leq & 2^{e+9} - 2^e & \\ l & \leq & 2^{e+9} & \text{approximation} \\ \log_2 l & \leq & e + 9 & \\ \lfloor \log_2 l \rfloor & \leq & e + 9 & \text{approximation} \\ \mathrm{msb}(l) & \leq & e + 9 & \text{index of most significant set bit is floor of } \log_2 \\ 32 - \mathrm{clz}(l) & \leq & e + 9 & \text{also expressible as count leading zeros for 32-bit length} \\ 23 - \mathrm{clz}(l) & \leq & e & \\ e & = & 23 - \mathrm{clz}(l) & \text{since we require the smallest e} \end{array}$$

Since e must be greater than or equal to zero the count leading zeros should be limited to the top 23 bits of l (lengths smaller than 9-bits are expressed with $e = 0$). Since the exponent is limited to 4-bits, exponents greater than 14 are mapped to the special maximum exponent, 24, which is encoded as 15:

$$e' = \begin{cases} e, \text{ if } e \leq 14 \\ 24, \text{ otherwise} \end{cases}$$

Having chosen the exponent, the relevant bits of **base** and **top**, $t = b + l$, are extracted:

$$B = b[e' + 9 : e'] \qquad T = t[e' + 9 : e']$$

The bounds are exact if the bits below e' in both b and t are all zero. By discarding the lower bits of b the **base** is automatically rounded down to a representable value, but if the top is not exact then we must round it up by incrementing by one to ensure the encoded range includes the requested top. Note that the calculated e' was based on the requested length, but having rounded the bounds the resulting length may be larger and may exceed the maximum representable length for e'. To check for this we calculate the encoded length $T - B$ (in units of 2^e), and compare it with the maximum encodable length, $2^9 - 1$. Note that B and T above are one bit wider than the encoding can store for this purpose. If the maximum encodable length is exceeded we increment e by one and recompute B and T, this time with a guarantee that the resulting length is encodable. Finally, the oversized B and T can drop their extra most significant bit in the final encoding.

7.13.5 Representability checks

To enable capabilities to be used to implement pointers in the C language the capability encoding is designed to allow the **address** to vary within a limited range without changing the decoded bounds. The *representable range* of a capability is the set of addresses

for which the decoded bounds remain the same. We also wish to maintain the *monotonicity* invariant that the bounds of a valid capability must be a subset of the bounds of the valid capability from which it is derived. Therefore, whenever the **address** of a capability changes the hardware must check whether the new address remains within the representable range, otherwise the new bounds would violate this invariant. For example, if `CSetAddr` or `CIncAddr` detects that the new **address** is outside of the representable range then the **tag** of the result is cleared.

The representation guarantees that the bounds remain decodable provided the address, a, and base, b, satisfy $b \leq a$ and $a < b + 2^{e+9}$. Additionally, if e is 24 then all addresses are representable. The representable range always includes **top**, although in some cases it is the highest representable address. Therefore the representation meets the C-language requirement that pointers may range within object bounds or 'one byte past the end'. Other CHERI implementations include much larger representable ranges than this minimum in order to accommodate common C programming practices. However, this comes at the cost of bits in the representation and our experience so far has shown that it is not necessary for embedded systems.

The following instructions all set the capability **address** and therefore require a representability check:

- `AUIPCC`
- `AUICGP`
- `CSetAddr`
- `CIncAddr`

Note that although `CJAL` and `CJALR` also set the address on the link register, it is guaranteed to be representable because its **address** can be at most equal to **PCC**.**top** given that the jump itself is in bounds. Therefore no representability check is required for these instructions.

Similarly, the value placed in **MEPCC** on exception should always be representable given that **PC** is always in bounds (or equal to **PCC**.**top** in the case of stepping off the end of **PCC**). One exception to this is if **MTCC** is configured in vectored mode and a subsequent exception goes to a **PC** that is out of the bounds of **MTCC**. This would cause a **PCC** bounds exception and in this case **MEPCC** might not be representable, in which case its tag should be cleared. It may be preferable not to support vectored mode, although note that care should also be taken when legalizing `mtvec` (**MTCC**.**address**) to ensure that this does not violate sealing or representability. legalization of `mepc` (clearing the least significant bit) may also cause values read from **MEPCC** to be unrepresentable if it has been written with an unaligned address. This includes the implicit read by `MRET`. In these cases the unrepresentable **MEPCC** that results from the **PCC** bounds exception should

have its tag cleared.

7.13.6 The NULL capability

The NULL capability is defined as an untagged capability with an **address** of zero and an encoding of all zeroes. This definition is for maximum compatibility with the C language, where it is used to represent NULL pointers. The NULL capability is also used as the value of the `$c0` register and to store integer results by setting the **address** to the required value. Although capability fields other than the **address** are not meaningful on untagged capabilities they may be queried using the `CGetX` instructions. Thus it can be observed that the NULL capability decodes as an untagged, unsealed capability, with no permissions, **base** 0 and **length** 0[1]. Note that NULL-derived capabilities with a non-zero address may have non-zero **base** and **top**, but will be untagged.

7.13.7 Zero length capabilities

The capability encoding described supports capabilities with zero length, where **base** is equal to **top**. Such capabilities do not authorize access to any memory (or sealing rights), so it may be tempting to use them as unforgeable tokens (e.g. to implement file handles), however they come with a big drawback: zero length capabilities can be derived with **base** equal to the **top** of an existing capability, even though that capability does not authorize access to **top**. To give an example of this suppose a memory allocator gives out two capabilities with adjacent ranges $[a, b)$ and $[b, c)$. Later it may receive a call to 'free' with a zero length capability $[b, b)$ and it has no way to tell which of the two ranges it was derived from. If it relies only on the **base** of the capability and does not validate the length the allocator may incorrectly free $[b, c)$. The same problem arises during revocation sweeps as performed by Cornucopia [9], meaning it is unable to revoke zero length capabilities. Therefore we strongly discourage the use of zero length capabilities and encourage validating the length of untrusted capabilities. As an alternative we suggest using capabilities of length one derived from the sealing root but without PERMIT_SEAL or PERMIT_UNSEAL. In this case USER_PERM0 may be used as a software defined permission.

[1] On other CHERI architectures the NULL capability is defined to have maximum length. This could be achieved by tweaking the encoding (e.g. by inverting the encoded exponent and making the zero value a special case), but there is no clear advantage to doing this.

7.13.8 Zero permission capabilities

In a similar vein to zero length capabilities the encoding also supports capabilities with no permissions. These are encoded using the sealing format but may be derived from any of the roots. We therefore caution against using zero permission capabilities as tokens, because the ability to derive the same capability from either a memory or a sealing root may break the expected unforgeability property. They may also behave unexpectedly with respect to revocation, since sealing capabilities are not subject to revocation but memory capabilities are. Given these limitations it is not clear that zero permission capabilities should be allowed at all and support may be removed in future revisions.

7.13.9 Capability layout in memory

While Figure 7.2 shows the nominal capability format, for microarchitectural reasons it may be preferable for the capability fields to appear in a different arrangement in memory. Future versions of the architecture may also specify a different capability format. Therefore software should not rely on the exact layout of capabilities in memory. This said, we expect that certain software will need knowledge of the capability format. For example, memory allocators and the toolchain will need to be aware of alignment requirements and a debugger will need to be able to decode capabilities from memory dumps.

7.13.10 Sail implementation

Appendix A contains Sail code implementing the capability encoding described here as well as properties validated using SMT.

7.14 Instruction compression

Compressed load and store extensions from the standard RV32 C extension decompress to their capability equivalents. Similar to the encodings of `CLC` and `CSC` in uncompressed instructions, the C extension uses compressed `C.LD` and `C.SD` from RV64 to encode `CLC` and `CSC`. In RV32 these opcodes are used to encode `C.FLW` and `C.FSW`, meaning these compressed encodings for floating point loads and stores are no longer available. Note that this applies to C instructions with implicit stack operands, that is we use `C.LDSP` (RV64) and `C.SDSP` (RV64) as capability loads and stores relative to the stack capability, replacing the `C.FLWSP` (RV32) and `C.FSWSP` (RV32) encodings. Such a decision is justified because capability loads and stores greatly outnumber floating point instructions in embedded code, and most devices this ISA is targeting do not have a floating point unit at all.

Implicit stack pointer arithmetic instructions are not useful with CHERI, as adding an offset with `ADD` will produce an untagged integer. These instructions are modified to decode to `CIncAddr` to produce valid stack-derived capabilities. As a result, `C.ADDI16SP imm` is decoded into `CIncAddr $csp, $csp, imm` and `C.ADDI4SPN $rd, imm` into `CIncAddr $cd, $csp, imm`.

The CHERIoT ISA introduces changes to mappings between certain compressed and uncompressed instructions, but no changes to the encodings of compressed instructions themselves. This translates to minimum logic modifications when adding CHERIoT ISA support to an existing RISC-V CPU. However, experiments in Appendix D show that further code size reduction can be achieved by introducing changes in the encoding themselves, to accommodate RV32E and CHERI instructions.

Chapter 8

Instruction encoding summary

8.1 Primary new instructions

The RISC-V specification reserves 4 major opcodes for extensions: 11 (0xb / 0b0001011), 43 (0x2b / 0b0101011), 91 (0x5b / 0b1011011), and 123 (0x7b / 0b1111011). The proposed CHERI encodings use major opcode 0x5b for all capability instructions.

All register-register operations use the RISC-V R-type or I-type encoding formats.

8.1.1 Capability-Inspection Instructions

31 25	24 20	19 15	14 12	11 7	6 0	
0x7f	0x0	cs1	0x0	rd	0x5b	`CGetPerm` rd, cs1
0x7f	0x1	cs1	0x0	rd	0x5b	`CGetType` rd, cs1
0x7f	0x2	cs1	0x0	rd	0x5b	`CGetBase` rd, cs1
0x7f	0x3	cs1	0x0	rd	0x5b	`CGetLen` rd, cs1
0x7f	0x4	cs1	0x0	rd	0x5b	`CGetTag` rd, cs1
0x7f	0xf	cs1	0x0	rd	0x5b	`CGetAddr` rd, cs1
0x7f	0x17	cs1	0x0	rd	0x5b	`CGetHigh` rd, cs1
0x7f	0x18	cs1	0x0	rd	0x5b	`CGetTop` rd, cs1

8.1.2 Capability-Modification Instructions

31 25	24 20	19 15	14 12	11 7	6 0	
0xb	cs2	cs1	0x0	cd	0x5b	`CSeal` cd, cs1, cs2
0xc	cs2	cs1	0x0	cd	0x5b	`CUnseal` cd, cs1, cs2
0xd	rs2	cs1	0x0	cd	0x5b	`CAndPerm` cd, cs1, rs2
0x10	rs2	cs1	0x0	cd	0x5b	`CSetAddr` cd, cs1, rs2
0x11	rs2	cs1	0x0	cd	0x5b	`CIncAddr` cd, cs1, rs2
imm[11:0]		cs1	0x1	cd	0x5b	`CIncAddrImm` cd, cs1, imm
0x8	rs2	cs1	0x0	cd	0x5b	`CSetBounds` cd, cs1, rs2
0x9	rs2	cs1	0x0	cd	0x5b	`CSetBoundsExact` cd, cs1, rs2
uimm[11:0]		cs1	0x2	cd	0x5b	`CSetBoundsImm` cd, cs1, uimm
0x16	rs2	cs1	0x0	cd	0x5b	`CSetHigh` cd, cs1, rs2
0x7f	0xb	cs1	0x0	cd	0x5b	`CClearTag` cd, cs1

`CSetBoundsExact` may not be required.

8.1.3 Pointer-Arithmetic Instructions

31 25	24 20	19 15	14 12	11 7	6 0	
0x14	cs2	cs1	0x0	rd	0x5b	`CSub` rd, cs1, cs2
0x7f	0xa	cs1	0x0	cd	0x5b	`CMove` cd, cs1

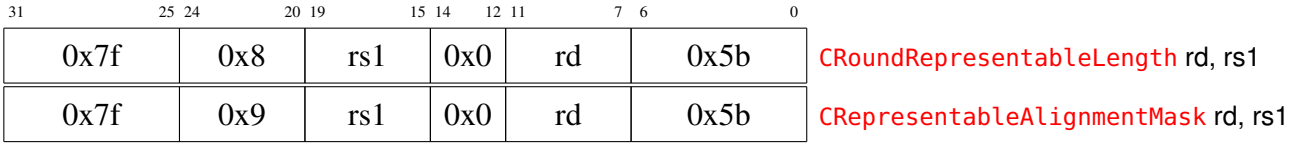
8.1.4 Pointer-Comparison Instructions

31 25	24 20	19 15	14 12	11 7	6 0	
0x20	cs2	cs1	0x0	rd	0x5b	`CTestSubset` rd, cs1, cs2
0x21	cs2	cs1	0x0	rd	0x5b	`CSetEqualExact` rd, cs1, cs2

8.1.5 Special Capabilty Register Access Instructions

31 25	24 20	19 15	14 12	11 7	6 0	
0x1	scr	cs1	0x0	cd	0x5b	`CSpecialRW` cd, scr, cs1

8.1.6 Adjusting to Compressed Capability Precision Instructions



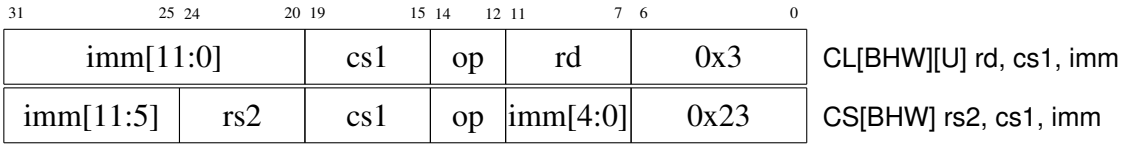
8.2 Modifications to existing RISC-V instructions

8.2.1 Control-Flow Instructions

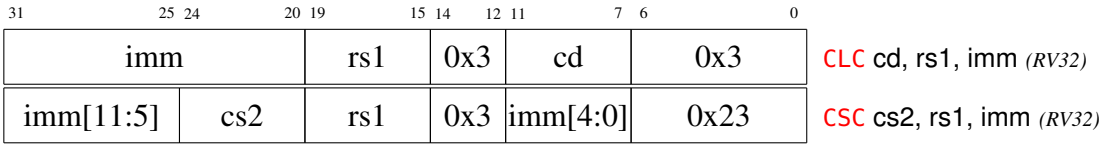
No special new control flow instructions are added, although RISC-V `JAL` / `JALR` become capability instructions `CJAL` / `CJALR`. The branch instructions also check that **PCC** permits at least one 2-byte instruction to be loaded from the target address, otherwise they raise a capability bounds exception.

8.2.2 Memory-Access Instructions

The standard RV32 load and store instructions are modified to take a capability as the base address:



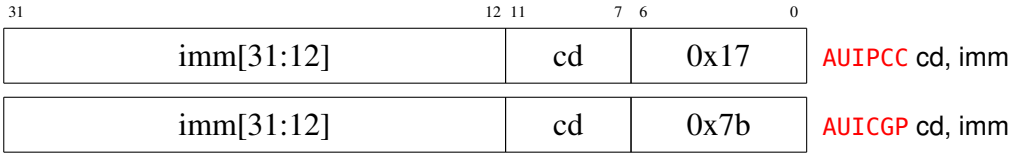
The RV64 instructions `LD` and `SD` are reused to behave as load capability (`LC`) and store capability (`SC`) respectively:



8.2.3 Address Construction Instructions

The `AUIPC` instruction is replaced by `AUIPCC`, which derives capabilities from **PCC**. Our ABI also requires a new instruction, `AUICGP`, that is similar to `AUIPCC` but derives from `$c3`

(`$cgp`). This required allocating a new major opcode, although we expect that further support for linker relaxation may remove the need for `AUICGP`.



8.3 Encoding Summary

The CHERIoT ISA shares encodings with CHERI-RISC-V. The general-purpose instructions use the 0x5b major opcode and use the RISC-V R-type or I-type encoding formats. CHERI-RISC-V uses the funct3 field from bits 14-12 as a top-level opcode, and funct7 as a secondary opcode for standard 3-register operand instructions. Two-register operand instructions and single-register operand instructions are a subset of the 3-register operand encodings.

Top-level encoding allocation (funct3 field)

	00	01	10	11
0	Two Source & Dest	CIncAddrImm	CSetBoundsImm	-
1	-	-	-	-

Two Source & Dest encoding allocation (funct7 field)

All three-register-operand (two sources, one destination) CHERI-RISC-V instructions use the RISC-V R-type encoding format, with the same funct field stored in funct7 and a 0 value in funct3.

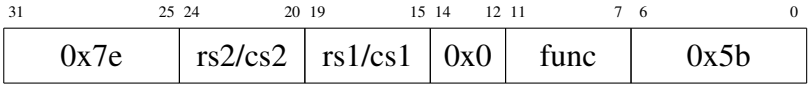
func	rs2/cs2	cs1	0x0	cd	0x5b

	00	01	10	11
00000	-	CSpecialRW	-	-
00001	-	-	-	-
00010	CSetBounds	CSetBoundsExact	-	CSeal
00011	CUnseal	CAndPerm	-	-
00100	CSetAddr	CIncAddr	-	-
00101	CSub	-	CSetHigh	-
00110	-	-	-	-
00111	-	-	-	-
01000	CTestSubset	CSEQX	-	-
01001	-	-	-	-
01010	-	-	-	-
01011	-	-	-	-
01100	-	-	-	-
01101	-	-	-	-
01110	-	-	-	-
01111	-	-	-	-
10000	-	-	-	-
10001	-	-	-	-
10010	-	-	-	-
10011	-	-	-	-
10100	-	-	-	-
10101	-	-	-	-
10110	-	-	-	-
10111	-	-	-	-
11000	-	-	-	-
11001	-	-	-	-
11010	-	-	-	-
11011	-	-	-	-
11100	-	-	-	-
11101	-	-	-	-
11110	-	-	-	-
11111	Stores	Loads	Two Source	Source & Dest

†Reserved for future use.

Two Source encoding allocation (rd field)

There are currently no two source instructions but they would be of the following form:

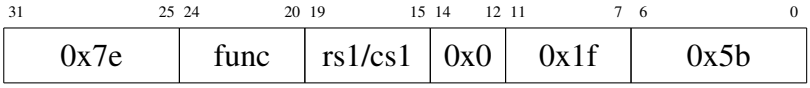


	00	01	10	11
000	-	-	-	-
001	-	-	-	-
010	-	-	-	-
011	-	-	-	-
100	-	-	-	-
101	-	-	-	-
110	-	-	-	-
111	-	-	-	One Source

†Reserved for future use.

One Source encoding allocation (rs2 field)

There are currently no one source instructions but they would be of the following form:

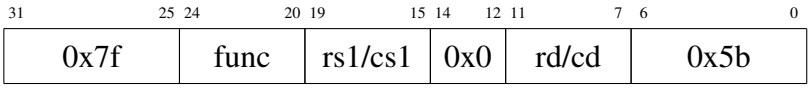


	00	01	10	11
000	-	-	-	-
001	-	-	-	-
010	-	-	-	-
011	-	-	-	-
100	-	-	-	-
101	-	-	-	-
110	-	-	-	-
111	-	-	-	-

†Reserved for future use.

Source & Dest encoding allocation (rs2 field)

Source & Dest instructions are of the following form:

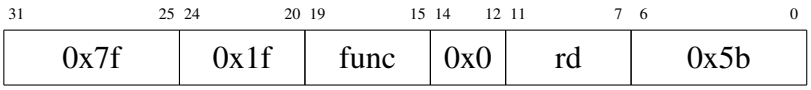


	00	01	10	11
000	CGetPerm	CGetType	CGetBase	CGetLen
001	CGetTag	-	-	-
010	CRRL	CRAM	CMove	CClearTag
011	-	-	-	CGetAddr
100	-	-	-	-
101	-	-	-	CGetHigh
110	CGetTop	-	-	-
111	-	-	-	Dest-Only

†Reserved for future use.

Dest-Only encoding allocation (rs1 field)

We do not currently have any one-register-operand instructions, but any future dest-only instructions will be of the following form:



	00	01	10	11
000	-	-	-	-
001	-	-	-	-
010	-	-	-	-
011	-	-	-	-
100	-	-	-	-
101	-	-	-	-
110	-	-	-	-
111	-	-	-	-

Chapter 9

Instruction reference

In this chapter, we specify each instruction via both informal descriptions and code in the Sail language. To allow for more succinct code descriptions, we rely on a number of common function definitions and constants also described in this chapter.

9.1 Sail language used in instruction descriptions

The instruction descriptions contained in this chapter are accompanied by code in the Sail language [4, 18] taken from the CHERIoT Sail implementation [15], which is derived from the CHERI-RISC-V Sail implementation [5]. Sail is a domain specific imperative language designed for describing processor architectures. It has a compiler that can output executable code in OCaml or C for building executable models, and can also translate to various theorem prover languages for automated reasoning about the ISA. The following is a brief description of the Sail language features used in document. For a full description see the Sail language documentation [18].
Types used in Sail:

- `int` Sail integers are of arbitrary precision (therefore there are no overflows) but can be constrained using simple first-order constraints. As a common case integer range types can be defined using `range(a,b)` to indicate an integer in the range a to b inclusive. Operations on integers respect the constraints on their operands so, for example, if `x` and `y` have type `range(a, b)` then `x + y` has type `range(a + a, b + b)`. Integer literals are written in decimal.
- `bits(n)` is a bit vector of length `n`. Vectors are indexed using square bracket notation with index 0 being the least significant bit. Arithmetic and logical operations on vectors are defined on two vectors of equal length producing a result of the same

length and truncating on overflow. Where signedness is significant it is indicated in the operator name, for example `<_s` performs signed comparison of bit vectors . Bit vector literals are written in hexadecimal for multiples of four bits or in binary with `0x` or `0b` prefixes, e.g. `0x3` means '0011' and `0b11` means '11'. The at symbol, `@`, indicates concatenation of vectors.

- `structs` are similar to C structs with named, typed fields accessed with a dot as in `struct_val.field_name`. Struct copying with field updates is also supported as in `{struct_val with field_name=new_val}`.
- Registers in Sail contain the architectural state that is modified by instruction execution. By convention register names in the CHERI specification start with a capital letter to distinguish them from local variables. Sail also supports a form of 'assignment' to function calls as in `wGPR(rd)= result`. This is just syntactic sugar for an extra argument to the function call. This syntax is used by functions that write registers or memory and have special behavior such as `wGPR`, `writeCapReg` and `MEMw`.

The following operators and expression syntax are used in the Sail code:

- Boolean operators: `not`, `|` (logical OR), `&` (logical AND), `^` (exclusive OR)
- Integer operators: `+` (addition), `-` (subtraction), $*$ (multiplication), `%` (modulo)
  Sail operations on integers are the usual mathematical operators. Note `a % b` is the modulo operator that, for $b > 0$ returns a value in the range 0 to $b - 1$ regardless of the sign of a. Although Sail integers are notionally infinite in range, CHERI instructions can be implemented with finite arithmetic.
- Bit vector operators: `&` (bitwise AND), `<_s` (signed less than), `@` (bit vector concatenation)
- Equality: `==` (equal), `!=` (not equal)
- Vector slice:
  `v[a..b]`
  Creates a sub-range of a vector from index a down to b inclusive.
- Local variables:
  `mutable_var = exp;`
  `let` `immutable_var = exp;`
  Mutable variables are introduced by simply assigning to them (optionally prefixed with keyword `var`). An explicit type may be given following a colon, but types can usually be inferred. Sail supports mutable or immutable variables where immutable ones are introduced by **`let`** and assigned only once when created.
- Functional if:
  `if` `cond` **`then`** `exp1` **`else`** `exp2`
  May return a value, similar to C ternary operator.

- Foreach loop:
```
foreach(i from start_exp to end_exp) {
  body
};
```
- Function invocation:
```
func_id (arg1, arg2)
```
- Field selection from struct:
```
struct_val.field
```
Returns the value of the given field from structure.
- Functional update of structure:
```
{struct_val with field=exp}
```
A copy of the structure with the named field replaced with another value.

9.2 Constant Definitions

The following constants are used in various type and function definitions throughout the specification.

```
type xlen       : Int = 32
type cap_addr_width : Int = xlen
```
Cap length width is one larger than address space width in order to represent maximum top / length of 2**xlen.
```
type cap_len_width : Int = cap_addr_width + 1
```
Width of expanded exponent field in bits
```
type cap_E_width : Int = 5
```
Width of compressed exponent field in bits
```
type cap_cE_width : Int = 4
```
Exponent chosen to permit representing the entire address space.
```
type cap_max_E : Int = 24
type cap_size : Int = 8
```
Width of T and B field in bits
```
type cap_mantissa_width : Int = 9
```
Width of the architectural perms returned by CGetPerms.
```
type cap_perms_width : Int = 12
```
Width of compressed perms field in bits
```
type cap_cperms_width : Int = 6
```
Width of architectural otype field in bits
```
type cap_otype_width : Int = 4
```
Width of compressed otype field in bits

```
type cap_cotype_width : Int = 3
```
There is one revocation bit for every 8 bytes.
```
type log2_revocation_granule_size : Int = 3
```

9.3 Function Definitions

This section contains descriptions of convenience functions used by the Sail code featured in this chapter.

Functions for integer and bit vector manipulation

The following functions convert between bit vectors and integers and manipulate bit vectors:

```
unsigned : forall 'n. bits('n) -> range(0, 2 ^ 'n - 1)
```
converts a bit vector of length n to an integer in the range 0 to $2^n - 1$.

```
signed : forall 'n, 'n > 0. bits('n) -> range(- (2 ^ ('n - 1)), 2 ^ ('n - 1) -
    1)
```
converts a bit vector of length n to an integer in the range -2^{n-1} to $2^{n-1} - 1$ using twos-complement.

```
to_bits : forall 'l, 'l >= 0. (int('l), int) -> bits('l)
```

```
bool_to_bit : bool -> bit
```
`bool_to_bit` converts a boolean to a bit in the conventional way by mapping `true` to `bitone` and `false` to `bitzero`.

```
bool_to_bits : bool -> bits(1)
```

```
truncate : forall 'm 'n, ('m >= 0 & 'm <= 'n). (bits('n), int('m)) -> bits('m)
```
`truncate(v, n)` truncates `v`, keeping only the *least* significant `n` bits.

```
truncateLSB : forall 'm 'n, ('m >= 0 & 'm <= 'n). (bits('n), int('m)) -> bits('
    m)
```
`truncateLSB(v, n)` truncates `v`, keeping only the *most* significant `n` bits.

```
pow2 : forall 'n. int('n) -> int(2 ^ 'n)
```

```
align_down : forall 'n 'm, ('n >= 1 & 'm > 'n). (int('n), bits('m)) -> bits('m)
```
align_down(n, bv) returns the given bit vector, bv, aligned down to a power of two by clearing the least significant n bits.

```
EXTZ
```

Adds zeros in most significant bits of vector to obtain a vector of desired length.

`EXTS`
Extends the most significant bits of vector preserving the sign bit.

`zeros`
Produces a bit vector of all zeros

`ones`
Produces a bit vector of all ones

Types used in function definitions

```
type CapBits = bits(8 * cap_size)
type CapAddrBits  = bits(cap_addr_width)
type CapLenBits   = bits(cap_len_width)
type CapPermsBits = bits(cap_perms_width)
```

Many functions also use `struct Capability`, a structure holding a partially-decompressed representation of CHERI capabilities.

Functions for reading and writing register and memory

```
C(n) : regno -> Capability
C(n) : (regno, Capability) -> unit
```

The overloaded function `C(n)` is used to read or write capability register `n`.

```
X(n) : regno -> xlenbits
X(n) : (regno, xlenbits) -> unit
```

The overloaded function `X(n)` is used to read or write integer register `n`.

```
mem_read_cap : (xlenbits, bool, bool, bool) -> MemoryOpResult(Capability)
```

`mem_read_cap`(addr, aq, rl, res) reads a capability from the `cap_size` aligned `addr` and returns either a `MemValue` with the partially decoded `Capability` or a `MemException` indicating failure. `aq`, `rl` and `res` are flags controlling relaxed memory ordering and atomic memory access (not used on single-core MCU).

```
mem_read_cap_revoked : xlenbits -> bool
```

`mem_read_cap_revoked`(addr) reads the revocation bit corresponding with the revocation granule aligned address.

```
mem_write_ea_cap : (xlenbits, bool, bool, bool) -> MemoryOpResult(unit)
```

`mem_write_ea_cap`(addr, aq, rl, res) announces the effective address for an intended capability write to the `cap_size` aligned address, `addr`. It returns either `MemValue(unit)` for succeess, or `MemException(e)` indicating an exception, `e`. `aq`, `rl` and `res` are flags controlling relaxed memory ordering and atomic accesses (not used on single-core MCU).

The non-exception case should be followed by a call to `mem_write_cap` with the value to be written. The write is split between address announcement and value write in order to permit certain relaxed memory ordering behaviours when integrated with the RMEM concurrency tool, but the address annoucement may be ignored in sequential execution.

```
mem_write_cap : (xlenbits, Capability, bool, bool, bool) -> MemoryOpResult(bool
    )
```

`mem_write_cap`(addr, cap, aq, rl, res) writes the Capability value `cap` (including tag) to the `cap_size` aligned address, `addr`. It returns either `MemValue(bool)`, indicating success (with store-conditional result) or `MemException(e)` in case of exception `e`. `aq`, `rl` and `res` are flags controlling relaxed memory ordering and atomic accesses (not used on single-core MCU).

Functions for ISA exception behavior

```
handle_exception : ExceptionType -> unit
```

```
handle_illegal : unit -> unit
```

```
handle_mem_exception : (xlenbits, ExceptionType) -> unit
```

```
handle_cheri_cap_exception : (CapEx, capreg_idx) -> unit
```

Causes the processor to raise a capability exception by writing the given capability exception cause and a 6-bit register number to the xtval register then signalling an exception. If bit 5 of the register number is set then bits 0-4 refer to the index of a special capability register, otherwise they refer to a general purpose register.

```
handle_cheri_reg_exception : (CapEx, regidx) -> unit
```

Causes the processor to raise a capability exception by writing the given capability exception cause and register number to the xtval register then signalling an exception. The register number refers to a general purpose register, so will be prepended with a zero bit.

```
min_instruction_bytes : unit -> CapAddrInt
```

```
legalize_epcc : Capability -> Capability
```

```
legalize_tcc : (Capability, Capability) -> Capability
```

Functions for manipulating capabilities

The Sail code abstracts the capability representation using the following functions for getting and setting fields in the capability. The base of the capability is the address of the first byte of memory to which it grants access and the top is one greater than the last byte, so the set of dereferenceable addresses is:

$$\{a \in \mathbb{N} \mid base \leq a < top\}$$

Note that the capability format can encode top of 2^{32}, meaning the entire 32-bit address space can be addressed.

```
getCapBounds : Capability -> (CapAddrInt, CapLen)
```

```
getCapBaseBits : Capability -> CapAddrBits
```

```
getCapTop : Capability -> CapLen
```

```
getCapLength : Capability -> CapLen
```

```
inCapBounds : (Capability, CapAddrBits, CapLen) -> bool
```
`inCapBounds`(cap, addr, len) checks whether the capability cap includes the region specified by addr and len. Specifically it checks whether cap.base $\leq$ addr AND addr + len $\leq$ cap.top. Note that this definition returns true if len is zero and addr $=$ cap.top.

The following functions adjust the bounds and address of capabilities. Not all combinations of bounds and address are representable, so these functions return a boolean value indicating whether the requested operation was successful. Even in the case of failure a capability is still returned, although it may not preserve the bounds of the original capability.

```
setCapBounds : (Capability, CapAddrBits, CapAddrBits) -> (bool, Capability)
```

```
setCapAddr : (Capability, CapAddrBits) -> (bool, Capability)
```
Returns the given capability with the address set to the given value and a boolean indicating whether the new address is within representable range (decoded bounds remain the same).

```
incCapAddr : (Capability, CapAddrBits) -> (bool, Capability)
```
Returns the given capability with the address incremented by the given value and a boolean indicating whether the new address is within representable range (decoded bounds remain the same).

```
getRepresentableAlignmentMask : xlenbits -> xlenbits
```
For a given length computes the exponent, e, that would be used to represent a capability of that length and returns a mask consisting of e zeros in the least significant bits and ones in the upper bits.

```
getRepresentableLength : xlenbits -> xlenbits
```
For a given length, l, returns a value greater than or equal to l that will result in a precisely representable capability when used with a base that is suitably aligned as per `getRepresentableAlignmentMask`.

```
sealCap : (Capability, bits(cap_otype_width)) -> Capability
```

```
unsealCap : Capability -> Capability
```

`isCapSealed : Capability -> bool`
Returns whether the given capability is sealed i.e. whether its otype is zero.

`clearTag : Capability -> Capability`
Returns the given capability with the tag set to false.

`clearTagIf : (Capability, bool) -> Capability`
Returns the given capability with the tag cleared if the second argument is true, otherwise returns it with the original tag value preserved.

`clearTagIfSealed : Capability -> Capability`
Returns the given capability with the tag cleared if the capability is sealed, otherwise returns it with the original tag value preserved.

The architectural permissions as described in Section 7.13.1 are accessed using the following functions:

```
getCapPerms : Capability -> CapPermsBits
```

```
setCapPerms : (Capability, CapPermsBits) -> Capability
```

Checking for availability of ISA features

```
haveRVC : unit -> bool
```

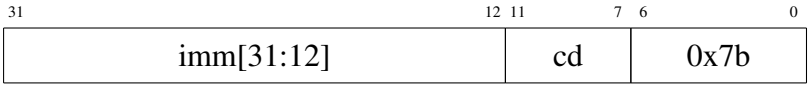
```
haveFExt : unit -> bool
```

9.4 CHERIoT Instructions

AUICGP

Format

AUICGP cd, imm



Description

Capability register *cd* is replaced with the contents of **CGP** (c3), with the **address** replaced with **CGP**.**address** + *imm* × 2048.

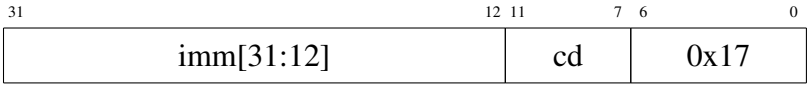
Semantics

```
let off : xlenbits = EXTS(imm) << 11;
let cgp_val = C(CGP_IDX); /* $c3 */
let (representable, newCap) = incCapAddr(cgp_val, off);
C(cd) = clearTagIf(newCap, isCapSealed(cgp_val) | not(representable));
RETIRE_SUCCESS
```

AUIPCC

Format

AUIPCC cd, imm



Description

Capability register *cd* is replaced with the contents of **PCC**, with the **address** replaced with **PCC.address** + *imm* × 2048.

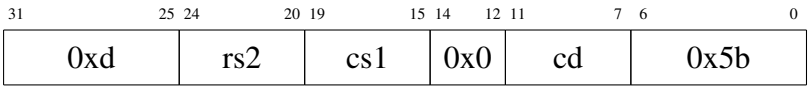
Semantics

```
let off : xlenbits = EXTS(imm) << 11;
let (representable, newCap) = setCapAddr(PCC, PC + off);
C(cd) = clearTagIf(newCap, not(representable));
RETIRE_SUCCESS
```

CAndPerm

Format

CAndPerm cd, cs1, rs2



Description

Capability register *cd* is replaced with the contents of capability register *cs1* with the **perms** field set to the bitwise AND of its previous value and bits 0 to `cap_perms_width` -1 of integer register *rs2*. If the resulting set of permissions cannot be represented by the capability encoding then the result will have a (possibly empty) subset of the ANDed permissions. If *cs1* was sealed then *cd*.**tag** is cleared.

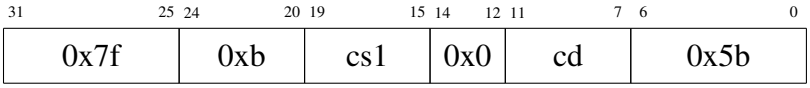
Semantics

```
let cs1_val = C(cs1);
let rs2_val = X(rs2);

let perms = getCapPerms(cs1_val);
let mask = truncate(rs2_val, cap_perms_width);

let inCap = clearTagIfSealed(cs1_val);
let newCap = setCapPerms(inCap, (perms & mask));

C(cd) = newCap;
RETIRE_SUCCESS
```

CClearTag

Format

CClearTag cd, cs1



Description

Capability register *cd* is replaced with the contents of *cs1*, with the **tag** field cleared.

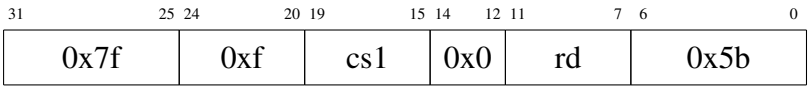
Semantics

```
let cs1_val = C(cs1);
C(cd) = clearTag(cs1_val);
RETIRE_SUCCESS
```

CGetAddr

Format

CGetAddr rd, cs1



Description

Integer register *rd* is set equal to the **address** field of capability register *cs1*.

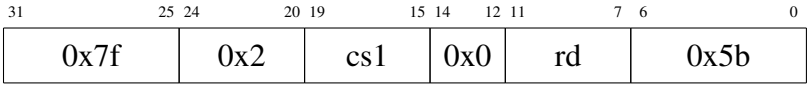
Semantics

```
let capVal = C(cs1);
X(rd) = capVal.address;
RETIRE_SUCCESS
```

CGetBase

Format

CGetBase rd, cs1



Description

Integer register *rd* is set equal to the **base** field of capability register *cs1*.

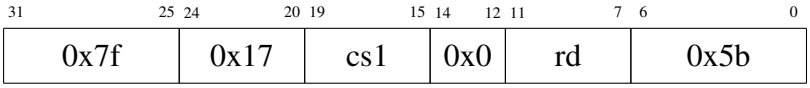
Semantics

```
let capVal = C(cs1);
X(rd) = getCapBaseBits(capVal);
RETIRE_SUCCESS
```

CGetHigh

Format

CGetHigh rd, cs1



Description

Integer register *rd* is set equal to the **high half** of capability register *cs1*.
The bits returned here are of the **in-memory** form of the capability, which may differ from microarchitectural forms in use within implementations. That is, applying `CGetHigh` to a capability loaded from address *m* will yield the same result as loading the high half of the capability-sized granule at *m* (that is, bits above **XLEN** when a capability is interpreted as a twice-**XLEN**-bit integer).

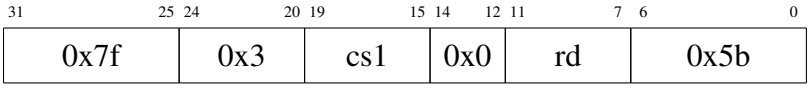
Semantics

```
let capVal : Capability = C(cs1);
X(rd) = capToBits(capVal)[sizeof(xlen) * 2 - 1 .. sizeof(xlen)];
RETIRE_SUCCESS
```

CGetLen

Format

CGetLen rd, cs1



Description

Integer register *rd* is set equal to the **length** field of capability register *cs1*.

Semantics

```
let capVal = C(cs1);
let len = getCapLength(capVal);
X(rd) = to_bits(sizeof(xlen), if len > cap_max_addr then cap_max_addr else len)
    ;
RETIRE_SUCCESS
```

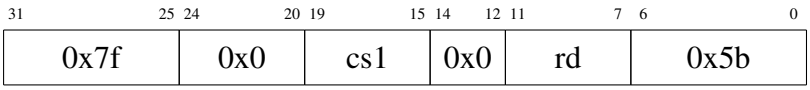
Notes

- Due to the compressed representation of capabilities, the actual length of capabilities can be 2^{xlen}; `CGetLen` will return the maximum value of $2^{xlen} - 1$ in this case.

CGetPerm

Format

CGetPerm rd, cs1



Description

The least significant bits of integer register *rd* are set equal to the **perms** field of capability register *cs1*. The other bits of *rd* are set to zero.

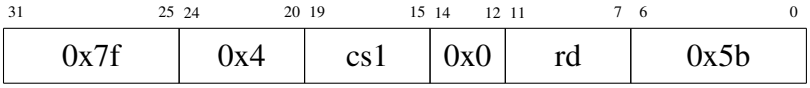
Semantics

```
let capVal = C(cs1);
X(rd) = EXTZ(getCapPerms(capVal));
RETIRE_SUCCESS
```

CGetTag

Format

CGetTag rd, cs1



Description

The low bit of integer register *rd* is set to the **tag** field of *cs1*. All other bits of *rd* are cleared.

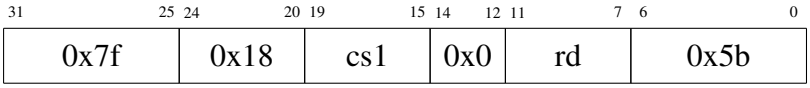
Semantics

```
let capVal = C(cs1);
X(rd) = EXTZ(bool_to_bits(capVal.tag));
RETIRE_SUCCESS
```

CGetTop

Format

CGetTop rd, cs1



Description

Integer register *rd* is set equal to the **top** field of capability register *cs1*.

Semantics

```
let capVal = C(cs1);
let top = getCapTop(capVal);
X(rd) = to_bits(sizeof(xlen), if top > cap_max_addr then cap_max_addr else top)
    ;
RETIRE_SUCCESS
```

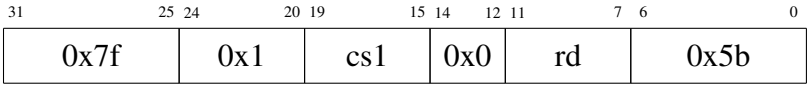
Notes

- Due to the compressed representation of capabilities, the actual top of capabilities can be 2^{xlen}; `CGetTop` will return the maximum value of $2^{xlen} - 1$ in this case.

CGetType

Format

CGetType rd, cs1



Description

Integer register *rd* is set equal to the **otype** field of capability register *cs1*.

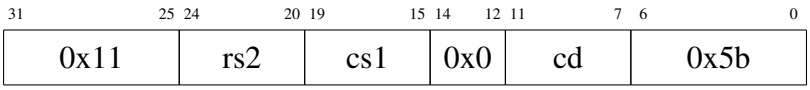
Semantics

```
let capVal = C(cs1);
X(rd) = EXTZ(capVal.otype);
RETIRE_SUCCESS
```

CIncAddr

Format

CIncAddr cd, cs1, rs2



`CIncOffset` is an alias for this instruction.

Description

Capability register *cd* is set equal to capability register *cs1* with its **address** replaced with *cs1*.**address** + *rs2*. If the resulting capability cannot be represented exactly, or if *cs1* was sealed, then *cd*.**tag** is cleared. The remaining capability fields are set to what the in-memory representation of *cs1* with the address set to *cs1*.**address** + *rs2* decodes to.

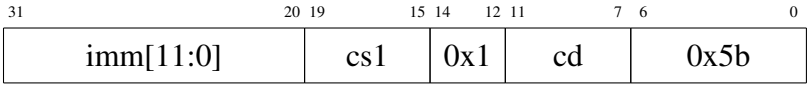
Semantics

```
let cs1_val = C(cs1);
let rs2_val = X(rs2);

let inCap = clearTagIfSealed(cs1_val);
let (success, newCap) = incCapAddr(inCap, rs2_val);

C(cd) = clearTagIf(newCap, not(success));
RETIRE_SUCCESS
```

CIncAddrImm

Format

CIncAddrImm cd, cs1, imm



`CIncOffsetImm` is an alias for this instruction.

Description

Capability register *cd* is set equal to capability register *cs1* with its **address** replaced with *cs1*.**address** + *imm*. If the resulting capability cannot be represented exactly, or if *cs1* was sealed, then *cd*.**tag** is cleared. The remaining capability fields are set to what the in-memory representation of *cs1* with the address set to *cs1*.**address** + *imm* decodes to.

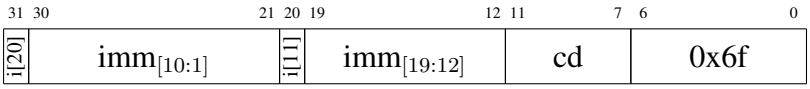
Semantics

```
let cs1_val = C(cs1);
let immBits : xlenbits = EXTS(imm);

let inCap = clearTagIfSealed(cs1_val);
let (success, newCap) = incCapAddr(inCap, immBits);

C(cd) = clearTagIf(newCap, not(success));
RETIRE_SUCCESS
```

CJAL

Format

CJAL cd, imm



Description

Capability register *cd* is replaced with the next instruction's **PCC** and sealed as a sentry. **PCC.address** is incremented by *imm*.

Semantics

```
let off : xlenbits = EXTS(imm);
let newPC = PC + off;
if not(inCapBounds(PCC, newPC, min_instruction_bytes())) then {
  handle_cheri_cap_exception(CapEx_BoundsViolation, PCC_IDX);
  RETIRE_FAIL
} else if newPC[1] == bitone & ~(haveRVC()) then {
  handle_mem_exception(newPC,  E_Fetch_Addr_Align());
  RETIRE_FAIL
} else {
  let (success, linkCap) = setCapAddr(PCC, nextPC); /* Note that nextPC
     accounts for compressed instructions */
  assert(success, "Link cap should always be representable.");
  assert(not (isCapSealed(linkCap)), "Link cap should always be unsealed");
  let sentry_type = if mstatus.MIE() == 0b1 then otype_sentry_ie else
     otype_sentry_id;
  C(cd) = sealCap(linkCap, to_bits(cap_otype_width, sentry_type));
  nextPC = newPC;
  RETIRE_SUCCESS
}
```

Exceptions

An exception is raised if:

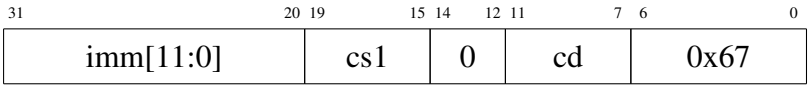
- **PCC.address** + *imm* < **PCC.base**.

- **PCC**.**address** + *imm* + min_instruction_bytes > **PCC**.**top**.
- **PCC**.**address** + *imm* is unaligned, ignoring bit 0.

CJALR

Format

CJALR cd, cs1, imm



Description

Capability register *cd* is replaced with the next instruction's **PCC** and sealed as a sentry. **PCC** is replaced with the value of capability register *cs1* with its **address** incremented by *imm* and the 0th bit of its **address** set to 0, and is unsealed if it is a sentry.

Semantics

```
let cs1_val = C(cs1);
let off : xlenbits = EXTS(imm);
let newPC = [cs1_val.address + off with 0 = bitzero]; /* clear bit zero as for
    RISCV JALR */
if not (cs1_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cs1);
  RETIRE_FAIL
} else if isCapSealed(cs1_val) & (not(isCapSentry(cs1_val)) | imm != zeros())
    then {
  handle_cheri_reg_exception(CapEx_SealViolation, cs1);
  RETIRE_FAIL
} else if not (cs1_val.permit_execute) then {
  handle_cheri_reg_exception(CapEx_PermitExecuteViolation, cs1);
  RETIRE_FAIL
} else if not(inCapBounds(cs1_val, newPC, min_instruction_bytes())) then {
  handle_cheri_reg_exception(CapEx_BoundsViolation, cs1);
  RETIRE_FAIL
} else if newPC[1] == bitone & ~(haveRVC()) then {
  handle_mem_exception(newPC,  E_Fetch_Addr_Align());
  RETIRE_FAIL
} else {
  let (success, linkCap) = setCapAddr(PCC, nextPC); /* Note that nextPC
      accounts for compressed instructions */
  assert(success, "Link cap should always be representable.");
```

```
  assert(not (isCapSealed(linkCap)), "Link cap should always be unsealed");
  let sentry_type = if mstatus.MIE() == 0b1 then otype_sentry_ie else
      otype_sentry_id;
  C(cd) = sealCap(linkCap, to_bits(cap_otype_width, sentry_type));
  nextPC = newPC;
  nextPCC = unsealCap(cs1_val);
  if unsigned(cs1_val.otype) == otype_sentry_id then
    mstatus->MIE() = 0b0;
  if unsigned(cs1_val.otype) == otype_sentry_ie then
    mstatus->MIE() = 0b1;
  RETIRE_SUCCESS
}
```

Exceptions

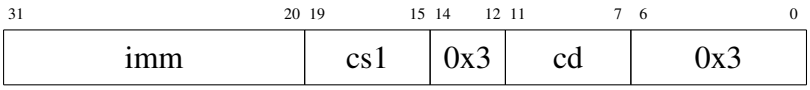
An exception is raised if:

- *cs1*.**tag** is not set.
- *cs1* is sealed and is not a sentry.
- *cs1* is a sentry and $imm \neq 0$.
- *cs1*.**perms** does not grant PERMIT_EXECUTE.
- *cs1*.**address** + *imm* < *cs1*.**base**.
- *cs1*.**address** + *imm* + min_instruction_bytes > *cs1*.**top**.
- *cs1*.**base** is unaligned.
- *cs1*.**address** + *imm* is unaligned, ignoring bit 0.

CLC

Format

CLC cd, cs1, imm



Description

Capability register *cd* is replaced with the capability located in memory at *cs1*.**address** + *imm*, and if *cs1*.**perms** does not grant PERMIT_LOAD_CAPABILITY then *cd*.**tag** is cleared.

Semantics

```
let offset : xlenbits = EXTS(imm);
let auth_val = C(cs1);
let vaddrBits = auth_val.address + offset;
if not(auth_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cs1);
  RETIRE_FAIL
} else if isCapSealed(auth_val) then {
  handle_cheri_reg_exception(CapEx_SealViolation, cs1);
  RETIRE_FAIL
} else if not (auth_val.permit_load) then {
  handle_cheri_reg_exception(CapEx_PermitLoadViolation, cs1);
  RETIRE_FAIL
} else if not(inCapBounds(auth_val, vaddrBits, cap_size)) then {
  handle_cheri_reg_exception(CapEx_BoundsViolation, cs1);
  RETIRE_FAIL
} else if not(is_aligned_addr(vaddrBits, cap_size)) then {
  handle_mem_exception(vaddrBits, E_Load_Addr_Align());
  RETIRE_FAIL
} else match translateAddr(vaddrBits, Read(Cap)) {
  TR_Failure(E_Extension(_), _) => { internal_error("unexpected cheri exception
       for cap load") },
  TR_Failure(e, _) => { handle_mem_exception(vaddrBits, e); RETIRE_FAIL },
  TR_Address(addr, ptw_info) => {
    let c = mem_read_cap(addr, false, false, false);
    match c {
```

```
      MemValue(v) => {
        var cr = v;
        if cr.tag & not(auth_val.permit_load_global) then {
          cr.global = false;
          cr.permit_load_global = false;
        };
        if cr.tag & not(auth_val.permit_load_mutable) & not(isCapSealed(cr))
            then {
          cr.permit_store = false;
          cr.permit_load_mutable = false;
        };
        if ptw_info.ptw_lc == PTW_LC_CLEAR  | not(auth_val.
            permit_load_store_cap) then {
          cr.tag = false;
        };
        /* Sealing capabilities are excluded from revocation */
        let isSealingCap = cr.permit_seal | cr.permit_unseal | cr.perm_user0;
        if (cr.tag & not(isSealingCap)) then {
          let base = getCapBaseBits(cr);
          let granule_addr = align_down(log2_revocation_granule_size, base);
          let revoked = mem_read_cap_revoked(granule_addr);
          cr.tag = cr.tag & not(revoked);
        };
        C(cd) = cr;
        RETIRE_SUCCESS
      },
      MemException(e) => {handle_mem_exception(vaddrBits, e); RETIRE_FAIL }
    }
  }
}
```

Exceptions

An exception is raised if:

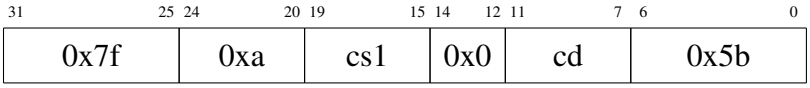
- *cs1*.**tag** is not set.
- *cs1* is sealed.
- *cs1*.**perms** does not grant PERMIT_LOAD.
- *cs1*.**address** + *imm* < *cs1*.**base**.
- *cs1*.**address** + *imm* + **CLEN** / 8 > *cs1*.**top**.

- *cs1*.**address** + *imm* is unaligned, regardless of whether the implementation supports unaligned data accesses.

CMove

Format

CMove cd, cs1



Description

Capability register *cd* is replaced with the contents of *cs1*.

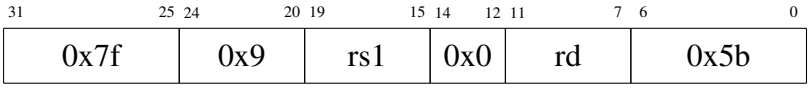
Semantics

```
C(cd) = C(cs1);
RETIRE_SUCCESS
```

CRepresentableAlignmentMask

Format

CRepresentableAlignmentMask rd, rs1



Description

Integer register *rd* is set to a mask that can be used to round addresses down to to a value that is sufficiently aligned to set exact bounds for the nearest representable length of *rs1* (as obtained by CRRL).

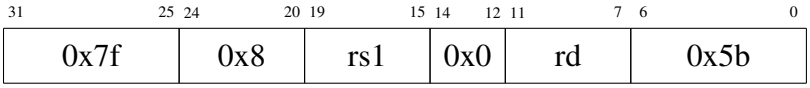
Semantics

```
let len = X(rs1);
X(rd) = getRepresentableAlignmentMask(len);
RETIRE_SUCCESS
```

CRoundRepresentableLength

Format

CRoundRepresentableLength rd, rs1



Description

Integer register *rd* is set to the smallest value greater or equal to *rs1* that can be used as a length to set exact bounds on a capability that has a suitably aligned base (as obtained with the help of CRAM). Note that this could round up the length to 2^{xlen} in which case *rd* will be set to zero. Users should be careful to account for this case.

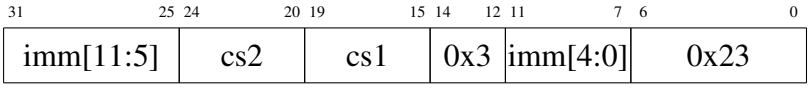
Semantics

```
let len = X(rs1);
X(rd) = getRepresentableLength(len);
RETIRE_SUCCESS
```

CSC

Format

CSC cs2, cs1, imm



Description

The capability located in memory at *cs1*.**address** + *imm* is replaced with capability register *cs2*.

Semantics

```
let offset : xlenbits = EXTS(imm);
let auth_val = C(cs1);
let vaddrBits = auth_val.address + offset;
let cs2_val = C(cs2);
if not(auth_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cs1);
  RETIRE_FAIL
} else if isCapSealed(auth_val) then {
  handle_cheri_reg_exception(CapEx_SealViolation, cs1);
  RETIRE_FAIL
} else if not (auth_val.permit_store) then {
  handle_cheri_reg_exception(CapEx_PermitStoreViolation, cs1);
  RETIRE_FAIL
} else if not (auth_val.permit_load_store_cap) & cs2_val.tag then {
  handle_cheri_reg_exception(CapEx_PermitStoreCapViolation, cs1);
  RETIRE_FAIL
} else if not (auth_val.permit_store_local_cap) & cs2_val.tag & not(cs2_val.
    global) then {
  handle_cheri_reg_exception(CapEx_PermitStoreLocalCapViolation, cs1);
  RETIRE_FAIL
} else if not(inCapBounds(auth_val, vaddrBits, cap_size)) then {
  handle_cheri_reg_exception(CapEx_BoundsViolation, cs1);
  RETIRE_FAIL
} else if not(is_aligned_addr(vaddrBits, cap_size)) then {
  handle_mem_exception(vaddrBits, E_SAMO_Addr_Align());
```

```
  RETIRE_FAIL
} else match translateAddr(vaddrBits, Write(if cs2_val.tag then Cap else Data))
    {
  TR_Failure(e, _) => { handle_mem_exception(vaddrBits, e); RETIRE_FAIL },
  TR_Address(addr, _) => {
    let eares : MemoryOpResult(unit) = mem_write_ea_cap(addr, false, false,
        false);
    match (eares) {
      MemException(e) => { handle_mem_exception(vaddrBits, e); RETIRE_FAIL },
      MemValue(_) => {
        let res : MemoryOpResult(bool) = mem_write_cap(addr, cs2_val, false,
            false, false);
        match (res) {
          MemValue(true)  => RETIRE_SUCCESS,
          MemValue(false) => internal_error("store got false from
              mem_write_value"),
          MemException(e) => { handle_mem_exception(vaddrBits, e); RETIRE_FAIL
              }
        }
      }
    }
  }
}
```

Exceptions

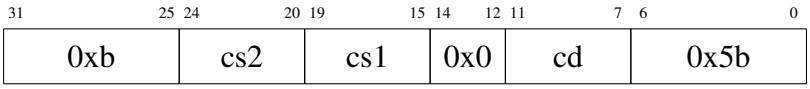
An exception is raised if:

- *cs1*.**tag** is not set.
- *cs1* is sealed.
- *cs1*.**perms** does not grant PERMIT_STORE.
- *cs1*.**perms** does not grant PERMIT_STORE_CAPABILITY and *cs2*.**tag** is set.
- *cs1*.**perms** does not grant PERMIT_STORE_LOCAL_CAPABILITY, *cs2*.**tag** is set and *cs2*.**perms** does not grant GLOBAL.
- *cs1*.**address** + *imm* < *cs1*.**base**.
- *cs1*.**address** + *imm* + **CLEN** / 8 > *cs1*.**top**.

CSeal

Format

CSeal cd, cs1, cs2



Description

Capability register *cd* is replaced with capability register *cs1*, and is sealed with **otype** equal to the **address** field of capability register *cs2*. If *cs2* is unable to authorize the sealing, or if *cs1* was already sealed, then the **tag** field of *cd* is cleared.

Semantics

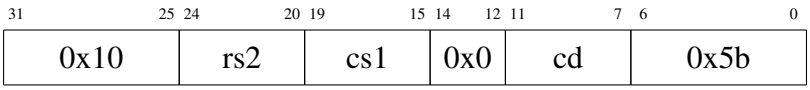
```
let cs1_val = C(cs1);
let cs2_val = C(cs2);

let cs2_addr = unsigned(cs2_val.address);
let (cs2_base, cs2_top) = getCapBounds(cs2_val);

let isPermittedOtype : bool =
  if cs1_val.permit_execute then
    match (cs2_addr) {
      /* 0 is unsealed */
      otype_sentry => true,
      otype_sentry_id => true,
      otype_sentry_ie => true,
      /* 4 and 5 are reserved */
      6 => true,
      7 => true,
      _ => false
    }
  else
    (8 < cs2_addr) & (cs2_addr <= 15);

let permitted = cs2_val.tag
              & not(isCapSealed(cs2_val))
              & cs2_val.permit_seal
```

```
              & (cs2_addr >= cs2_base)
              & (cs2_addr < cs2_top)
              & isPermittedOtype;

let inCap = clearTagIfSealed(cs1_val);
let newCap = sealCap(inCap, to_bits(cap_otype_width, cs2_addr));

C(cd) = clearTagIf(newCap, not(permitted));
RETIRE_SUCCESS
```

CSetAddr

Format

CSetAddr cd, cs1, rs2



Description

Capability register *cd* is set equal to capability register *cs1* with its **address** replaced with *rs2*. If the resulting capability cannot be represented exactly, or if *cs1* was sealed, then *cd*.**tag** is cleared. The remaining capability fields are set to what the in-memory representation of *cs1* with the address set to *rs2* decodes to.

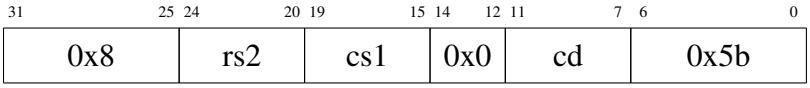
Semantics

```
let cs1_val = C(cs1);
let rs2_val = X(rs2);

let inCap = clearTagIfSealed(cs1_val);
let (representable, newCap) = setCapAddr(inCap, rs2_val);

C(cd) = clearTagIf(newCap, not(representable));
RETIRE_SUCCESS
```

CSetBounds

Format

CSetBounds cd, cs1, rs2



Description

Capability register *cd* is set to capability register *cs1* with its **base** field replaced with *cs1*.**address** and its **length** field replaced with integer register *rs2*. If the resulting capability cannot be represented exactly the **base** will be rounded down and the **length** will be rounded up by the smallest amount needed to form a representable capability covering the requested bounds. The **tag** field of the result is cleared if the bounds of the result exceed the bounds of *cs1*, or if *cs1* was sealed.

Semantics

```
let cs1_val = C(cs1);
let rs2_val = X(rs2);

let newBase = cs1_val.address;
let newLen  = rs2_val;
let inBounds = inCapBounds(cs1_val, newBase, unsigned(newLen));

let inCap = clearTagIfSealed(cs1_val);
let (_, newCap) = setCapBounds(inCap, newBase, newLen);

C(cd) = clearTagIf(newCap, not(inBounds)); /* ignore exact */
RETIRE_SUCCESS
```

Notes

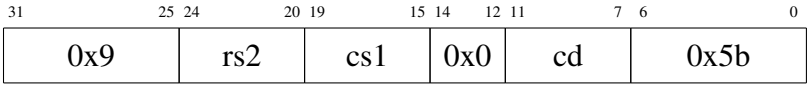
- This Sail code actually does the bounds check on the *requested* bounds, not the bounds that result from `setCapBounds`. This is an important distinction because the resulting bounds may be larger than the requested bounds, which could potentially lead to non-monotonic behaviour. However, providing that `setCapBounds` always returns the most precise encodable bounds it is safe to do the check on the requested

bounds because, in the worst case, it would return the existing bounds. This is desirable because in hardware the bounds checking can be performed in parallel with computing the new bounds.

CSetBoundsExact

Format

CSetBoundsExact cd, cs1, rs2



Description

Capability register *cd* is set to capability register *cs1* with its **base** field replaced with *cs1*.**address** and its **length** field replaced with integer register *rs2*. If the resulting capability cannot be represented exactly, the **tag** field will be cleared (unlike `CSetBounds`), the **base** will be rounded down and the **length** will be rounded up by the smallest amount needed to form a representable capability covering the requested bounds. The **tag** field of the result is cleared if the bounds of the result exceed the bounds of *cs1*, or if *cs1* was sealed.

Semantics

```
let cs1_val = C(cs1);
let rs2_val = X(rs2);

let newBase = cs1_val.address;
let newLen = X(rs2);
let inBounds = inCapBounds(cs1_val, newBase, unsigned(newLen));

let inCap = clearTagIfSealed(cs1_val);
let (exact, newCap) = setCapBounds(inCap, newBase, newLen);

C(cd) = clearTagIf(newCap, not(inBounds & exact));
RETIRE_SUCCESS
```

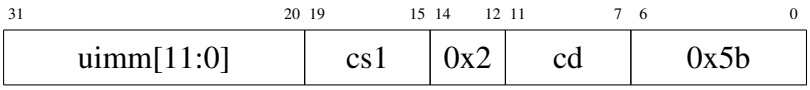
Notes

- The same caveat regarding the order of the bounds check applies as for `CSetBounds`.

CSetBoundsImm

Format

CSetBoundsImm cd, cs1, uimm



Description

Capability register *cd* is set to capability register *cs1* with its **base** field replaced with *cs1*.**address** and its **length** field replaced with *uimm*. If the resulting capability cannot be represented exactly the **base** will be rounded down and the **length** will be rounded up by the smallest amount needed to form a representable capability covering the requested bounds. The **tag** field of the result is cleared if the bounds of the result exceed the bounds of *cs1*, or if *cs1* was sealed.

Semantics

```
let cs1_val = C(cs1);

let newBase = cs1_val.address;
let newLen : CapAddrBits = EXTZ(uimm);
let inBounds = inCapBounds(cs1_val, newBase, unsigned(newLen));

let inCap = clearTagIfSealed(cs1_val);
let (_, newCap) = setCapBounds(inCap, newBase, newLen);

C(cd) = clearTagIf(newCap, not(inBounds)); /* ignore exact */
RETIRE_SUCCESS
```

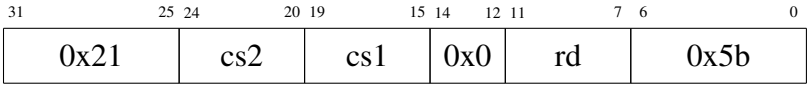
Notes

- The same caveat regarding the order of the bounds check applies as for `CSetBounds`.

CSetEqualExact

Format

CSetEqualExact rd, cs1, cs2



Description

Integer register *rd* is set to 1 if the **tag** fields and in-memory representations of capability registers *cs1* and *cs2* are identical, including any reserved encoding bits, otherwise it is set to 0.

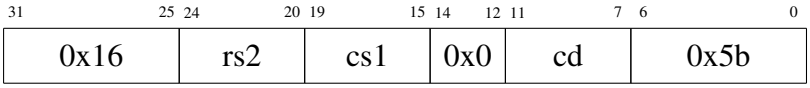
Semantics

```
let cs1_val = C(cs1);
let cs2_val = C(cs2);
X(rd) = EXTZ(bool_to_bits(cs1_val == cs2_val));
RETIRE_SUCCESS
```

CSetHigh

Format

CSetHigh cd, cs1, rs2



Description

Capability register *cd* comes to hold the capability from *cs1* with its high bits replaced with the value in the integer register *rs2*. The tag of *cd* is cleared.
rs2 holds the **in-memory** form of capability bits. That is, this instruction yields the same result as writing *cs1* out to memory, overwriting the high word with *rs2*, and loading that capability-sized granule into *cd*, although without the memory mutation side-effects.

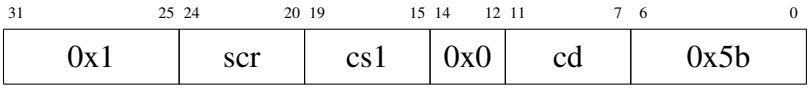
Semantics

```
let capVal = C(cs1);
let intVal = X(rs2);
let capLow : xlenbits = capToBits(capVal)[sizeof(xlen) - 1 .. 0];
let newCap : Capability = capBitsToCapability(false, intVal @ capLow);
C(cd) = newCap;
RETIRE_SUCCESS
```

CSpecialRW

Format

CSpecialRW cd, scr, cs1



Description

Capability register *cd* is set equal to special capability register *scr*, and *scr* is set equal to capability register *cs1* if *cs1* is not **C0**.

Semantics

```
let specialExists : bool = match unsigned(scr) {
  28 => true,
  29 => true,
  30 => true,
  31 => true,
  _  => false
};
if (not(specialExists)) then {
  handle_illegal();
  RETIRE_FAIL
} else if not(PCC.access_system_regs) then {
  handle_cheri_cap_exception(CapEx_AccessSystemRegsViolation, 0b1 @ scr);
  RETIRE_FAIL
} else {
  let cs1_val = C(cs1);
  C(cd) = match unsigned(scr) {
    28 => MTCC,
    29 => MTDC,
    30 => MScratchC,
    31 => legalize_epcc(MEPCC),
    _  => {assert(false, "unreachable"); undefined}
  };
  if (cs1 != zeros()) then {
    match unsigned(scr) {
      28 => MTCC = legalize_tcc(MTCC, cs1_val),
```

```
      29 => MTDC = cs1_val,
      30 => MScratchC = cs1_val,
      31 => MEPCC = cs1_val,
      _  => assert(false, "unreachable")
    }
  };
  RETIRE_SUCCESS
}
```

Exceptions

An exception is raised if:

- *scr* does not exist.
- *scr* requires PERMIT_ACCESS_SYSTEM_REGISTERS and that is not granted by **PCC**.**perms**.

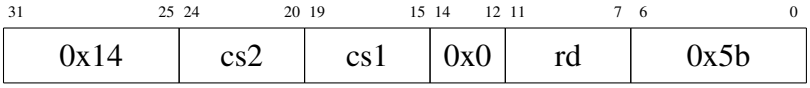
Notes

- Writing **NULL** to a special capability register cannot be done with **C0** as that only performs a read. An alternative implementation would allocate a separate two-operand CSpecialR instruction and interpret *cs1* being **C0** as a write of **NULL** if the need to use a temporary capability register proves to be overly problematic for software.

CSub

Format

CSub rd, cs1, cs2



Description

Integer register *rd* is set equal to (*cs1*.**address** − *cs2*.**address**) mod 2^{xlen}.

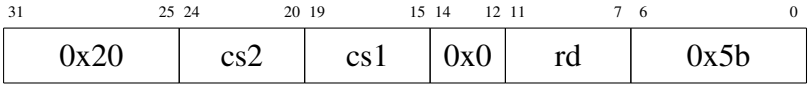
Semantics

```
let cs2_val = C(cs2);
let cs1_val = C(cs1);

X(rd) = cs1_val.address - cs2_val.address;
RETIRE_SUCCESS
```

CTestSubset

Format

CTestSubset rd, cs1, cs2



Description

Integer register *rd* is set to 1 if the **tag** fields of capability registers *cs1* and *cs2* are the same and the bounds and permissions of *cs2* are a subset of those of *cs1*.

Semantics

```
let cs1_val = C(cs1);
let cs2_val = C(cs2);

let (cs2_base, cs2_top) = getCapBounds(cs2_val);
let (cs1_base, cs1_top) = getCapBounds(cs1_val);
let cs2_perms = getCapPerms(cs2_val);
let cs1_perms = getCapPerms(cs1_val);

let result = if cs1_val.tag != cs2_val.tag then
               0b0
             else if cs2_base < cs1_base then
               0b0
             else if cs2_top > cs1_top then
               0b0
             else if (cs2_perms & cs1_perms) != cs2_perms then
               0b0
             else
               0b1;

X(rd) = EXTZ(result);
RETIRE_SUCCESS
```

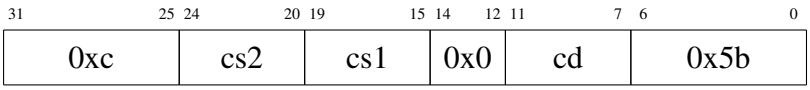
Notes

- The operand order for this instruction is reversed compared with the normal RISC-V comparison instructions, but this may be changed in future.
- The **otype** field is ignored for this instruction, but an alternative implementation might wish to consider capabilities with distinct **otype**s as unordered as is done for the **tag** field.

CUnseal

Format

CUnseal cd, cs1, cs2



Description

Capability register *cd* is replaced with capability register *cs1* and is unsealed, using capability register *cs2* as the authority for the unsealing operation. If *cs2*.**perms** does not grant GLOBAL then *cd*.**perms** is stripped of GLOBAL. If *cs2* is unable to authorize the unsealing, the **tag** field of *cd* is cleared.

Semantics

```
let cs1_val = C(cs1);
let cs2_val = C(cs2);
let cs2_addr = unsigned(cs2_val.address);
let (cs2_base, cs2_top) = getCapBounds(cs2_val);
let permitted = cs2_val.tag
              & isCapSealed(cs1_val)
              & not(isCapSealed(cs2_val))
              & (cs2_addr == unsigned(cs1_val.otype))
              & cs2_val.permit_unseal
              & (cs2_addr >= cs2_base)
              & (cs2_addr < cs2_top);
let new_global = cs1_val.global & cs2_val.global;
let newCap = {unsealCap(cs1_val) with global=new_global};
C(cd) = clearTagIf(newCap, not(permitted));
RETIRE_SUCCESS
```

Appendix A

Sail listings for capability encoding

This chapter contains Sail types and functions that implement the capability encoding scheme.

`EncCapability` represents capabilities as stored in memory.

```
struct EncCapability = {
  reserved   : bits(1),
  cperms     : bits(cap_cperms_width),
  cotype     : bits(cap_cotype_width),
  cE         : bits(cap_cE_width),
  T          : bits(cap_mantissa_width),
  B          : bits(cap_mantissa_width),
  address    : bits(cap_addr_width)
}
```

`Capability` represents a partially decompressed capability. The permissions, E and otype are expanded from their compressed format by `encCapabilityToCapability` and compressed by `capToEncCap`.

```
struct Capability = {
  tag                    : bool,
  perm_user0             : bool,
  permit_seal            : bool,
  permit_unseal          : bool,
  permit_execute         : bool,
  access_system_regs     : bool,
  permit_load_store_cap  : bool,
  permit_load            : bool,
  permit_store_local_cap : bool,
  permit_load_mutable    : bool,
```

```
  permit_store           : bool,
  permit_load_global     : bool,
  global                 : bool,
  reserved               : bits(1),
  E                      : bits(cap_E_width),
  B                      : bits(cap_mantissa_width),
  T                      : bits(cap_mantissa_width),
  otype                  : bits(cap_otype_width),
  address                : bits(cap_addr_width)
}
```

Partially decompress a capability from bits to a `Capability` struct. Permissions, otype and exponent are decompressed, but the bounds are left in the form of B and T fields.

```
function encCapabilityToCapability(t,c) : (bool, EncCapability) -> Capability =
    {
  var perm_user0             : bool = false;
  var permit_seal            : bool = false;
  var permit_unseal          : bool = false;
  var permit_execute         : bool = false;
  var access_system_regs     : bool = false;
  var permit_load_store_cap  : bool = false;
  var permit_load            : bool = false;
  var permit_store_local_cap : bool = false;
  var permit_load_mutable    : bool = false;
  var permit_store           : bool = false;
  var permit_load_global     : bool = false;
  var global                 : bool = bit_to_bool(c.cperms[5]);
  var isExe : bool = false;
  match c.cperms[4..0] {
    0b11 @ [SL, LM, LG] => {
      /* mem-rw-cap format */
      permit_load = true;
      permit_load_store_cap = true;
      permit_store = true;
      permit_store_local_cap = bit_to_bool(SL);
      permit_load_mutable = bit_to_bool(LM);
      permit_load_global = bit_to_bool(LG);
    },
    0b101 @ [LM, LG] => {
      /* mem-ro-cap format */
      permit_load = true;
```

```
    permit_load_store_cap = true;
    permit_load_mutable = bit_to_bool(LM);
    permit_load_global = bit_to_bool(LG);
  },
  0b10000 => {
    /* mem-wo-cap */
    permit_store = true;
    permit_load_store_cap = true;
  },
  0b100 @ [LD, SD] => {
    /* mem-data */
    permit_load = bit_to_bool(LD);
    permit_store = bit_to_bool(SD);
  },
  0b01 @ [SR, LM, LG] => {
    /* Executable format */
    isExe = true;
    permit_execute = true;
    permit_load = true;
    permit_load_store_cap = true;
    access_system_regs  = bit_to_bool(SR);
    permit_load_mutable = bit_to_bool(LM);
    permit_load_global  = bit_to_bool(LG);
  },
  0b00 @ [U0, SE, US] => {
    /* Sealing format */
    perm_user0    = bit_to_bool(U0);
    permit_seal   = bit_to_bool(SE);
    permit_unseal = bit_to_bool(US);
  }
};
/* The otype of executable caps is mapped to 1-7 and others to 9-15. Unsealed
  is always 0. */
let otype = (if isExe | c.cotype == 0b000 then 0b0 else 0b1) @ c.cotype;
/* The 4-bit exponent is expanded to 5 bits, using 0xf to encode a cap_max_E
  value that enables representing the entire address space. */
let E = if c.cE == 0xf then cap_max_E_bits else EXTZ(c.cE);
return struct {
  tag                   = t,
  perm_user0            = perm_user0           ,
  permit_seal           = permit_seal          ,
```

```
    permit_unseal          = permit_unseal         ,
    permit_execute         = permit_execute        ,
    access_system_regs     = access_system_regs    ,
    permit_load_store_cap  = permit_load_store_cap ,
    permit_load            = permit_load           ,
    permit_store_local_cap = permit_store_local_cap,
    permit_load_mutable    = permit_load_mutable   ,
    permit_store           = permit_store          ,
    permit_load_global     = permit_load_global    ,
    global                 = global                ,
    reserved               = c.reserved,
    E                      = E,
    B                      = c.B,
    T                      = c.T,
    otype                  = otype,
    address                = c.address
  }
}
```

Compress a `Capability` back to the bits representation. This is simply the reverse of `encCapabilityToCapability`, although it relies on the fields having valid values. In particular E must be either cap_max_E or less than 15, and otype is either 0 (unsealed) or in the correct range for the capability format. Both of these will be true for things returned by `encCapabilityToCapability`. `CSetBounds` ensures a sensible value for E and nothing else sets it. For the otype we rely on the fact that sealed capabilities cannot be modified and unsealed is always encoded as zero. It is mildy surprsing that `CAndPerm` on a sealed capability may result in a untagged capability with a different otype, but otypes have no significance on untagged capabilities anyway.

```
function capToEncCap(cap) : Capability -> EncCapability = {
  var cperms : bits(cap_cperms_width) = zeros();
  cperms[5] = bool_to_bit(cap.global);
  if cap.permit_execute & cap.permit_load & cap.permit_load_store_cap then {
    /* Executable format */
    cperms[4..3] = 0b01;
    cperms[2] = bool_to_bit(cap.access_system_regs);
    cperms[1] = bool_to_bit(cap.permit_load_mutable);
    cperms[0] = bool_to_bit(cap.permit_load_global);
  } else if cap.permit_load & cap.permit_load_store_cap & cap.permit_store then
      {
    /* mem cap-rw */
    cperms[4..3] = 0b11;
```

```
    cperms[2] = bool_to_bit(cap.permit_store_local_cap);
    cperms[1] = bool_to_bit(cap.permit_load_mutable);
    cperms[0] = bool_to_bit(cap.permit_load_global);
  } else if cap.permit_load & cap.permit_load_store_cap then {
    /* mem cap-ro */
    cperms[4..2] = 0b101;
    cperms[1] = bool_to_bit(cap.permit_load_mutable);
    cperms[0] = bool_to_bit(cap.permit_load_global);
  } else if cap.permit_store & cap.permit_load_store_cap then {
    /* mem cap-wo */
    cperms[4..0] = 0b10000;
  } else if cap.permit_load | cap.permit_store then {
    /* mem rw data */
    cperms[4..2] = 0b100;
    cperms[1] = bool_to_bit(cap.permit_load);
    cperms[0] = bool_to_bit(cap.permit_store);
  } else {
    /* Sealing format */
    cperms[4..3] = 0b00;
    cperms[2] = bool_to_bit(cap.perm_user0);
    cperms[1] = bool_to_bit(cap.permit_seal);
    cperms[0] = bool_to_bit(cap.permit_unseal);
  };
  return struct {
    cperms = cperms,
    reserved = cap.reserved,
    cotype = cap.otype[2..0], /* truncate otype when compressing */
    cE = if cap.E == cap_max_E_bits then 0xf else cap.E[3..0],
    T = cap.T,
    B = cap.B,
    address = cap.address
  };
}
```

Returns the decoded base and top of the given Capability.

```
function getCapBoundsBits(c) : Capability -> (CapAddrBits, CapLenBits) = {
  let E = unsigned(c.E);
  let a : CapAddrBits = c.address;
  /* Calculate corrections for upper bits of base and top based on relative
     positions of base, top and address with respect to
     2**(E+cap_mantissa_width) aligned regions */
```

```
  let a_mid = truncate(a >> E, cap_mantissa_width);
  /* If a_mid is less than B then a must be in region above base */
  let a_hi = if a_mid <_u c.B then 1 else 0;
  /* If T is less than B then top must be in region above base */
  let t_hi = if c.T <_u c.B then 1 else 0;
  /* If address is in region above base then we need to subtract one from a_top
    to get top bits of base */
  let c_b = 0 - a_hi;
  /* The correction for top can be -1, 0, or 1 depending on whether a and t lie
    in the same region and, if not, which is in the high region. This boils
        down
    to a subtraction. */
  let c_t = t_hi - a_hi;
  let a_top = a >> (E + cap_mantissa_width);
  /* Finally reconstruct the base and top using the corrections and truncate.
      */
  let base : CapAddrBits = truncate((a_top + c_b) @ c.B @ zeros(E),
      cap_addr_width);
  let top : CapLenBits  = truncate((a_top + c_t) @ c.T @ zeros(E),
      cap_len_width);
  (base, top)
}
```

Returns cap with E, B and T set such that the decoded bounds include the region specified by base and length. If the region is not precisely representable the base may be rounded down and the length up. Also returns a boolean indicating whether the bounds are precisely representable, and sets the address of the returned capability to the requested base.

```
function setCapBounds(cap, base, length) : (Capability, CapAddrBits,
    CapAddrBits) -> (bool, Capability) = {
  let ext_base = 0b0 @ base;
  /* Compute new top, note extra bit in case of overflow */
  let top : CapLenBits = ext_base + (0b0 @ length);
  /* Find smallest exponent that can represent required length.
   */
  let e : range(0, 23) = 23 - count_leading_zeros(truncateLSB(length, 23));
  /* Saturate e at max if it exceeds representable 4-bit value. */
  var e_sat : range(0,  cap_max_E) = if e > 14 then cap_max_E else e;
  /* Extract B and T bits from base and top, include a spare bit so that we can
   check for length overflow below. */
  var B = truncate(base >> e_sat, cap_mantissa_width + 1);
  var T = truncate(top >> e_sat, cap_mantissa_width + 1);
```

```
  /* Work out whether we have lost significant bits in top */
  var maskLo : CapLenBits = EXTZ(ones(e_sat));
  lostSignificantTop  = unsigned(top & maskLo) != 0;
  if lostSignificantTop then {
    /* we must increment T to make sure it is still above top even with lost
        bits */
    T = T + 1;
  };

  /* If the resulting length is greater than maximum possible, we must
     increment e by one and try again. This time overflow is impossible. */
  if  0b0111111111 <_u (T - B) then {
    if e_sat < 14 then {
      /* increment e_sat by one, note that min is only necessary to satisfy the
         Sail type checker */
      e_sat = min(e_sat + 1, cap_max_E);
    } else {
      e_sat = cap_max_E;
    };
    /* Recompute B, T, mask etc for new e_sat */
    B = truncate(base >> e_sat, cap_mantissa_width + 1);
    T = truncate(top >> e_sat, cap_mantissa_width + 1);
    maskLo = EXTZ(ones(e_sat));
    lostSignificantTop  = unsigned(top & maskLo) != 0;
    if lostSignificantTop then  {
      T = T + 1;
    };
  };
  /* Return cap with new bounds */
  let encE = to_bits(cap_E_width, e_sat);
  let newCap = {cap with address=base, E=encE, B=truncate(B, cap_mantissa_width
      ), T=truncate(T, cap_mantissa_width)};
  lostSignificantBase = unsigned(ext_base & maskLo) != 0;
  let exact = not(lostSignificantBase | lostSignificantTop);
  (exact, newCap)
}
```

The representable alignment mask for a given length depends on the exponent that would be used to represent a region of that length, assuming the base is sufficiently aligned. To compute this we resuse the implementation of `setCapBounds`

```
function getRepresentableAlignmentMask(len) = {
```

```
  let (exact, c) = setCapBounds(root_cap_mem, to_bits(sizeof(xlen), 0), len);
  var e = unsigned(c.E);
  ones(sizeof(xlen)) << e
}
```

Given the required alignment mask returned by `getRepresentableAlignmentMask` we can round up the length by bit twiddling.

```
function getRepresentableLength(len) = {
  let m = getRepresentableAlignmentMask(len);
  (len + ~(m)) & m
}
```

A.1 SMT validation of properties of the capability encoding

The Sail compiler can translate Sail code into a Satisfiability Modulo Theories (SMT) problem that can be given to a solver such as CVC4 or Z3 to check whether a given function returns true for all input values. We have used this to check important properties of the capability encoding as implemented in Sail.
Some helper functions are used in the Sail properties:

`encodeDecode` is a helper function to compress and then decompress a Capability. The `Capability` struct can hold non-encodable values therefore some properties encode then decode a Capability to check the effect that compression / decompression has. In general the bounds, address and tag should be unaffected but the permissions and otype might change.

```
function encodeDecode(c : Capability) -> Capability =  capBitsToCapability(c.
    tag, capToBits(c))
```

`capEncodable` is a helper to check that the given Capability is unaffected by compression / decompression.

```
function capEncodable(c : Capability) -> bool = c == encodeDecode(c)
```

The following functions have been checked to return true for all inputs.

`prop_decEnc` checks that going from bits to Capability and back preserves bits. This is a vital property to ensure that `memcpy` works.

```
function prop_decEnc(t : bool, c : CapBits) -> bool  = {
    let cap = capBitsToCapability(t, c);
    let b = capToBits(cap);
    (c == b) & (cap.tag == t)
}
```

prop_andperms checks that for any capability and permissions mask CAndPerm will result in a capability whose permissions are a subset of the original permissions and the mask.

```
function prop_andperms(b : CapBits, mask : CapPermsBits) -> bool = {
    let c = capBitsToCapability(false, b);
    let perms = getCapPerms(c);
    let newCap = setCapPerms(c, perms & mask);
    /* We must encode then decode the resulting Capability to see the effect
     * of permissions compression. */
    let newCap2 = encodeDecode(newCap);
    let newPerms = getCapPerms(newCap2);
    /* Check that newperms are a subset of original perms and requested perms
        */
    ((newPerms & ~(perms)) == zeros()) & ((newPerms & ~(mask)) == zeros())
}
```

prop_setbounds checks the basic properties of setBounds: that the resutling cap includes the requested region and that the exact flag is correct. Note that we restrict this to bounds where the top is less than or equal to the maximum possible, since the encoding cannot handle all such cases and the ISA should not allow creation of tagged capabilities with such top anyway.

```
function prop_setbounds(reqBase : CapAddrBits, reqLen : CapAddrBits) -> bool =
    {
    let (exact, c) = setCapBounds(root_cap_mem, reqBase, reqLen);
    let encodable = capEncodable(c);
    let (b2, t2) = getCapBoundsBits(c);
    let reqTop = (0b0 @ reqBase) + (0b0 @ reqLen);
    let saneTop = reqTop <=_u 0b1@0x00000000;
    saneTop --> (
        encodable &
        (c.address == reqBase) &
        ((exact & (reqBase == b2) & (reqTop == t2))
        | (not(exact) & (b2 <=_u reqBase) & (reqTop <=_u t2)))
    )
}
```

prop_setbounds_monotonic checks monotonicity preservation of setCapBounds. For this we use two pairs of base and length and set bounds first to one and then the other. In effect we perform all possible pairs of set bounds operation and check that the result respects monotonicity. This is not as trivial as it might seem due to the rounding that setCapBounds has to do. The reason we don't start with arbitrary capability bits is because some encodings are invalid and won't be generated by setCapBounds, in particular when the exponent

is at the maximum and T > B.

```
function prop_setbounds_monotonic(
        reqBase1 : CapAddrBits, reqLen1 : CapAddrBits,
        reqBase2 : CapAddrBits, reqLen2 : CapAddrBits) -> bool = {
    let (exact1, c1) = setCapBounds(root_cap_mem, reqBase1, reqLen1);
    let (b1, t1) = getCapBoundsBits(c1);
    let reqTop1 = (0b0 @ reqBase1) + (0b0 @ reqLen1);
    let saneTop1 = reqTop1 <=_u 0b1@0x00000000;
    let reqTop2 = (0b0 @ reqBase2) + (0b0 @ reqLen2);
    let saneTop2 = reqTop2 <=_u 0b1@0x00000000;
    let (exact2, c2) = setCapBounds(c1, reqBase2, reqLen2);
    let (b2, t2) = getCapBoundsBits(c2);
    let requestMonotonic = (b1 <=_u reqBase2) & (reqTop2 <=_u t1);
    let resultMonotonic = (b1 <=_u b2) & (t2 <=_u t1);
    (saneTop1 & saneTop2 & requestMonotonic) --> resultMonotonic
}
```

`prop_setaddr` checks that the flag returned by `setCapAddr` is sound with respect to the definition of representability i.e. if it returns 'representable' then the bounds remain unchanged. Note that a conservative implementation of the representability check may return false even though the bounds are unchanged, so the converse does not necessarily hold.

```
function prop_setaddr(reqBase : CapAddrBits, reqLen : CapAddrBits, newAddr :
    CapAddrBits) -> bool = {
    let (exact, c) = setCapBounds(root_cap_mem, reqBase, reqLen);
    let (representable, newCap) = setCapAddr(c, newAddr);
    let boundsEqual = capBoundsEqual(c, newCap);
    representable <--> boundsEqual
}
```

`prop_repbounds_c` checks the representable bounds match the minimum guarantee of the C language i.e. base..one past the end.

```
function prop_repbounds_c(reqBase : CapAddrBits, reqLen : CapAddrBits, newAddr
    : CapAddrBits) -> bool = {
    let (exact, c) = setCapBounds(root_cap_mem, reqBase, reqLen);
    let reqTop = (0b0 @ reqBase) + (0b0 @ reqLen);
    let saneTop = reqTop <=_u 0b1@0x00000000;
    let (b, t) = getCapBoundsBits(c);
    let (representable, newCap) = setCapAddr(c, newAddr);
    let inCBounds = (b <=_u newAddr) & ((0b0 @ newAddr) <=_u t);
    (saneTop & inCBounds) --> representable
}
```

`prop_repbounds` checks the representable bounds match expectations from the encoding, namely $[b, b + 2^{e+9})$.

```
function prop_repbounds(reqBase : CapAddrBits, reqLen : CapAddrBits, newAddr :
    CapAddrBits) -> bool = {
    let (exact, c) = setCapBounds(root_cap_mem, reqBase, reqLen);
    let reqTop = (0b0 @ reqBase) + (0b0 @ reqLen);
    let saneTop = reqTop <=_u 0b1@0x00000000;
    let (b, t) = getCapBoundsBits(c);
    let (representable, newCap) = setCapAddr(c, newAddr);
    let repTop = (0b0 @ b) + ((to_bits(33,1) << unsigned(c.E)) << 9);
    /* The representable bounds check: either E is max or address is in range
        */
    let inRepBounds = c.E == cap_max_E_bits | ((b <=_u newAddr) & ((0b0 @
        newAddr) <_u repTop));
    /* For any sane capability the inRepBounds check matches the flag returned
       by setCapAddr () */
    saneTop --> (inRepBounds <--> representable)
}
```

`prop_crrl_cram` checks that the `CRRL` and `CRAM` instructions can be used to derive a representable base and length for given requested base and length. Note that CRRL returns zero for non-zero `reqLen` in the case of rounding up to the max length.

```
function prop_crrl_cram(reqBase : CapAddrBits, reqLen : CapAddrBits) -> bool =
    {
    let mask = getRepresentableAlignmentMask(reqLen);
    let reqTop = (0b0 @ reqBase) + (0b0 @ reqLen);
    let saneTop = reqTop <=_u 0b1@0x00000000;
    let newBase = reqBase & mask;
    let newLen = getRepresentableLength(reqLen);
    let (exact, c) = setCapBounds(root_cap_mem, newBase, newLen);
    (saneTop & ((reqLen == zeros()) | (newLen != zeros())))-->
    (exact & reqLen <=_u newLen)
}
```

Appendix B

Permission compression rationale

To devise the permission compression scheme we initially observed a number of constraints on the useful permissions combinations:

$MC \rightarrow (LD \vee SD)$	Capability read / write is not useful without a load or store permission.
$LG \rightarrow (MC \wedge LD)$	Load global requires load capability.
$LM \rightarrow (MC \wedge LD)$	Load mutable requires load capability.
$SL \rightarrow (MC \wedge SD)$	Store local requires store capability.
$SR \rightarrow EX$	Access system registers only applies to executable capabilities.
$EX \rightarrow (MC \wedge LD)$	Executable capabilities require load capability for global access.
$\neg(EX \wedge SD)$	Executable capabilities should not be writable, to enhance security.
$\neg((SE \vee US \vee U0) \wedge (LD \vee SD))$	Memory permissions should be disjoint from sealing permissions due to separate namespaces.

If we apply all of these constraints we find there are only 33 useful permission combinations (excluding the global bit, which we take to be orthogonal). Eliminating just one of these combinations allows us to encode them using a 5-bit encoding. We see limited uses for write-only capabilities outside MMIO, so we chose write-only store-local as the least useful combination, leading to the encoding described in Section 7.13.1. This may be reassessed once we have a clearer picture of use cases for write-only capabilities.

Appendix C

Potential revised bound encoding

Here we describe some possible improvements to bounds encoding described in Section 7.13.3. In particular we seek to get a more efficient encoding (greater precision, fewer unusable encodings), and to reduce the complexity of the set bounds operation without using any more bits or excessive hardware complexity. The proposed encoding is a minor alteration to the existing one based on two observations:

1. That incrementing T in the set bounds operation would not be necessary if the lower e bits of top were decoded as ones, instead of zeros.
2. That zero length capabilities are of little use, and potentially even harmful (see Section 7.13.7)

As such we consider revising the existing bounds decoding as follows (note ones instead of zeros in low bits of top):

address, $a =$	$a_{\text{top}} = a[31 : e+9]$	$a_{\text{mid}} = a[e+8 : e]$	$a_{\text{low}} = a[e-1 : 0]$
base, $b =$	$a_{\text{top}} + c_{\text{b}}$	B	0
top, $t =$	$a_{\text{top}} + c_{\text{t}}$	T	$1 \ldots 1$

and further redefine the decoded top to be an *inclusive* bound instead of exclusive. The corrections c_b and c_t remain the same. This has a number of consequences:

- The set bounds implementation no longer has to increment the value of T if the requested top is not exactly represented because the decoding naturally rounds up as desired. This may slightly simplify the implementation.
- The smallest length encodable for a given e is 2^e (when $B = T$). Zero length capabilities are no longer supported.

- The largest length encodable for a given e is 2^{e+9} (when $T - B = 2^9 - 1$). However in this case the representable range is equal to the dereferenceable range, so it may be necessary to limit the maximum value of $T - B$ to $2^9 - 2$. This would ensure that 'one past the end' remains within representable range, but would make the maximum *usable* length $511 * 2^e$, which is the same as the current encoding.
- Bounds can be set to the entire address space using $B = 0$, $T = 2^9 - 1$, $e = 23$. This means the maximum exponent is smaller by one and therefore the worst case granularity is now 2^{23}, or 8 MiB instead of 16 MiB.

To retain software compatibility we do not propose to change the architectural definition of **top**, therefore `CGetLen` would have to take account of the inclusive top value (possibly by adding one to the result) and `CSetBounds` would have to subtract one from the requested length prior to encoding. Other instructions will have to adjust bounds checks accordingly. We have not yet fully evaluated this encoding to see if it is an overall improvement, but include it here for consideration.

Appendix D

Proposed compressed instruction encoding changes

Section 7.14 introduces changes to map certain compressed instructions to their CHERI counterparts, without changing the encoding of compressed instructions themselves. This brings minimum changes to the instruction decoder of an existing RISC-V CPU. However, further code size reduction can be achieved if we take advantage of the extra register index bits freed up by RV32E, or if we drastically redesign certain encodings in C extension entirely.

D.1 Compressed `CMove` and `CIncAddr`

The compiler generates `CMove` and `mv` to copy registers for capabilities and integers respectively. While `mv` has a compressed `c.mv` counterpart, `CMove` does not. Move instructions are often used to shuffle registers among argument, temporary and callee-saved registers for function calls, taking a significant proportion of the code size.

We take advantage of the freed bits from RV32E and use one bit to differentiate between `c.CMove` and `c.mv`. Initial code size investigations show a reduction of 5%, which is significant. An alternative for `c.CMove` is to conflate `c.CMove` with `c.mv`, which requires no extra bits from RV32E. However, implications of this conflation on ABI, architectural state and compiler have not been investigated.

Similarly, the extra bit in `c.addi` can encode `c.CIncAddr`. The benefit of a `c.CIncAddr` is less significant, at around 1-2%.

D.2 Three-operand compressed instructions

Currently, RISC-V compressed instructions take at most two operands, with certain instructions having full 5-bit register indices to address all 32 registers. Initial prototyping suggests that sacrificing the ability to address all registers and reducing immediate ranges to increase the number of operands gives further code size reduction. Increasing the number of operands increases the chance of pattern-matching uncompressed instructions with compressed ones, and the code size reduction outweighs the number of instructions that can no longer be compressed with a smaller immediate or register index. We currently see another 1-2% reduction on top of compressed `c.CMove` and `c.CIncAddr`. However, this is a drastic ISA change and needs further investigation for justification.

Appendix E

Standing on the Shoulders of Giants

The CHERIoT design is heavily based on prior work by the CHERI project and its myriad contributors and collaborators. Our targeted use case differs from those of earlier projects, so we present a brief tour through the existing landscape.

E.1 CTSRD CHERI, CHERI-RISC-V, Morello, and CheriBSD

The CTSRD project at the Cambridge University Computer Laboratory is the nucleation point for the growing CHERI ecosystem. The project's current architecture, as well as detailed rationale and discussion of its historical evolution, can be found in the CHERI ISA document (version 8, as of this writing) [22]. To summarize, however, its primary (micro)architectural focus has been on desktop- and server-class multi-core machines; these modern computers have 64-bit general data paths, Memory Management Units (MMUs) providing large virtual address spaces, caches, and so on. Its CHERI-RISC-V proposal extends RV64 specifically; similarly, Arm's experimental Morello extends 64-bit ARMv8 [3]. Correspondingly, its primary software focus has been on UNIX-like operating systems, centered around its adaptation of FreeBSD, CheriBSD [6].

E.1.1 Translation vs. Protection

CHERI is, informally, designed to "compose well" with modern (micro)architectures. By contrast to many prior capability systems, CHERI does not demand additional lookup tables to interpret its capabilities or define its protection policies. In the desktop/server space, CHERI sits atop existing MMUs: within a CheriBSD process, CHERI capabilities are interpreted relative to a MMU translation table. This interpretation continues to benefit from existing TLB designs.

By contrast, embedded systems have generally not had a notion of virtual addresses, conducting all their business using physical addresses as directly presented on a peripheral bus. As such, there has not been a convenient translation layer to press into service as a memory *protection* mechanism. Attempts to add lookup-table based "protection without translation" to physical memory systems, such as ARM's MPU [13] and RISC-V's PMP [21, §3.6], sit uncomfortably on the critical path for memory. The need to adjudicate every operation with a minimum of delay necessitates small tables fit into expensive, fast (T)CAMs. CHERI capabilities, by contrast, directly carry their permissions and bounds, obviating the need for tables and CAMs and necessitating only the check that each operation is authorized. Removing the need for tabular storage allows for greater system flexibility, as authority is now per reference rather than per address, and also removes risk of a (ephemerally) misconfigured table.

E.2 Incorporated CHERI Extensions

The CHERIoT ISA enshrines into its architecture several as-yet experimental aspects from the larger CHERI systems.

E.2.1 Multi-Root, Compressed Permission Encodings

The CHERIoT ISA is the first fully-elaborated example of a "multi-root," compressed permissions encoding scheme as first proposed in [22, §D.5]. Its three roots capture two of the suggested splittings: first, memory addresses from sealing types, and second, write from execute memory access.[1]

E.2.2 Recursive Permissions

The CHERIoT ISA has two "recursive" permissions (recall Section 7.13.1), borrowing an idea from many earlier systems. The first such is LM, directly analogous to Morello's recursive-load-mutable permission [3, §2.7.4] and the "weak" capabilities of KeyKOS and Coyotos [7, 10, 19]. A capability lacking LM and yet authorized to load capabilities, by having both LD ("load") and MC ("memory capabilities"), will cause both SD and LM to be stripped from capabilities loaded through it. That is, the capability in the register file resulting from a load instruction may differ from the capability loaded from memory by having these permission bits cleared. The other is LG and interacts with the 1-bit

[1] This latter splitting is often called either "W xor X" ("W^X"), following its introduction in OpenBSD 3.3 [17], or "Data Execution Prevention" (DEP), after Windows XP [11].

information flow system encoded in the GL ("global") and SL ("store local") permissions. Like lacking LM clears SD and LM, lacking LG clears GL and LG: all capabilities viewed transitively through a capability without LG appear to be local.

E.2.3 Architectural Seals

The CHERIoT ISA has a richer collection of architecturally-understood sealing types than the current larger CHERI-RISC-V baseline (see Section 7.13.2). (At the time of writing, Morello has its own set of additional architectural seals beyond those in CHERI-RISC-V.) In a future revision of the CHERIoT ISA, sentries will differentiate between forward and backward arcs, with the former requiring explicit authority to construct (unlike the larger systems, where the `CSealEnter` instruction has ambient authority) and the latter being constructable only as part of a jump.

E.2.4 Capability Load Barrriers and Memory-Capability Versioning

The CHERIoT ISA adapts the larger systems' MMU-based *capability load barriers* (CHERI-RISC-V [22, §5.3.10] and Morello [3, R_{CPRKD} in §2.14]) directly into the processor pipeline (recall Section 7.8). For this, CHERIoT adds a second metadata bit, in addition to the CHERI capability tag, to each capability-sized granule. Capabilities whose base refers to a granule with this second metadata bit asserted are invalid and will have their tag cleared when loaded into the register file. This behavior can be seen as a "1-color-bit" scaling-down of the proposed "Memory-Capability Versioning" scheme of CTSRD CHERI [22, §D.6], which builds on technologies like Arm's Memory Tagging Extension [14]. However, CHERIoT has done away with the "version" field within the capability format, as we found it straightforward for the heap allocator to always internally use capabilities whose bounds cover the entire heap and, importantly, whose base granule is never made invalid.

E.3 Esswood's CheriOS

While the CTSRD group's focus has been largely on UNIX-like software, there have been deviations over the years. Esswood's CheriOS [8] describes a green-field microkernel operating system that presumes a CHERI ISA. The entire system, OS and user programs alike, run in the same *single-address-space*; the MMU is used for page mapping tricks, but all software perceives the same address space (albeit through different capabilities). Like the CHERIoT RTOS, CheriOS is *ringless*; the traditional separation of a more-privileged supervisor operating over a less-privileged userspace is no more. The core Trusted Computing Base for CheriOS, analogous to the CHERIoT RTOS's switch routines, consists of

fewer than 2700 CHERI-MIPS instructions. Both systems move functionality traditionally placed in inner rings out to minimally-privileged compartments.

CheriOS software is generally composed of a number of libraries linked together; linkage policy articulates the trust relationships between callers and callees, offering all of mutually-trusting, sandbox, safebox, and mutually-distrusting call relationships. The CHERIoT RTOS also relies on linkage to define a capability graph, but it offers fewer flavors of cross-compartment calls (most calls are mutually-distrusting with library calls being a minimal safebox).

CheriOS offers full memory safety for C/C++ programs. As expected, CHERI provides integrity and monotonicity, and the memory allocators and compiler-generated code insert bounding instructions where appropriate for spatial safety. CheriOS achieves heap and stack temporal safety by exploiting the large address spaces of server-class machines. Heap temporal safety is achieved in the TCB by a combination of take-once "reservations" of address space and quarantining released virtual address space. A hardware barrier added to CHERI-MIPS facilitates revoking capabilities pointing into a large contiguous region of released address space, which the TCB may then recycle for new reservations. Stack temporal safety builds atop heap temporal safety, with the compiler arranging to construct "slinky stacks" [8, §4.1] that do not reuse possibly-escaped stack allocations' address space. When a segment of a slinky stack has too little usable address space, it is recycled as heap memory and another large heap allocation is made to provide a fresh segment. The CHERIoT RTOS, by contrast, must operate on physical addresses and so builds its heap temporal safety using additional CHERIoT ISA metadata and its slightly weaker notion of stack temporal safety using the store-local permission.

E.4 Xia's CheriRTOS

Similarly, there have been efforts within the CTSRD group to explore CHERI in smaller hardware. Xia (also an author of this report) developed, for his Ph.D. thesis, CheriRTOS [24, 25]. CheriRTOS introduces the first 64-bit CHERI capability encodings (for 32-bit address spaces) and explores its implementation in a derivative of CHERI-MIPS. The capability scheme used here was largely a straightforward scaling-down of CTSRD's server-oriented CHERI. While it was an important and admirable proof of concept, its deficiencies, especially with bounds precision and the large number of bits still devoted to permissions, significantly motivated the development of the CHERIoT ISA capability format.

CheriRTOS, like many RTOSes before it, takes a *task-centric* perspective on the world, pairing together threads and their associated data and relying on inter-task communication when work requires access to multiple tasks' state. Being the first of its kind,

CheriRTOS is focused on aspects of cross-compartment invocation and allocator security; tasks are manually-specified concepts and are compiled with only a limited understanding of CHERI capabilities.[2] It then uses CHERI mechanisms to provide *task isolation*. By contrast, the CHERIoT RTOS decomposes computations (threads and their runtime stacks) from persistent resources (compartments and their associated globals and import tables), allowing threads to enter and exit different compartments, as needed.

E.5 Almatary's CompartOS and CheriFreeRTOS

Almatary's Ph.D. thesis develops the "CompartOS" software model [2] and instantiates it with an adaptation of FreeRTOS, called CheriFreeRTOS [1]. The Trusted Computing Base for CompartOS is its secure *dynamic* code loader. The unit of compartmentalization is, as in the CHERIoT RTOS, a *linkage unit* rather than a task (though the CHERIoT RTOS provides only static code loading). CompartOS includes compartment *availability* in its design objectives, and so offers a multitude of fault-handling mechanisms, including per-compartment trap handlers and forced stack unwinds as offered in the CHERIoT RTOS. However, while CompartOS is a flexible model, the concrete CheriFreeRTOS does not consider temporal memory safety to be in scope; it lacks a mechanism for capability revocation and does not scrub stacks on compartment switches.
CheriFreeRTOS was used in an extensive evaluation which explored the relative costs and security of both task- and linkage-based compartmentalization models atop both CHERI- and MPU/PMP-enabled ISAs [1, §5]. This evaluation found that, while the microarchitectural area costs of CHERI and PMPs (and MMUs) were similar, the security benefits of CHERI were "unparalleled" and came with lower runtime cost than comparably-secure MPU/PMP-based approaches. Further, the cost of adapting FreeRTOS to the CompartOS model was found to be minimal, with many applications requiring no source-level changes.

[2] Specifically, task C/C++ code is compiled in a "hybrid" mode, where pointers lower to *integers* unless a capability is explicitly requested. These integers are interpreted relative to an architectural *capability* register, either the program counter (for relative control transfers or PC-relative loads) or a "default data capability" ("DDC," which interposes legacy integer-based load and store instructions) [22, §2.3.12]. The CHERIoT ISA does not have these legacy instructions or DDC; all CHERIoT RTOS code is expected to be compiled "purecap," with all pointers always lowered to capabilities.

Bibliography

[1] H. Almatary. 'CHERI compartmentalisation for embedded systems'. PhD thesis. University of Cambridge, Computer Laboratory, Nov. 2022. DOI: 10.48456/tr-976. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-976.pdf.

[2] H. Almatary, M. Dodson, J. Clarke, P. Rugg, I. Gomes, M. Podhradsky, P. G. Neumann, S. W. Moore and R. N. M. Watson. 'CompartOS: CHERI Compartmentalization for Embedded Systems'. In: (2022). URL: https://arxiv.org/abs/2206.02852.

[3] *Arm Architecture Reference Manual Supplement: Morello for A-profile Architecture*. Version A.k. Arm Limited. 2020. URL: https://developer.arm.com/documentation/ddi0606/latest.

[4] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami and P. Sewell. 'ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS'. In: ***POPL 2019**: Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*. Jan. 2019. DOI: 10.1145/3290384.

[5] Capability Hardware Enhanced RISC Instructions. *CTSRD-CHERI/sail-cheri-riscv: CHERI-RISC-V model written in Sail*. URL: https://github.com/CTSRD-CHERI/sail-cheri-riscv.

[6] B. Davis, R. N. M. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka, A. Joannou, B. Laurie, A. T. Markettos, J. E. Maste, A. Mazzinghi, E. T. Napierala, R. M. Norton, M. Roe, P. Sewell, S. Son and J. Woodruff. 'CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment'. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2019 (Providence, RI, USA). ACM, Apr. 2019, pp. 379–393. DOI: 10.1145/3297858.3304042. URL:

https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201904-asplos-cheriabi.pdf.

[7] M. S. Doerrie. 'Confidence in Confinement: An Axiom-free, Mechanized Verification of Confinement in Capability-based Systems'. PhD thesis. Johns Hopkins University, 2015. URL: http://www.doerrie.us/assets/doerrie-dissertation-jhu.pdf.

[8] L. G. Esswood. 'CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor'. PhD thesis. University of Cambridge, Computer Laboratory, Sept. 2021. DOI: 10.48456/tr-961. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-961.pdf.

[9] N. W. Filardo, B. F. Gutstein, J. Woodruff, S. Ainsworth, L. Paul-Trifu, B. Davis, H. Xia, E. T. Napierala, A. Richardson, J. Baldwin, D. Chisnall, J. Clarke, K. Gudka, A. Joannou, A. T. Markettos, A. Mazzinghi, R. M. Norton, M. Roe, P. Sewell, S. Son, T. M. Jones, S. W. Moore, P. G. Neumann and R. N. M. Watson. 'Cornucopia: Temporal Safety for CHERI Heaps'. In: *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, May 2020, pp. 1507–1524. DOI: 10.1109/SP40000.2020.00098. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2020oakland-cornucopia.pdf.

[10] N. Hardy. 'KeyKOS architecture'. In: *SIGOPS Operating Systems Review* 19.4 (1985), pp. 8–25. DOI: 10.1145/858336.858337.

[11] M. Inc. *Data Execution Prevention - Win32 apps*. URL: https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

[12] LF Projects, LLC. *seL4 FAQ*. https://www.sel4.systems/Info/FAQ/proof.pml. 2023.

[13] A. Limited. *Armv8-M Memory Model and Memory Protection User Guide*. URL: https://developer.arm.com/documentation/107565/0100/.

[14] A. Limited. *Armv8.5-A Memory Tagging Extension White Paper*. URL: https://developer.arm.com/documentation/102925/0100/.

[15] Microsoft. *CHERIoT ISA model written in Sail*. URL: https://github.com/microsoft/cheriot-sail.

[16] Microsoft. *CHERIoT RTOS*. URL: https://github.com/microsoft/cheriot-rtos.

[17] T. de Raadt. *OpenBSD 3.3*. URL: https://www.openbsd.org/33.html.

[18] REMS Project. *Sail Language*. https://www.cl.cam.ac.uk/~pes20/sail/. 2018.

[19] J. S. Shapiro and J. W. Adams. *Coyotos Microkernel Specification. Version 0.6+*. 10th Sept. 2007. URL: https://web.archive.org/web/20160904092954/http://www.coyotos.org:80/docs/ukernel/spec.html.

[20] A. Waterman and K. Asanović, eds. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL: https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[21] A. Waterman and K. Asanović, eds. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10*. May 2017. URL: https://content.riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf.

[22] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, D. Chisnall, J. Clarke, B. Davis, N. W. Filardo, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son and H. Xia. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)*. Tech. rep. UCAM-CL-TR-951. University of Cambridge, Computer Laboratory, Sept. 2020. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf.

[23] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. Norton, T. Baureiss, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, A. T. Markettos, M. Roe, P. G. Neumann, R. N. M. Watson and S. W. Moore. 'CHERI Concentrate: Practical Compressed Capabilities'. In: *IEEE Transactions on Computers* 68.10 (Oct. 2019), pp. 1455–1469. DOI: 10.1109/TC.2019.2914037. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2019tc-cheri-concentrate.pdf.

[24] H. Xia. 'Capability memory protection for embedded systems'. PhD thesis. University of Cambridge, Computer Laboratory, Feb. 2021. DOI: 10.48456/tr-955. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-955.pdf.

[25] H. Xia, J. Woodruff, H. Barral, L. Esswood, A. Joannou, R. Kovacsics, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Richardson, S. W. Moore and R. N. M. Watson. 'CheriRTOS: A Capability Model for Embedded Devices'. In: *2018 IEEE 36th International Conference on Computer Design (ICCD)*. 2018 IEEE 36th International Conference on Computer Design (ICCD). IEEE, Oct. 2018, pp. 92–99. DOI: 10.1109/ICCD.2018.00023. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201810-iccd2018-cheri-rtos.pdf.