Technical Report

UCAM-CL-TR-927
ISSN 1476-2986

Number 927

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7)

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

June 2019

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Approved for public release; distribution is unlimited. Sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 (“CTSRD”), FA8750-11-C-0249 (“MRC2”), HR0011-18-C-0016 (“ECATS”), and FA8650-18-C-7809 (“CIFV”) as part of the DARPA CRASH, MRC, and SSITH research programs. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government. Additional support was received from St John’s College Cambridge, the Google SOAAP Focused Research Award, a Google Chrome University Research Program Award, the RCUK’s Horizon Digital Economy Research Hub Grant (EP/G065802/1), the EPSRC REMS Programme Grant (EP/K008528/1), the EPSRC Impact Acceleration Account (EP/K503757/1), the EPSRC IOSEC grant (EP/EP/R012458/1), the ERC Advanced Grant ELVER (789108), the Isaac Newton Trust, the UK Higher Education Innovation Fund (HEIF), Thales E-Security, Microsoft Research Cambridge, Arm Limited, Google DeepMind, HP Enterprise, and a Gates Cambridge Scholarship.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

This technical report describes CHERI ISAv7, the seventh version of the Capability Hardware Enhanced RISC Instructions (CHERI) Instruction-Set Architecture (ISA) being developed by SRI International and the University of Cambridge. This design captures nine years of research, development, experimentation, refinement, formal analysis, and validation through hardware and software implementation. CHERI ISAv7 is a substantial enhancement to prior ISA versions. We differentiate an architecture-neutral protection model vs. architecture-specific instantiations in 64-bit MIPS, 64-bit RISC-V, and x86-64. We have defined a new CHERI Concentrate compression model. CHERI-RISC-V is more substantially elaborated. A new compartment-ID register assists in resisting microarchitectural side-channel attacks. Experimental features include linear capabilities, capability coloring, temporal memory safety, and 64-bit capabilities for 32-bit architectures.

CHERI is a *hybrid capability-system architecture* that adds new capability-system primitives to commodity 64-bit RISC ISAs, enabling software to efficiently implement *fine-grained memory protection* and *scalable software compartmentalization*. Design goals include incremental adoptability within current ISAs and software stacks, low performance overhead for memory protection, significant performance improvements for software compartmentalization, formal grounding, and programmer-friendly underpinnings. We have focused on providing strong, non-probabilistic, efficient architectural foundations for the principles of *least privilege* and *intentional use* in the execution of software at multiple levels of abstraction, preventing and mitigating vulnerabilities.

The CHERI system architecture purposefully addresses known performance and robustness gaps in commodity ISAs that hinder the adoption of more secure programming models centered around the principle of least privilege. To this end, CHERI blends traditional paged virtual memory with an in-address-space capability model that includes capability registers, capability instructions, and tagged memory. CHERI builds on the C-language fat-pointer literature: its capabilities can describe fine-grained regions of memory, and can be substituted for data or code pointers in generated code, protecting data and also improving control-flow robustness. Strong capability integrity and monotonicity properties allow the CHERI model to express a variety of protection properties, from enforcing valid C-language pointer provenance and bounds checking to implementing the isolation and controlled communication structures required for software compartmentalization.

CHERI's hybrid capability-system approach, inspired by the Capsicum security model, allows incremental adoption of capability-oriented design: software implementations that are more robust and resilient can be deployed where they are most needed, while leaving less critical software largely unmodified, but nevertheless suitably constrained to be incapable of having adverse effects. Potential deployment scenarios include low-level software Trusted Computing Bases (TCBs) such as separation kernels, hypervisors, and operating-system kernels, as well as userspace TCBs such as language runtimes and web browsers. We also see potential early-use scenarios around particularly high-risk software libraries (such as data compression, protocol parsing, and image processing), which are concentrations of both complex and historically vulnerability-prone code exposed to untrustworthy data sources, while leaving containing applications unchanged.

Acknowledgments

The authors of this report thank members of the CTSRD, MRC2, ECATS, CIFV, and REMS teams, our past and current research collaborators at SRI and Cambridge, as well as colleagues at other institutions who have provided invaluable feedback and continuing support throughout this work:

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

The CTSRD team also thanks past and current members of its external oversight group for significant support and contributions:

Lee Badger
Simon Cooper
Rance DeLong
Jeremy Epstein
Virgil Gligor
Li Gong
Mike Gordon
Steven Hand
Andrew Herbert
Warren A. Hunt Jr.
Doug Maughan
Greg Morrisett
Brian Randell
Kenneth F. Shotting
Joe Stoy
Tom Van Vleck
Samuel M. Weber

We would also like to acknowledge the late David Wheeler and Paul Karger, whose conversations with the authors about the CAP computer and capability systems contributed to our thinking, and whose prior work provided considerable inspiration.

Finally, we are grateful to Howie Shrobe, MIT professor and past DARPA CRASH and MRC program manager, who offered both technical insight and support throughout this work. We are also grateful to Robert Laddaga, Stu Wagner, and Jonathan Smith, who succeeded Howie in overseeing the CRASH program, John Launchbury (DARPA I2O office director), Dale Waters (DARPA AEO office director), Linton Salmon (DARPA SSITH program manager), Daniel Adams and Laurisa Goergen (DARPA I2O SETAs supporting the CRASH and MRC programs), and Marnie Dunsmore and John Marsh (DARPA MTO SETAs supporting the SSITH program).

Contents

Chapter 1

Introduction

CHERI (Capability Hardware Enhanced RISC Instructions) extends commodity RISC Instruction-Set Architectures (ISAs) with new capability-based primitives that improve software robustness to security vulnerabilities. The CHERI model is motivated by the *principle of least privilege*, which argues that greater security can be obtained by minimizing the privileges accessible to running software. A second guiding principle is the *principle of intentional use*, which argues that, where many privileges are available to a piece of software, the privilege to use should be explicitly named rather than implicitly selected. While CHERI does not prevent the expression of vulnerable software designs, it provides strong *vulnerability mitigation*: attackers have a more limited vocabulary for attacks, and should a vulnerability be successfully exploited, they gain fewer rights, and have reduced access to further attack surfaces. CHERI allows software privilege to be minimized at two granularities:

Fine-grained code protection CHERI provides support for *fine-grain protection* and *intentional use* through in-address-space *memory capabilities*, which replace integer virtual-address representations of code and data pointers. The aim here is to minimize the rights available to be exercised on an instruction-by-instruction basis, limiting the scope of damage from inevitable software bugs. CHERI capabilities protect the integrity and valid provenance of pointers themselves, as well as allowing fine-grained protection of the in-memory data and code that pointers refer to. These protection policies can, to a large extent, be based on information already present in program descriptions – e.g., from C-language types, memory allocators, and run-time linking. This application of least privilege and intentional use provides strong protection against a broad range of memory- and pointer-based vulnerabilities and exploit techniques – buffer overflows, format-string attacks, pointer injection, data-pointer-corruption attacks, control-flow attacks, and so on. Many of these goals can be achieved through code recompilation on CHERI.

Secure encapsulation At a coarser granularity, CHERI also supports *secure encapsulation* and *intentional use* through the robust and efficient implementation of highly scalable in-address-space *software compartmentalization* using *object capabilities*. The aim here is to minimize the set of rights available to larger isolated software components, building on efficient architectural support for strong software encapsulation. These protections are grounded in explicit descriptions of isolation and communication provided by software authors, such as through explicit software sandboxing. This application of least privilege and intentional use provides strong mitigation of application-level vulnerabilities, such as logical errors, downloaded malicious code, or software Trojans inserted in the software supply chain.

Effective software compartmentalization depends on explicit software structure, and can require significant code change. Where compartmentalization already exists in software, CHERI can be used to significantly improve compartmentalization performance and granularity. Where that structure is not yet present, CHERI can improve the adoption path for compartmantalization due to supporting in-address-space compartmentalization models.

CHERI is designed to support incremental adoption within current security-critical, C-language *Trusted Computing Bases (TCBs)*: operating-system (OS) kernels, key system libraries and services, language runtimes supporting higher-level type-safe languages, and applications such as web browsers and office suites. While CHERI builds on many historic ideas about capability systems (see Chapter 11), one of the key contributions of this work is CHERI's *hybrid capability-system architecture*. In this context, *hybrid* refers to combining aspects from conventional architectures, system software, and language/compiler choices with capability-oriented design. Key forms of hybridization in the CHERI design include:

A RISC capability system A capability-system model is blended with a conventional RISC user-mode architecture without disrupting the majority of key RISC design choices.

An MMU-enabled capability system A capability-system model is cleanly and usefully composed with conventional ring-based privilege and virtual memory based on MMUs (Memory Management Units).

A C-language capability system CHERI can be targeted by a C/C++-language compiler with strong compatibility, performance, and protection properties.

Hybrid system software CHERI supports a range of OS models including conventional MMU-based virtual-memory designs, hybridized designs that host capability-based software within multiple virtual address spaces, and pure single-address-space capability systems.

Incremental adoptability Within pieces of software, capability-aware design can be disregarded, partially adopted, or fully adopted with useful and predictable semantics. This allows incremental adoption within large software bases, from OS kernels to application programs.

We hope that these hybrid aspects of the design will support gradual deployment of CHERI features in existing software, rather than obliging a clean-slate software design, thereby offering a more gentle hardware-software adoption path.

In the remainder of this chapter, we describe our high-level design goals for CHERI, the notion that CHERI is an architecture-neutral protection model with architecture-specific mappings (such as CHERI-MIPS and CHERI-RISC-V), an introduction to the CHERI-MIPS concrete instantiation, a brief version history, an outline of the remainder of this report, and our publications to date on CHERI. A more detailed discussion of our research methodology, including motivations, threat model, and evolving approach from ISA-centered prototyping to a broader architecture-neutral protection model may be found in Chapter 10. Historical context and related work for CHERI may be found in Chapter 11. The Glossary at the end of the report contains stand-alone definitions of many key ideas and terms, and may be useful reference material when reading the report.

1.1 CHERI Design Goals

CHERI has three central design goals aimed at dramatically improving the security of contemporary C-language TCBs, through processor support for fine-grained memory protection and scalable software compartmentalization, whose (at times) conflicting requirements have required careful negotiation in our design:

Fine-grained memory protection improves software resilience to escalation paths that allow low-level software bugs involving individual data structures and data-structure manipulations to be coerced into more powerful software vulnerabilities; e.g., through remote code injection via buffer overflows, control-flow and data-pointer corruption, and other memory-based techniques. Unlike MMU-based memory protection, CHERI memory protection is intended to be driven by the compiler in protecting programmer-described data structures and references, rather than via coarse page-granularity protections. CHERI capabilities limit how pointers can be used by scoping the ranges of memory (via bounds) and operations that can be performed (via permissions). They also protect the integrity, provenance, and monotonicity of pointers in order to prevent inadvertent or inappropriate manipulation that might otherwise lead to privilege escalation.

Memory capabilities may be used to implement data pointers (protecting against a variety of data-oriented vulnerabilities such as overflowing buffers) and also to implement code pointers (supporting the implementation of control-flow integrity by preventing corrupted code pointers and return addresses from being used). Fine-grained protection also provides the foundation for expressing compartmentalization within application instances. We draw on, and extend, ideas from recent work in C-language *software bounds checking* by combining *fat pointers* with capabilities, allowing capabilities to be substituted for C pointers with only limited changes to program semantics.

CHERI permits efficient implementation of dialects of C and C++ in which various invalid accesses, deemed to be undefined behavior in those languages, and potentially giving arbitrary behavior in their implementations, are instead guaranteed to throw an exception.

Software compartmentalization involves the decomposition of software (at present, primarily application software) into isolated components to mitigate the effects of security vulnerabilities by applying sound principles of security, such as abstraction, encapsulation, type safety, and especially least privilege and the minimization of what must be trustworthy (and therefore sensibly trusted!). Previously, it seems that the adoption of compartmentalization has been limited by a conflation of hardware primitives for virtual addressing and separation, leading to inherent performance and programmability problems when implementing fine-grained separation. Specifically, we seek to decouple the virtualization from separation to avoid scalability problems imposed by MMUs based on translation look-aside buffers (TLBs), which impose a very high performance penalty as the number of protection domains increases, as well as complicating the writing of compartmentalized software.

A viable transition path must be applicable to current software and hardware designs. CHERI hardware must be able to run most current software without significant modification, and allow incremental deployment of security improvements starting with the most critical software components: the TCB foundations on which the remainder of the system rests, and software with the greatest exposure to risk. CHERI's features must significantly improve security, to create demand for upstream processor manufacturers from their downstream mobile and embedded device vendors. These CHERI features must at the same time conform to vendor expectations for performance, power use, and compatibility to compete with less secure alternatives.

We draw on *formal methodologies* wherever feasible, to improve our confidence in the design and implementation of CHERI. This use is necessarily subject to real-world constraints of timeline, budget, design process, and prototyping, but it has helped increase our confidence that CHERI meets our functional and security requirements. Formal methods can also help to avoid many of the characteristic design flaws that are common in both hardware and software. This desire requires us not only to perform research into CPU and software design, but also to develop new formal methodologies, and adaptations and extensions of existing ones.

We are concerned with satisfying the need for trustworthy systems and networks, where *trustworthiness* is a multidimensional measure of how well a system or other entity satisfies its various requirements – such as those for security, system integrity, and reliability, as well as human safety, and total-system survivability, robustness, and resilience, notably in the presence of a wide range of adversities such as hardware failures, software flaws, malware, accidental and intentional misuse, and so on. Our approach to trustworthiness encompasses hardware and software architecture, dynamic and static evaluation, formal and non-formal analyses, good software-engineering practices, and much more.

1.2 Architecture Neutrality and Architectural Instantiations

CHERI consists of an architectural-neutral protection model, and a set of instantiations of that model across multiple ISAs. Our initial mapping into the 64-bit MIPS ISA has allowed us to develop the CHERI approach; we have now expanded to include a more elaborated mapping into the 64-bit RISC-V ISA, and a sketch mapping into the x86-64 ISA. In doing so, we have attempted to maximize the degree to which specification is architecture neutral, and minimize the degree to which it is architecture specific. Even within a single ISA, there are multiple potential instantiations of the CHERI protection model, which offer different design tradeoffs

– for example, decisions about whether to have separate integer and capability register files or to merge them into a single register file.

The successful mapping into multiple ISAs has led us to believe that the CHERI protection model is a portable protection model, that support portable software stacks in much the same way that portable virtual-memory-based operating systems can be implemented across a variety of architectural MMUs. Unlike MMUs, whose software interactions are primarily with the operating system, CHERI interacts directly with compiler-generated code, key system libraries, compartmentalization libraries, and applications; across all of these, we have found that an architecture-neutral approach can be highly effective, offering portability to the vast majority of CHERI-aware C/C++ code. We first consider the architecture-neutral model, and then applications of our approach in specific ISAs.

1.2.1 The Architecture-Neutral CHERI Protection Model

The aim of the CHERI protection model, as embodied in both the software stack (see Chapter 2) and architecture (see Chapter 3), is to support two vulnerability mitigation objectives: first, fine-grained pointer and memory protection within address spaces, and second, primitives to support both scalable and programmer-friendly compartmentalization within address spaces. The CHERI model is designed to support low-level TCBs, typically implemented in C or a C-like language, in workstations, servers, mobile devices, and embedded devices. In contrast to MMU-based protection, this is done by protecting *references to code and data* (pointers), rather than the *location of code and data* (virtual addresses). This is accomplished via an *in-address-space capability-system model*: the architecture provides a new primitive, the *capability*, that software components (such as the OS, compiler, run-time linker, compartmentalization runtime, heap allocator, etc.) can use to implement strongly protected pointers within virtual address spaces.

In the security literature, capabilities are tokens of authority that are unforgeable and delegatable. *CHERI capabilities* are integer virtual addresses that have been extended with metadata to protect their integrity, limit how they are manipulated, and control their use. This metadata includes a *tag* implementing strong integrity protection (differentiating valid and invalid capabilities), *bounds* limiting the range of addresses that may be dereferenced, *permissions* controlling the specific operations that may be performed, and also *sealing*, used to support higher-level software encapsulation. Protection properties for capabilities include the ISA ensuring that capabilities are always derived via valid manipulations of other capabilities (*provenance*), that corrupted in-memory capabilities cannot be dereferenced (*integrity*), and that rights associated with capabilities are non-increasing (*monotonicity*).

CHERI capabilities may be held in registers or in memories, and are loaded, stored, and dereferenced using CHERI-aware instructions that expect capability operands rather than integer virtual addresses. On hardware reset, initial capabilities are made available to software via special and general-purpose capability registers. All other capabilities will be derived from these initial valid capabilities through valid capability transformations.

In order to continue to support non-CHERI-aware code, dereference of integer virtual addresses via legacy instruction is transparently indirected via a *default data capability* (**DDC**) for loads and stores, or a *program-counter capability* (**PCC**) for instruction fetch.

A variety of programming-language and code-generation models can be used with a CHERI-extended ISA. As integer virtual addresses continue to be supported, C or C++ compilers might choose to always implement pointers via integers, selectively implement certain pointers as capabilities based on annotations or type information (i.e., a hybrid C interpretation), or alternatively always implement pointers as capabilities except where explicitly annotated (i.e., a *pure-capability* interpretation). Programming languages may also employ capabilities internal to their implementation: for example, to protect return addresses, vtable pointers, and other virtual addresses for which capability protection can provide enhanced vulnerability mitigation.

When capabilities are being used to implement pointers (e.g., to code or data) or internal addresses (e.g., for return addresses), they must be constructed with suitably restricted rights, to accomplish effective protection. This is a run-time operation performed using explicit instructions (e.g., to set bounds, mask permissions, or seal capabilities) by the operating system, run-time linker, language runtime and libraries, and application code itself:

The operating-system kernel may narrow bounds and permissions on pointers provided as part of the start-up environment when executing a program binary (e.g., to arguments or environmental variables), or when returning pointers from system calls (e.g., to new memory mappings).

The run-time linker may narrow bounds and permissions when setting up code pointers or pointers to global variables.

The system library may narrow bounds and permissions when returning a pointer to newly allocated heap memory.

The compartmentalization runtime may narrow bounds and permissions, as well as seal capabilities, enforcing compartment isolation (e.g., to act as sandboxes).

The compiler may insert instructions to narrow bounds and permissions when generating code to take a pointer to a stack allocation, or when taking a pointer to a field of a larger structure allocated as a global, on the stack, or on the heap.

The language runtime may narrow bounds and permissions when returning pointers to newly allocated objects, or when setting up internal linkage, as well as seal capabilities to non-dereferenceable types.

The application may request changes to permissions, bounds, and other properties on pointers, in order to further subset memory allocations and control their use.

The CHERI model can also be used to implement other higher-level protection properties. For example, tags on capabilities in memory can be used to support accurate C/C++-language temporal safety via revocation or garbage collection, and sealed capabilities can be used to enforce language-level encapsulation and type-checking features. The CHERI protection model and its implications for software security are described in detail in Chapter 2.

CHERI is an *architecture-neutral protection model* in that, like virtual memory, it can be deployed within multiple ISAs. In developing CHERI, we initially considered it as a concrete extension to the 64-bit MIPS ISA; using it, we could explore the implications downwards into the microarchitecture, and upwards into the software stack. Having developed a mature hardware-software protection model, we used this as the baseline in deriving an architecture-neutral CHERI protection model. This architecture-neutral model is discussed in detail in Chapter 3. We have demonstrated the possibility of adding CHERI protection to more than one base ISA by providing a detailed concrete instantiation for the 64-bit MIPS ISA (Chapter 4), a draft instantiation in the RISC-V ISA (Chapter 5), and a lightweight architectural sketch for the x86-64 ISA (Chapter 6).

1.2.2 An Architecture-Specific Mapping into 64-bit MIPS

The CHERI-MIPS ISA (see Chapter 4) is an instantiation of the CHERI protection model as an extension to the 64-bit MIPS ISA [50]. CHERI adds the following features to the MIPS ISA[1] to support granular memory protection and compartmentalization within address spaces:

Capability registers describe the rights (*protection domain*) of the executing thread to access memory, and to invoke object references to transition between protection domains. We model these registers as a separate *capability register file*, supplementing the general-purpose integer register file.

Capability registers contain a tag, object type, permission mask, base, length, and offset (allowing the description of not just a bounded region, but also a pointer into that region, improving C-language compatibility). Capability registers are suitable for describing both data and code, and can hence protect both data integrity/confidentiality and control flow. Certain registers are reserved for use in exception handling; all others are available to be managed by the compiler using the same techniques used with conventional registers. Over time, we imagine that software will increasingly use capabilities rather than integers to describe data and object references.

Another potential integration into the ISA (which would maintain the same CHERI protection semantics) would be to extend the existing general-purpose integer registers so that they could also hold capabilities. This might reduce the hardware resources required to implement CHERI support. However, we selected our current approach to maintain consistency with the MIPS ISA extension model (in which coprocessors have independent register files), and to minimize *Application Binary Interface (ABI)* disruption on boundaries between legacy and CHERI-aware code for the purposes of rapid architectural and software iteration. We explore the potential space of mappings from the CHERI model into the ISA in greater detail in Section 3.10.1, as well as in Chapters 5 and 6 where we consider alternative mappings into non-MIPS ISAs.

Capability instructions allow executing code to create, constrain (e.g., by reducing bounds or permissions), manage, and inspect capability register values. Both unsealed (memory) and sealed (object) capabilities can be loaded and stored via memory capability registers

[1] Formally, CHERI instructions are added to MIPS as a *MIPS coprocessor* – a reservation of opcode space intended for third-party use. Despite the suggestive term “coprocessor”, CHERI support will typically be integrated tightly into the processor pipeline, memory subsystem, and so on. We therefore eschew use of the term.

(i.e., dereferencing). Object capabilities can be invoked, via special instructions, allowing a transition between protection domains, but are *immutable* and *non-dereferenceable*, providing encapsulation of the code or data that they refer to. Capability instructions implement *guarded manipulation*: invalid capability manipulations (e.g., to increase rights or length) and invalid capability dereferences (e.g., to access outside of a bounds-checked region) result in an exception that can be handled by the supervisor or language runtime. A key aspect of the instruction-set design is *intentional use of capabilities*: explicit capability registers, rather than ambient authority, are used to indicate exactly which rights should be exercised, to limit the damage that can be caused by exploiting bugs. Tradeoffs exist around intentional use, and in some cases compatibility or opcode utilization may dictate implicit capability selection; for example, legacy MIPS load and store instructions implicitly dereference a Default Data Capability as they are unable to explicitly name a capability register. Most capability instructions are part of the user-mode ISA, rather than the privileged ISA, and will be generated by the compiler to describe application data structures and protection properties.

Tagged memory associates a 1-bit tag with each capability-aligned and capability-sized word in physical memory, which allows capabilities to be safely loaded and stored in memory without loss of integrity. Writes to capability values in memory that do not originate from a valid capability in the capability register file will clear the tag bit associated with that memory, preventing accidental (or malicious) dereferencing of invalid capabilities.

This functionality expands a thread's effective protection domain to include the transitive closure of capability values that can be loaded via capabilities via those present in its register file. For example, a capability register representing a C pointer to a data structure can be used to load further capabilities from that structure, referring to further data structures, which could not be accessed without suitable capabilities.

Non-bypassable tagging of unforgeable capabilities enables not only reliable and secure enforcement of capability properties, but also reliable and secure identification of capabilities in memory for the purposes of implementing other higher-level protection properties such as temporal safety.

In keeping with the RISC philosophy, CHERI instructions are intended for use primarily by the operating system and compiler rather than directly by the programmer, and consist of relatively simple instructions that avoid (for example) combining memory access and register value manipulation in a single instruction. In our current software prototypes, there are direct mappings from programmer-visible C-language pointers to capabilities in much the same way that conventional code generation translates pointers into general-purpose integer register values; this allows CHERI to continuously enforce bounds checking, pointer integrity, and so on. There is likewise a strong synergy between the capability-system model, which espouses a separation of policy and mechanism, and RISC: CHERI's features make possible the implementation of a wide variety of OS, compiler, and application-originated policies on a common protection substrate that optimizes fast paths through hardware support.

Our prototype of this approach, instantiating our ideas about CHERI capability access to a specific instruction set (the 64-bit MIPS ISA) has necessarily led to a set of congruent implementation decisions about register-file size, selection of specific instructions, exception handling, memory alignment requirements, and so on, that reflect that starting-point ISA. These decisions might be made differently with another starting-point ISA as they are simply surface features of the underlying approach; we anticipate that adaptations to ISAs such as ARM, RISC-V, and x86-64 would adopt instruction-encoding conventions, and so on, more in keeping with their specific flavor and design (see Chapters 5 and 6).

Other design decisions reflect the goal of creating a platform for prototyping and exploring the design space itself; among other choices, this includes the initial selection of 256-bit capabilities, giving us greater flexibility to experiment with various bounds-checking and capability behaviors. However, a 256-bit capability introduces potentially substantial cache overhead for pointer-intensive applications – so we have also developed a "compressed" 128-bit in-memory representation. This approach exploits redundancy between the virtual address represented by a capability and its lower and upper bounds – but necessarily limits granularity, leading to stronger alignment requirements.

In our CHERI-MIPS prototype implementation of the CHERI model, capability support is tightly coupled with the existing processor pipeline: instructions propagate values between general-purpose integer registers and capability registers; capabilities transform interpretation of virtual addresses generated by capability-unaware instructions including by transforming the program counter; capability instructions perform direct memory stores and loads both to and from general-purpose integer registers and capability registers; and capability-related behaviors deliver exceptions to the main pipeline. By virtue of having selected the MIPS-centric design choice of exposing capabilities as a separate set of registers, we maintain a separate capability register file as an independent hardware unit – in a manner comparable to vector or floating-point units in current processor designs. The impacts of this integration include additional control logic due to maintaining a separate register file, and a potentially greater occupation of opcode space, whereas combining register files might permit existing instructions to be reused (with care) across integer and capability operations.

Wherever possible, CHERI systems make use of existing hardware designs: processor pipelines and register files, cache memory, system buses, commodity DRAM, and commodity peripheral devices such as NICs and display cards. We are currently focusing on enforcement of CHERI security properties on applications running on a general-purpose processor; in future work, we hope to consider the effects of implementing CHERI in peripheral processors, such as those found in Network Interface Cards (NICs) or Graphical Processing Units (GPUs).

1.2.3 Architectural Neutrality: CHERI-RISC-V and CHERI-x86-64

We believe that the higher-level memory protection and security models we describe encompass not only a number of different potential expressions within a single ISA (e.g., whether to have separate capability registers or to extend general-purpose integer registers to also optionally hold capabilities), but also be applied to other RISC (and CISC) ISAs. This should allow reasonable source-level software portability (leaving aside language runtime and OS assembly code, and compiler code generation) across the CHERI model implemented in different architectures – in much the same way that conventional OS and application C code, as well as APIs for virtual memory, are moderately portable across underlying ISAs.

We have therefore developed two further mappings of the CHERI protection model into specific ISAs: CHERI-RISC-V (Chapter 5) and CHERI-x86-64 (Chapter 6). CHERI-RISC-V is a draft architecture that we are in the process of defining and implementing: RISC-V derives many of it foundational design choices from MIPS, with some more contemporary architectural choices such as hardware page-table walking, and the adaptation to CHERI is very similar. In some areas, we have chosen to leave open specific aspects of the design, learning from our work on CHERI-MIPS, to allow evaluation of performance tradeoffs – e.g., as relates to using a split or merged capability register file. CHERI-x86-64 is an architectural sketch that we have developed to better understand how the CHERI model might apply to more CISC instruction sets. Despite substantive underlying differences between x86-64 and MIPS, we find that many aspects of our approach carry through. We do not yet have implementation aims for CHERI-x86-64, although we hope to explore this further in the future.

1.3 Deterministic Protection

CHERI has been designed to provide strong, non-probabilistic protection rather than depending on short random numbers or truncated cryptographic hashes that can be leaked and reinjected, or that could be brute forced. Essential to this approach is using out-of-band memory tags that prevent confusion between data and capabilities. Software stacks can use these features to construct higher-level protection properties, such as preventing the transmission of pointers via Inter-Process Communication (IPC) or network communications. They are also an essential foundation to strong compartmentalization, which assumes a local adversary.

1.4 Formal Modeling and Provable Protection

The design process for CHERI has used formal semantic models as an important tool in various ways. Our goal here has been to understand how we can support the CHERI design and engineering process with judicious use of mathematically rigorous methods, both in lightweight ways (providing engineering and assurance benefits without the costs of full formal verification), and using machine-checked proof to establish high confidence that the architecture design provides specific security properties.

The basis for all this has been use of formal specifications of the ISA instruction behavior as a fundamental design tool, initially for CHERI-MIPS in L3 [37], and now for CHERI-MIPS and CHERI-RISC-V in Sail [8]. L3 and Sail are domain-specific languages specifically designed for expressing instruction behavior, encoding data, etc. Simply moving from the informal pseudocode commonly used to describe instruction behavior to parsed and type-checked artifacts already helps maintain clear specifications. The CHERI-MIPS instruction descriptions in Chapter 7 are automatically included from the Sail model, keeping documentation and model in sync.

Both L3 and Sail support automatic generation of executable models (variously in SML, OCaml, or C) from these specifications. These executable models have been invaluable, both as golden models for testing our hardware prototypes, and as emulators for testing CHERI software above. The fact that they are automatically generated from the specifications again helps keep things in sync, enabling regression testing on any change to the specification, and makes for easy experimentation with design alternatives. The generated emulators run fast enough to boot FreeBSD in a few minutes (booting cheribsd currently takes around 250s, roughly 320kips).

We have also used the models to automatically generate ISA test cases, both via simple random instruction generation, and using theorem-prover and SMT approaches [16].

Finally, the models support formal verification, with mechanised proof, of key architectural security properties. L3 and Sail support automatic generation of versions of the models in the definition languages of (variously) the HOL4, Isabelle, and Coq theorem provers, which we have used as a basis for proofs. Key architectural verification goals including proving not just low-level properties, such as the monotonicity of each individual instruction and properties of the CHERI Concentrate compression scheme, but also higher-level goals such as compartment monotonicity, in which arbitrary code sequences isolated within a compartment are unable to construct additional rights beyond those reachable either directly via the register file or indirectly via loadable capabilities. We have proven a number of such properties about the CHERI-MIPS ISA, to be documented in future papers and reports.

The CHERI design process has also benefitted from an interplay with our work on rigorous semantics for C [79, 78].

1.5 CHERI ISA Version History

A complete version history, including detailed notes on instruction-set changes, can be found in Appendix A. A short summary of key ISA versions is presented here:

CHERI ISAv1 - 1.0–1.4 - 2010–2012 Early versions of the CHERI ISA explored the integration of capability registers and tagged memory – first in isolation from, and later in composition with, MMU-based virtual memory. CHERI-MIPS instructions were targeted only by an extended assembler, with an initial microkernel ("Deimos") able to create compartments on bare metal, isolating small programs from one another. Key early design choices included:

- to compose with the virtual-memory mechanism by being an in-address-space protection feature, supporting complete MMU-based OSes,
- to use capabilities to implement code and data pointers for C-language TCBs, providing reference-oriented, fine-grained memory protection and control-flow integrity,
- to impose capability-oriented monotonic non-increase on pointers to prevent privilege escalation,
- to target capabilities with the compiler using explicit capability instructions (including load, store, and jumping/branching),
- to derive bounds on capabilities from existing code and data-structure properties, OS policy, and the heap and stack allocators,
- to have both in-register and in-memory capability storage,

- to use a separate capability register file (to be consistent with the MIPS coprocessor extension model),
- to employ tagged memory to preserve capability integrity and provenance outside of capability registers,
- to enforce monotonicity through constrained manipulation instructions,
- to provide software-defined (sealed) capabilities including a “sealed” bit, user-defined permissions, and object types,
- to support legacy integer pointers via a Default Data Capability (**DDC**),
- to extend the program counter (**PC**) to be the Program-Counter Capability (**PCC**),
- to support not just fine-grained memory protection, but also higher-level protection models such as software compartmentalization or language-based encapsulation.

CHERI ISAv2 - 1.5 - August 2012 This version of the CHERI ISA developed a number of aspects of capabilities to better support C-language semantics, such as introducing tags on capability registers to support capability-oblivious memory copying, as well as improvements to support MMU-based operating systems.

UCAM-CL-TR-850 - 1.9 - June 2014 This technical report accompanied publication of our ISCA 2014 paper on CHERI memory protection. Changes from CHERI ISAv2 were significant, supporting a complete conventional OS (CheriBSD) and compiler suite (CHERI Clang/LLVM), a defined CCall/CReturn mechanism for software-defined object capabilities, capability-based load-linked/store-conditional instructions to support multi-threaded software, exception-handling improvements such as a CP2 cause register, new instructions CToPtr and CFromPtr to improve compiler efficiency for hybrid compilation, and changes relating to object capabilities, such as user-defined permission bits and instructions to check permissions/types.

CHERI ISAv3 - 1.10 - September 2014 CHERI ISAv3 further converges C-language pointers and capabilities, improves exception-handling behavior, and continues to mature support for object capabilities. A key change is shifting from C-language pointers being represented by the base of a capability to having an independent “offset” (implemented as a “cursor”) so that monotonicity is imposed only on bounds, and not on the pointer itself. Pointers are allowed to move outside of their defined bounds, but can be dereferenced only within them. There is also a new instruction for C-language pointer comparison (CPtrCmp), and a NULL capability has been defined as having an in-memory representation of all zeroes without a tag, ensuring that BSS (pre-zeroed memory) operates without change. The offset behavior is also propagated into code capabilities, changing the behavior of **PCC**, **EPCC**, CJR, CJALR, and several aspects of exception handling. The sealed bit was moved out of the permission mask to be a stand-alone bit in the capability, and we went from independent CSealCode and CSealData instructions to a single CSeal instruction, and the CSetType instruction has been removed. While the object type originates as a virtual address in an authorizing capability, that interpretation is not mandatory due to use of a separate hardware-defined permission for sealing.

UCAM-CL-TR-864 - 1.11 - January 2015 This technical report refines CHERI ISAv3's convergence of C-language pointers and capabilities; for example, it adds a CIncOffset instruction that avoids read-modify-write accesses to adjust the offset field, as well as exception-handling improvements. TLB permission bits relating to capabilities now have modified semantics: if the load-capability bit is not present, than capability tags are stripped on capability loads from a page, whereas capability stores trigger an exception, reflecting the practical semantics found most useful in our CheriBSD prototype.

CHERI ISAv4 / UCAM-CL-TR-876 - 1.15 - November 2015 This technical report describes CHERI ISAv4, introducing concepts required to support 128-bit compressed capabilities. A new CSetBounds instruction is added, allowing adjustments to both lower and upper bounds to be simultaneously exposed to the hardware, providing more information when making compression choices. Various instruction definitions were updated for the potential for imprecision in bounds. New chapters were added on the protection model, and how CHERI features compose to provide stronger overall protection for secure software. Fast register-clearing instructions are added to accelerate domain switches. A full set of capability-based load-linked, store-conditional instructions are added, to better support multi-threaded pure-capability programs.

CHERI ISAv5 / UCAM-CL-TR-891 - 1.18 - June 2016 CHERI ISAv5 primarily serves to introduce the CHERI-128 compressed capability model, which supersedes prior candidate models. A new instruction, CGetPCCSetOffset, allows jump targets to be more efficiently calculated relative to the current **PCC**. The previous multiple privileged capability permissions authorizing access to exception-handling state has been reduced down to a single system privilege to reduce bit consumption in capabilities, but also to recognize their effective non-independence. In order to reduce code-generation overhead, immediates to capability-relative loads and stores are now scaled.

CHERI ISAv6 / UCAM-CL-TR-907 - 1.20 - April 2017 CHERI ISAv6 introduces support for kernel-mode compartmentalization, jump-based rather than exception-based domain transition, architecture-abstracted and efficient tag restoration, and more efficient generated code. A new chapter addresses potential applications of the CHERI protection model to the RISC-V and x86-64 ISAs, previously described relative only to the 64-bit MIPS ISA. CHERI ISAv6 better explains our design rationale and research methodology.

CHERI ISAv7 / UCAM-CL-TR-927 - 7.0 - June 2019 CHERI ISAv7 differentiates an architecture-neutral CHERI protection model vs. its architecture-specific instantiations in 64-bit MIPS, 64-bit RISC-V, and x86-64. A new capability compression scheme, CHERI Concentrate, is defined, and the previous scheme, CHERI-128, is deprecated. CHERI-MIPS now supports special-purpose capability registers, which have been moved out of the numbered general-purpose capability register space. New special-purpose capability registers, including those for thread-local storage, have been defined. CHERI-RISC-V is more substantially elaborated. A new compartment-ID register assists in resisting microarchitectural side-channel attacks. New optimized instructions with immediate fields improve the performance of generated code. Experimental 64-bit capabilities have been defined for 32-bit architectures, as well as instructions to accelerate spatial and temporal memory safety. The opcode reencoding begun in prior CHERI ISA specification versions has now been completed.

1.5.1 Changes in CHERI ISA 7.0-ALPHA1

This release of the *CHERI Instruction-Set Architecture* is an interim version intended for submission to DARPA/AFRL to meet the requirements of CTSRD deliverable A001:

- The CHERI ISA specification version numbering scheme has changed to include the target major version in the draft version number.

- A significant refactoring of early chapters in the report has taken place: there is now a more clear distinction between architecture-neutral aspects of CHERI, and those that are architecture specific. The CHERI-MIPS ISA is now its own chapter distinct from architecture-neutral material. We have aimed to maximize architecture-neutral content – e.g., capability semantics and contents, in-memory representation, compression, etc. – using the architecture-specific chapters to address only architecture-specific aspects of the mapping of CHERI into the specific architecture – e.g., as relates to register-file integration, exception handling, and the Memory Management Unit (MMU). In some areas, content must be split between architecture-neutral and architecture-specific chapters, such as behavior on reset, handling of the System_Access_Registers permission and its role in controlling architecture-specific behavior, and the integration of CHERI with virtual memory, where the goals are largely architecture neutral but mechanism is architecture specific.

- There are now dedicated chapters for each of our applications of CHERI to each of three ISAs: 64-bit MIPS (Chapter 4), 64-bit RISC-V (Chapter 5), and x86-64 (Chapter 6).

- Our CHERI-RISC-V prototype has been substantially elaborated, and now includes an experimental encoding in Appendix C. We have somewhat further elaborated our x86-64 model, including addressing topics such as new page-table bits for CHERI, including a hardware-managed capability dirty bit. We also consider potential implications for RISC-V compressed instructions.

- We have completed an opcode renumbering for CHERI-MIPS. The "proposed new encoding" from CHERI ISAv6 has now become the established encodings; the prior encodings are now documented as "deprecated encodings".

- Substantial improvements have been made to descriptive text around memory protection, with the concept of "pointer protection" – i.e., as implemented via tags – more clearly differentiated from memory protection.

- We now more clearly describe how terms like "lower bound" and "upper bound" relate to the base, offset, and length fields.

- We now more clearly differentiate language-level capability semantics from capability use in code generation and the ABI, considering pure-capability and hybrid C as distinct from pure-capability and hybrid code generation. We explain that different language-level integer interpretations of capabilities are supportable by the architecture, depending on compiler code-generation choices.

- Potential software policies for revocation, garbage collection, and capability flow control based on CHERI primitives are described in greater detail.

- Monotonicity is more clearly described, as are the explicit opportunities for non-monotonicity around exception handling and CCall Selector 1. Handling of disallowed requests for non-monotonicity or bypass of guarded manipulation by software is more explicitly discussed, including the opportunities for both exception throwing and tag stripping to maintain CHERI's invariants.

- Further notes have been added regarding the in-memory representation of capabilities, including the storage of NULL capabilities, virtual addresses for non-NULL capabilities, and how to store integer values in untagged capability registers. These values now appear in the bottom 64 bits of the in-memory representation. Topics such as endianness are also considered.

- NULL capabilities are now defined as having a base of 0x0, the maximum length supported in a particular representation (2^{64} for 128-bit capabilities, and $2^{64}-1$ for 256-bit capabilities), and no granted permissions. NULL capabilities continue to have an all zeros in-memory representation. This allows integers to be stored in the offset of an untagged capability without concern that they may hold values that are unrepresentable with respect to capability bounds.

- New instructions CReadHwr and CWriteHwr have been added. These have allowed us to migrate special capability registers (SCRs) out of the general-purpose capability register file, including **DDC**, the new user TLS register (**CULR**), the new privileged TLS register (**CPLR**), **KR1C**, **KR2C**, **KCC**, **KDC**, and **EPCC**. Access to privileged special registers continues to be authorized by the Access_System_Registers permission on **PCC**.

- With this migration, **C0** is now available to use as a NULL capability register, which is more consistent with the baseline MIPS ISA in which **R0** is the zero register. The only exception to this is in capability-relative load and store instructions, and the CTestSubset instruction, in which an operand of **C0** specifies that **DDC** should be used.

- Various instruction pseudo-ops to access special registers, such as `CGetDefault`, now expand to special capability register access instructions instead of capability move instructions.

- With consideration of merged rather than split integer and capability register files for RISC-V and x86-64, and a separation between general-purpose capability registers and special capability registers (SCRs) on 64-bit MIPS, we avoid describing the integer register file as the "general-purpose register file". We describe a number of tradeoffs around ISA design relating to using a split vs. merged register file; avoiding the use of specific capability registers as special registers assists in supporting both register-file approaches.

- The CPU reset state of various capability registers is now more clearly defined. Most capability registers are initialized to NULL on reset, with the exception of **DDC**, **PCC**, **KCC**, and **EPCC**. These defaults authorize initial access to memory for the boot process, and are designed to allow CHERI-unaware code to operate oblivious to the capability-system feature set.

- We more clearly describe design choices around failure-mode choices, including throwing exceptions and clearing tag bits. Here, concerns in conclude stylistic consistency with the host architecture, potential use cases, and interactions with the compiler and operating system.

- In general, we now refer to software-defined permissions rather than user-defined permissions, as these permissions without an architectural interpretation may be used in any ring.

- Permission numbering has been rationalized so that 128-bit and 256-bit microarchitectural permission numbers consistently start at 15.

- The existing permission Permit_Seal, which authorized sealing and explicit unsealing of sealed capabilities, has now been broken out into two separate permissions: Permit_Seal, which authorizes sealing, and Permit_Unseal, which authorizes explicit unsealing. This will allow privilege to be reduced where unsealing is desirable (e.g., within object implementations, or in C++ vtable use) by not requiring that permission to seal for the object type is also granted.

- The ISA quick reference has been updated to reflect new instructions, as well as to more correctly reflect endianness.

- We have added a reference to a new technical report, *Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks* [147], which considers the potential interactions between CHERI and the recently announced Spectre and Meltdown microarchitectural side-channel attacks. CHERI offers substantial potential to assist in mitigating aspects of these attacks, as long as the microarchitecture performs required capability checks before performing any speculative memory accesses.

- We have added two new instructions, Get the architectural Compartment ID (`CGetCID`) and Set the architectural Compartment ID (`CSetCID`), which allow information on compartments to be passed to via architecture to microarchitecture in order to support mitigation of side-channel attacks. This could be used to tag branch-predictor entries to control the compartments in which they can be used, for example. A new Permit_Set_CID permission allows capabilities to delegate use of ranges of CIDs.

- Bugs have been fixed in the definitions of capability-relative load and store instructions: permission checks involving the Permit_Load, Permit_Load_Cap, Permit_Store, and Permit_Store_Cap permissions were not properly updated from our shift from an untagged capability register file to a tagged register file. All loads now require Permit_Load. If Permit_Load_Cap is also present, then tags will not be stripped when loading into a capability register. All stores now require Permit_Store. If Permit_Store_Cap is also present, then storing a tagged capability will not generate an exception.

- New Capability Set Bounds From Immediate (CSetBoundsImm) and Capability Increment Offset From Immediate (CIncOffsetImm) instructions have been added. These instructions optimize global-variable setup and stack allocations by reducing the number of instructions and registers required to adjust pointer values and set bounds.

- New Capability Branch if Not NULL (CBNZ) and Capability Branch if NULL (CBEZ) instructions have been added, which optimize pointer comparisons to NULL.

- A new Capability to Address (CGetAddr) instruction allows the direct retrieval of a capability's virtual address, rather than requiring the base and offset to be separately retrieved and added together. This facilitates efficient implementation of a CHERI C variant in which all casts of capabilities to integers have virtual-address rather than offset interpretation. A capability's virtual address is now more directly defined when we specify capability fields.

- We more clearly describe CCall Selector 1 as "exception-free domain transition" rather than "userspace domain transition", as it is also intended to be used in more privileged rings.

- We have shifted to more consistently throwing an exception at jump instructions (e.g., CJR) that go out of bounds, rather than throwing the exception when fetching the first instruction at the target address. This provides more debugging information when using compressed capabilities, as otherwise **EPCC** might have unrepresentable bounds in the event that the jump target is outside of the representable region.

- The exception vectors use during failures of Selector 0 and Selector 1 CCall have been clarified. The general-purpose exception vector is used for all failure modes with CCall Selector 1.

- New experimental instruction Test that Capability is a Subset of Another (CTestSubset) has been added. This instruction is intended to be used by garbage collectors that need to rapidly test whether a capability points into the range of another capability.

- A new experimental 64-bit capability format for 32-bit virtual addresses has been added (Section D.7).

- A description of an experimental *linear capability* model has been added (Section D.10). This model introduces the concept that a capability may be linear – i.e., that it can only be moved rather copied in memory-to-register, register-to-register, and register-to-memory operations. This introduces two new instructions, Linear Load Capability Register (LLCR) and Linear Store Capability Register (LSCR). This functionality has not yet been fully specified.

- An experimental appendix considers possible implementations of *indirect capabilities*, in which a capability value points at an actual capability to utilize, allowing table-based capability lookups (Section D.11).

- An experimental appendix considering potential forms of compression for capability permissions has been added (Section D.8).

- We have added a reference to our ICCD 2017 paper, *Efficient Tagged Memory*, which describes how to efficiently implement tagged memory in memory subsystems not supporting inline tags directly in DRAM [54].

1.5.2 Changes in CHERI ISA 7.0-ALPHA2

This version of the *CHERI Instruction-Set Architecture* is an interim version distributed for review by DARPA and our collaborators:

- We have removed the range check from the `CToPtr` specification, as this has proven microarchitecturally challenging. We anticipate that current consumers requiring this range check can use the new `CTestSubset` instruction alongside `CToPtr`.

- Use of a branch-delay slot with `CCall` Selector 1 has been removed.

- With the addition of `CReadHwr` and `CWriteHwr` and shifting of special capability registers out of the general-purpose capability register file, we have now removed the check for the Access_System_Registers permission for all registers in the general-purpose capability register file.

- A new `CCheckTag` instruction is added, which throws an exception if the tag is not set on the operand capability. This instruction could be used by a compiler to shift capability-related exception behavior from invalid dereference to calculation of an invalid capability via a non-exception-throwing manipulation.

- We have added a new `CLCBI` instruction that allows capability-relative loads of capabilities to be performed using a substantially larger immediate (but without a general-purpose integer-register operand). This substantially accelerates performance in the presence of CHERI-aware linkage by avoiding multi-instruction sequences to load capabilities for global variables.

- We have added new discussion relating to microarchitectural side channels such as Spectre and Meltdown (Section 2.5).

- We have added a reference to our ASPLOS 2019 paper, *CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment*, which describes how to adapt a full MMU-based OS design to support ubiquitous use of capabilities to implement C and C++ pointers in userspace [28].

- We have added a reference to our POPL 2019 paper, *ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS*, which describes a formal modeling approach for instruction-set architectures, as well as a formal model of the CHERI-MIPS ISA [8].

- We have added a reference to our POPL 2019 paper, *Exploring C Semantics and Pointer Provenance*, which describes a formal model for C pointer provenance, and is evaluated in part using pure-capability CHERI code [78].

- We have added a description of an experimental compact capability coloring scheme, a possible candidate to replace our Local-Global capability flow-control model (Section D.13). In the proposed scheme, a series of orthogonal “colors” can be set or cleared on capabilities, authorized by a color space implemented in a style similar to the sealed-capability object-type space using a single permission. For a single color implementing the Local-Global model, two bits are still used. However, for further colors, only a single bit is used. This could make available further colors to use for kernel-user separation, inter-process isolation, and so on.

- An experimental Permit_Recursive_Mutable_Load permission is described, which, if not present, causes further capabilities loaded via that capability to be loaded without store permissions (see Section D.6).

- We have added a new experimental `CLoadTags` instruction that allows tags to be loaded for a cache line without pulling data into the cache.

- A new experimental *sealed entry capability* feature is described, which permits entry via jump but otherwise do not allow dereferencing (Section D.12). These are similar to enter capabilities from the M-Machine [18], and could provide utility in providing further constraints on capability use for the purposes of memory protection – e.g., in the implementation of C++ v-tables.

- A new experimental *memory type token* feature is described, which provides an alternative mechanism to object types within pairs of sealed capabilities (Section D.14).

1.5.3 Changes in CHERI ISA 7.0-ALPHA3

This version of the *CHERI Instruction-Set Architecture* is an interim version distributed for review by DARPA and our collaborators:

- The CHERI Concentrate capability compression format is now documented, with a more detailed rationale section than the prior CHERI-128 section.

- The `CLCBI` (Capability Load Capability with Big Immediate) instruction, which accelerates position-independent access to global variables, is no longer considered experimental.

- The architecture-neutral description of tagged memory has been clarified.

- The maximum supported lengths for both compressed and uncompressed capabilities has been updated: 2^{64} for 128-bit +capabilities, and $2^{64} - 1$ for 256-bit capabilities.

- It is clarified that CLoadTags instruction must provide cache coherency consistent with other load instructions. We recommend "non-temporal" behavior, in which unnecessary cache-line fills are avoided to limit cache pollution during revocation.

- We now define the object type for unsealed capabilities, returned by the CGetType instruction, as $2^{64} - 1$ rather than 0.

- An experimental section has been added on how CHERI capabilities might compose with memory-versioning schemes such as Sparc ADI and Arm MTE (see Section D.9).

- Pseudocode throughout the CHERI ISA specification is now generated from our Sail formal model of the CHERI-MIPS ISA [8].

- The Glossary has been updated for CHERI ISAv7 changes including CHERI-RISC-V, split vs. merged register files, capabilities for physical addresses, and special capability registers.

- Capability exception codes are now shared across architectures.

- CHERI-RISC-V now includes capability-relative floating-point load and store instructions. We have clarified that existing RISC-V floating-point load and store instructions are constrained by **DDC**.

- CHERI-RISC-V now throws exceptions, rather than clearing tags, when non-monotonic register-to-register capability operations are attempted.

- While a specific encoding-mode transition mechanism is not yet specified for CHERI-RISC-V, candidate schemes are described and compared in greater detail.

- CHERI-RISC-V's "capability encoding mode" now has different impacts for uncompressed instructions vs. compressed instructions: In the compressed ISA, jump instructions also become capability relative.

- CHERI-RISC-V page-table entries now contain a "capability dirty bit" to assist with tracking the propagation of capabilities.

- Throwing an exception on an out-of-bounds capability-relative jump rather than on the target fetch is now more clearly explained: This improves debuggability by maintaining precise information about context state on jump, whereas after the jump, bounds may not be representable due to capability compression. When an inappropriate **EPCC** is installed, the exception will still be thrown on instruction fetch.

- A new **ErrorEPCC** special register has been defined, to assist with exceptions thrown within exception handlers; its behavior is modeled on the existing MIPS **ErrorEPC** special register.

1.5.4 Changes in CHERI ISA 7.0-ALPHA4

This version of the *CHERI Instruction-Set Architecture* is an interim version distributed for review by DARPA and our collaborators:

- We have added new instructions `CSetAddr` (Set capability address to value from register), `CAndAddr` (Mask address of capability – experimental), and `CGetAndAddr` (Move capability address to an integer register, with mask – experimental), which optimize common virtual-address-related operations in language runtimes such as WebKit's Javascript engine. These instructions cater better to a language mapping from C's `intptr_t` type to the virtual address, rather than offset, of a capability, which has been our focus previously. These complement the previously added `CGetAddr` that allows easier compiler access to a capability's virtual address.

- We have added two new experimental instructions, `CRAM` (Retrieve Mask to Align Capabilities to Precisely Representable Address) and `CRRL` (Round to Next Precisely Representable Value), which allow software to retrieve alignment information for the base and length for a proposed set of bounds.

- `CMove`, which was previously an assembler pseudo-operation for `CIncOffset`, is now a stand-alone instruction. This avoids the need to special case sealed capabilities when `CIncOffset` is used solely to move, not to modify, a capability.

- The names of the instructions `CSetBoundsImmediate` and `CIncOffsetImmediate` have been shortened to `CSetBoundsImm` and `CIncOffsetImm`.

- The instructions `CCheckType` and `CCheckPerm` have been deprecated, as they have not proven to be particularly useful in implementing multi-protection-domain systems.

- We have added a new pseudo-operation, `CAssertInBounds`, described in Section 7.5.5, allows an exception to be thrown if the address of a capability is not within bounds.

- The instruction `CCheckTag` has now been assigned an opcode.

- We have revised the encodings of many instructions in our draft CHERI-RISC-V specification in Appendix C.

- We more clearly specify that when a special register write occurs to **EPC**, the result is similar to `CSetOffset` but with the tag bit stripped, in the event of a failure, rather than an exception being thrown.

- We have added a reference to our TaPP 2018 paper, *Pointer Provenance in a Capability Architecture*, which describes how architectural traces of pointer behavior, visible through the CHERI instruction set, can be analyzed to understand software and structure.

- We have added a reference to our ICCD 2018 paper, *CheriRTOS: A Capability Model for Embedded Devices*, which describes an embedded variant of CHERI using 64-bit capabilities for 32-bit addresses, and how embedded real-time operating systems might utilize CHERI features.

- We have revised our description of conventions for capability values, including when used as pointers, to hold integers, and for NULL value, to more clearly describe their use. We more clearly describe the requirements for the in-memory representation of capabilities, such as a zeroed NULL capability so that BSS behaves as desired. We provide more clear architecture-neutral explanations of pointer dereferencing, capability permissions and their composition, the namespaces protected by capability permissions, exception handling, exception priorities, virtual memory, and system reset. These definitions appear in Chapter 3. Chapter 4, which describes CHERI-MIPS, has been shortened as a variety of content has been made architectural neutral.

- More detailed rationale is provided for our composition of CHERI with MIPS exception handling.

- We are more careful to use the term “pointer” to refer to the C-language type, verses integer or capability values that maybe used by the compiler to implement pointers.

- With the advent of ISA variations utilizing a merged register file, we are more careful to differentiate integer registers from general-purpose registers, as general-purpose registers may also hold capabilities.

- We more clearly define the terms “upper bound” and “lower bound”.

- We now more clearly describe the effects of our *principle of intentionality* on capability-aware instruction design in Section 3.6.

- We better describe the rationale for tagged capabilities in registers and memory, in contrast to cryptographic and probabilistic protections, in Section 8.2.

- We have made a number of improvements to the CHERI-x86-64 sketch, described in Chapter 6, to improve realism around trap handling and instruction design.

- We have rewritten our description of the interaction between CHERI and Direct Memory Access (DMA) in Section 3.8.4. to more clearly describe tag-stripping and capability-aware DMA options.

1.5.5 Changes in CHERI ISA 7.0

This version of the *CHERI Instruction-Set Architecture* is a full release of the Version 7 specification:

- We have now deprecated the CHERI-128 capability compression format, in favor of CHERI Concentrate.

- The RISC-V `AUIPC` instruction now returns a **PCC**-relative capability in the capability encoding mode.

- Capabilities now contain a **flags** field, which will hold state that can be changed without affecting privilege. Corresponding experimental `CGetFlags` and `CSetFlags` instructions have been added. These are described in greater detail in Section D.1.

- The capability encoding-mode bit in CHERI-RISC-V is specified as a bit in the **flags** field of a capability. The current mode is defined as the flag bit in the currently installed **PCC**. Design considerations and other potential options are described in Chapter 8.

- We now more explicitly describe the reset states of special and general-purpose capability registers for CHERI-MIPS and CHERI-RISC-V.

- Compressed capabilities now contain a dedicated **otype** field that always holds an object type (see sections 2.3.7 and 3.3.1), rather than stealing bounds bits for object type when sealing. Now, any representable capability may be sealed. Several object type values are reserved for architectural experimentation (see table 3.2).

- More detail is provided regarding the integration of CHERI Concentrate with special registers, its alignment requirements, and so on.

- Initial discussion of a disjoint capability tree for physical addresses and hardware facilities using these has been added to the experimental appendix, in appendix D.16.

- Initial discussion of a hybrid 64/128-bit capability design has been added to the experimental appendix, in appendix D.15.

- We have added formal Sail instruction semantics for CHERI-RISC-V; this is currently in Appendix C.

- We have added a reference to our IEEE TC 2019 paper, *CHERI Concentrate: Practical Compressed Capabilities*, which describes our current approach to capability compression.

- We have added a reference to Alexandre Joannou's PhD dissertation, *High-performance memory safety: optimizing the CHERI capability machine*, which describes approaches to improving the efficiency of capability compression and tagged memory.

1.6 Experimental Features

Appendix D describes a number of experimental features that extend CHERI with new functionality. These include several architectural features:

- Capability flags that allow non-security bit-wise metadata to be associated with capabilities

- Instructions to assist with memory-allocation alignment

- Fast capability subset testing and non-temporal tag loading to better support sweeping revocation for temporal memory safety

- Efficient tag rederivation for use with swapping, memory compression, memory encryption, and virtual-machine migration
- A recursive mutable load permission that limits the store rights via future capability loads
- 64-bit capabilities for 32-bit architectures
- More efficient capability permission representations
- Memory versioning for use with capabilities
- Linear capabilities
- Indirect capabilities
- Sealed entry capabilities (with dedicated, hardware object type)
- Capability coloring for capability flow control
- Sealing with large object type fields in memory
- A system for mixing 64-bit and 128-bit capabilities
- Capabilities referencing physical addresses
- Use of capabilities across a system for peripherals and accelerators
- New instructions to improve code density

We believe that these represent interesting, and in some cases promising, portions of the design space beyond the baseline CHERI. However, they appear in an appendix because: (1) we do not yet recommend their use; (2) they have not been thoroughly evaluated across architecture, hardware, and software with respect to utility, security, compatibility, microarchitectural realism, nor performance; and/or (3) their preservation of essential CHERI security properties has not been formally proven. They are therefore included to provide insight into potential future directions or interesting potential alternative points in the overall design space.

1.7 Document Structure

This document is an introduction to, and a reference manual for, the CHERI protection model and instruction-set architecture.

Chapter 1 introduces the CHERI protection model, our architecture-neutral approach, and specific CHERI-MIPS and CHERI-RISC-V ISAs.

Chapter 2 describes the high-level model for the CHERI approach in terms of architectural features, software protection objectives, and software mechanism.

Chapter 3 provides a detailed description of architecture-neutral aspects of the CHERI protection model, including capability and tagged-memory models, categories of new instructions, etc.

Chapter 4 describes an architecture-specific mapping of the CHERI protection model into the 64-bit MIPS architecture. This includes specification of the CHERI-MIPS capability coprocessor, register file, Translation Look-aside Buffer (TLB), privilege model, and other ISA-specific semantics.

Chapter 5 describes a draft architecture-specific mapping of the CHERI protection model into the 64-bit RISC-V architecture. This includes specification of the CHERI-RISC-V architecture extension, register file, Memory Management Unit (MMU), privilege models, and other ISA-specific semantics.

Chapter 6 provides an "architectural sketch" of how the CHERI protection model might be mapped into the x86-64 ISA, a decidedly non-RISC instruction set.

Chapter 7 provides a detailed description of each new CHERI-MIPS instruction, its pseudo-operations, and how compilers should handle floating-point loads and stores via capabilities.

Chapter 8 discusses the design rationale for many aspects of the CHERI-MIPS ISA, as well as our thoughts on future refinements based on lessons learned to date.

Chapter 9 outlines a detailed (but not formally proved) argument for why a reference monitor above CHERI provides certain security properties, and touches on some issues in the specification that formal proof has to deal with.

Chapter 10 describes the motivations and hardware-software co-design research approach taken in developing CHERI, including major phases of the research. Chapter 11 describes the historical context for this work, including past systems that have influenced our approach.

Chapter 12 discusses our short- and long-term plans for the CHERI protection model and CHERI-MIPS ISA, considering both our specific plans and open research questions that must be answered as we proceed.

Appendix A provides a more detailed version history of the CHERI protection model and CHERI-MIPS ISA.

Appendix B is a quick reference for CHERI-MIPS instructions and encodings.

Appendix C is a quick reference for the proposed CHERI-RISC-V instructions and encodings.

Appendix D specifies a number of CHERI-MIPS instructions that we still consider experimental, and hence are not included in the main specification.

Appended E describes our prior (now deprecated) CHERI-128 compression scheme, which has been superseded by CHERI Concentrate.

The report also includes a Glossary defining many key CHERI-related terms.

Future versions of this document will continue to expand our consideration of the CHERI model and CHERI-MIPS instruction-set architecture, its impact on software, and evaluation strategies and results. Additional information on our prototype CHERI hardware and software implementations, as well as formal methods work, can be found in accompanying reports.

1.8 Publications

As our approach has evolved, and project developed, we have published a number of papers and reports describing aspects of the work. Our conference papers contain greater detail on the rationale for various aspects of our hardware-software approach, along with evaluations of micro-architectural impact, software performance, compatibility, and security:

- In the International Symposium on Computer Architecture (ISCA 2014), we published *The CHERI Capability Model: Revisiting RISC in an Age of Risk* [153]. This paper describes our architectural and micro-architectural approaches with respect to capability registers and tagged memory, hybridization with a conventional Memory Management Unit (MMU), and our high-level software compatibility strategy with respect to operating systems.

- In the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015), we published *Beyond the PDP-11: Architectural support for a memory-safe C abstract machine* [21], which extends our architectural approach to better support convergence of pointers and capabilities, as well as to further explore the C-language compatibility and performance impacts of CHERI in larger software corpora.

- In the IEEE Symposium on Security and Privacy (IEEE S&P, or "Oakland", 2015), we published *CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization* [146], which describes a hardware-software architecture for mapping compartmentalized software into the CHERI capability model, as well as extends our explanation of hybrid operating-system support for CHERI.

- In the ACM Conference on Computer and Communications Security (CCS 2015), we published *Clean Application Compartmentalization with SOAAP* [46], which describes our higher-level design approach to software compartmentalization as a a form of vulnerability mitigation, including static and dynamic analysis techniques to validate the performance and effectiveness of compartmentalization.

- In the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016), we published *Into the depths of C: elaborating the de facto standards* [79], which develops a formal semantics for the C programming language. As part of that investigation, we explore the effect of CHERI on C semantics, which led us to refine a number of aspects of CHERI code generation, as well as refine the CHERI ISA. In the other direction, understanding the changes needed to port existing software to CHERI has informed our views on what C semantics should be.

- In the September-October 2017 issue of IEEE Micro, we published *Fast Protection-Domain Crossing in the CHERI Capability-System Architecture* [143], expanding on architectural and microarchitectural aspects of the CHERI object-capability compartmentalization model described in our Oakland 2015 paper.

- In the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2017), we published *CHERI-JNI: Sinking the Java security model into the C* [20]. This paper describes how to use CHERI memory safety and compartmentalization to isolate Java Native Interface (JNI) code from the Java Virtual Machine, imposing the Java memory and security model on native code.

- In the MIT Press book, *New Solutions for Cybersecurity*, we published two chapters on CHERI. *Balancing Disruption and Deployability in the CHERI Instruction-Set Architecture (ISA)* discusses our research and development approach, and how CHERI hybridizes conventional architecture, microarchitecture, operating systems, programming languages, and general-purpose software designs with a capability-system model [136]. *Fundamental Trustworthiness Principles in CHERI* discusses how CHERI fulfills a number of critical trustworthiness principles [92].

- In the International Conference on Computer Design (ICCD 2017), we published *Efficient Tagged Memory* [54]. This paper describes how awareness of the architectural semantics of tagged pointers can be used to improve performance and reduce DRAM access overheads for tagging implemented over DRAM without innate tag storage.

- In the International Conference on Computer Design (ICCD 2019), we published *CheriRTOS: A Capability Model for Embedded Devices* [157]. This paper describes an embedded variant on CHERI using 64-bit capabilities for 32-bit addresses, and how embedded real-time operating systems might utilize CHERI features.

- In the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2019), we published *ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS*, which describes a formal modeling approach and formal models for several instruction sets including CHERI-MIPS [8].

- In the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2019), we published *Exploring C Semantics and Pointer Provenance*, describing a formal model for C pointer provenance and its practical evaluation, including via pure-capability C code on the CHERI architecture [78].

- In the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019), we published *CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment* [28]. This paper describes how to adapt a full MMU-based OS design to support ubiquitous use of capabilities to implement C and C++ pointers in userspace.

- In IEEE Transactions on Computers, we published *CHERI Concentrate: Practical Compressed Capabilities* [152]. This paper describes our compressed 128-bit and 64-bit capability formats, evaluating the effects of precision loss in bounds, and the potential performance impact of the approach.

We have additionally released several technical reports, including this document, describing our approach and prototypes. Each has had multiple versions reflecting evolution of our approach:

- This report, the *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture* [137, 138, 141, 142, 140], describes the CHERI ISA, both as a high-level, software-facing model and the specific mapping into the 64-bit MIPS instruction set. Successive versions have introduced improved C-language support, support for scalable compartmentalization, and compressed capabilities.

- The *Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide* [135] describes in greater detail our mapping of software into instruction-set primitives in both the compiler and operating system; earlier versions of the document were released as the *Capability Hardware Enhanced RISC Instructions: CHERI User's Guide* [133].

- The *Bluespec Extensible RISC Implementation: BERI Hardware Reference* [144, 145] describes hardware aspects of our prototyping platform, including physical platform and practical user concerns.

- The *Bluespec Extensible RISC Implementation: BERI Software Reference* [132, 134] describes non-CHERI-specific software aspects of our prototyping platform, including software build and practical user concerns.

- The technical report, *Clean application compartmentalization with SOAAP (extended version)* [45], provides a more detailed accounting of the impact of software compartmentalization on software structure and security using conventional designs, with potential applicability to CHERI-based designs as well.

- The technical report, *Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks* [147] explores the potential interactions between CHERI, a fundamentally architectural protection technique, and the recently announced Spectre and Meltdown microarchitectural side-channel attacks. The report describes a modest architecture extension identifying CHERI compartment identifiers to the microarchitecture, and also explores opportunities for Spectre mitigation arising from performing capability checks in speculation.

The following technical reports are PhD dissertations that describe both CHERI and our path to our current design:

- Robert Watson's PhD dissertation, *New approaches to operating system security extensibility*, describes the operating-system access-control and compartmentalization approaches, including FreeBSD's MAC Framework and Capsicum, which motivated our work on CHERI [128, 129].

- Jonathan Woodruff's PhD dissertation, *CHERI: A RISC capability machine for practical memory safety*, describes our CHERI1 prototype implementation [154].

- Robert Norton's PhD dissertation, *Hardware support for compartmentalisation*, describes how hardware support is provided for optimized domain transition using the CHERI2 prototype implementation [96].

- Alexandre Joannou's PhD dissertation, *High-performance memory safety: optimizing the CHERI capability machine*, describes hardware optimizations for efficient implementation of CHERI capabilities such as capability compression for a 128-bit capability format and a hierarchical tag cache for efficient tagged memory [55].

As our research proceeded, and prior to our conference and journal articles, we published a number of workshop papers laying out early aspects of our approach:

- Our philosophy in revisiting of capability-based approaches is described in *Capabilities Revisited: A Holistic Approach to Bottom-to-Top Assurance of Trustworthy Systems*, published at the Layered Assurance Workshop (LAW 2010) [95], shortly after the inception of the project.

- Mid-way through creation of both the BERI prototyping platform, and CHERI protection model and CHERI-MIPS ISA, we published *CHERI: A Research Platform Deconflating Hardware Virtualization and Protection* at the Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012) [148].

- Jonathan Woodruff, whose PhD dissertation describes our initial CHERI prototype, published a workshop paper on this work at the CEUR Workshop's Doctoral Symposium on Engineering Secure Software and Systems (ESSoS 2013): *Memory Segmentation to Support Secure Applications* [95].

- In the USENIX Workshop on the Theory and Practice of Provenance (TaPP), we published *Pointer Provenance in a Capability Architecture* [74]. This paper describes how architectural traces of pointer behavior, visible through the CHERI instruction set, can be analyzed to understand software structure and security.

Further research publications and technical reports will be forthcoming.

Chapter 2

The CHERI Protection Model

This chapter describes the portable *CHERI protection model*, its use in software, and its impact on potential software vulnerabilities; concrete mappings into computer architecture are left to later chapters. We consider a number of topics from a more abstract, software-facing perspectives: the principles underlying the model, our goals for capabilities, hybridization with conventional architectural designs, implications for operating-system and language support and compatibility, and concerns around microarchitectural side channels.

There are many potential concrete mappings of this abstract software-facing protection model into specific Instruction-Set Architectures (ISAs), but most key aspects of the model can be shared across target architectures, including the capability protection model, composition with virtual memory, and tagged memory. Whether used for memory protection or compartmentalization, CHERI's properties should hold with considerable uniformity across underlying architectural implementations (e.g., regardless of capability size, whether capabilities are stored in their own register file or as extensions to general-purpose integer registers, etc.), and should support common (and ideally portable) programming models and approaches.

We detail cross-architecture aspects of CHERI in Chapter 3. Our current instantiations within concrete ISAs include the mature CHERI-MIPS ISA (Chapter 4), a draft CHERI-RISC-V ISA (Chapter 5), and a high-level sketch of a CHERI-x86-64 (Chapter 6). CHERI-MIPS remains our reference instantiation, and has been validated with a complete end-to-end hardware-software stack including ISA-level simulations, FPGA implementation, operating system, compiler, linker, debugger, and application suite. CHERI-RISC-V is a draft specification that has not yet seen significant use. Our motivations for targeting this second ISA are detailed in Chapter 5; they include demonstrating the portability of the CHERI approach, a desire to use a more contemporary ISA as a baseline, and the potential opportunity for technology transition. We include our x86-64 sketch to explore how the CHERI protection model might apply to the dominant non-RISC architecture.

2.1 Underlying Principles

The design of CHERI is influenced by two broad underlying principles that are as much philosophical as architectural, but are key to all aspects of the design:

The principle of least privilege It should be possible to express and enforce a software design in which each program component can execute with only the privileges it requires to perform its function. This is expressed in terms of architectural privileges (e.g., by allowing restrictions to be imposed in terms of bounds, permissions, etc., encapsulating a software-selected but hardware-defined set of rights) and at higher levels of abstraction in software (e.g., by allowing sealed capabilities to refer to encapsulated code and data incorporating both a software-selected and software-defined set of rights). This principle has a long history in the research literature, and has been explored (with varying degrees of granularity) both in terms of the expression of reduced privilege (i.e., through isolation and compartmentalization) and the selection of those privileges (e.g., through hand separation, automated analysis, and so on).

The principle of intentional use When multiple rights are available to a program, the selection of rights used to authorize work on behalf of the program should be explicit, rather than implicit in the architecture or another layer of software abstraction. The effect of this principle is to avoid the accidental or unintended exercise of rights that could lead to a violation of the intended policy. It helps counter what are classically known as 'confused deputy' problems, in which a program will unintentionally exercise a privilege that it holds legitimately, but on behalf of another program that does not (and should not) exercise that privilege [49]. This principle, common to many capability systems but usually not explicitly stated, has been applied throughout the CHERI design, from architectural privileges (e.g., the requirement to explicitly identify capability registers used for load or store) through to the sealed capability mechanism that can be used to support object-capability models such as found in CheriBSD.

These principles, which offer substantial mitigations against software vulnerabilities or malicious code, guide the integration of a capability-system model with the general-purpose instruction set – and its exposure in the software model. A more detailed exploration of the design principles embodied in and supported by CHERI can be found in *Fundamental Trustworthiness Principles in CHERI* [92].

2.2 CHERI Capabilities: Strong Protection for Pointers

The purpose of the CHERI ISA extensions is to provide strong protection for pointers within virtual address spaces, complementing existing virtual memory provided by Memory Management Units (MMUs). These protections apply to the storage and manipulation of pointers, and also accesses performed via pointers. The rationale for this approach is two-fold:

1. A large number of vulnerabilities in Trusted Computing Bases (TCBs), and many of the application exploit techniques, arise out of bugs involving pointer manipulation, corruption, and use. These occur in several ways, with bugs such as those permitting attackers to coerce arbitrary integer values into dereferenced pointers, or leading to undesirable arithmetic manipulation of pointers or buffer bounds. These can have a broad variety of impacts – including overwriting or leaking sensitive data or program metadata, injection

of malicious code, and attacks on program control flow, which in turn allow attacker privilege escalation.

Virtual memory fails to address these problems as (a) it is concerned with protecting data mapped at virtual addresses rather than being sensitive to the context in which a pointer is used to reference the address – and hence fails to assist with misuse of pointers; and (b) it fails to provide adequate *granularity*, being limited to page granularity – or even more coarse-grained "large pages" as physical memory sizes grow.

2. Strong integrity protection, fine-grained bounds checking, encapsulation, and monotonicity for pointers can be used to construct efficient *isolation* and *controlled communication*, foundations on which we can build scalable and programmer-friendly compartmentalization within address spaces. This facilitates deploying fine-grained application sandboxing with greater ubiquity, in turn mitigating a broad range of logical programming errors higher in the software stack, as well as resisting future undiscovered vulnerability classes and exploit techniques.

   Virtual memory also fails to address these problems, as (a) it scales poorly, paying a high performance penalty as the degree of compartmentalization grows; and (b) it offers poor programmability, as the medium for sharing is the virtual-memory page rather than the pointer-based programming model used for code and data sharing within processes.

Consequently, *CHERI capabilities* are designed to represent language-level pointers with additional metadata to protect their integrity and provenance, enforce bounds checks and permissions (and their monotonicity), and hold additional fields supporting undereferenceable (i.e., sealed) software-defined pointers suitable to implement higher-level protection models such as separation and efficient compartmentalization. Unlike virtual memory, whose functions are intended to be managed by low-level operating-system components such as kernels, hypervisors, and system libraries, CHERI capabilities are targeted at compiler and language-runtime use, allowing program structure and dynamic memory allocation to direct their use. We anticipate CHERI being used within operating-system kernels, and also in userspace libraries and applications, for the purposes of both memory protection and compartmentalization.

Significant attention has gone into providing strong compatibility with the C and C++ programming languages, widely used in off-the-shelf TCBs such as OS kernels and language runtimes, and also with conventional MMUs and virtual-memory models – which see wide use today and continue to operate on CHERI-enabled systems. This is possible by virtue of CHERI having a *hybrid capability model* that securely composes a capability-system model with conventional architectural features and programming-language pointer interpretation. CHERI is designed to support incremental migration via selective recompilation (e.g., transforming pointers into capabilities, as discussed below). It provides several possible strategies for selectively deploying changes into larger code bases – constructively trading off source-code compatibility, binary compatibility, performance, and protection.

Most source code can be recompiled to employ CHERI capabilities transparently by virtue of existing pointer syntax and semantics, which the compiler can map into capability use just as it currently maps that functionality into integer virtual-address use – while providing additional metadata to the architecture allowing the implementation of stronger memory safety. Code in

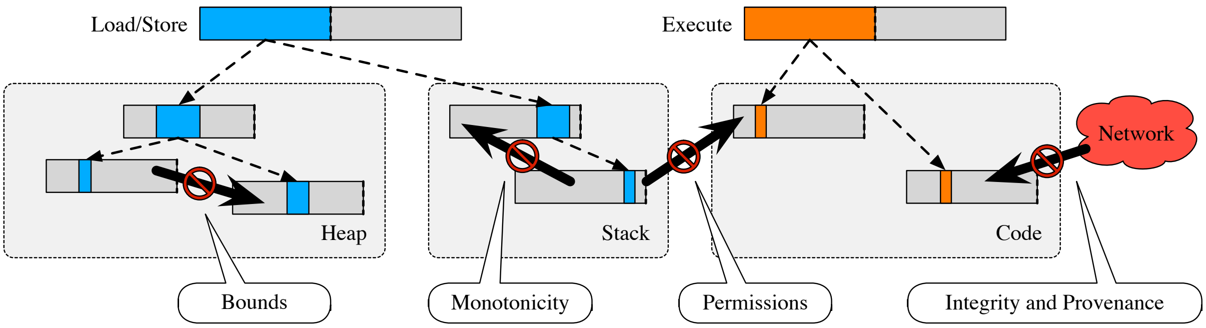

Figure 2.1: CHERI enforces strict *integrity*, *provenance validity*, *monotonicity*, *bounds*, *permissions*, and *encapsulation* on pointers, mitigating common vulnerabilities and exploit techniques.

which all pointers (and implied virtual addresses) are implemented solely using capabilities is referred to as *pure-capability code*. Capability use can also be driven selectively, albeit less transparently, through annotation of C pointers and types to indicate that hybrid capability code generation should be used when operating on those pointers – referred to as *hybrid-capability code*. It is also possible to imagine compilers making automatic policy-based decisions about capability use on a case-by-case basis, based on trading off compatibility, performance, and protection with only limited programmer intervention. It is further worth observing that, although the primary focus of CHERI has been protecting pointers using capabilities, capabilities are a more generalizable hardware data type that can be used to protect other types from corruption and mis-manipulation.

2.3 Architectural Capabilities

In current systems, pointers are integer values that are commonly stored in two architectural forms: in integer registers, and in memory. Capabilities are likewise stored in registers and memory, and contain integer values interpreted as virtual addresses; they also contain additional metadata to implement protection properties around pointers, such as bounds. Capabilities are therefore larger than the virtual addresses they protect – typically between 2× (e.g., 128-bit compressed capabilities on a 64-bit architecture) and 4× (e.g., 256-bit uncompressed capabilities on a 64-bit architecture). The majority of the capability is stored in a register or in addressable memory, as is the case for current integer pointers; however, there is also a 1-bit tag that may be inspected via the instruction set, but is not visible via byte-wise loads and stores. This tag is used to record whether the capability is valid; it is preserved by legal capability operations but cleared by other operations on that memory. Some of CHERI's protections are for pointers themselves (e.g., their integrity and provenance validity), whereas others are for the pointee data or code referenced by pointers (e.g., bounds and permissions). CHERI's sealing feature protects both a pointer (via immutability) and the pointee (via non-dereferenceability).

Extending architectures with capability registers and suitable memory storage naturally aligns with many current architectural and microarchitectural design choices, as well as software-facing considerations such as compiler code generation, stack layout, operating-system behavior, and so on. However, the generalized CHERI protection model can be mapped into architectures in many different forms. For example, an early design choice might be between holding capabilities in a dedicated *capability register file* or extending existing 64-bit registers to hold 128-bit capabilities. While this and many other choices will affect a variety of factors in the architecture and microarchitecture, the resulting protection model can be considered *portable* in that common protection properties and usage patterns can be mapped into various architectural instantiations. These topics are considered further in Chapter 3.

In the remainder of this section, we describe the high-level protection properties and other functionality that capabilities grant to pointers and the execution environment (see Figure 2.1):

- Capability tags for pointer integrity and provenance (Section 2.3.1)
- Capability bounds to limit the dereferenceable range of a pointer (Section 2.3.2)
- Capability permissions to limit the use of a pointer (Section 2.3.3)
- Capability monotonicity and guarded manipulation to prevent privilege escalation (Section 2.3.4)
- Capability sealing to implement software encapsulation (Section 2.3.6)
- Capability object types to enable a software object-capability model (Section 2.3.7)
- Sealed capability invocation to implement non-monotonic domain transition (Section 2.3.8)
- Capability control flow to limit pointer propagation (Section 2.3.10)
- Capability compression to reduce the in-memory overhead of pointer metadata (Section 2.3.11)
- Hybridization with integer pointers (Section 2.3.12)
- Hybridization with MMU-based virtual memory (Section 2.3.13)
- Hybridization with ring-based privilege (Section 2.3.14)
- Failure modes and exception delivery (Section 2.3.15)
- Capability revocation (Section 2.3.16)

These features allow capabilities to be architectural primitives upon which higher-level software protection and security models can be constructed (see Section 2.4).

2.3.1 Tags for Pointer Integrity and Provenance

Each location that can hold a capability – whether a capability register or a capability-sized, capability-aligned word of memory – has an associated 1-bit tag that consistently and atomically tracks capability validity for the value stored at that location:

Capability registers each have a 1-bit tag tracking whether the in-register value is a valid capability. This bit will be set or cleared only as permitted by guarded manipulation.

Capability-sized, capability-aligned words of memory each have a 1-bit tag associated with the location, which is not directly addressable via data loads or stores: *tagged memory*. Depending on the ISA variant, this may be at 128-bit or 256-bit granularity. The capability's virtual address, as well as its other metadata such as bounds and permissions, are stored within the capability in addressable memory; these fields are protected by the corresponding unaddressable tag bit. If untagged memory exists in the system, the tags of capability values stored to those locations are discarded, and all loaded capability values will have the tag bit unset.

Tags atomically follow capabilities into and out of capability registers when their values are loaded from, or stored to, tagged memory. Stores of other non-capability types – e.g., of bytes or half words – automatically and atomically clear the tag in the destination memory location. This allows in-memory pointer corruption by data stores to be detected on next attempted dereference – for example, this prevents arbitrary data received over the network from being directly dereferenced as a pointer.

The capability tag controls which operations can be performed using a capability. Attempting controlled operations on an untagged capability will cause an precise exception.

Regardless of the value of the tag bit, capability register fields can be accessed: they can be extracted and, subject to guarded manipulation, modified. Similarly, addressable portions of the capability can be read from memory using ordinary data load and store instructions. Capability values can also be loaded and stored via other valid capabilities regardless of the validity of the loaded or stored capability. An untagged capability value is simply data: allowing capability registers to hold untagged values allows them to be used for capability-oblivious operations. For example, a region of memory can be copied via capability registers, including pointers within data structures, preserving the value of the tag bit for each copied location.

However, other operations that *dereference* or otherwise use a capability require that the capability have its tag set – i.e., be a *valid capability*. Dereferencing refers to using the capability to load or store data or other capabilities, or to fetch instructions. This includes the implied dereference associated with the Default Data Capability controlling legacy integer-relative loads and stores. A valid tag is also required to use a capability to seal or unseal another capability, to jump to that capability, to use it to set the architectural compartment ID, or to call it for the purposes of domain transition. Detailed information on which instructions require capabilities to have valid tags, or operate on untagged capability values, may be found in the instruction reference.

Valid capabilities can be constructed only by deriving them from existing valid capabilities, which ensures *pointer provenance* (Figure 2.1). In almost all cases, a new capability value will be derived from a single capability value – e.g., as a result of reducing bounds or permissions. In a few cases, a capability may derive from multiple other capability values. For example, a sealed capability is derived from both the authorizing sealing capability and an original data capability. Similarly, an explicitly unsealed capability is derived from both the sealed capability and the capability that authorizes its unsealing.

Implementing C pointers as tagged capabilities allows them to be reliably identified in the virtual address space, which can help support techniques such as garbage collection. The CHERI ISA has been designed to avoid leakage of virtual addresses out of tagged capabilities (e.g., into general-purpose integer registers) during normal memory allocation, comparison, manipulation, and dereference, to facilitate reliable detection of pointers in both registers and memory. Virtual addresses can be extracted from capabilities – e.g., for debugging purposes – but avoiding doing so in code generation supports potential use of techniques such as copying garbage collection.

Our CHERI prototype implements tagged memory using partitioned memory, with tags and associated capability-sized units linked and propagated by the cache hierarchy in order to provide suitable atomicity. However, it is also possible to imagine implementations in which DRAM or non-volatile memory is extended to store tags with capability-sized units as well – which might be more suitable for persistent memory types where atomicity is not simply a property of coherent access through the cache. We similarly assume that DMA will clear tags when writing to memory, although it is possible to imagine future DMA implementations that are able to propagate tags (e.g., to maintain tags on pointers in descriptor rings).

2.3.2 Bounds on Pointers

Capabilities contain lower and upper bounds for each pointer; while the pointer may move out of bounds (and perhaps back in again), attempts to dereference an out-of-bounds pointer will throw a hardware exception. This prevents exploitation of buffer overflows on global variables, the heap, and the stack, as well as out-of-bounds execution. Allowing pointers to sometimes be out-of-bounds with respect to their buffers – without faulting – is important for de-facto C-language compatibility. The 256-bit capability variant allows pointers to stray arbitrarily out of bounds. The 128-bit scheme imposes some restrictions, as bounds compression depends on redundancy between the pointer and bounds, which may not be present if the pointer is substantially outside of its bounds (see Section 3.4.4 for details).

Bounds originate in allocation events. The operating system places bounds on pointers to initial address-space allocations during process startup (e.g., via the initial register file, and ELF auxiliary arguments), and on an ongoing basis as new address-space mappings are made available (e.g., via `mmap` system calls). Most bounds originate in the userspace language runtime or compiler-generated code, including the run-time linker for function pointers and global data, the heap allocator for pointers to heap allocations, and generated code for pointers taken to stack allocations. Programming languages may also offer explicit subsetting support to allow software to impose its own expectations on suitable bounds for memory accesses to complex objects (such as in-memory video streams) or in their own memory allocators.

2.3.3 Permissions on Pointers

Capabilities additionally extend each pointer with a permissions mask controlling how the pointer may be used; for example, the run-time linker or compiler may set the permissions so that pointers to data cannot be reused as code pointers, or so that pointers to code cannot be used to store data. Further permissions control the ability to load and store capabilities themselves, allowing the compiler to implement policies such as *dereferenceable code and data pointers cannot be loaded from character strings.* Permissions can also be made accessible to higher-level aspects of the run-time and programmer model, offering dynamic enforcement of concepts similar to `const`.[1] Languages may provide further facilities to allow programmer-directed refinement of permissions – for example, for use in Just-in-Time (JIT) compilers.

Permissions changes, as with bounds setting, are often linked to allocation events. Permissions on capabilities for initial memory memory mappings will be introduced by the kernel during process startup; further capabilities returned for new mappings will also have their permissions restricted based on intended use. Executable capabilities representing function pointers and return addresses will be refined by the run-time linker. Read-only and read-write capabilities referring to data will be refined by the run-time linker, heap allocator, and stack allocator.

Permissions also control access to the sealing facility used for encapsulation (see Section 2.3.6). While sealing permission could be granted with all data and code capabilities, best practice in privilege minimization suggests that a separate hierarchy of sealing pointers should be maintained instead. Returning independent sealing capabilities via a dedicated system-call interface reduces opportunities for arbitrary code and data capabilities being used improperly for this purpose.

2.3.4 Capability Monotonicity via Guarded Manipulation

Capability monotonicity is a property of the CHERI ISA design ensuring that new capabilities must be derived from existing capabilities only via valid manipulations that may narrow (but never broaden) rights ascribed to the original capability. This property prevents broadening the bounds on pointers, increasing the permissions on pointers, and so on, eliminating many manipulation attacks and inappropriate pointer reuses. Monotonicity also underlies effective isolation for software compartmentalization by ensuring that delegated capabilities cannot be used to reach other resources despite further manipulation. CHERI enforces capability monotonicity via four mechanisms:

Limited expressivity Some instructions are prevented, by design, from expressing an increase of rights due to the expression of their operands and implementation. For example, permissions on capabilities are modified using a bitwise ‘and’ operation, and hence cannot express an increase in permissions.

Exceptions on monotonicity violation Some instructions may be able to represent non-monotonic operations, but attempts to use them non-monotonically will lead to an exception

[1] The C-language `const` qualifier conflates several orthogonal properties and thus can not be enforced automatically. Our language extensions include more constrained `__input` and `__output` qualifiers.

being delivered. For example, an attempt to broaden bounds on a capability might throw an exception without writing back the non-monotonically modified capability. Throwing an exception at the point of violation may ease debugging close to the point of violation.

Stripping the tag in register write-back As an alternative to throwing an exception, a non-monotonic operation might succeed in writing back a new capability – but with the tag bit cleared, preventing future dereference. Clearing the tag allows the failure to be discovered by an explicit software check, or on the next attempt to dereference. This may make debugging more expensive (if additional checks are introduced, perhaps with help from the compiler) or more tricky (if loss of the tag is only discovered substantially later).

Stripping the tag during memory store Tagged memory ensures that attempts to directly modify capability fields (whether non-monotonically or otherwise) will clear the tag, causing later attempts to dereference the capability to fail. This ensures that attempts to modify capabilities cannot bypass guarded manipulation.

Selecting which enforcement mechanism to use will reflect the specific operation being implemented, concerns about about ease of debugging, as well as the context of the surrounding architecture. For example, in some architectures, exceptions can be thrown on any instruction (e.g., MIPS), while in others it is preferable for exceptions to be thrown only on memory accesses (e.g., ARMv8). As a result of these combined architectural features, guarded manipulation implements *non-bypassable capability monotonicity*.

Monotonicity allows reasoning about the set of reachable rights for executing code, as they are limited to the rights in any capability registers, and inductively, the set of any rights reachable from those capabilities – but no other rights, which would require a violation of monotonicity. Monotonicity is a key foundation for fine-grained compartmentalization, as it prevents delegated rights from being used to gain access to other undelegated areas of memory. More broadly, monotonicity contributes to the implementation of the principle of intentional use, in that capabilities not only cannot be used for operations beyond those for which they are authorized, but also cannot inadvertently be converted into capabilities describing more broad rights.

The two notable exceptions to capability monotonicity are invocation of sealed capabilities (see Section 2.3.8) and exception delivery (see Section 2.3.15). Where non-monotonicity is present, control is transferred to code trusted to utilize a gain in rights appropriately – for example, a trusted message-passing routine in the userspace runtime, or an OS-provided exception handler. This non-monotonicity is required to support protection-domain transition from one domain holding a limited set of rights to destination domain that holds rights unavailable to the originating domain – and is therefore also a requirement for fine-grained compartmentalization (see Section 2.4.4).

2.3.5 Capability Flags

Capabilities include a flags field that can be manipulated freely. Unlike the permissions field, it does not determine privilege, i.e., the state of this field is orthogonal to capability monotonicity. Currently, there are only architecture-specific interpretations for this field: CHERI-RISC-V uses it to control opcode interpretation on instruction fetch. In the future, other non-security behavioral flags relating to capabilities may be placed here.

2.3.6 Sealed Capabilities

Capability *sealing* allows capabilities to be marked as *immutable* and *non-dereferenceable*, causing hardware exceptions to be thrown if attempts are made to modify, dereference, or jump to them. This enables capabilities to be used as unforgeable tokens of authority for higher-level software constructs grounded in *encapsulation*, while still allowing them to fit within the pointer-centric framework offered by CHERI capabilities. Sealed capabilities are the foundation for building the CheriBSD *object-capability model* supporting in-address-space compartmentalization, where pairs of sealed code and data capabilities are object references whose invocation triggers a protection-domain switch. Sealed capabilities can also be used to support other operating-system or language robustness features, such as representing other sorts of delegated (non-hardware-defined) rights, or ensuring that pointers are dereferenced only by suitable code (e.g., in support of language-level memory or type safety).

2.3.7 Capability Object Types

Capabilities contain an additional piece of metadata, an *object type*, updated when a capability undergoes (un)sealing. Object types allow multiple sealed capabilities to be indelibly (and indivisibly) linked, so that the kernel or language runtime can avoid expensive checks (e.g., via table lookups) to confirm that they are intended to be used together. For example, for object-oriented compartmentalization models (such as the CheriBSD object-capability model), pairs of sealed capabilities can represent objects: one is the code capability for a class, and the other is a data capability representing the data associated with a particular instance of an object. In the CheriBSD model, these two sealed capabilities have the same value in their object-type field, and two candidate capabilities passed to object invocation will not be accepted together if their object types do not match.

The object-type field is set when a capability is sealed based on a second input capability authorizing use of the type space – itself simply a capability permission authorizing sealing within a range of values specified by the capability's bounds. A similar model authorizes *unsealing*, which permits a sealed capability to be restored to a mutable and dereferenceable state – if a suitable capability to have sealed it is held. This is used in the CheriBSD model during object invocation to grant the callee access to its internal state.

A similar model could be achieved without using an unsealing mechanism: a suitably privileged component could inspect a sealed capability and rederive its unsealed contents. However, authorizing both sealing and unsealing based on type capabilities allows the right to construct encapsulated pointers to be delegated, without requiring recourse to a privileged software supervisor at the cost of additional domain transitions – or exercise of unnecessary privilege.

2.3.8 Sealed Capability Invocation

CHERI supports two forms of non-monotonicity: jump-like capability invocation, and exception handling (see Section 2.3.15). In CHERI-MIPS, the `CCall` instruction (optionally paired with use of the `CReturn` instruction) accepts a pair of sealed capability operands on which various checks are performed (for example, that they are valid, sealed, and have matching object types). If all tests are passed, then additional capabilities become available to the executing CPU context – either by virtue of unsealing of the operand registers (jump-like `CCall`) or by control transferring to the exception handler (exception-based `CCall`).

For both models, the destination execution environment has well-defined and reliable properties, such as a controlled target program-counter capability and additional data capability that can be used to authorize domain transition. The jump-like model avoids the microarchitectural overhead of exception delivery, behaving much like a conventional jump to register, permitting an in-address-space domain switch without changing rings.

In both cases, the newly executing code has the ability to further manipulate execution state, and impose semantics such as call-return secure function invocation (CheriBSD) or secure asynchronous message passing (microkernel), which will likely be followed by a privilege de-escalation as a target domain is entered (see Section 2.4.4).

Object-Capability Policies in CHERI

Consider an execution environment having access to several capabilities sealed with the same `otype`. The tests required by the jump-like `CCall` mechanism describe a *Cartesian product* of method rights (indicated by the sealed code capability) and object rights (sealed data capability) to this environment. Regardless of how the environment came to have these sealed capabilities, it is free to pair any sealed code capability with any sealed data capability and have the `CCall` tests pass.

Non-Cartesian and/or stateful policies can, however, be encoded by indirection, using memory to store additional data to be checked by the invoked subsystem on entry. The sealed data pointers given out by the invoked subsystem now no longer directly reference objects; instead, they reference "data trampolines" describing the pairing of object(s) *and remote agent(s)* with associated access rights information. Attenuation of access rights is no longer necessarily an ambiently available action and requires either the explicit construction of membranes (i.e., proxy objects) or active cooperation of the invoked subsystem (or an agent acting on its behalf) to create new data trampoline(s).

2.3.9 Capability Protection for Non-Pointer Types

While the design of CHERI capabilities is primarily focused on the protection of pointers, the pointer interpretation of capabilities depends entirely on a capability's permissions mask. If the mask authorizes load, store, and fetch instructions, then the capability has a pointer interpretation. Capabilities are not required to have those permissions set, however, allowing capabilities to be used for other purposes – for example, to protect other critical data types from in-memory corruption (such as implementing UNIX file descriptors or stack canaries), or to authorize access to system services (such as authorizing use of specific system calls identified by the capability). Sealed capabilities and a set of software-defined permissions bits facilitate these use cases by permitting non-architecture-defined capability interpretations while retaining capability-based protections.

2.3.10 Capability Flow Control

The CHERI capability model is designed to support the implementation of language-level pointers: tagged memory allows capabilities to be stored in memory, and in particular, embedded within software-managed data structures such as objects or the stack. CHERI is therefore particularly subject to a historic criticism of capability-system models – namely, that capability propagation makes it difficult to track down and revoke rights (or to garbage collect them). To address this concern, CHERI has three mechanisms by which the flow of capabilities can be constrained:

Capability TLB bits extend the existing load and store permissions on TLB entries (or, in architectures with hardware page-table walkers, page-table entries) with new permissions to authorize loading and storing of capabilities. This allows the operating system to maintain pages from which tagged capabilities cannot be loaded (tags will be transparently stripped on load), and to which capabilities cannot be stored (a hardware exception will be thrown). This can be used, for example, to prevent tagged capabilities from being stored in memory-mapped file pages (as the underlying object might not support tag storage), or to create regions of shared memory through which capabilities cannot flow.

Capability load and store permission bits extend the load and store permissions on capabilities themselves, similarly allowing a capability to be used only for data access – if suitably configured. This can be used to create regions of shared memory within a virtual address space through which capabilities cannot flow. For example, it can prevent two separated compartments from delegating access to one another's memory regions, instead limiting communication to data traffic via the single shared region.

Capability control-flow permissions "color" capabilities to limit propagation of specific types of capabilities via other capabilities. This feature marks capabilities as *global* or *local* to indicate how they can be propagated. Global capabilities can be stored via any capability authorized for capability store. Local capabilities can be stored only via a capability specifically authorized as *store local*. This can be used, for example, to prevent propagation of temporally sensitive stack memory between compartments, while still allowing garbage-collected heap memory references to be shared.

This feature remains under development, as we hope to generalize it to further uses such as limiting the propagation of ephemeral DRAM references in persistent-memory systems. However, it is used successfully in the CheriBSD compartmentalization model to improve memory safety and limit obligations of garbage collection.

The decision to strip tags on load, but throw an exception on store, reflects pragmatic software utilization goals: language runtimes and system libraries often need to implement *capability-oblivious memory copying*, as the programmer may not wish to specify whether a region of memory must (or must not) contain capabilities. By stripping tags rather than throwing an exception on load, a capability-oblivious memory copy is safe to use against arbitrary virtual addresses and source capabilities – without risk of throwing an exception. Software that wishes to copy only data from a source capability (excluding tag bits due to a non-propagation goal) can simply remove the load-capability permission from the source capability before beginning a memory copy.

On the other hand, it is often desirable to detect stripping of a capability on store via a hardware exception, to ease debugging. For example, it is typically desirable to catch storing a tagged capability to a file as early as possible in order to avoid debugging a later failed dereference due to loss of a tag. Similarly, storing a tagged capability to a virtual-memory page might be an indicator to a garbage collector that it may now be necessary to scan that page in search of capabilities.

This design point conserves TLB and permission bits; there is some argument that completing the space (i.e., shifting to three or four bits each) would offer functional improvements – for example, the ability to avoid exceptions on a capability-oblivious memory copy via a capability that does not authorize capability store, or the ability to transparently strip tags on store to a shared memory page. However, we have not yet found these particular combinations valuable in our software experimentation,

2.3.11 Capability Compression

The 256-bit in-memory representation of CHERI capabilities provides full accuracy for pointer lower bounds and upper bounds, as well as a large *object type space* with software-defined permissions. The 128-bit implementation of CHERI uses floating-point-like *fat-pointer compression techniques* that rely on redundancy between the three 64-bit virtual addresses. The compressed representation exchanges stronger alignment requirements (proportional to object size) for a more compact representation. The CHERI Concentrate compression model (see Section 3.4.4) maintains the monotonicity inherent in the 256-bit model: no ISA manipulation of a capability can grant increased rights, and when unrepresentable cases are generated (e.g., a pointer substantially out of bounds, or a very unaligned object), the pointer becomes un-dereferenceable. Memory allocators already implement alignment requirements for heap and stack allocations (word, pointer, page, and superpage alignments), and these algorithms require only minor extension to ensure fully accurate bounds for large memory allocations. Small allocations require no additional alignment, where the definition of ‘small’ depends on the compression format used and might be from 4 kiB to 1 MiB. Relative to a 64-bit pointer, the 128-bit design reduces per-pointer memory overhead (with a strong influence on cache footprint for some software designs) by roughly two thirds, compared to the 256-bit representation.

2.3.12 Hybridization with Integer Pointers

Processors implementing CHERI capabilities also support existing programs compiled to use conventional integer pointers rather than capabilities, using two special capabilities:

Default Data Capability indirects and controls non-capability-based pointer-based load and store instructions.

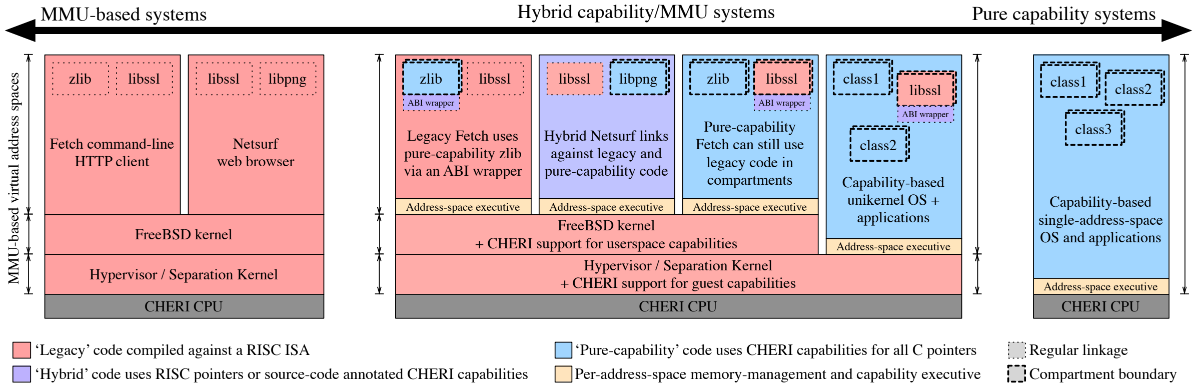
MMU-based systems
Hybrid capability/MMU systems
Pure capability systems
MMU-based virtual address spaces
zlib
libssl
Fetch command-line HTTP client
libssl
libpng
Netsurf web browser
FreeBSD kernel
Hypervisor / Separation Kernel
CHERI CPU
zlib
ABI wrapper
libssl
Legacy Fetch uses pure-capability zlib via an ABI wrapper
Address-space executive
libssl
libpng
Hybrid Netsurf links against legacy and pure-capability code
Address-space executive
zlib
libssl
ABI wrapper
Pure-capability Fetch can still use legacy code in compartments
Address-space executive
class1
libssl
ABI wrapper
class2
Capability-based unikernel OS + applications
Address-space executive
FreeBSD kernel + CHERI support for userspace capabilities
Hypervisor / Separation Kernel + CHERI support for guest capabilities
CHERI CPU
class1
class2
class3
Capability-based single-address-space OS and applications
Address-space executive
CHERI CPU
'Legacy' code compiled against a RISC ISA
'Hybrid' code uses RISC pointers or source-code annotated CHERI capabilities
'Pure-capability' code uses CHERI capabilities for all C pointers
Per-address-space memory-management and capability executive
Regular linkage
Compartment boundary

Figure 2.2: CHERI supports a wide range of operational software models including: unmodified MMU-based RISC operating systems; hybrid operating systems utilizing the MMU to support a process model and/or virtualization while using CHERI within virtual address spaces; and pure single-address-space CHERI-based operating systems.

Program Counter Capability extends the conventional program counter with capability metadata, indirecting and controlling instruction fetches.

Programs compiled to use capabilities to represent pointers (whether implicitly or via explicit program annotations) will not use the default data capability, instead employing capability registers and capability-based instructions for pointer operations and indirection. The program-counter capability will be used regardless of the code model employed, although capability-aware code generation will employ constrained program-counter bounds and permissions to implement control-flow robustness rather than using a single large code segment. Support for legacy loads and stores can be disabled by installing a sufficiently constrained (e.g., untagged) default data capability.

Different compilation modes and ABIs provide differing levels of compatibility with existing code – but include the ability to run entirely unmodified non-CHERI binaries, to execute non-CHERI code in sandboxes within CHERI-aware applications, and CHERI-aware code in sandboxes within CHERI-unaware applications.

2.3.13 Hybridization with Virtual Addressing

The above features compose naturally with, and complement, the Virtual-Memory (VM) models commonly implemented using commodity Memory Management Units (MMUs) in current OS designs (Figure 2.2). Capabilities are *within* rather than *between* address spaces; they protect programmer references to data (pointers), and are intended to be driven primarily by the compiler rather than by the operating system. In-address-space compartmentalization complements process isolation by providing fine-grained memory sharing and highly efficient domain switching for use between compartments in the same application, rather than between independent programs via the process model. Operating-system kernels will also be able to use capabilities to improve the safety of their access to user memory, as user pointers cannot be accidentally used to reference kernel memory, or accidentally access memory outside of user-provided buffers. Finally, the operating system might choose to employ capabilities internally, and even in its interactions with userspace, in referencing kernel data structures and objects.

2.3.14 Hybridization with Architectural Privilege

Conventional architectures employ ring-based mechanisms to control use of architectural privilege: only code executing in "supervisor" or "kernel" mode is permitted to access the virtual address space with supervisor rights, but also to control the MMU, certain cache management operations, interrupt-related features, system-call return, and so on. The ring model prevents unprivileged code from manipulating the virtual address space (and other processor features) in such a way as to bypass memory protection and isolation configured by the operating system. Contemporary instantiations may also permit virtualization of those features, allowing unmodified operating systems to execute efficiently over microkernels or hypervisors. CHERI retains support for these models with one substantial modification: use of privileged features within privileged rings, other than in accessing virtual memory as the supervisor, depends on the program-counter capability having a suitable hardware permission set.

This feature similarly allows code *within* kernels, microkernels, and hypervisors to be compartmentalized, preventing bypass of the capability model within the kernel virtual address space through control of virtual memory features. The feature also allows vulnerability mitigation by allowing only explicit use of privileged features: kernel code can be compiled and linked so that most code executes with a program-counter capability that does not authorize use of privilege, and only by jumping to selected program-counter capabilities can that privilege be exercised, preventing accidental use. Finally, this feature paves the way for process and object models in which the capability model is used without recourse to rings.

2.3.15 Failure Modes and Exceptions

Bounds checks, permissions, monotonicity, and other properties of the CHERI protection model inevitably introduce the possibility of new ISA-visible failure modes when software violates rules imposed through capabilities (whether due to accident or malicious intent). In general, in our prototyping, we have selected to deliver *hardware exceptions* as early as possible when such events occur; for example, on attempts to perform disallowed load and store operations, to broaden bounds, and so on. This allows the operating system (which in turn may delegate to the userspace language runtime or application) the ability to catch and handle failures in various ways – such as by emulating disallowed accesses, converting to a language-visible exception, or performing some diagnostic or mitigation activity.

Different architectures express differing design philosophies for when exceptions may be delivered, and there is flexibility in the CHERI model in when exceptions might be delivered. For example, while an attempt to broaden (rather than narrow) bounds could generate an immediate exception (our prototyping choice), the operation could instead generate a non-dereferenceable pointer as its output, in effect deferring an exception until the time of an attempted load, store, or instruction fetch. The former offers slightly improved debuggability (by exposing the error earlier), whereas the latter can offer microarchitectural benefits by reducing the set of instructions that can throw exceptions. Both of these implementations ensure monotonicity by preventing derived pointers from improperly allowing increased access following guarded manipulation, and are consistent with the model.

2.3.16 Capability Revocation, Garbage Collection, and Flow Control

Revocation is a key design concern in capability systems, as revocation is normally implemented via table indirection – an approach in tension with the CHERI design goal of avoiding table-based lookups or indirection on pointer operations. As described in Section 2.3.10, CHERI provides explicit ISA-level features to constrain the flow of capabilities in order to reduce the potential overhead in walking through memory to find outstanding capabilities to resources (e.g., to implement garbage collection or sweeping revocation). There are also explicit features in the instruction-set architecture that directly support the implementation of both pointer and object-capability revocation:

MMU-based virtual-address revocation As CHERI capabilities are evaluated prior to virtual addressing (i.e., they are pointers within address spaces), the MMU can be used not only to maintain virtual address spaces, but also to explicitly prevent the dereferencing of pointers to virtual address ranges – regardless of the capability mechanism. Combined with a policy of either non-reuse of virtual address space (as distinct from non-reuse of physical address space), sweeping revocation, or garbage collection, this allows all outstanding capabilities (and any further capabilities derived from them) to be revoked without the need to search for those capabilities in the register file or memory. This revocation is subject to the granularity and scalability limitations of MMUs: for example, it is not possible to revoke portions of the virtual address space smaller than one page.

This low-level hardware mechanism must be combined with suitable software management of the virtual address space in order for it to be effective. For example, a policy of non-reuse of the virtual address space at allocation time will prevent stale capabilities from referring to a new allocation after an old one has been freed. A further policy of revoking MMU mappings for the region of virtual address space will prevent use of the freed memory as a communications channel from the point of free. Asynchronous and batched revocations will improve performance, subject to windows of opportunity in which use after free (but not use after re-allocation) might still be possible. It is also worth observing explicitly that non-reuse of the virtual address space in no way implies non-reuse of physical memory, as memory underlying revoked virtual addresses can be safely reused. An alternative to virtual address-space non-reuse is garbage collection, in which outstanding references to freed (and perhaps revoked) virtual address space are sought and explicitly invalidated.

Use of the MMU for virtual address-space revocation is subject to a number of limits depending on the non-reuse and garbage-collection policies adopted. For example, if small, sub-page-size, tightly packed memory allocations are freed in a manner that leads to fragmentation (i.e., both allocated and freed memory within the same virtual page), then revocation will not be possible – as it would prevent access to valid allocations (which could be emulated only at great expense). Similarly, fragmentation of the virtual address space may lead to greater overhead in the OS's virtual-memory subsystem, due to the need to maintain many individual small mappings, as well as the possibility of reduced opportunity to use superpages should revocations occur that are expressed in terms of smaller page sizes.

However, overall, the MMU provides a non-bypassable means of preventing use of all outstanding capabilities to a portion of the virtual address space, permitting strong revocation to be used where appropriate.

Accurate garbage collection Traditional implementations of C are not amenable to accurate garbage collection because unions and types such as `intptr_t` allow a register or memory location to contain either an integer value or a pointer. CHERI-C does not have this limitation: The tag bit makes it possible to accurately identify all memory locations that contain data that can be interpreted as a pointer. In addition, the value of the pointer (encoded in the offset) is distinct from the base and length; thus, code that stores other data in low bits of the pointer will not affect the collector. Garbage collection is the logical dual of revocation: garbage collection extends the lifetime of objects as long as they have valid references, whereas revocation curtails the lifetime of references once the objects to which they refer are no longer valid. A simple stop-the-world mark-and-sweep collector for C can perform both tasks, scanning all reachable memory, invalidating all references to revoked objects, and recycling unreachable memory.

More complex garbage collectors typically rely on read or write barriers (i.e., mechanisms for notifying the collector that a reference has been read or written). These are typically inserted by the compiler; however, in the context of revocation the compiler-generated code must be treated as untrusted. It may be possible to use the permission bits – either in capabilities themselves or in page-table entries – to introduce traps that can be used as barriers.

Capability tags for sweeping revocation In addition to supporting garbage collection, capability tags in registers and memory also allow the reliable identification of capabilities for the purposes of explicit revocation. Subject to safety in the presence of concurrency (e.g., by suspending software execution in the virtual address space, or temporarily limiting access to portions of the virtual address space), software can reliably sweep through registers and memory, clearing the tags (or otherwise replacing) for capabilities that are to be revoked. This comes at potentially significant cost, which can be mitigated through use of the MMU – e.g., to prevent capabilities from being used in certain pages intended only to store data, or to track where capabilities have been stored via a capability dirty bit in virtual-memory metadata.

Revocation of sealed capabilities When the interpretation of sealed capabilities is performed by a trustworthy software exception handler, there is the opportunity for that exception handler to implement revocation semantics explicitly. For example, the `CCall` selector 0/`CReturn` exception handler could interpret the virtual address of a sealed capability as pointing to a table entry within the kernel, rather than directly encapsulating a pointer to user memory. The address could be split into two parts: a table index, and a generation counter. The table entry could then itself contain a generation counter. Sealed object-capability references to the table entry would incorporate the value of the counter at the time of sealing, and the `CCall` mechanism would check the generation count, rejecting invocation on a mismatch. When object-capability revocation is desired, the table generation counter could be bumped, preventing any further use of outstanding references. This approach would be subject to limits on table-entry reuse and the size of the table; for example, a reasonable design might employ a 24-bit table index (permitting up to 2^{24} objects in the system at a time) and a 40-bit generation counter. Use of the 24-bit object-type could further increase the number of objects permissible in the system concurrently. Many other similar schemes incorporating explicit checks for revocation based on software interposition employing counters, tables, etc., can be imagined.

CHERI includes several architectural features to facilitate techniques such as garbage collection and sweeping revocation. Tags allow capabilities to be accurately identified in both registers and memory. In addition, CHERI can limit the flow of capabilities via various mechanisms, limiting the memory areas that must be swept for the two techniques: MMU permissions controlling capability load and store via specific pages; capability permissions controlling capability load and store via specific capabilities; and the local-global feature that controls the propagation of subsets of capabilities. These primitives may be combined to support higher-level software policies such as:

- “capabilities may not be shared between address spaces”
- “local stack capabilities may be stored only to the local stack”
- “this shared-memory buffer can be used only for data sharing, not capability sharing”
- “capabilities can flow only one way through this shared buffer”
- “only the TCB can introduce capabilities to shared memory between compartments”
- “supervisor involvement is required to share sealed capabilities between compartments”
- “first store of a capability to any page will deliver an exception to the supervisor”

As a result, garbage collection and sweeping revocation can rely on strong invariants about capability propagation that limit the areas of memory that must be swept for garbage collection or revocation.

2.4 Software Protection and Security Using CHERI

The remainder of the chapter explores these ideas in greater detail, describing the high-level semantics offered by the ISA and how they are mapped into programmer-visible constructs such as C-language features. The description in this chapter is intended to be agnostic to the specific Instruction-Set Architecture (ISA) in which CHERI is implemented. Whereas the implementation described in later chapters maps into the 64-bit MIPS ISA, the overall CHERI strategy is intended to support a variety of ISA backends, and could be implemented in the 64-bit ARMv8, SPARCv9, or RISC-V ISAs with only modest localization. In particular, it is important that programmers be able to rely on the properties described in this chapter – regardless of the ISA-level implementation – and that software abstractions built over these properties have consistent behavior that can be depended upon to mitigate vulnerabilities.

2.4.1 Abstract Capabilities

The CHERI architecture imposes tight constraints on capability manipulation and use including provenance validity and monotonicity. While these rules generally permit the execution of current C and C++ code without significant modification, there are occasions on which the programmer model of pointer properties (for example) may violate rules for capabilities. For example, the architecture maintains provenance validity of capabilities from reset, permitting them to remain valid only if they are held in tagged memory or registers. In practice, operating systems may swap memory pages from DRAM to disk and back, violating architectural provenance validity. The OS kernel is able to maintain the appearance of provenance validity for swapped pages by saving tags when swapping out, and re-deriving capabilities from valid architectural capabilities when swapped back in – maintaining the *abstract capabilities* that compiler-generated code works with. Our ASPLOS 2019 paper on CheriABI explores this issue in detail [28], covering topics such as context switching, the C-language runtime, virtual-memory behavior, and debugging.

2.4.2 C/C++ Language Support

CHERI has been designed so that there are clean mappings from the C and C++ programming language into these protection properties. Unlike conventional virtual memory, the compiler (rather than the operating system) is intended to play the primary role in managing these protections. Protection is within address spaces, whether in a conventional user process, or within the operating-system kernel itself in implementing its own services or in accessing user memory:

Spatial safety CHERI protections are intended to directly protect the *spatial safety* of userspace types and data structures. This protection includes the integrity of pointers to code and data, as well as implied code pointers in the form of return addresses and vtable entries; bounds on heap and stack allocations; the prevention of executable data, and modification of executable code via permission.

Temporal safety CHERI provides instruction-set foundations for higher-level *temporal safety* properties, such as non-reuse of heap allocations via garbage collection and revocation, and compiler clearing of return addresses on the stack. In particular, the capability tags on registers and in memory allows pointers to be reliably located and atomically replaced with a different value (including an invalid capability). Acceleration features allow capabilities to be located more efficiently than simply sweeping all of physical memory.

Software compartmentalization CHERI provides hardware foundations for highly efficient *software compartmentalization*, the fine-grained decomposition of larger software packages into smaller isolated components that are granted access only to the memory (and also software-defined) resources they actually require.

Enforcing language-level properties CHERI's software-defined permission bits and sealing features can also be used to enforce other language-level protection objectives (e.g., opacity of pointers exposed outside of their originating modules) or to implement hardware-assisted type checking for language-level objects (e.g., to more robustly link C++ objects with their corresponding vtables).

CHERI protections are implemented by a blend of functionality:

Compiler and linker responsible for generating code that manipulates and dereferences code and data pointers, compile-time linkage, and stack allocation.

Language runtime responsible for ensuring that program run-time linkage, memory allocation, and exceptions implement suitable policies in their refinement and distribution of capabilities to the application and its libraries.

Operating-system kernel responsible for interactions with conventional virtual memory, maintaining capability state across context switches, reporting protection failures via signals or exceptions, and implementing domain-transition features used with compartmentalization.

Application program and libraries responsible for distributing and using pointers, allocating and freeing memory, and employing higher-level capability-based protection features such as compartmentalization during software execution.

Data-Pointer Protection

Depending on the desired compilation mode, some or all data pointers will be implemented using capabilities. We anticipate that memory allocation (whether from the stack or heap, or via kernel memory mapping) will return capabilities whose bounds and permissions are suitable for the allocation, which will then be maintained for any derived pointers, unless explicitly narrowed by software. This will provide the following general classes of protections:

Pointer integrity protection Overwriting a pointer in memory with data (e.g., received over a socket) will not be able to construct a dereferenceable pointer.

Pointer provenance checking and monotonicity Pointers must be derived from prior pointers via manipulations that cannot increase the range or permissions of the pointer.

Bounds checking Pointers cannot be moved outside of their allocated range and then be dereferenced for load, store, or instruction fetch.

Permissions checking Pointers cannot be used for a purpose not granted by its permissions. In as much as the kernel, compiler, and run-time linker restrict permissions, this will (for example) prevent data pointers from being used for code execution.

Bounds or permissions subsetting Programmers can explicitly reduce the rights associated with a capability – e.g., by further limiting its valid range, or by reducing permissions to perform operations such as store. This might be used to narrow ranges to specific elements in a data structure or array, such as a string within a larger structure.

Flow control on pointers Capability (and hence pointer) flow propagation can be limited using CHERI's capability flow-control mechanism, and used to enforce higher-level policies such as that *stack capabilities cannot be written to global data structures*, or that *non-garbage-collectable capabilities cannot be passed across domain transitions*.

Code-Pointer Protection

Again with support of the compiler and linker, CHERI capabilities can be used to implement control-flow robustness that prevents code pointers from being corrupted or misused. This can limit various forms of control-flow attacks, such as overwriting of return addresses on the stack, as well as pointer re-use attacks such as *Return-Oriented Programming (ROP)* and *Jump-Oriented Programming (JOP)*. Potential applications include:

Return-address protection Capabilities can be used in place of pointers for on-stack return addresses, preventing their corruption.

Function-pointer protection Function pointers can also be implemented as capabilities, preventing corruption.

Exception-state protection On-stack exception state and signal frame information also contain pointers whose protection will limit malicious control-flow attacks.

C++ vtable protection A variety of control-flow attacks rely on either corrupting C++ vtables, or improper use of vtables, which can be detected and prevented using CHERI capabilities to implement both pointers to, and pointers in, vtables.

2.4.3 Protecting Non-Pointer Types

One key property of CHERI capabilities is that although they are designed to represent pointers, they can also be used to protect other types – whether those visible directly to programmers through APIs or languages, or those used only in lower-level aspects of the implementation to improve robustness. A capability can be stripped of its hardware interpretation by masking all hardware-defined permission bits (e.g., those authorizing load, store, and so on). A set of purely software-defined permission bits can be retrieved, masked, and checked using suitable instructions. Sealed capabilities further impose immutability on capability fields. These non-pointer capabilities benefit from tag-based integrity and provenance protections, monotonicity, etc. There are many possible use cases, including:

- Using CHERI capabilities to represent hardware resources such as physical addresses, interrupt numbers, and so on, where software will provide implementation (e.g., allocation, mapping, masking), but where capabilities can be stored and delegated.

- Using CHERI capabilities as canaries in address spaces: while stripping any hardware-defined interpretation, tagged capabilities can be used to detect undesired memory writes where bounds may not be suitable.

- Using CHERI capabilities to represent language-level type information, where there is not a hardware interpretation, but unforgeable tokens are required – for example, to authorize use of vtables by suitable C++ objects.

2.4.4 Isolation, Controlled Communication, and Compartmentalization

In *software compartmentalization*, larger complex bodies of software (such as operating-system kernels, language runtimes, web browsers, and office suites) are decomposed into multiple components that run in isolation from one another, having only selectively delegated rights to the broader application and system, and limited further attack surfaces. This allows the impact of exploited vulnerabilities or faults to be constrained, subject to software being suitably structured – i.e., that its privileges and functionality have been suitable decomposed and safely represented. Software sandboxing is one example of compartmentalization, in which particularly high-risk software is tightly isolated due to the risks it poses – for example, in rendering HTML downloaded from a web site, or in processing images attached to e-mail. Compartmentalization is a more general technique, of which sandboxing is just one design pattern, in which privileges are delimited and minimized to improve software robustness [57, 101, 131, 46]. Software compartmentalization is one of the few known techniques able to mitigate future unknown classes of software vulnerability and exploitation, as its protective properties do not depend on the specific vulnerability or exploit class being used by an attacker.

Software compartmentalization is build on two primitives: *software isolation* and *controlled communication*. CHERI hybridizes two orthogonal mechanisms exist to construct isolation and controlled communication: the conventional MMU (using multiple virtual address spaces as occurs in widely used sandboxed process models), and CHERI's in-address-space capability mechanism (by constructing closures in the graph of reachable capabilities). These mechanisms can be combined to construct fine-grained software compartmentalization within virtual address spaces, which may complement (or even replace) a virtual-address-based process model.

To constrain software execution using CHERI, a more privileged software runtime must arrange that only suitable capabilities are delegated to software that must run in isolation. For example, the runtime might grant software access to its own code, a stack, global variables, and heap storage, but not to the private privileged state of the runtime, nor to the internal state of other isolated software components. This is accomplished by suitably initializing the thread register file of the software (and hence CPU register file when it begins execution) to point into an initial set of delegated code and allocation capabilities, and then exercising discretion in storing capabilities into any further memory that it can reach. Capability nonforgeability, monotonicity, and provenance validity ensure that new rights cannot be created by constrained software, and that existing rights cannot be escalated. As isolation refers not just to the initial state, but also the continuing condition of software, discretion in delegating capabilities must be continued throughout execution, in much the same way that software isolation using the MMU depends not just on safe initial configuration, but safe continuing configuration as code executes.

In order to achieve compartmentalization, and not simply isolation, CHERI's selective non-monotonic mechanisms can be used: exception handling, and jump-based invocation. If the software supervisor arranges that additional rights will be acquired by the exception handler (using more privileged kernel code and data capabilities), then the exception handler will be able to perform non-monotonic transformations on the set of capabilities in the register file, accessing memory (and other resources) unavailable to the isolated code. Sealed capabilities allow encapsulated handles to resources to be delegated to isolated code in such a manner that the sealed capabilities and resources they describe can be protected from interference. CHERI's jump-based invocation mechanism allows those resources to be unsealed in a controlled manner, with control flow transferred to appropriate receiving code in a way that protects both the caller and callee. This source of non-monotonicity can also be used to implement domain transition by having the caller discard rights prior to performing the jump, and the callee acquire any necessary rights via unsealing of its capabilities. It is essential to CHERI's design that exercise of non-monotonicity support reliable transfer of control to code trusted with newly acquired rights.

Efficient controlled communication can persist across domain transitions through the appropriate delegation of capabilities to shared memory, as well as the delegation of sealed capabilities allowing selected domain switching. CHERI's permissions allow uses of shared memory to be constrained in a variety of ways. The software configuring compartmentalization might choose to delegate load-only or load-execute access to shared code or read-only data segments. Other permissions constrain the propagation of capabilities; for example, the software supervisor might allow communication only using data and not capabilities via a communication ring between two mutually distrusting phases in a processing pipeline. Similarly, CHERI's local-global protections might be utilized to prevent capabilities for non-garbage-collectable memory from being shared between mutually distrusting components, while still allowing garbage-collectable heap allocations to be delegated.

Collectively, these mechanisms allow a variety of software-defined compartmentalization models to be constructed. We have experimented with several, including the CheriBSD in-process compartmentalization mechanism, which models domain transition on a secure function call with trusted stack maintained by the operating-system kernel via exception-based invocation [146, 143], and microkernel-based systems that utilize jump-based domain transition within a single-address-space operating system, which model domain transition on asynchronous or synchronous message passing. Effective software compartmentalization relies not only on limiting access to memory, but also a variety of other properties such as appropriate (perhaps fair or prioritized) scheduling, resource allocation, and non-leakage of data or rights via newly allocated or freshly reused memory, which are higher-level properties that must be ensured by the software supervisor. While many of these concerns exist in MMU-based software compartmentalization, they can take on markedly different forms or implications. For example, the zeroing of memory before reuse prevents the leakage of rights, and not just data, in the capability model. As with MMU-based isolation and compartmentalization, CHERI provides strong architectural primitives, and is not intended to directly address microarchitectural concerns such as cache side channels or information leakage through branch predictors, performance counters, or other state.

Substantially different architectural underpinnings for capability-based, rather than MMU-based, compartmentalization give it quite different practical properties. For example, two protection domains sharing access to a region of memory will not experience increased page-table and TLB footprint by virtue of sharing a virtual address space. Similarly, the model for delegating shared memory is substantially different: simple pointer delegation, rather than page-table construction, has far lower overhead. On the other hand, revoking access to shared memory via the capability model requires either non-reuse of portions of the virtual address space, sweeping capability revocation, or garbage collection (see Section 2.3.16). We have found that the two approaches complement one another well: virtual memory continues to provide a highly useful underpinning for conventional coarse-grained virtual-machine and process models, whereas CHERI compartmentalization works extremely well within applications as it caters to rapid domain switching and large amounts of sharing between fine-grained and tightly coupled components.

2.4.5 Source-Code and Binary Compatibility

CHERI supports Application Programming Interfaces (APIs) and Application Binary Interfaces (ABIs) with compatibility properties intended to facilitate incremental deployment of its features within current software environments. For example, an OS kernel can be extended to support CHERI capabilities in selected userspace processes with only minor extensions to context switching and process setup, allowing both conventional and CHERI-extended programs to execute – without implying that the kernel itself needs to be implemented using capabilities. Further, given suitable care with ABI design, CHERI-extended libraries can exist within otherwise unmodified programs, allowing fine-grained memory protection and compartmentalization to be deployed selectively to the most trusted software (i.e., key system libraries) or least trustworthy (e.g., video CODECs), without disrupting the larger ecosystem. CHERI has been tested with a large range of system software, and efficiently supports a broad variety of C programming idioms poorly supported by the state of the art in software memory protection. It provides strong and reliable hardware-assisted protection in eliminating common exploit paths that today can be mitigated only by using probabilistically correct mechanisms (e.g., grounded in address-space randomization) that often yield to determined attackers.

2.4.6 Code Generation and ABIs

Compilers, static and dynamic linkers, debuggers, and operating systems will require extension to support CHERI capabilities. We anticipate multiple conventions for code generation and binary interfaces, including:

Conventional code generation Unmodified operating systems, user programs, and user libraries will work without modification on CHERI processors. This code will not receive the benefits of CHERI memory protection – although it may execute encapsulated within sandboxes maintained by CHERI-aware code, and thus can participate in a larger compartmentalized application. It will also be able to call hybrid code.

Hybrid code generation Conventional code generation, calling conventions, and binary interfaces can be extended to support (relatively) transparent use of capabilities for selected pointers – whether hand annotated (e.g., with a source-code annotation) or statically determined at compile time (e.g., return addresses pushed onto the stack). Hybrid code will generally interoperate with conventional code with relative ease – although conventional code will be unable to directly dereference capability-based types. CHERI memory-protection benefits will be seen only for pointers implemented via capabilities – which can be adapted incrementally based on tolerance for software and binary-interface modification.

Pure-capability code generation Software can also be compiled to use solely capability-based instructions for memory access, providing extremely strong memory protection. Direct calling in and out of pure-capability code from or to conventional code or hybrid code requires ABI wrappers, due to differing calling conventions. Extremely strong memory protection is experienced in the handling of both code and data pointers.

Compartmentalized code is accessed and can call out via object-capability invocation and return, rather than by more traditional function calls and returns. This allows strong isolation between mutually distrusting software components, and makes use of a new calling convention that ensures, among other properties, non-leakage of data and capabilities in unused argument and return-value registers. Compartmentalized code might be generated using any of the above models; although it will experience greatest efficiency when sharing data with other compartments if a capability-aware code model is used, as this will allow direct loading and storing from and to memory shared between compartments. Containment of compartmentalized components does not depend on the trustworthiness of the compiler used to generate code for those components.

Entire software systems need not utilize only one code-generation or calling-convention model. For example, a kernel compiled with conventional code, and a small amount of CHERI-aware assembly, can host both hybrid and pure-capability userspace programs. A kernel compiled to use pure-capability or hybrid code generation could similarly host userspace processes using only conventional code. Within the kernel or user processes, some components might be compiled to be capability-aware, while others use only conventional code. Both capability-aware and conventional code can execute within compartments, where they are sandboxed with limited rights in the broader software system. This flexibility is critical to CHERI's incremental adoption model, and depends on CHERI's hybridization of the conventional MMU, OS models, and C programming-language model with a capability-system model.

2.4.7 Operating-System Support

Operating systems may be modified in a number of forms to support CHERI, depending on whether the goal is additional protection in userspace, in the kernel itself, or some combination of both. Typical kernel deployment patterns, some of which are orthogonal and may be used in combination, might be:

Minimally modified kernel The kernel enables CHERI support in the processor, initializes register state during context creation, and saves/restores capability state during context switches, with the goal of supporting use of capabilities in userspace. Virtual memory is extended to maintain tag integrity across swapping, and to prevent tags from being used with objects that cannot support them persistently – such as memory-mapped files. Other features, such as signal delivery and debugging support require minor extensions to handle additional context. The kernel can be compiled with a capability-unaware compiler and limited use of CHERI-aware assembly. No additional protection is afforded to the kernel in this model; instead, the focus is on supporting fine-grained memory protection within user programs.

Capability domain switching in userspace Similar to the minimally modified kernel model, only modest changes are made to the kernel itself. However, some additional extensions are made to the process model in order to support multiple mutually distrusting security domains within user processes. For example, new `CCall` and `CReturn` exception handlers are created, which implement kernel-managed 'trusted stacks' for each user thread. Access to system calls is limited to authorized userspace domains.

Fine-grained capability protection in the kernel In addition to capability context switching, the kernel is extended to support fine-grained memory protection throughout its design, replacing all kernel pointers with capabilities. This allows the kernel to benefit from pointer tagging, bounds checking, and permission checking, mitigating a broad range of pointer-based attacks such as buffer overflows and return-oriented programming.

Capability domain switching in the kernel Support for a capability-aware kernel is extended to include support for fine-grained, capability-based compartmentalization within the kernel itself. This in effect implements a microkernel-like model in which components of the kernel, such as filesystems, network processing, etc., have only limited access to the overall kernel environment delegated using capabilities. This model protects against complex threats such as software supply-chain attacks against portions of the kernel source code or compiled kernel modules.

Capability-aware system-call interface Regardless of the kernel code generation model, it is possible to add a new system-call Application Binary Interface (ABI) that replaces conventional pointers with capabilities. This has dual benefits for both userspace and kernel safety. For userspace, the benefit is that system calls operating on its behalf will conform to memory-protection policies associated with capabilities passed to the kernel. For example, the `read` system call will not be able to overflow a buffer on the userspace stack as a result of an arithmetic error. For the kernel, referring to userspace memory only through capabilities prevents a variety of *confused deputy problems* in which kernel bugs in validating userspace arguments could permit the kernel to access kernel memory when userspace access is intended, perhaps reading or overwriting security-critical data. The capability-aware ABI would affect a variety of user-kernel interactions beyond system calls, including ELF auxiliary arguments during program startup, signal handling, and so on, and resemble other pointer-compatibility ABIs – such as 32-bit compatibility for 64-bit kernels.

These points in the design space revolve around hybrid use of CHERI primitives, with a continued strong role for the MMU implementing a conventional process model. It is also possible to imagine operating systems created without taking this view:

Pure-capability operating system A clean-slate operating-system design might choose to minimize or eliminate MMU use in favor of using the CHERI capability model for all protection and separation. Such a design might reasonably be considered a *single address-space system* in which capabilities are interpreted with respect to a single virtual address space (or the physical address space in MMU-free designs). All separation would be implemented in terms of the object-capability mechanism, and all memory sharing in terms of memory capability delegation. If the MMU is retained, it might be used simply for full-system virtualization (a task for which it is well suited), or also support mechanisms such as paging and revocation within the shared address space.

2.5 Protection Against Microarchitectural Side-Channels

While CHERI has been designed as an architectural security mechanism – i.e., one concerned with explicit access to memory contents or control of system functions – recent publication of highly effective attacks against microarchitectural side channels has caused us to reconsider CHERI's potential role [61]. Several of these attacks (e.g., Spectre variants) rely on overly optimistic speculative execution of paths that violate invariants embedded in the executing code. For example, code may contain explicit bounds checks, but by suitably training a branch predictor, an attacker can cause the code to bypass those checks in speculative execution, which then leaves behind a measurable result in the instruction or data cache. CHERI offers new opportunities to bound speculative execution such that it observes security properties otherwise not explicitly available to the microarchitecture. Possible bounds on speculative execution grounded in CHERI features include:

- Enforcing capability tag checks in speculation, preventing code or data pointers without valid provenance from being used.
- Enforcing capability bounds checks in speculation, preventing any out-of-bounds memory accesses for data load/store or instruction fetch.
- Enforcing capability permission checks in speculation, preventing inappropriate loads or stores or instruction fetch.
- Enforcing other capability protections, such as being sealed, to ensure encapsulation is implemented in speculation.
- Limiting data-value speculation for capability values, or for values that will be combined with capabilities (e.g., integer values that are added to a capability offset to calculate a new capability).
- Limiting speculation across protection-domain boundary transitions.

In addition, we have extended CHERI with new instructions to get and set a software-defined *compartment ID* (CID). Unlike with conventional MMU-based virtual address spaces that have specific address-space identifiers or page-table roots identifying protection domains, CHERI protection domains are emergent from the dynamic delegation of capabilities. The

CID might be used by microarchitectures to limit speculation of sharing of microarchitectural state. For example, branch-predictor entries may be tagged with a CID to prevent them from being used with the wrong compartment. This would necessarily need to be combined with an address-space identifier (ASID), as addresses (and hence corresponding capabilities) may have different interpretations in different address spaces.

As with other CHERI features, CID management is authorized using a capability, allowing regions of CIDs to be delegated to domains or switchers for their own selective use. Where strong side-channel-free confidentiality is not required between a set of domains, the CID may be left as-is. Otherwise, a suitably authorized software domain switcher will be able to set the CID to a new value.

Protective effects rely, of course, on appropriate implementation in the microarchitecture. Further notes on our thoughts on CHERI and microarchitectural side channels may be found in our technical report, *Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks* [147].

Chapter 3

Mapping CHERI Protection into Architecture

In this chapter, we explore architecture-neutral aspects of the mapping from the abstract CHERI protection model into Instruction-Set Architectures (ISAs). We consider the high-level architectural goals in mappings and the implications of our specific capability-system model before turning to the concrete definitions associated with CHERI's architectural capabilities, register files, tagged memory, and its composition with various existing architectural features such as exception handling and virtual memory.

We conclude with a consideration of "deep" versus "surface" design choices: where there is freedom to make different choices in instantiating the CHERI model in a specific ISA, with an eye towards both the adaptation design space and also applications to further non-MIPS ISAs, and where divergence might lead to protection inconsistency across architectures. These topics are revisited in greater detail in Chapters 4 (CHERI-MIPS), 5 (CHERI-RISC-V), and 6 (CHERI-x86-64), addressing specify emerging CHERI variants (or, in the case of CHERI-x86-64, a conceptual variant).

3.1 High-Level Architectural Goals

In addition to the broad abstract goal of supporting pointer-centric protection with strong compatibility and performance objectives, we have pursued the following architectural goals in integrating CHERI into contemporary instruction-set architectures:

1. When mapping the CHERI model into RISC architectures, CHERI's extensions should subscribe to the RISC design philosophy: a load-store instruction set intended to be targeted by compilers, with more complex instructions motivated by quantitative analysis. While current page-table structures (or in the case of MIPS, simply TLB mechanisms) are retained for functionality and compatibility, new table-oriented structures are avoided in describing new security primitives. In general, instructions that do not access memory or trigger an exception should be single-cycle register-to-register operations.

2. New primitives, such as tagged memory and capabilities, are aligned closely with current microarchitectural designs (e.g., as relates to register files, pipelined and superscalar pro-

cessors, memory subsystems, and buses), offering minimal disruption necessary to offer substantial semantic and performance improvements that would be difficult to support with current architectures. Where current de-facto approaches to microarchitecture must be changed to support CHERI – such as through the adoption of architectural tagged memory – there are efficient implementations.

3. CHERI composes sensibly with MMU-based memory protection: current MMU-based operating systems should run unmodified on CHERI designs, and as CHERI support is introduced in an MMU-based operating system, it should compose naturally while allowing both capability-aware and legacy programs to run side-by-side. This allows software designers to view the system as a set of more conventional virtual address spaces within which CHERI offers protection – or as a single-address-space system environment as use of the MMU is minimized.

4. As protection pressure shifts from conventional MMU-based techniques to reference-oriented protection using CHERI capabilities, page-table efficiency increases as larger page sizes cease to penalize protection.

5. Utilization of protection primitives is common-case, not exceptional, occurring in performance-centric code paths such as stack and heap allocation, on pointer arithmetic, and on pointer load and store, rather than being an infrequent higher-cost activity that can be amortized.

6. The principles of least privilege and intentional use dictate a number of aspects of CHERI ISA design, including requiring that no confusion arise between the use of capabilities as pointers versus integers as pointers. Load, store, and jump instructions will never automatically select semantics based on presence of a tag – for example, to avoid opportunities accidental use of the wrong right (e.g., by virtue of a capability tag being cleared due to an exploitable software vulnerability leading to its interpretation as an integer virtual address). Similarly, associative lookups of capabilities are entirely avoided.

   Trade-offs around this design goal inevitably exist. For example, to run unmodified software, CHERI provides a Default Data Capability that is transparently dereferenced when legacy integer-pointer-based code accesses memory, which we deem necessary for compatibility reasons. Similarly, we do not currently choose to provide granular control over the use of ring-based processor privilege, in order to avoid the complexity and disruption of implementing entirely new interfaces for interrupt and MMU management, using a single permission on code capabilities rather than a broad set of possible capabilities representing different privileges. A purer (non-hybridized) capability-system design would avoid these design choices.

7. Just as C-language pointers map cleanly and efficiently into integers today, pointers must similarly map cleanly, efficiently, and vastly more robustly, into capabilities. This should apply both to language-visible data and code pointers, but also pointers used in implementing language features, such as references to C++ vtables, return addresses, etc.

8. Flexibility exists to employ only legacy integer pointers or capabilities as dictated by software design and code generation, trading off compatibility, protection, and performance – while ensuring that security properties are consistently enforced and can be reasoned about cleanly.

9. When used to implement isolation and controlled communication in support of compartmentalization, CHERI's communication primitives scale with the actual data footprint (i.e., the working set of the application). Among other things, this implies that communication should not require memory copying costs that grow with data size, nor trigger TLB aliasing that increases costs as the degree of sharing increases. Our performance goal is to support at least two orders of magnitude more active protection domains per core than current MMU-based systems support (going from tens or hundreds to at least tens of thousands of domains), and similarly to reduce effective domain-crossing cost by at least two orders of magnitude.

10. When sharing memory or object references between protection domains, programmers should see a unified namespace connoting efficient and comprehensible delegation.

11. When implementing efficient protection-domain switching, the architecture supports a broad range of software-defined policies, calling conventions, and memory models. Where possible, software TCB paths should be avoided – but where necessary for semantic flexibility, they should be supported safely and efficiently. As with MMU-based protection-domain representation and crossing, CHERI supports both synchronous and asynchronous communication patterns.

12. Where possible, we make use of provable, deterministic protection, avoiding probabilistic techniques. For example, we avoid the use of cryptographic hashes that must be truncated to small numbers of bits within a pointer or capability, instead making use of tagging. This not only avoids brute-force attempts against short hashes, but also allows stronger non-reinjection properties: pointers leaked via network communications or IPC cannot be reinjected, despite having previously been valid. This in turn allows stronger temporal safety properties to be enforced by software, due to having stronger guarantees. Provability is an essential aspect to our work: CHERI's architectural safety properties must be formally expressible, and mechanically provable from that expression.

13. More generally, we seek to exploit hardware performance gains wherever possible: in eliminating repeated software-generated checks by providing richer semantics, in providing stronger underlying atomicity for pointer integrity protection that would be very difficult to provide on current architectures, and in providing more scalable models for memory sharing between mutually distrusting software components. By making these operations more efficient, we encourage their more extensive use.

These and other design goals permeate CHERI's abstract architecture-neutral design as well as its architecture-specific instantiations.

3.2 Capability-System Model

In CHERI, capabilities are unforgeable tokens of authority through which programs access all memory and services within an address space. Capabilities are a fundamental hardware type that may be held in registers (where they can be inspected, manipulated, and dereferenced using capability instructions), or in memory (where their integrity is protected). They include an integer virtual address, bounds, permissions, and other protective metadata including an object type and one-bit tag.

Capability permissions determine what operations (if any) are available via the architecture. Commonly used permissions include those authorizing memory loads, memory stores, and instruction fetches. Where permissions authorize memory access, *capability bounds* limit the range of addresses that may be accessed; for other permissions, bounds constrain other forms of access (e.g., use of the object-type space). Memory capabilities (those authorizing memory access) may be used to load other capabilities into registers for use. Capabilities may also be sealed in order to make their fields immutable and the capability non-dereferenceable.

While motivated by the goal of representing pointers (protected virtual addresses), they are also able to protect non-pointer values. For example, *sealed capabilities* without memory-access permissions may be used to represent references to protection domains that can be transitioned to via software-defined object invocation.

Unforgeability is implemented by two means: tag bits and guarded manipulation. Each capability register (and each capability-aligned physical memory location) is associated with a tag bit indicating that a capability is valid. Attempts to directly overwrite a capability in memory using data (rather than capability) stores automatically clears the tag bit. When data is loaded into a register, its tag bit is also loaded; while data without a valid tag can be loaded into a register, attempts to dereference or invoke such a register will trigger an exception.

Guarded manipulation is enforced by virtue of the ISA: instructions that manipulate capability register fields (e.g., base, offset, length, permissions, type) are not able to increase the rights associated with a capability. Similarly, sealed capabilities can be unsealed only via the invocation mechanism, or via the unseal instruction subject to similar monotonicity rules. This enforces encapsulation, and prevents unauthorized access to the internal state of objects.

Collectively, unforgeability and guarded manipulation ensure that dereferenceable capabilities (those with their tag set) have *valid provenance*: they are derived only from other valid capabilities, and only through valid manipulations. All other capabilities will not have their tag set, hence cannot be dereferenced.

Intentionality avoids the automatic selection of a capability from among a set in order to locate rights to authorize a requested operation. It is always clear for every instruction what capability will authorize its action, e.g., whether for the executing code capability (to authorize privileged ISA operations such as MMU management), explicit operand capabilities (to query, modify, or dereference), or implicit use of the Default Data Capability (e.g., when constraining legacy load and store instructions). There are no associative lookups of capabilities to select from among several options, and instructions are always clearly defined as expecting an integer or a tagged capability as an operand, failing if that expectation is not met.

We anticipate that many languages will expose capabilities to the programmer via pointers or references – e.g., as qualified pointers in C, or mapped from object references in Java.

Similarly, capabilities may be used to bridge communication between different languages more safely – for example, by imposing Java memory-protection and security properties on native code compiled against the Java Native Interface (JNI). In general, we expect that languages will not expose registers directly for management by programmers, instead using them for instruction operands and as a cache of active values, as is the case for integer pointers today. On the other hand, we expect that there will be some programmers using the equivalent of assembly-language operations, and the CHERI compartmentalization model does not place trust in compiler correctness for non-TCB code.

3.3 Architectural Capabilities

CHERI capabilities are an architectural data type, directly implemented by the CPU hardware in a manner similar to integers or floating-point values. Capabilities may be held in registers or in tagged memory. On RISC ("load-store") architectures, CHERI-aware code can use new capability instructions to inspect, manipulate, and dereference capabilities held in registers. On CISC architectures, direct use of capabilities in memory may also be possible. In-register modification of capability values is subject to guarded manipulation (e.g., to enforce monotonicity), and dereference is subject to appropriate checks (e.g., for a valid tag, sealing, appropriate permissions, and suitable bounds). In-memory modification of capability values is protected by tagged memory.

3.3.1 Capability Contents

Capabilities contain a number of software-accessible architectural fields, which may differ in content and size from the microarchitectural implementation or in-memory representation:

- Tag bit ("**tag**", 1 bit "out of band" from addressable memory)
- Permissions mask ("**perms**", parameterizable size)
- Software-defined permissions mask ("**uperms**", parameterizable size)
- Flags ("**flags**", parameterizable size)
- Object type ("**otype**", 64 bits)
- Offset ("**offset**", 64 bits)
- Base virtual address ("**base**", 64 bits)
- Length in bytes ("**length**", 64 bits)

Bit	Name	Tag?	Seal?	Bounds?
0	Global	✓	-	-
1	Permit_Execute	✓	Unsealed	Address
2	Permit_Load	✓	Unsealed	Address
3	Permit_Store	✓	Unsealed	Address
4	Permit_Load_Capability	✓	Unsealed	-
5	Permit_Store_Capability	✓	Unsealed	-
6	Permit_Store_Local_Capability	✓	Unsealed	-
7	Permit_Seal	✓	Unsealed	Object Type
8	Permit_CCall	✓	Sealed	-
9	Permit_Unseal	✓	Unsealed	Object Type
10	Access_System_Registers	✓	Unsealed	-
11	Permit_Set_CID	✓	Unsealed	CID

Table 3.1: Architectural permission bits for the **perms** capability field, along with checks usually used alongside that permission: *Tag?* Require a valid tag; *Seal?* Require the capability to be sealed or unsealed; *Bounds?* Perform a bounds check authorizing access to the listed namespace. See the instruction-set reference for detailed per-instruction requirements.

Tag Bit

The **tag** bit indicates whether an in-register capability or a capability-sized, capability-aligned location in physical memory contains a valid capability. If **tag** is set, the capability is valid and can be dereferenced (subject to other checks). If **tag** is clear, the remainder contains 256 (or 128) bits of normal data; an untagged capability cannot be dereferenced. If capabilities are held in dedicated registers, those registers must still have tag bits in order to allow untagged data to move through those registers – e.g., to implement capability-oblivious memory-copy and sort operations. Section 3.4.2 describes the behavior of tagged memory.

Permission Bits

The **perms** bit vector governs the architecturally defined permissions of the capability including read, write, and execute permissions. Bits 0–11 of this field, which control use and propagation of the capability, and also limit access to privileged instructions, are defined in Table 3.1. Permissions grant access only subject to constraints imposed by the current architectural ring – that is, they always restrict relative to the existing architectural security model. Permissions are also contingent on the capability **tag** bit being set, and specific permissions may depend on the capability being sealed (or unsealed), or bounds checks against **base** and **length**, when used:

Global Allow this capability to be stored via capabilities that do not themselves have Permit_Store_Local_Capability set.

Permit_Execute Allow this capability to be used in the **PCC** register as a capability for the program counter, constraining control flow.

Permit_Load Allow this capability to be used to load untagged data; also requires Permit_Load_Capability to permit loading a tagged value.

Permit_Store Allow this capability to be used to store untagged data; also requires Permit_Store_Capability to permit storing a tagged value.

Permit_Load_Capability Allow this capability to be used to load capabilities with valid tags; Permit_Load is also required.

Permit_Store_Capability Allow this capability to be used to store capabilities with valid tags; the permission Permit_Store is also required.

Permit_Store_Local_Capability Allow this capability to be used to store non-global capabilities; also requires Permit_Store and Permit_Store_Capability.

Permit_Seal Allow this capability to authorize the sealing of another capability with a **otype** equal to this capability's **base** + **offset**.

Permit_CCall Allow this sealed capability to be used with a "direct" `CCall` (i.e., without passing through a software exception handler).

Permit_Unseal Allow this capability to be used to unseal another capability with a **otype** equal to this capability's **base** + **offset**.

Permit_Set_CID Allow the architectural compartment ID to be set to this capability's **base** + **offset** using `CSetCID`.

In general, permissions on a capability relate to its implicit or explicit use in authorizing an operation that uses the capability – e.g., in fetching an instruction via **PCC**, branching to a code capability, loading or storing data via a capability, loading or storing a capability via a capability, performing sealing or unsealing operations, or controlling capability propagation. In addition, a further *privileged permission* controls access to privileged aspects of the instruction set such as exception-handling, which are key to the security of the model and yet do not fit the "capability as an operand" model:

Access_System_Registers Allows access to privileged processor permitted by the architecture (e.g., by virtue of being in supervisor mode), with architecture-specific implications. This bit limits access to features such as MMU manipulation, interrupt management, processor reset, and so on. The operating system can remove this permission to implement constrained compartments within the kernel.

A richer conversion to a capability architecture might replace existing privileged instructions (e.g., to flush the TLB) with new instructions that accept an authorizing capability as an operand, and adopt a more granular model for authorizing architectural privileges using capabilities than this all-or-nothing approach.

The Permit_Store_Local_Capability permission bit is used to limit capability propagation via software-defined policies: local capabilities (i.e., those without the Global permission set)

otype value	Interpretation
$2^{64} - 1$	Unsealed capability
$2^{64} - 2$	Reserved (experimental "enter capabilities"; appendix D.12)
$2^{64} - 3$	Reserved (experimental "memory type tokens"; appendix D.14)
$2^{64} - 4$ through $2^{64} - 16$	Reserved
other	Capability sealed by `CSeal`

Table 3.2: Object types and their architecture-specified roles.

can be stored only via capabilities that have Permit_Store_Local_Capability set. Normally, this permission will be set only on capabilities that, themselves, have the Global bit cleared. This allows higher-level, software-defined policies, such as "Disallow storing stack references to heap memory" or "Disallow passing local capabilities via cross-domain procedure calls," to be implemented. We anticipate both generalizing and extending this model in the future in order to support more complex policies – e.g., relating to the propagation of garbage-collected pointers, or pointers to volatile vs. non-volatile memory.

Software-Defined Permission Bits

The **uperms** bit vector may be used by the kernel or application programs for software-defined permissions. They can be masked and retrieved using the same `CAndPerm` and `CGetPerm` instructions that operate on hardware-defined permissions, and also checked using the `CCheckPerm` instruction. When using 256-bit capabilities, 16 software-defined permission bits are available; with 128-bit capabilities, 4 software-defined permission bits are available.

Software-defined permission bits can be used in combination with existing hardware-defined permissions (e.g., to annotate code or data capabilities with further software-defined rights), or in isolation of them (with all hardware-defined permissions cleared, giving the capability only software-defined functionality). For example, software-defined permissions on code capabilities could be employed by a userspace runtime to allow the kernel to determine whether a particular piece of user code is authorized to perform system calls. Similarly, user permissions on sealed data capabilities might authorize use of specific methods (or sets of methods) on object capabilities, allowing different references to objects to authorize different software-defined behaviors. By clearing all hardware-defined permissions, software-defined capabilities might be used as unforgeable tokens authorizing use of in-application or kernel services.

Flags

The **flags** field can be read with the `CGetFlags` instruction and written with the `CSetFlags` instruction.

There are no architecture-neutral flags currently defined, therefore the size and interpretation of this field are entirely architecture specific.

Object Type

The 64-bit **otype** field indicates whether a capability is sealed and, if so, what “type” it has; see table 3.2 for defined values. CHERI uses multiple object types to allow software to create unforgeable associations between sealed capabilities.[1] If a capability is sealed, it becomes non-dereferenceable (i.e., cannot be used for load, store, or instruction fetch) and immutable (i.e., whose fields cannot be manipulated). Capability unsealing is mediated either by capabilities (via the `CUnseal` instruction) or by control transfers (via the `CCall` instruction); see section 4.10). One potential application of sealed capabilities is for use as object-capability references – i.e., as references to software-defined objects with architecturally enforced encapsulation. However, they are available to software for more general use in constructing architecturally protected references.

While defined as a 64-bit space, the object types available to an *implementation* of CHERI may be a smaller space. If so, the implementation values in **otype** fields are translated to the abstract space as if by sign extension. Attempts to seal capabilities to types that cannot be expressed by the implementation will fail in an implementation-specified way, but generally similarly to any other representability failure.

Base

The 64-bit **base** field is the base virtual address of the segment described by a capability. The **base** field is the *lower bound* of the capability: dereferencing an effective virtual address below **base** will throw an exception. In the presence of compressed capabilities, not all possible 64-bit values of **base** will be representable (see Section 3.4.4).

Offset

The 64-bit **offset** field holds a free-floating pointer that will be added to the base when dereferencing a capability. The value can float outside of the range described by the capability – e.g., as a result of using `CSetOffset` to set the offset to a negative value, or to a value greater than **length** – but an exception will be thrown if a requested dereference is out of range. A non-zero offset may be used when a language-level pointer refers to a location within a memory allocation or data structure; for example, to point into the middle of a string, or at a non-zero index within an array. A non-zero offset may also be used when the lower bound of a memory allocation is insufficiently aligned to permit precise description with the **base** field of a compressed capability (see Section 3.4.4).

Virtual Address

The virtual address, or **cursor**, of a capability is the sum of its **base** and **offset** fields. The components of the virtual address may be accessed separately (e.g., via `CGetOffset`), or as a

[1] While earlier versions of the CHERI-MIPS ISA interpreted this field as an address, recent versions treat this as a software-managed value without architectural interpretation. The width of the object type field is defined by the architectural implementation and may be less than the 64 bits suggested here; in such cases, the implementation field must be the least significant bits of the architectural **otype** and should be *sign extended* upon interpretation.

single combined entity (e.g., via CGetAddr and CSetAddr) depending on the software use case. For example, an integer cast of a C-language pointer might return either the offset or the virtual address, depending on the C-language interpretation being used.

Length

The 64-bit **length** field is the length of the segment described by a capability. The sum of **base** and **length** is the *upper bound* of the capability: accessing at or above **base** + **length** will throw an exception. In the presence of compressed capabilities, not all possible 64-bit values of **length** will be representable (see Section 3.4.4).

3.3.2 Capability Values

Pointer Values in Capabilities

In general, C and C++-language pointers are suitable to be represented as memory capabilities (i.e., those that are unsealed and have a memory interpretation by virtue of memory-related permissions). This includes both data pointers, which may have enabled permissions that include Permit_Load, Permit_Store, Permit_Load_Capability, and Permit_Store_Capability, and code pointers, which may have enabled permissions that include Permit_Load, Permit_Execute, and Permit_Load_Capability. Other permissions, such as Global or Permit_CCall, may also be present. The following architectural values will normally be used:

- The **tag** is set.
- The capability is unsealed (has **otype** of $2^{64} - 1$).
- **perms** contains a suitable combination of load, store, and execute permissions, as well as other possible permissions.
- **base** will point to the bottom of the memory allocation, allowing for suitable alignment if bounds compression is used.
- **offset** will point within the memory allocation (but may point outside in some circumstances).
- The virtual address will be equal to the integer value of the pointer.
- **length** will be the length of the memory allocation, allowing for suitable alignment if bounds compression is used.

Code pointers will normally include Permit_Load and Permit_Load_Capability so that constant islands and global variables can be accessed via the code segment. In the presence of bounds compression (i.e., with 128-bit capabilities), the memory allocation may require stronger alignment so as to ensure precise bounds. Implied pointers in the run-time environment, originating in compiler-generated code or the run-time linker, such as Program Linkage Table (PLT) entries, Global Offset Table (GOT) entries, the Thread-Local Storage (TLS) pointer, C++ v-table pointers, and return addresses, will typically have similar values. Note that the **flags** field may have an architecture-specific default value.

The NULL Capability

When representing C-language pointers as capabilities, it is important to have a definition of NULL with as close-as-possible semantics to today's definition that NULL has an integer value of 0. We choose to define a NULL capability that has the following architecture values set:

- **tag** is cleared.
- The capability is unsealed (has **otype** of $2^{64} - 1$).
- **perms** is 0x0.
- **flags** is 0x0.
- **base** is 0x0.
- **offset** is 0x0.
- By implication, the virtual address of the capability is 0x0.
- **length** is the largest permitted length ($2^{64} - 1$ on 256-bit CHERI, and 2^{64} on 128-bit CHERI).

The NULL capability is used in several places in the architecture, including being the value returned when **C0** is used as an operand to many (but not all) instructions, when a 0 integer value is passed to CToPTR, and when comparisons with NULL are performed by the CBEZ and CBNZ instructions.

3.3.3 Integer Values in Capabilities

In the C language, the intptr_t type is intended to be an integer type large enough to hold a pointer, and sees two common uses: an opaque field that can hold either an integer or pointer type; or an integer type permitting arithmetic and other integer operations on pointer values. We find it convenient to store an integer value in a capability using the following conventions:

- **tag** is cleared.
- The capability is unsealed (has **otype** of $2^{64} - 1$).
- **perms** is 0x0.
- **flags** is 0x0.
- **base** is 0x0.
- **offset** is the integer value to be stored.
- By implication, the virtual address of the capability is the integer value to be stored.
- **length** is the largest permitted length ($2^{64} - 1$ on 256-bit CHERI, and 2^{64} on 128-bit CHERI).

3.3.4 General-Purpose Capability Registers

General-purpose capability registers are registers that are able to load, store, inspect, manipulate, and dereference capabilities while preserving their 1-bit tag and full set of structured fields. New capability-aware instructions (see Section 3.6) allow use of new registers or new fields added to existing registers, and via guarded manipulation must implement properties such as tag preservation, monotonic transformation, and so on. Capability registers are tagged so that capability-oblivious operations – such as tag-preserving memory copies of regions containing both data and capabilities – can be performed, preserving both set and unset tag bits. This means that all capability-aware instructions dereferencing a capability must check for a valid tag, as capability registers may contain data values that are not permitted to be dereferenced.

Architectures may be extended with capability registers in two ways: first, by introducing new architectural registers that supplement existing registers, or second, by extending existing registers with new capability-register fields that can be accessed through new instructions. These two approaches may be combined in the same architecture, as in CHERI-MIPS (see Chapter 4), where we introduce both a new capability register file supplementing the integer register file, and also extend certain special registers, as described in Section 3.3.5, to become capability registers – such as the Program Counter (**PC**) becoming the Program-Counter Capability (**PCC**).

An alternative approach would extend the existing general-purpose integer register file to allow it to hold both 64-bit integers and capabilities, with instructions selecting the desired semantics when utilizing a register. We refer to this design as a *merged register file*. This is similar to extension of 32-bit registers to 64-bit registers, in which 32-bit load, store, and manipulation can take place despite the full register size being large enough to hold a 64-bit value. A similar set of constraints applies: when an integer is loaded into a capability-width register, the tag bit and remainder of the non-integer data bits in the register must be zeroed, in similar manner to the use of zero or sign extension when loading a smaller integer into a larger integer register. When a register containing a tagged capability is used as an input to an integer arithmetic operation, we recommend that the virtual address of a capability be used as the integer value used for input.

When integer and capability instructions share a common underlying register file, it is essential that intentionality be maintained: instructions must not select between integer and capability interpretations based on the tag value. Instead, instructions must specifically interpret input and output registers as integers or as capabilities. If a capability dereference is expected, an exception must be thrown if the input register does not contain a valid tag. If an integer dereference is to be performed, only the integer portion of the capability register will be used (per above, the virtual address of the capability), and it will be indirected through an appropriate implied capability such as the Program-Counter Capability (**PCC**) or Default Data Capability (**DDC**).

When utilizing a merged register file, not all integer registers may be extended to hold capabilities. A tradeoff exists around the extension of existing well-supported ABIs, such as the calling convention, vs. the impact of register-file growth and opcode utilization. Larger numbers of capability registers will increase the memory footprint of context switching and the cost of stack spillage (where a callee cannot know whether a register requires saving as a full capability or whether integer width would be sufficient). Similarly, larger numbers of available capability registers increase the opcode footprint of capability-relative instructions. While this opcode space is no greater than for integer-relative instructions, in some architectures (e.g., ARMv8), opcode space is at a substantial premium, and adding new capability variants of all load/store/jump instructions will over-consume or exhaust the space. Reducing the number of capability registers comes at other costs, such as potentially disrupting current ABI design choices, and increasing register pressure for pointer-intensive workloads. Here, a variety of design points are available, but one option would be to limit capabilities to a subset of the full register file, allowing a smaller number of bits to name the available capability registers. This pressure is especially acute in variable-size instruction sets (e.g., with the RISC-V compressed instruction set). Other options to avoid this pressure include the introduction of new opcode modes in which existing opcodes can be reused to refer to capabilities instead of integers, at a cost to binary compatibility. The most straightforward choice, where opcode space is plentiful with respect to the vocabulary of load-store instructions, is to allow all existing general-purpose integer registers to hold capabilities.

Microarchitectural and in-memory representations of capabilities may differ substantially from the architectural representation in terms of size and contents, but these differences will not be exposed via instructions operating on capability-register fields. We define two variants with 256-bit and 128-bit in-memory representations of a conceptual 256-bit capability register, with the latter employing capability compression (Section 3.4.4) to reduce the register-file and memory footprint.

Register-File Implications for Integer Values in Capabilities

The convention for storing integers in capabilities, described in Section 3.3.3, is not currently defined in architecture when using a split register file. However, it is convenient that:

- Adding an integer value to the offset of a NULL capability (e.g., using `CIncOffset`) gives a capability that follows these conventions.
- Maximal bounds allow the virtual address to take on any value without risking a bounds representability failure during arithmetic – in contrast to using a maximum length of 0, which might otherwise seem intuitive.

With a merged register file, this is instead an architectural definition, and is used when an integer value is moved or loaded into a capability register. This might occur during an ordinary data load into a register, or as a result of (for example) an integer arithmetic operation writing back to a register. Sign extension will occur as normal for the architecture to fill the offset field, with remaining fields being set to the above values.

3.3.5 Special Capability Registers

In addition to the general-purpose capability registers available for use via capability load, store, jump, query, and manipulation instructions, there are also a set of *Special Capability Registers* (SCRs). These capability registers are accessed via special get and set instructions (and, in some architectures, swap or direct manipulation), and serve specific architectural functions. Access to special capability registers is controlled on a case-by-case basis (`CReadHwr` and

CWriteHwr), but may include universal read and write access, read and write access only when holding Permit_Access_System_Registers, or based on execution ring or exception-handling state. The specific registers vary by underlying architecture, but will include the following (or natural variations of them):

Program Counter Capability (PCC) extends the existing Program Counter (**PC**) to be a full capability, imposing validity, permission, bounds, and other checks on instruction fetch.

Default Data Capability (DDC) indirects legacy non-capability loads and stores, controlling and relocating data accesses to memory.

Exception Program Counter Capability (EPCC) Just as conventional architectures save the **PC** in **EPC** following an exception, and restore **EPC** into **PC** on exception return, **EPCC** extends **EPC** to hold a copy of the full saved **PCC**.

Kernel Code Capability (KCC) When an exception is taken, the value in **PCC** is replaced with the value in **KCC**, installing a suitable execution and security context for the exception handler. Note: A better name for this capability might be the Exception Code Capability.

Kernel Data Capability (KDC) When an exception is taken, **KDC** holds a suitable data capability for use by the exception handler. Note: A better name for this capability might be the Exception Data Capability.

User Thread-Local Storage (CULR) A capability extended version of a Thread-Local Storage (TLS) register, available to any executing code.

Privileged Thread-Local Storage (CPLR) A capability register intended to be used only by privileged code within a ring to implement Thread-Local Storage (TLS).

Although these capability special registers may be viewed as extensions to existing special registers (e.g., **EPC**), CHERI introduces new capability-based instructions to get and set their values, rather than conflating them with existing integer-based special-register instructions in the architecture ISA, in order to ensure intentional use in the presence of a merged register file.

Where existing special registers, such as the Program Counter (**PC**) or Exception Program Counter (**EPC**), are extended to become capabilities, the semantics of accessing the integer interpretation must be determined with care. Unlike with the general-purpose integer register file, it may be desirable for reasons of compatibility to modify the capability while retaining its tag and other metadata (such as bounds and permissions) without modification – subject to maintaining monotonicity. For example, when modifying **PC**, it is desirable to leave other fields (such as bounds of **PCC**) unmodified, and further to have integer accesses be performed on **offset** rather than on the capability virtual address, so that capability-unaware code can jump within its code segment without experiencing a tag violation or being exposed to absolute virtual addresses. A similar argument applies to **EPC**, where capability-unaware exception-handling code may be able to continue to operate. These design choices allow accesses to be relocated relative to each of these capabilities.[2] In the case of **EPC**, it is desirable that attempted writes be considered equivalent to `CSetOffset`, but with failures – such as might occur if a sealed capability has been placed in **EPCC** – leading to the tag being cleared rather than an exception being through in what is likely to already be an exception-handling context.

3.4 Capabilities in Memory

Maintaining the integrity and provenance validity of capabilities stored to, and later read from, memory, is an essential feature of the CHERI architecture. Capabilities may be stored to memory in a broad variety of circumstances, including, when language-level pointers are implemented using capabilities, operating-system context switching, stack spills of capability registers, stack storage for local pointer variables, pushing return capabilities to the stack on function call, the capabilities held in Global Offset Table (GOT) structures to reach global variables, global variables themselves holding types implemented via capabilities, Procedure Linkage Table (PLT) entries holding code capabilities that can be jumped to, and so on. As tagged memory maintains tag bits at capability-sized, capability-aligned intervals, stores of capabilities to memory will retain their tags only if at suitable alignment. This allows capabilities to be held at any suitably aligned memory location, interleaved arbitrarily with other data – such as is commonly the case with pointers and other data today.

3.4.1 In-Memory Representation

As implemented in CHERI-MIPS and CHERI-RISC-V, all in-memory capability bits are directly addressable via ordinary data accesses (e.g., byte loads) except for the tag bit, which is stored "out-of-band" as a 129th or 257th bit. The in-memory capability representation will typically not be a direct mapping of architectural capability fields into memory, as fields may be stored as partially computed values to improve performance (e.g., storing a virtual address rather than base and offset), to reduce size (e.g., through bounds compression), or to utilize multiple formats (e.g., for unsealed vs. sealed capabilities). Given the prior definitions, we impose several constraints on the in-memory representation:

The NULL capability has an in-memory representation of all zero bytes and a cleared tag. This definition allows zero-filled memory to be interpreted as NULL-filled memory when loaded as a capability, providing greater consistency with the C-language expectations for NULL pointers[3].

The bottom 64 bits of a capability hold its virtual address. Supporting casts between a capability and an ordinary integer type sized to correspond to the size of a virtual address has significant utility in practical C code.

[2] This is a design point on which we have had considerable discussion, and for which other approaches would also be reasonable. For example, a virtual-address interpretation of **PC** or **EPC** would also be meaningful, but would place greater constraints on how capabilities were used to constrain access by unmodified software.

[3] This design choice has a number of implications, including that the architectural length cannot be stored untransformed for 256-bit capabilities – it might instead need to be the XOR of the length, and that as NULL pointers do not have tags set, they cannot be differentiated from zeroes in memory.

A 256-bit format is described in Section 3.4.3. A 128-bit compressed format is described in Section 3.4.4. These formats vary in terms of the number of permission bits they offer, and also bounds precision effects stemming from capability compression. Concrete architectures may additionally allocate bits for the **flags** field.

Software authors are discouraged from directly interpreting the in-memory capability representation to improve the chances of software portability (e.g., across architectures) and forward compatibility (e.g., with respect to newly added permissions or other changes in field behavior). This also allows multi-endian architectures or heterogenous designs to utilize a single endianness for in-memory capability storage (e.g., little endian) to avoid ambiguities in which the same in-memory bit pattern might otherwise describe two different sets of rights depending on where it is loaded and interpreted. This is also important given the desire to be able to retrieve the virtual address or integer value of an in-memory capability by loading from the bottom 64 bits of the capability.

Despite the software benefits from avoiding encoding the in-memory capability representation, it is important that the in-memory representation be considered architectural (i.e., having a defined and externally consistent representation) to better support systems software functions such as swap, core dumps, debuggers, virtual-machine migration, and efficient run-time linking, which may embed that representation within file formats or network protocols.

3.4.2 Tagged Memory

CHERI relies on tagged physical memory: the association of a 1-bit *tag* with each capability-sized, capability-aligned location in physical memory. Associating tags with physical memory ensures that if memory is mapped at multiple virtual addresses, the same tags will be loaded and stored regardless of the virtual address through which it is accessed. Tags must be atomically bound to the data they protect. As a result, it is expected that tags will be cached with the memory they describe within the cache hierarchy.

When a capability-sized value in a capability register is written to a capability-aligned area of memory using a capability store instruction, and the capability via which the store takes place has suitable permissions, the tag bit on the capability register will be stored atomically in memory with the capability value. Other stores of untagged capability values or other types (e.g., bytes, half words, words, floats, doubles, and double words) across one or more capability-aligned locations in memory will atomically clear the corresponding tag bits for that memory.

When a capability-sized value is loaded into a capability register from a capability-aligned location in memory using a capability load instruction, and the capability via which the load takes place has suitable permissions, the tag associated with that memory is loaded atomically into the register along with the capability value. Otherwise, loads will clear the capability register tag bit.

Strong atomicity properties are required such that it is not possible to partially overwrite a capability value in memory while retaining the tag, or partially load a capability and have the tag bit set. These strong atomicity properties ensure that tag bits are set only on capability values that have valid provenance – i.e., that have not been corrupted due to data stores into their contents, or undergone non-monotonic transformations. Our use of atomicity, in this context, has primarily to do with the visibility of partial or interleaved results (which must

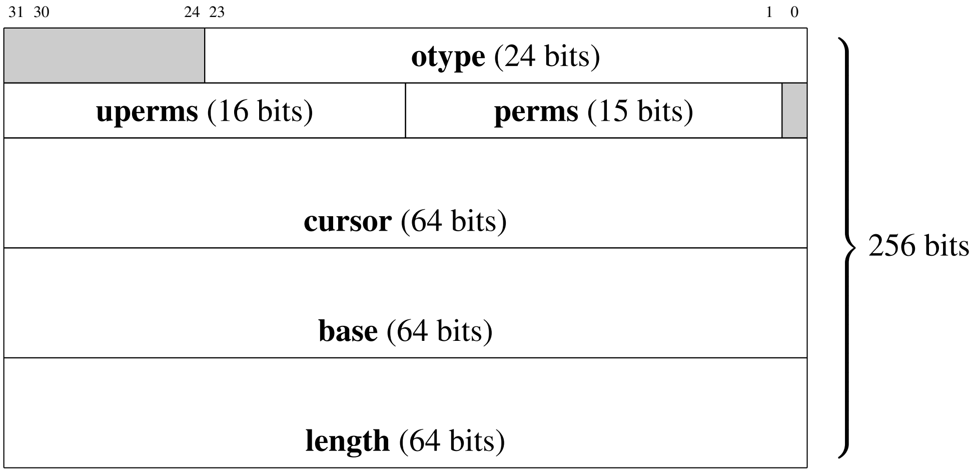

Figure 3.1: 256-bit memory representation of a capability

not occur for capability stores or tag clearing during data overwrite, or there is a risk that corrupted capabilities might be dereferenceable), rather than ordering or visibility progress guarantees (where we accept the memory model of the host architecture). This provides a set of properties that falls out naturally from current microarchitectures and coherent memory-subsystem designs: atomicity is with respect only to lines in the local cache, and not global state.

3.4.3 256-bit Capability Format

A 256-bit format for representing capabilities is shown in Figure 3.1. This is the format that is currently used by the 256-bit versions of the Bluespec implementation, the L3 formal model, and the CHERI-enabled QEMU MIPS emulator. Programs should not rely on this memory representation, as there are alternative capability representations (see, for example, the 128-bit format in Section 3.4.4), and it may change in future. Instead, programs should access the fields through the instructions provided in the ISA.

Note that there is a significant difference between the architecturally defined fields and the in-memory representation: this format implements **offset** as **cursor** − **base**, where the **cursor** field is internal to the implementation. These fields are stored in memory in a big-endian format. The CHERI processor prototype is currently always defined to be big-endian, in contrast to the traditional MIPS ISA, which allows endianness to be selected by the supervisor. This is not fundamental to our approach; rather, it is expedient for early prototyping purposes.

In this representation, **uperms** is a 16 bit field and **perms** is a 15-bit field. **otype** is taken to be the least-significant 24-bits of the architectural **otype** value and is sign-extended to the architectural value.[4]

[4] Prior versions of the architecture had a separate **s** flag within the 256-bit capability encoding, used to discriminate sealed and unsealed capabilities. In version 7 of the ISA, unsealed capabilities were redefined as having **otype** of $2^{64} - 1$ and this bit was reclaimed as reserved.

3.4.4 Compressed Capabilities

256-bit capabilities offer high levels of precision and software compatibility, but at a cost: quadrupling the size of pointers. This has significant software and micro-architectural costs to cache footprint, memory bandwidth, and also in terms of the widths of memory paths in the design. However, CHERI is designed to be largely agnostic to the in-memory representation, permitting alternative "compressed" representations while retaining largely compatible software behavior. Compression is possible because the base, length, and pointer values in capabilities are frequently redundant. For example the pointer is often within bounds and the length small, so the most significant bits of the pointer, base and upper bound are likely to be the same. This can be exploited by increasing the alignment requirements on bounds associated with a pointer (while retaining full precision for the pointer itself) and encoding the bounds relative to the pointer with limited precision. Space can further be recovered by enforcing stronger alignment requirements on sealed capabilities than for data capabilities (as unsealed capabilities have a particular object type), and by reducing the number of permission and reserved bits.

Using this approach, it is possible to usefully represent capabilities via a compressed 128-bit in-memory representation, while retaining a 256-bit architectural view. Compression results in a loss of precision, exposed as a requirement for stronger bounds alignment, for larger memory allocations. Because of the representation, we are able to vary the requirement for alignment based on the size of the allocation, and for small allocations (< 4 KiB), impose no additional alignment requirements. The design retains full monotonicity: no setting of bounds or adjustment of the pointer value can cause bounds to increase, granting further rights – but care must be taken to ensure that intended reductions in rights occur where desired. Some manipulations of pointers could lead to unrepresentable bounds (as the bounds are no longer redundant to content in the pointer): in this case, which occurs when pointers are moved substantially out of bounds, the tag will be cleared preventing further dereferencing.

For bounds imposed by memory allocators, this is not a substantial cost: heap, stack, and OS allocators already impose alignment in order to achieve natural word, pointer, page, or superpage alignment in order to allow fields to be accessed and efficient utilization of virtual-memory features in the architecture. For software authors wishing to impose narrower bounds on arbitrary subsets of larger structures, the precision effects can become visible: it is no longer possible to arbitrarily subset objects over the 4 KiB threshold without alignment adjustments to bounds. This might occur, for example, if a programmer explicitly requested small and unaligned bounds within a much larger aligned allocation – such as might be the case for video frame data within a 1 GiB memory mapping. In such cases, care must be taken to ensure that this cannot lead to buffer overflows with potential security consequences. Alignment requirements are further explored in Section 3.4.5 and Appendix E.3.4.

Different representations might be used for unsealed data capabilities versus sealed capabilities used for object-capability invocation. Data capabilities experience very high levels of precision intended to support string subsetting operations on the stack, in-memory protocol parsing, and image processing. Sealed capabilities require additional fields, such as the object type and further permissions, but because they are unused by current software, and represent coarser-grained uses of memory, greater alignment can be enforced in order to recover space for these fields. Even stronger alignment requirements could be enforced for the default data capability in order to avoid further arithmetic addition in the ordinary RISC load and store paths, where a bitwise or, rather than addition, is possible due to zeroed lower bits in strongly aligned bounds.

We specify two different 128-bit compression schemes for capabilities. CHERI Concentrate (Section 3.4.5) is our current compression format. CHERI-128 (Appendix E) is our previous compression format, and is now considered deprecated.

3.4.5 CHERI Concentrate Compression

CHERI Concentrate is a compressed capability encoding that uses a floating point representation to encode the bounds relative to the capability's address [152]. It is a development from the CHERI-128 compression format described in Appendix E. For a more detailed rational behind some of the encoding decisions see Section 8.25.

Figure 3.2 shows the capability format and decoding method for 128-bit CHERI concentrate. The format contains a 64-bit address, a, 16 permission bits (4 user defined and 12 hardware defined), an 18-bit object type and 27 bits that encode the bounds relative to the address. The following definitions are used in the description of the bounds encoding:

MW is the *mantissa width*, a parameter of the encoding that determines the precision of the bounds. For 128-bit capabilities we use $MW = 14$, but this could be adjusted depending on the number of bits available in the capability format.

B **and** T are MW-bit values that are substituted into the capability address to form the base and top. They are stored in a slightly compressed form in the encoding, in one of two formats depending on the I_E bit.

I_E is the *internal exponent* bit that selects between two formats. If the bit is set then an exponent is stored instead of the lower three bits of B and T fields (B_E and T_E), reducing the precision available by three bits. Otherwise the exponent is implied to be zero and the full width of B and T are used.

E is the 6-bit *exponent*. It determines the position at which B and T are inserted in a. Larger values allow larger regions to be encoded but impose stricter alignment restrictions on the bounds.

In more detail the base, b, and top, t, are derived from the address by substituting the MW 'middle bits' (bits E to $E + MW$) of a, a_{mid}, with B and T respectively and clearing the lower E bits. In order to allow for memory regions that span alignment boundaries and so that a can roam over a larger region while maintaining the original bounds the most significant bits of a may be adjusted up or down by one using corrections c_b and c_t which are described later.

The I_E bit selects between two cases: the $I_E = 0$ case with zero exponent for regions less than 2^{12} bytes long or the *internal exponent* case with E stored in the lower bits of T and B. In the latter case E is chosen such that the most significant non-zero bit of the length of the region aligns with $T[12]$ in the decoded top. This means that the top two bits of T can be derived from B using the equality $T = B + L$, where $L[12]$ is known from the values of I_E and E and a carry out is implied if $T[11..0] < B[11..0]$ (because we know that the top is more than the base). Storing the exponent in the lower bits of T and B means that there is less

63							0
p'16	▨	otype'18	I_E	$T[11:3]$	T_{E}'3	$B[13:3]$	B_{E}'3
a'64							

p: permissions otype: object type a: pointer address

If $I_E = 0$:

$$\begin{aligned} E &= 0 \\ T[2:0] &= T_{\text{E}} \\ B[2:0] &= B_{\text{E}} \\ L_{\text{carry\_out}} &= \begin{cases} 1, & \text{if } T[11:0] < B[11:0] \\ 0, & \text{otherwise} \end{cases} \\ L_{\text{msb}} &= 0 \end{aligned}$$

If $I_E = 1$:

$$\begin{aligned} E &= \{T_{\text{E}}, B_{\text{E}}\} \\ T[2:0] &= 0 \\ B[2:0] &= 0 \\ L_{\text{carry\_out}} &= \begin{cases} 1, & \text{if } T[11:3] < B[11:3] \\ 0, & \text{otherwise} \end{cases} \\ L_{\text{msb}} &= 1 \end{aligned}$$

Reconstituting the top two bits of T:

$$T[13:12]{=}B[13:12] + L_{\text{carry\_out}} + L_{\text{msb}}$$

Decoding the bounds:

address, $a =$	$a_{\text{top}} = a[63 : E + 14]$	$a_{\text{mid}} = a[E + 13 : E]$	$a_{\text{low}} = a[E - 1 : 0]$
top, $t =$	$a_{\text{top}} + c_{\text{t}}$	$T[13:0]$	0'E
base, $b =$	$a_{\text{top}} + c_{\text{b}}$	$B[13:0]$	0'E

To calculate corrections c_{t} and c_{b}:

$$\begin{aligned} A_3 &= a[E + 13 : E + 11] \\ B_3 &= B[13:11] \\ T_3 &= T[13:11] \\ R &= B_3 - 1 \end{aligned}$$

$A_3 < R$	$T_3 < R$	c_{t}	$A_3 < R$	$B_3 < R$	c_{b}
false	false	0	false	false	0
false	true	$+1$	false	true	$+1$
true	false	-1	true	false	-1
true	true	0	true	true	0

Figure 3.2: CHERI Concentrate 128-bit capability format and decoding

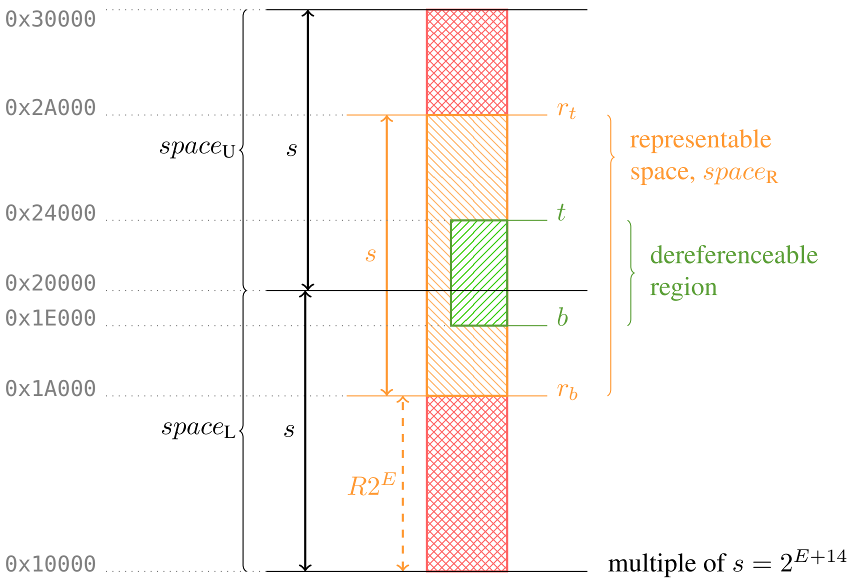

Figure 3.3: Graphical representation of memory regions encoded by CHERI Concentrate. The example addresses on the left are for a `0x6000`-byte object located at `0x1E000`; the representable region extends `0x4000` below the object's base and `0x8000` above the object's limit.

bounds precision for non-zero exponents, but we consider this an acceptable compromise to save encoding bits given that larger objects are more likely to have aligned bounds or be easily padded to alignment boundaries.

If we required that t and b had the same a_{top} bits above $E + 14$ the lower bits of a would give us a space of $s = 2^{E+14}$ values over which a can range without changing the decoded bounds. However, this would be an unacceptable restriction for software so instead we make use of the same 'spare' encodings but define a representable limit, R, relative to the base by subtracting one from the top three bits of B. If B, T or a_{mid} is less than R we infer that they lie in the 2^{E+14} aligned region above R labelled space$_U$ in figure 3.3. This allows us to compute the corrections to a_{top}, c_b and c_t, shown in the tables in figure 3.2. The overall effect is that we guarantee at least $\frac{1}{8}s$ bytes below the base and $\frac{1}{4}s$ above top where a can roam out-of-bounds while still allowing us to recover the bounds.

Additionally there is one corner case in the decoding that must be correctly handled: to allow the entire 64-bit address space to be addressable we permit t to be up to 2^{64} (i.e. a 65-bit value), but this bit-size mismatch introduces some additional complication when decoding. The following condition is required to correct t for capabilities whose representable region wraps the edge of the address space:

$$\textbf{if } ((E < 52)\,\&((t[64:63] - b[63]) > 1)) \textbf{ then } t[64] = !t[64]$$

That is, if the decoded length of the capability is larger than E allows, invert the most significant bit of t.

CHERI Concentrate Encoding (Set Bounds)

To encode a capability with requested base, b, length, l, and top, $t = b + l$, using this encoding we must first determine E by finding the most significant set bit of l. We select an E that aligns $T[12]$ with the most significant set bit of l as required for the top two bits of T to be inferred correctly when decoded:

$$E = 52 - \text{CountLeadingZeros}(l[64 : 13])$$

Note that l is a 65-bit value allowing the maximum possible length of 2^{64} to be encoded with $E = 52$, $T = 0$ and $B = 0$. We exclude the lower 12 bits of l because lengths less than this are encoded with $E = 0$ and I_E set depending on the value of $l[12]$ (L_{msb}):

$$I_E = \begin{cases} 0, & \text{if } E = 0 \text{ and } l[12] = 0 \\ 1, & \text{otherwise} \end{cases}$$

The values of B and T are formed by extracting the relevant bits from b and t. For $I_E = 0$ this means:

$$B = b[14 : 0] T = t[12 : 0]$$

With $I_E = 1$, we discard the lower bits and also lose three bits of each to store the exponent:

$$B = b[E + 14 : E + 3] T = t[E + 12 : E + 3]$$

If in truncating t we have rounded it down (i.e., if there were any set bits in $t[E + 2 : 0]$) then we must increment T by one to ensures that the encoded region includes the requested top as required by CSetBounds. In rounding b and t to 2^{E+3} aligned values we may have increased the length, so if the truncation and rounding results in $L[13]$ being set (where $L = T - B$) then we must increase E by one and re-derive B and T as above (and this time we will be guaranteed that L will not overflow).

CHERI Concentrate Alignment Requirements

For a requested base and top to be exactly representable the CHERI concentrate format may require additional alignment requirements:

- For allocations with $I_E = 0$ (i.e. lengths less than 4 kiB for $MW = 14$) there is no specific alignment requirement.
- For larger allocations the base and top must be aligned to 2^{E+3} byte boundaries (i.e. the $E + 3$ least significant bits are zero) where E is determined from the length, l, by $E = 52 - \text{CountLeadingZeros}(l[64 : 13])$
- No additional alignment requirements are currently placed on sealed capabilities or on **DDC**.

Note that there is a jump in required alignment from 1-byte to 8-bytes at the transition between $I_E = 0$ and $I_E = 1$ caused by using the lower 3 bits of T and B to store the exponent.

CHERI Concentrate Microarchitectural Considerations

CHERI concentrate decoding can be done quite efficiently in hardware. In particular when decoding the bounds calculating and comparing against the representable limit, R, requires only three-bit arithmetic and reconstituting the top two bits of T requires a 12-bit comparison and two bit arithmetic. Deriving the bounds is then a case of shifting and masking, and an up to 49-bit increment or decrement of a_{top}.

However, in our prototype FPGA implementation we found that `CSetBounds` required some optimization in order to meet timing.

We found it useful to construct a mask from l using a 'smear right' operation. This uses a tree of shift rights and ORs to set all bits from the most significant set bit down and can be done in parallel with computing E. This mask, shifted appropriately, can then be used to test for loss of significance bits in t and b and also for incrementing T when needed.

Rather than calculating $T - B$ to detect length overflow we instead detect when l is at its maximum (all ones below most significant set bit) and preemptively increment E. This results in slightly less tight bounds than the algorithm presented above and some encodings that are not used, but is conservative.

The fast representable bounds check presented in Appendix E can be used when modifying the address to detect when the bounds can no longer be correctly reconstructed and the capability should become untagged.

3.4.6 64-bit Capabilities for 32-bit Architectures

We describe an experimental application of these compression ideas to a 32-bit architecture, yielding 64-bit capabilities, in Appendix D.7.

3.5 Capability State on CPU Reset

Although the architecture-neutral description of CHERI does not define a specific set of capability registers (or capability extensions to existing registers), there are architecture-neutral invariants that must be maintained from the time of processor reset. An initial set of strong *root capabilities* must be available from inception for use by software. Most critically, the *program-counter capability* must authorize the execution of code following reset, and will typically cover the entire virtual address space. Similarly, at least one suitable root *data capability* is necessary to authorize access for data loads and stores; this will typically also cover the entire virtual address space.

An important design question is whether multiple roots are present, and if so, whether they define disjoint trees of potential capabilities. For example, the initial program-counter capability might grant load and execute permissions but not store permission; similarly, an initial data capability might grant load and store permissions but not execute permissions. Due to monotonicity rules, this would prevent the later creation of any capability holding both store and execute permissions ("W^X"). Similarly, it is easy to imagine using additional independent capability roots for orthogonal architectural rights, such as sealing and unsealing permission vs. memory access, which utilize independent namespaces (object types vs. virtual addresses). Additional discussion may be found in appendix D.8.

In general, we have taken the view that initial architectural root capabilities should hold all permissions, both architecture-defined and software-defined, allowing software the flexibility to implement any suitable models. This impacts higher-level software behavior substantially: for example, certain current POSIX APIs (e.g., `mmap()` combined with `mprotect()`) assume that decisions about load, store, and execute combinations can be made dynamically, and that it is possible to have pointers that hold all three permissions.

Depending on compatibility and security goals, software might choose to expose independent roots in its own structure – e.g., by not granting sealing permission to user code using code or data capabilities, instead returning a specific sealing root capability via a separate system call, allowing only certain object types to be used directly by userspace. The main downsides to this view are that the architecture itself does not directly embody invariants such as W^X, and that this also prevents use of different formats for disjoint provenance trees of capabilities with orthogonal functions – e.g., the use of different formats for memory-access vs. sealing capabilities. We choose to accept these costs in return for a more flexible software model in which all root capabilities at processor reset hold all permissions.

3.5.1 Capability Registers on Reset

When the CPU is hard reset, all capability registers intended to act as roots will be initialized to the following values:

- The **tag** bit is set.
- **offset** = 0, except for the program-counter capability, which will have its **offset** initialized to an appropriate boot vector address. Other architecture-specific capability registers may have other initial values – e.g., as relates to exception vectors.
- **base** = 0
- **length** = $2^{64} - 1$ for 256-bit capabilities, and 2^{64} for compressed capability representations.
- **otype** = $2^{64} - 1$ (truncated as required by the implementation's encoding).
- All available permission bits are set. When the 256-bit capability representation is used, 31 permission bits are available, including 16 software-defined permissions. When the 128-bit capability representation is used, 15 permission bits are available, including 4 software-defined permissions. Software-defined permissions bits that are not available are set to zero.
- Concrete architectures specify the reset value of **flags** for root capabilities.
- All unused bits are cleared.

If the architecture-specific approach is to extend existing integer registers to also hold tagged capabilities, then those registers may instead be initialized to hold untagged values:

- The **tag** bit is unset.
- **offset** = 0 (or some other value appropriate to the register).
- **base** = 0.
- **length** = $2^{64} - 1$ for 256-bit capabilities, and 2^{64} for compressed capability representations.
- **otype** = $2^{64} - 1$ (truncated as required by the implementation's encoding).
- All available permission bits are unset.
- **flags** = 0x0.
- All unused bits are cleared.

3.5.2 Tagged Memory on Reset

In an ideal world, all tags in memory are cleared on CPU reset, as this avoids the unpredictable introduction of additional capability roots. However, this is not straightforward to offer architecturally or microarchitecturally. We instead rely on firmware or software supervisors to ensure that pages placed into use, especially with untrustworthy code, have been properly cleared. While this property is often already enforced by real-world hardware and systems – whether due to Error-Correcting Codes (ECC),[5] or because of page zeroing by the OS. However, the criticality of this behavior becomes quite high given the risks associated with errant tagged values.

3.6 Capability-Aware Instructions

A key design choice in the CHERI protection model is *intentionality*: the use of explicit instructions that accept (and require) capability operands rather than overloading existing instructions, allowing selection of integer-relative or capability-relative semantics. In particular, it is essential that selection of integer or capability semantics never be conditional on the value of the operand's tag. This requires not just the introduction of instructions to inspect, manipulate, load, and store capabilities, as a new CPU data type, but also a set of explicit load, store, and control-flow instructions accepting capability operands as the base address or jump target where the baseline ISA would accept explicit integer operands.

We have generally attempted to minimize the number of new instructions. However, in some cases multiple variants are required to optimize important code paths – for example, capability bounds can be set using both an integer register operand (`CSetBounds`), where there is a dynamically defined size, such as when using `malloc`, and an immediate operand (`CSetBoundsImm`), where there is a compilation-time size available, such as for most stack-allocated buffers.

[5] To avoid any potential confusion, we note that ECC is also widely used for Elliptic-Curve Cryptography.

Where possible, the structure and semantics of capability instructions have been aligned with similar core MIPS instructions, similar calling conventions, and so on. CHERI depends on introducing several new classes of instructions to the baseline ISA. In some cases these are congruent to similar instructions relating to general-purpose integer registers, control-flow manipulation, and memory accesses, in the form of capability-register manipulation, jumps to capabilities, and capability-relative memory accesses. Others have functions specific to CHERI, such as those manipulating capability fields, and those relating to protection-domain transition. The semantics of these instructions implements many aspects of the protection model; for example, constraints on permission and bounds manipulation in capability field manipulation instructions contribute to enforcing CHERI's capability monotonicity properties. These instructions are described in detail in Chapter 7:

Retrieve capability fields These instructions extract specific capability-register fields and move their values into general-purpose (integer) registers: `CGetAddr`, `CGetAndAddr`, `CGetBase`, `CGetFlags`, `CGetLen`, `CGetOffset`, `CGetPerm`, `CGetSealed`, `CGetTag`, and `CGetType`.

Capability move This instruction moves a capability from one register to another without change: `CMove`.

Conditional capability move These instructions conditionally move a value based on whether a capability is NULL or non-NULL: `CMOVN` and `CMOVZ`.

Manipulate capability fields These instructions modify capability-register fields, setting them to values moved from integer registers, subject to constraints such as monotonicity and representability: `CAndAddr`, `CAndPerm`, `CClearTag`, `CIncOffset`, `CIncOffsetImm`, `CSetAddr`, `CSetBounds`, `CSetBoundsExact`, `CSetBoundsImm`, `CSetFlags`, and `CSetOffset`.

Derive integer pointers from capabilities, or capabilities from integer pointers The `CToPtr` and `CFromPtr` instructions efficiently convert between integer pointers and capabilities, performing suitable bounds checks against contextual capabilities. These support efficient hybrid code, in which use of integer pointers and capabilities are intermixed.

Capability pointer comparison and arithmetic These instructions provide C-language-centric pointer comparison and subtraction behavior: `CPtrCmp` and `CSub`.

Load or store via a capability These instructions access memory indirected via an explicitly named capability register, and will ideally correspond to a full range of contemporary indexing modes present in the baseline ISA – for example, allowing aligned or unaligned access to zero-extended and sign-extended integers of varying widths, as well as loading and storing of capabilities themselves. Further, software stacks dependent on atomic operations on pointers will require a suitable suite of atomic operations loading, modifying, and storing capabilities – e.g., load-linked, store-conditional instructions, or atomic test-and-set instructions, depending on the underlying architecture. These instructions include: `CL[BHWD][U]`, `CLCBI`, `CLL[BHWD][U]`, `CLLC`, `CSC`, `CS[BHWD]`, `CSC[BHWD]`, and `CSCC`.

These correspond in semantics to the similar baseline ISA instructions, but are constrained by the properties of the named capability including tag check, permissions, bounds, seal check, and so on; if capability protections would be violated, then an exception will be thrown. Capability restrictions can be used to implement spatial safety via permissions and bounds.

Program-Counter Capability Generated code makes frequent reference to **PCC** in common position-independent code structures, such as references to the Global Offset Table (GOT) or Program Linkage Table (PLT). The instructions `CGetPCC` and `CGetPCCSetOffset` allow **PCC** to be retrieved.

Capability jumps Capability-based code pointers allow the implementation of control-flow robustness by limiting the permissions and bounds on jump targets (e.g., preventing store, and limiting fetchable instructions). Depending on the underlying ISA, different jump variations may be required – for example, adding capability variants of jump-and-link register, jump register, and so on, including: `CJALR` and `CJR`.

Branch on capability fields These instructions branch within the current program-counter capability (i.e., to an immediate relative to the current program counter) dependent on capability tags or the capability holding a NULL value. These include: `CBEZ`, `CBNZ`, `CBTS`, and `CBTU`.

Capability checks The `CCheckPerm`, `CCheckTag`, and `CCheckType` instructions compare capability fields with expected permissions and types, throwing an exception if they do not match, or that the capability tag is set. These are used to validate arguments on entry to protected subsystem, and for compiler-inserted tag assertions. Argument validation instructions are in the process of being deprecated, as they have not seen practical use.

Capability sealing The `CSeal` and `CUnseal` instructions seal or unseal capabilities given a suitable authorizing capability (i.e., one with the Permit_Seal or Permit_Unseal permission as appropriate). Sealed capabilities allow software to implement encapsulation, such as is required for software compartmentalization.

Protection-domain switching The `CCall` and `CReturn` instructions are primitives upon which protection-domain switching can be implemented. Both instructions can be implemented in terms of hardware-assisted exception handlers; `CCall` has a further jump-based semantic that unseals its sealed code and data capability-register operands. Both calling semantics allow software-controlled non-monotonicity by granting access to additional state via exception-handler registers or unsealing.

Fast register clear The `CClearRegs` instruction clears a range of capability, integer, or floating-point registers to support fast protection-domain transition.

Special capability registers Special capability registers are read and written via special `CReadHwr` and `CWriteHwr`.

Tag loading and rederivation Certain system operations, such as process or virtual-machine checkpointing and memory compression, require that tagged memory have its tags saved and then restored. Memory locations can be iteratively loaded into capability registers to check for tags; tags can then be later restored by manually rederived manually using instructions such as CAndPerm and CSetBounds. However, these instruction sequence is complex and can incur substantial overhead when used during bulk restoration. The CLoadTags instruction allows tags to be loaded for a cache line of memory (non-temporally), and the CBuildCap, CCopyType, and CCSeal instructions allow tags to be efficiently restored.

Compartment identifiers CHERI protection domains, when constructed purely of graphs of capabilities, do not allow the microarchitecture to explicitly identify one domain from another. In order to allow tagging of microarchitectural state, such as branch-predictor entries, to avoid side channels, instructions are present to allow software to explicitly identify compartment boundaries where confidentiality requirements preclude more extensive microarchitectural sharing: CGetCID and CSetCID.

In addition, architectures may require additional capability-related instructions related to conditional moves and exception delivery.

3.7 Handling Failures

Instruction-set architectures have various resources in the event that a “failure” occurs, with common choices being to set special status bits (on ISAs that have status registers), to write back a special value to a general-purpose integer register, or to throw an exception. CHERI introduces several new potential failure modes:

Instruction-fetch failures Because the program counter is extended to be a capability, it is possible for CHERI to deny access for instruction fetch. For example, the program counter may move out of bounds, software may jump to an untagged or otherwise insufficiently authorized capability, or an exception handler may install an untagged or insufficiently authorized capability on return.

We explored two variations on failure reporting: to report the failure via an exception at the time that the new program-counter capability is installed (e.g., on the jump instruction), or at the time that the instruction fetch is requested (i.e., when execution of the new instruction is requested). Throwing an exception on fetch leads to the most consistent general behavior, and also better handles installation of an invalid exception program-counter capability by avoiding an exception within the kernel when the exception program-counter capability is written to. Throwing an exception prior to writing the new value to **PC**, on the other hand, provides more complete debugging information: the errant jump **PC** is available to the exception handler. With compressed capabilities, this also provides access to the target virtual address and fully precise bounds; in the event of a substantially out-of-bounds target address, either the target virtual address or the bounds would have to be discarded to ensure a representable capability.

Ultimately, both approaches are consistent with our security goals. We therefore err on the side of improved debuggability, throwing exceptions on jump where possible. We also require checking of capability properties on instruction fetch to catch cases such as exception return to an invalid or out-of-bounds capability.

Load and store failures When dereferencing a capability for data access occurs, ISAs generally report this failure via an exception at the time of the attempted access, which CHERI in general does as well. These exceptions fit existing patterns of exception delivery in MMU-based architectures and operating systems, which are designed to handle faults on memory access.

There are two cases in which an alternative approach is taken: when the Permit_Load capability permission or equivalent page-table or TLB permission is not present, any tag on the loaded capability is instead stripped. This avoids an exception that depends on the loaded data value, which is awkward in some architectures (e.g., ARMv8), but also facilitates writing code for tag-stripping memory copies, which arise frequently around protection-domain boundaries.

Guarded manipulation failures A new class of register-to-register instructions in CHERI can experience failures when attempts are made to violate rules imposed via guarded manipulation – for example, attempts to perform non-monotonic operations, or transformations that lead to non-representable bounds with compressed capabilities. In our initial CHERI-MIPS design, we took the perspective that reporting failures early allowed the greatest access to debugging information, and favored throwing an exception at the earliest possible point: the instruction attempting to violate guarded manipulation.

Another potential design choice is to instead strip the tag from the value being written back to a target capability register, which maintains our security safety properties, but defers exception delivery until an attempted dereference – e.g., an instruction load via the resulting untagged capability. There are two arguments for this latter behavior: first, that some architectures by design limit the set of instructions that throw exceptions to facilitate superscalar scheduling (e.g., ARMv8); and second, that exception delivery means that failures that could otherwise be easily detected and handled by a compiler or language run time via an explicit tag check are now complex to handle.

When using tag stripping in ISAs with status registers (e.g., ARMv8), the cost of checking results for frequent operations can be amortized via a single status check. For ISAs without status registers, checking results can come at a significant cost, and a deferred exception delivery at time of dereference will be the best choice for performance-critical code.

We therefore make design choices about exception delivery for violations of guarded manipulation in a case-by-case basis, taking into account more general architectural design philosophies, and also specific use cases where software may benefit from tag clearing rather than exception delivery.

3.8 Composing Architectural Capabilities with Existing ISAs

In applying CHERI to an architecture, the aim is to impose the key properties of the abstract CHERI model in a manner keeping with the design philosophy and approach of each architecture: strong compatibility with MMU-based, C-language TCBs; strong fine-grained memory protection supporting language properties; and incrementally deployable, scalable, fine-grained compartmentalization. This should allow the construction of portable, CHERI-aware software stacks that have consistent protection properties across a range of underlying architectures and architectural integration strategies.

ISAs vary substantially in their representation and semantics, but have certain common aspects:

- One or more operation encoding (opcode) spaces representing specific instructions as fetched from memory;
- A set of architectural registers managed by a compiler or hand-crafted assembly code, which hold intermediate values during computations;
- Addressable memory, reached via a variety of segmentation and paging mechanisms that allow [optional] implementation of virtual addressing;
- An instruction set allowing memory values to be loaded and stored, values to be computed upon, control flow to be manipulated, and so on, with respect to both general-purpose integer and floating-point values – and vectors of values for an increasing number of ISAs;
- An exception mechanism allowing both synchronous exceptions (e.g., originating from instructions such as divide-by-zero, system calls, unimplemented instructions, and page-table misses) and asynchronous events from outside of the instruction flow (timers, inter-processor interrupts, and external I/O interrupts) that cause a controlled transition to a supervisor;
- A set of control instructions or other (perhaps memory-mapped) interfaces permitting interaction with the boot environment, management of interrupt mechanisms, privileged state, virtual addressing features, timers, debugging features, energy management features, and performance-profiling features.

  Depending on the architecture, these might be strictly part of the ISA (e.g., implemented explicit instructions to flush the TLB, mask interrupts, or reset the register state), or they may be part of a broader platform definition with precise architectural behavior dependent on the specific processor vendor (e.g., having firmware interfaces that flush TLBs or control interrupt state, or register values at the start of OS boot rather than CPU reset).

Implementations of these concepts in different ISAs differ markedly: opcodes may be of fixed or variable lengths; instructions might strictly separate or combine memory access and computation; page tables may be a purely software or architectural constructs; and so on. Despite these differences in underlying software representation, a large software corpus (implemented in both low-level languages (e.g., C, C++) and higher-level managed languages – e.g., Java) can be written and maintained in a portable manner across multiple mainstream architectures.

The CHERI protection model is primarily a transformation of memory access mechanisms in the instruction set, substituting a richer capability mechanism for integer pointers used with load and store instructions (as well as instruction fetch). However, it has broad impact across all of the above ISA aspects, as it is by design explicitly integrated with register use (to ensure intentionality of access) rather than implicit in existing memory access (as is the case with virtual memory). CHERI must also integrate with the exception mechanism, as handling an exception implies a change in effective protection domain, control of privileged operations such as management of virtual memory, and so on.

CHERI-MIPS is an application of the CHERI protection model to the 64-bit MIPS ISA. CHERI-MIPS is grounded in MIPS's load-storage architecture (instructions either load/store data with respect to memory, or compute on register values, but never both), the software-managed TLB (page tables are a purely software construct), and the MIPS ISA "coprocessor" opcode space reserved for ISA extensions. As a result, a number of concrete design choices are made that are in many ways specific to MIPS: a decision to separate general-purpose integer files and capability register files; occupation of the coprocessor opcode space; and TLB rather than page-table additions to control the use of capabilities. These low-level design choices will apply to only a limited degree in other ISAs – but the objectives achieved through these choices must also appear in other ISAs implementing the CHERI model: explicit use of capabilities for addressing relative to virtual-address spaces, monotonicity enforcement via guarded manipulation, tagged memory protecting valid pointer provenance in memory, suitable support in the exception mechanism to allow current OS approaches combining user and kernel virtual-address spaces, and so on.

In the following chapters we present high-level sketches of applications of the CHERI protection model to three ISAs: 64-bit MIPS (in which our ideas were first developed and prototyped), RISC-V (a contemporary load-store instruction set – which in many ways is a descendant of the MIPS ISA); and the x86-64 ISA (which has largely independent lineage of Complex Instruction Set (CISC) architectures). The CHERI model applies relatively cleanly to all three, with many options available in how specifically to apply its approach, and yet with a consistent overall set of implications for software-facing design choices. Wherever possible, we aim to support the same operating-system, language, compiler, run-time, and application protection and security benefits, which will be represented differently in machine code and low-level software support, but be largely indistinguishable from a higher-level programming perspective. These instantiations should retain the highly compatible strong protection and compartmentalization scalability properties seen with CHERI extensions for MIPS.

It is possible to imagine less tight integration of CHERI's features with the instruction set. Microcontrollers, for example, are subject to tighter constraints on area and power, and yet might benefit from the use of capabilities when sharing memory with software running on a fully CHERI-integrated application processor. For example, a microcontroller might perform DMA on behalf of a CHERI-compiled application, and therefore desire to constrain its access to those possible through capabilities provided by the application. In this scenario, a less complete integration might serve the purposes of that environment, such as by providing a small number of special capability registers sufficient to perform capability-based loads and stores, or to perform tag-preserving memory copies, but not intended to be used for the majority of general-purpose operations in a small, fixed-purpose program for which strong static checking or proof of correctness may be possible.

3.8.1 Architectural Privilege

In operating-system design, *privileges* are a special set of rights exempting a component from the normal protection and access-control models – perhaps for the purposes of system bootstrapping, system management, or low-level functionality such as direct hardware access. In CHERI, three notions of privilege are defined, complementing current notions of architectural privilege:

Ring-based privilege derives from the widely used architectural notion that code executes within a *ring*, typically indicated by the state of a privileged status register, authorizing access to architectural protection features such as MMU configuration or interrupt management. Code executing in lower rings, such as a microkernel, hypervisor, or full operating-system kernel, has the ability to manage state giving it control over state in higher, but not lower, rings. When a privileged operation is attempted in a higher ring, an architectural exception will typically be thrown, allowing a supervisor to emulate the operation, or handle this as an error by delivering a signal or terminating a process. More recent hardware architectures allow privileged operations to be virtualized, improving the performance of full-system virtualization in which code that would historically have run in the lowest ring (i.e., the OS kernel) now runs over a hypervisor.

CHERI retains and extends this notion of privilege into the capability model: when an unauthorized operation is performed (such as attempting to expand the rights associated with a capability), the processor will throw an exception and transition control to a lower ring. The exception mechanism itself is modified in CHERI, in order to save and restore the capability register state required within the execution of each ring – to authorize appropriate access for the exception handler. The lower ring may hold the privilege to perform the operation, and emulate the unauthorized operation, or perform exception-handling operations such as delivering a signal to (or terminating) the user process.

Capability control of ring-related privileges refers to limitations that can be placed on ring-related privileges using the capability model. Normally, code executing in lower protection rings (e.g., the supervisor) has access to privileged functions, such as MMU, cache, and interrupt management, by virtue of ambient authority. CHERI permits that ambient authority to be constrained via capability permissions on the *program-counter capability*, preventing less privileged code (still executing within a low ring) from exercising virtual-memory features that might allow bypassing of in-kernel sandboxing. More generally, this allows vulnerability mitigation by requiring explicit (rather than implicit) exercise of privilege, as individual functions can be marked as able to exercise those features, with other kernel code unable to do so.

These models can be composed in a variety of ways. For example, if a compartmentalization model is implemented in userspace over a hybrid kernel, the kernel might choose to accept system calls from only suitably privileged compartments within userspace – such as by requiring those compartments to have a specific software-defined permission set on their program-counter capability.

Layering Software Privilege over Capability Privilege

In addition to these purely architectural views of privilege, privileged software (e.g., the OS kernel running in supervisor mode) is able to selectively proxy access to architectural privilege via system calls. This facility is used extensively in contemporary designs. For example, requests to memory map files or anonymous memory, after processing by many levels of abstraction, lead to page-table updates, TLB flushes, and so on. Similarly, requests to configure in-process signal timers or time out I/O events, many levels of abstraction lower, are translated into operations to manage hardware timers and interrupts.

Similar structures can be implemented using the CHERI capability model. **Privilege through capability context** is a new, and more general, notion of privilege arising solely from the capability model, based on a set of rights held by an execution context connoting privilege within an address space. When code begins executing within a new address space, it will frequently be granted full control over that address space, with initial capabilities that allow it to derive any required code, data, and object capabilities it might require. This notion of privilege is fully captured by the capability model, and no recourse is required to a lower ring as part of privilege management in this sense. This approach follows the spirit of Paul Karger's paper on limiting the damage potential of discretionary Trojan horses [57], and extends it further. Certain operations, such as domain transition, do employ the ring mechanism, in order to represent controlled privilege escalation – e.g., via the object-capability call and return instructions.

3.8.2 Traps, Interrupts, and Exception Handling

CHERI retains and extends existing architectural exception support, as triggered by traps, system calls, and interrupts. CHERI affects the situations in which exceptions are triggered, and changes aspects of exception delivery, state management within exceptions, and also exception return. Exception handling is also one of the means by which non-monotonic state transition takes place: as exception handlers are entered, they gain access to capabilities unavailable to general execution, allowing them to implement mechanisms such as domain transition to more privileged compartments. As exception support varies substantially by architecture – how exception handlers are registered, what context is saved and restored, and so on – CHERI integration necessarily varies substantially. However, certain general principles apply regardless of the specific architecture.

New Exceptions for Existing and New Instructions

New exception opportunities are introduced for both existing and new instructions, which may trap if insufficient rights are held, or an invalid operation is requested. For example:

- Instruction fetch may trap if it attempts to fetch an instruction in a manner not authorized by the installed Program-Counter Capability (**PCC**).
- Existing integer-relative load and store instructions will trap if they attempt to access memory locations in a manner not authorized by the installed Default Data Capability (**DDC**).
- New capability-relative load and store instructions will trap if they attempt to access memory locations in a manner not authorized by the explicitly presented capability.
- New capability-manipulation instructions may trap if they violate guarded-manipulation rules, such as by attempting to increase the bounds on a capability.

In general, CHERI attempts to provide useful cause information when exceptions fire, including to identify whether an exception was triggered by using an invalid capability, dereferencing a sealed capability, or an access request not being authorized by capability permissions or bounds (see Section 3.8.2 for details).

Exception Delivery

The details of exception delivery vary substantially by architecture; however, CHERI adaptations are in general fairly consistent across architectures:

Interrupt state Interrupts will typically be disabled on exception entry. System software will typically leave interrupts disabled during low-level processing, but re-enable interrupts so as to allow preemption during normal kernel operation. CHERI does not change this behavior.

Control-flow state The Program Counter (**PC**) will be saved to an Exception Program Counter (**EPC**). System status state, such as the ring in which the interrupted code was executing, as well as possibly other state such as interrupt masks, will be saved in a special status register. System software will typically save this and any other register state associated with the preempted code, allowing to to establish a full execution context for the exception handler, or to switch to another thread. CHERI extends **PC** to become a Program-Counter Capability (**PCC**) and **EPC** to become an Exception Program-Counter Capability (**EPCC**). Depending on the architecture, status registers may be extended to also contain CHERI-related information, such as whether opcode interpretation for loads and stores is integer relative or capability relative (as in CHERI-RISC-V), allowing that state to differ between interrupted code and the exception handler.

Other architectural state In addition to general-purpose registers, architectures may provide access to a set of special registers, such as for Thread-Local Storage (TLS). Additional context banking or saving may also occur, to facilitate fast exception delivery. For example, in ARMv8, the stack pointer register is banked, allowing exception handlers to use their own stack pointer to save remaining registers. In x86-64, the full register context will be saved to a pre-configured memory location. CHERI extensions are also required to these additional pieces of architectural context management; for example, TLS integer registers must be extended to become TLS capabilities. The banked ARMv8 stack pointer would need to widened to a full capability, and the x86-64 context-management implementation would also need to save full capability state.

Exception-handler entry In order to execute an exception handler, the architecture will switch to an appropriate ring (often the supervisor ring), and set **PC** to the address of the desired exception vector. Exception delivery may also change other aspects of execution, such 32-bit vs. 64-bit execution, so as to enter the exception handler in the execution mode that is expected. CHERI introduces two new capabilities: the Kernel Code Capability (**KCC**) and the Kernel Data Capability (**KDC**), which provide additional rights to the exception handler authorizing its execution. **KCC** will automatically be combined with exception vector address and installed in **PCC** to execute the vector. **KDC** becomes available by virtue of **PCC** having the Access_System_Registers permission, and can be used to reach exception-handler data such as global variables.

Safe exception state handling

In some architectures, partial register banking or reserved exception-only registers mean that exception handlers must utilize only a subset of registers unless they explicitly save them. With CHERI, it is essential that capability register values not just be saved and restored, to ensure correct functionality, but that capability register values are also not leaked, as this may undesirably grant privilege. For example, even if the ABI does not require that a system call or trap maintain the values of certain registers over exception handling, the exception handler must restore or clear those values to ensure that capabilities used by the exception handler or another context are not leaked.

Exception Return

Exception return unwinds the effects described in the previous section, restoring **PC** from **EPCC**, restoring the saved ring and interrupt-enable state, swapping banked registers, and so on. The changes made to support CHERI exception entry must also be made to exception return, such as restoring the full **EPCC** to **PCC**.

Capability Exception Causes

In each of the target ISAs (MIPS, RISC-V, and x86-64), we introduce a new capability status code register that reports further details of the most recent capability-related exception. Use of the capability status register is indicated by a new high-level exception code in the ISA's native exception cause register. Depending on the ISA, the capability status register may be banked (i.e., there may be different independent instances in different rings). The register is formatted as shown in Figure 3.4. The possible values for *ExcCode* are shown in Table 3.8.2. If the last instruction to throw an exception did not throw a capability exception, then the *ExcCode* field of **capcause** will be *None*. *ExcCode* values from 128 to 255 are reserved for use by application programs.

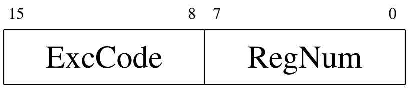

Figure 3.4: Capability Cause Register

Value	Description
0x00	None
0x01	Length Violation
0x02	Tag Violation
0x03	Seal Violation
0x04	Type Violation
0x05	Call Trap
0x06	Return Trap
0x07	Underflow of trusted system stack
0x08	Software-defined Permission Violation
0x09	MMU prohibits store capability
0x0a	Requested bounds cannot be represented exactly
0x0b	*reserved*
0x0c	*reserved*
0x0d	*reserved*
0x0e	*reserved*
0x0f	*reserved*
0x10	Global Violation
0x11	Permit_Execute Violation
0x12	Permit_Load Violation
0x13	Permit_Store Violation
0x14	Permit_Load_Capability Violation
0x15	Permit_Store_Capability Violation
0x16	Permit_Store_Local_Capability Violation
0x17	Permit_Seal Violation
0x18	Access_System_Registers Violation
0x19	Permit_CCall Violation
0x1a	Access_CCall_IDC Violation
0x1b	Permit_Unseal Violation
0x1c	Permit_Set_CID Violation
0x1d	*reserved*
0x1e	*reserved*
0x1f	*reserved*

Table 3.3: Capability Exception Codes

The *RegNum* field of **capcause** will hold the number of the capability register whose permission was violated in the last exception, if this register was not the unnumbered register **PCC**. If the capability exception was raised because **PCC** did not grant access to a numbered reserved register, then **capcause** will contain the number of the reserved register to which access was denied. If the exception was raised because **PCC** did not grant some other permission (e.g., permission to read **capcause** was required, but not granted) then *RegNum* will hold 0xff.

Capability Exception Priority

Exception handling in most architectures involves an architectural cause code that describes the type of event that triggered the exception – for example, indicating that a trap has been caused by a read or write page fault. Exception types are prioritized so that if more than one exception code could be delivered – e.g., there is the potential for both an alignment fault and also a page fault triggered by a particular load or store – a single cause is consistently reported.

Capability-triggered exceptions in general have a high priority, above that for either alignment faults or MMU-related faults (such as page-table or TLB misses), as capability processing logically occurs "before" a virtual address is interpreted. This also prevents undesirable (or potentially insecure) behaviors, such as the ability to trigger a page fault on a virtual address outside the bounds of a capability being dereferenced: instead, the bounds error should be reported. Similarly, if an operating system implements emulation of unaligned loads and stores by catching unaligned-access exceptions, having capability checks occur in preference to alignment exceptions avoids having alignment emulation also perform capability checks – e.g., of its length or permissions. Other priority rules are less security critical, but are defined by this specification so that exception processing is deterministic. Each architecture defines its own exception priority, and architecture-specific instantiations of CHERI must define an architecture-specific prioritization for capability-related exceptions relative to other exception types.

If an instruction could potentially throw more than one capability exception, **capcause** is set to the highest priority exception (numerically lowest priority value) as shown in Table 3.4. The *RegNum* field of **capcause** is set to the register which caused the highest priority exception.

If an instruction throws more than one capability exception with the same priority (e.g., both the source and destination register are reserved registers), then the register that is furthest to the left in the assembly language opcode has priority for setting the *RegNum* field.

3.8.3 Virtual Memory

Where virtual memory is present and enabled, CHERI capabilities are interpreted with respect to the current virtual address space. In CHERI-MIPS, this means that the embedded address in a capability is always a virtual address, as the virtual-address translation cannot be disabled. In others, such as CHERI-RISC-V, where virtual-address translation can be enabled or disabled dynamically, the embedded address will be interpreted as a physical address when translation is disabled, and a virtual address when virtual addressing is enabled.

Capabilities do not embed Address-Space IDentifiers (ASIDs), and so will be interpreted relative to the current virtual address space; this means that, as with virtual addresses them-

Priority	Description
1	Access_System_Registers Violation
2	Tag Violation
3	Seal Violation
4	Type Violation
5	Permit_Seal Violation
	Permit_CCall Violation
	Access_CCall_IDC Violation
	Permit_Unseal Violation
	Permit_SetSID Violation
6	Permit_Execute Violation
7	Permit_Load Violation
	Permit_Store Violation
8	Permit_Load_Capability Violation
	Permit_Store_Capability Violation
9	Permit_Store_Local_Capability Violation
10	Global Violation
11	Length Violation
12	Requested bounds cannot be represented exactly
13	Software-defined Permission Violation
14	MMU prohibits store capability
15	Call Trap
	Return Trap

Table 3.4: Exception Priority

selves, the interpretation of a specific capability value depends on the address space that they are used in. The operating system or other TCBs may wish to limit the flow of capabilities between address spaces for this reason.

Processing of capabilities is therefore "before" virtual-address translation, with the result of each memory access via a capability being an access control decision (allow or reject the access) and a virtual address and length for the authorized operation. The operation then proceeds through the normal memory access paths for instruction fetch, load, or store. The capability mechanism therefore never enables new operations not already supported by existing MMU-based checks.

Authorizing MMU Control

Memory Management Units (MMUs) typically come in two flavors: those supporting architectural (hardware) page-table walking; and those supporting software TLB management. In the former category, a series of instructions or control registers configures parameters such as the page-table format being used, the current page-table root, and can selectively or fully flush the Translation Look-aside Buffer (TLB). The page table has an architecturally defined format, consisting of a multi-level tree of Page-Table Directory Entries (PTDEs) and leaf-node Page-Table Entries (PTEs), and may not only be read but also written to if dirty bits are supported. The architecture will perform a series of memory reads to locate the correct page-table entry to satisfy a lookup, filling a largely microarchitectural TLB. In the latter category, instructions directly manipulate architectural TLB entries as a result of TLB miss (or other) exceptions. In both cases, exceptions may fire if operations are rejected as a result of page permission checks (e.g., and attempt to store to a read-only page). CHERI composes with these mechanisms in several ways:

- CHERI controls use of privileged instructions and control registers that configure the MMU, including enabling and disabling translation, configuring a page-table root if supported, and flushing the TLB. To perform these operations, the Access_System_Registers permission must be present on **PCC**.
- On systems with a software-managed TLB, such as CHERI MIPS, retrieving and inserting TLB entries also depends on Access_System_Registers being present on **PCC**. The TLB exception handler will use **KCC** and **KDC**, available in the exception context, to authorize access to the software-managed page table.
- On systems with a hardware page-table walker, CHERI currently *does not* control memory accesses performed by the walker via physical addresses. In a more ideal future world, the page-table walker would be given an initial, likely physical, capability to use as the root, and have further access authorized by capabilities embedded in page-table directory entries.

Page-Table-Entry or TLB-Entry Permissions

Virtual-address translation is itself unmodified, but permission checking is extended with two new page permissions in the TLB entries or PTEs (depending on the architecture):

Page-Table Load Capability Permission If this permission is present, as well as the existing page-table read permission, then loading tagged capabilities is permitted. Otherwise, if a capability load operation would load a capability value that has the tag bit set, the tag bit will be stripped before register write back.

Page-Table Store Capability Permission If this permission is present, as well as the existing page-table write permission, then storing tagged capabilities is permitted. Otherwise, if a capability store operation occurs with a capability value that has the tag bit set, an exception will be thrown.

With the page-table store-capability permission, it is also imaginable that the architecture might choose to strip the tag bit before performing the store, rather than throw an exception, if the permission is required but not present. This would avoid a data-dependent exception, which may simplify the microarchitecture. However, this would disallow the dynamic tracking of possible capability locations using this permission bit, in a manner similar to emulated dirty page support. As this support may be important in improving performance for revocation and garbage collection, it would be desirable to provide some other mechanism in that case.

Capability Dirty Bit

In architectures that support tracking dirty pages in the page table, by performing updates to page-table entries when a page has been dirtied, it is imaginable that a new *capability dirty bit* might provide a suitable substitute for trapping on a failed capability store. This bit would be set atomically if a new tagged capability value is stored via the page. In as much as the architecture supported false positives for the page dirty bit – i.e., that the dirty bit could be set even though there wasn't a committed data write – that would also be permissible for the capability dirty bit. However, false negatives – in which the dirty bit is not set despite the page becoming dirty – would not be permissible for the capability dirty bit. Otherwise, there is a risk that revocation or garbage collection might "miss" a capability, violating a temporal security or safety policy.

Memory Compression, Memory Encryption, Swapping, and Migration

When memory pages are stored to a non-tag-bearing medium, such as by virtue of being compressed in DRAM, encrypted, swapped, or perhaps migrated to another system by virtue of process or virtual-machine migration, tags must also be saved and restored. Architecturally, this can be performed by reading through the page of memory, checking for tags, and preserving them out-of-band – e.g., in a swap meta-data structure. They can then be restored by rederiving the capability value from some suitably privileged authorizing capability. We offer specific instructions to support efficiently restoring tags without software inspecting the in-memory format: `CBuildCap` and `CCSeal`. The `CLoadTags` allows efficient gathering of tag data from full cache lines, and will have non-temporal behavior – i.e., will not perform cache allocation, despite being coherent, to avoid sweeping passes pulling all the corresponding data into the cache. It is imaginable that a `CStoreTags` instruction might be desirable to set tags bulk, but this would require some care with privilege to avoid an arbitrary `CSetTag` implementation rather than controlled rederivation.

3.8.4 Direct Memory Access (DMA)

As described in this chapter, the CHERI capability model is a property of the instruction-set architecture of the CPU, and imposed on code executing on that CPU. However, in most computer systems, Direct Memory Access (DMA) is used by non-application cores, accelerators, and peripheral devices to transfer data into and out of system memory without explicit instruction execution for each byte transferred: device drivers configure and start DMA using device or DMA-engine control registers, and then await completion notification through an interrupt or by polling. Used in isolation, nothing about the CHERI ISA implies that device memory access would be constrained by capabilities.

DMA Stores with Tag Stripping

Our first recommendation is that, in the absence of additional support, DMA access to memory be unable to write tagged values, and that it implicitly strip tags associated with stored memory locations as all writes will be data and not capabilities. This implements a conservative model in which only the CPU is able to introduce capabilities into the system, and DMA stores do not risk errantly (or maliciously) introducing capabilities without valid provenance, or corrupt CPU-originated capabilities– as all such writes will involve data and not capabilities.

Capability-Aware DMA and IOMMUs

Our second recommendation is that “capability-aware DMA” – i.e., DMA that can load and store tagged values – be the remit of only trustworthy DMA engines that will preserve valid provenance, ensure monotonicity, and so on. As with capabilities on general-purpose CPUs, capabilities must be evaluated with respect to an address space. In the event that no IOMMU is present, this will be a (possibly “the”) physical address space. With an IOMMU, this will be one of potentially many I/O virtual address spaces. As with multiple virtual address spaces on an MMU-enabled general-purpose CPU, care will need to be taken to ensure that capabilities can be used only in address spaces where they have appropriate meaning.

There is a more general question about the *reachability* of all capabilities: a general-purpose OS can reasonably be expected to find all available capabilities through awareness of architectural registers and tag-enabled memory, for the purposes of revocation or garbage collection. Capabilities held by devices will require additional work to locate or revoke, and will likely require awareness of the specific device. This is an area for further research.

3.9 Implications for Software Models and Code Generation

3.9.1 C and C++ Language and Code Generation Models

CHERI capabilities are an architectural primitive that can be used in a variety of ways to support different aspects of software robustness. This is especially true because of CHERI's hybrid approach, which supports incremental deployment within both source languages and code generation. We have explored three different C and C++ language models:

Pure Integer Pointers In this C-language variant, all pointers are assumed to be implemented as integer virtual addresses.

Hybrid Pointers In this C-language variant, pointers may be implemented as integer virtual addresses or as capabilities depending on language-level types or other annotations. While we have primarily explored the use of a simple qualifier, `__capability`, which indicates that a pointer type should be implemented as a capability, a variety of other mechanisms can or could be used. For example, policy for the use of capabilities might be dictated by binary compatibility constraints: public APIs and ABIs for a library might utilize integer pointers, but all internal implementation might use capabilities.

Pure-Capability Pointers In this C-language variant, all pointers are implemented as capabilities.

Alongside these language-level models, we have also developed a set of binary code-generation and binary interface conventions regarding software-managed capabilities. These are similar to those used in non-capability designs, including features such as caller-save and callee-save registers, a stack pointer, etc. We have explored three different Application Binary Interfaces (ABIs) that utilize capabilities to varying degrees:

Native ABI The native ABI(s) for the architecture: capability registers and capability instructions are unused. Generated code relies on CHERI compatibility features to interpret integer pointers with respect to the program-counter and default-data capabilities.

Hybrid ABI Capability-aware code makes selective use of capability registers and instructions, but can transparently interoperate with MIPS-ABI code when capability arguments or return values are unused. The programmer may annotate pointers or types to indicate that data pointers should be implemented in terms of capabilities; the compiler and linker may be able utilize capabilities in further circumstances, such as for pointers that do not escape a scope, or are known to pass to other hybrid code. They may also use capabilities for other addresses or values used in generated code, such as to protect return addresses or for on-stack canaries. The goal of this ABI is binary compatibility with, where requested by the programmer, additional protection. This is used within hybrid applications or libraries to provide selective protection for key allocations or memory types, as well as interoperability with pure-capability compartments.

Pure-Capability ABI Capabilities are used for all language-level pointers, but also underlying addresses in the run-time environment, such as return addresses. The goal of this ABI is strong protection at significant cost to binary interoperability. This is used for both compartmentalized code, and also pure-capability ("CheriABI") applications.

3.9.2 Object Capabilities

As noted above, the CHERI design calls for two forms of capabilities: capabilities that describe regions of memory and offer bounded-buffer "segment" semantics, and object capabilities that permit the implementation of protected subsystems. In our model, object capabilities are represented by a pair of sealed code and data capabilities, which provide the necessary information to implement a protected subsystem domain transition. Object capabilities are "invoked" using the `CCall` instruction (which is responsible for unsealing the capabilities, performing a safe security-domain transition, and argument passing), followed by `CReturn` (which reverses this process and handles return values).

In traditional capability designs, invocation of an object capability triggered microcode responsible for state management. Initially, we implemented `CCall` and `CReturn` as software exception handlers in the kernel, but are now exploring optimizations in which `CCall` and `CReturn` perform a number of checks and transformations to minimize software overhead. In the longer term, we hope to investigate the congruence of object-capability invocation with message-passing primitives between architectural threads: if each register context represents a security domain, and one domain invokes a service offered by another domain, passing a small number of general-purpose integer and capability registers, then message passing may offer a way to provide significantly enhanced performance.[6] In this view, architectural thread contexts, or register files, are simply caches of thread state to be managed by the processor.

Significant questions then arise regarding rendezvous: how can messages be constrained so that they are delivered only as required, and what are the interactions regarding scheduling? While this structure might appear more efficient than a TLB (by virtue of not requiring objects with multiple names to appear multiple times), it still requires an efficient lookup structure (such as a TCAM).

In either instantiation, a number of design challenges arise. How can we ensure safe invocation and return behavior? How can callers safely delegate arguments by reference for the duration of the call to bound the period of retention of a capability by a callee (which is particularly important if arguments from the call stack are passed by reference)?

How should stacks themselves be handled in this light, since a single logical stack will arguably be reused by many different security domains, and it is undesirable that one domain in execution might 'pop' rights from another domain off of the stack, or reuse a capability to access memory previously used as a call-by-reference argument.

These concerns argue for at least three features: a logical stack spanning many stack frag-

[6] This appears to be another instance of the isomorphism between explicit message passing and shared memory design. If we introduce hardware message passing, then it will in fact blend aspects of both models and use the explicit message-passing primitive to cleanly isolate the two contexts, while still allowing shared arguments using pointers to common storage, or delegation using explicit capabilities. This approach would allow application developers additional flexibility for optimization.

ments bound to individual security domains, a fresh source of ephemeral stacks ready for reuse, and some notion of a do-not-transfer facility in order to prevent the further propagation of a capability (perhaps implemented via a revocation mechanism, but other options are readily apparent). PSOS explored similar notions of propagation-limited capabilities with similar motivations.

Our current software `CCall`/`CReturn` maintains a 'trusted stack' in the kernel address space and provides for reliable return, but it is clear that further exploration is required. Our goal is to support many different semantics as required by different programming languages, from an enhanced C language to Java. By adopting a RISC-like approach, in which traps to a lower ring occur when architecture-supported semantics is exceeded, we will be able to supplement the architectural model through modifications to the supervisor.

3.10 Concluding Notes

3.10.1 Deep Versus Surface Design Choices

In adapting an ISA to implement the CHERI protection model, there are both deeper design choices (e.g., to employ tagged memory and registers) that might span multiple possible applications to an ISA, and more surface design choices reflecting the specific possible integrations (e.g., the number of capability registers). Further, applications to an ISA are necessarily sensitive to existing choices in the ISA – for example, whether and how page tables are represented in the instruction set, and the means by which exception delivery takes place. In general, the following aspects of CHERI are fundamental design decisions that it is desirable to retain in applying CHERI concepts in any ISA:

- Capabilities can be used to implement pointers into virtual address spaces (or physical address spaces for processors without MMUs);
- Tags on registers determine whether they are valid pointers for loading, fetching, or jumping to;
- Tagged registers can contain both data and capabilities, allowing (for example) capability-oblivious memory copies;
- Tags on pointer-sized, pointer-aligned units of memory preserve validity (or invalidity) across loads and stores to memory;
- Tags are associated with physical memory locations – i.e., if the same physical memory is mapped at two different virtual addresses, the same tags will be used;
- Attempts to store data into memory that has a valid tag will atomically clear the tag;
- Capability loads and stores to memory offer strong atomicity with respect to capability values and tags preventing race conditions that might yield combinations of different capability values, or the tag remaining set when a corrupted capability is reloaded;

- Pointers are extended to include bounds and permissions; the "pointer" is able to float freely within (and to varying extents, beyond) the bounds;

- Permissions are sufficient to control both data and control-flow operations;

- Guarded manipulation implements monotonicity: rights can be reduced but not increased through valid manipulations of pointers;

- Invalid manipulations of pointers violating guarded-manipulation rules lead to an exception or clearing of the valid tag, whether in a register or in memory, with suitable atomicity;

- Loads via, stores via, and jumps to capabilities are constrained by their permissions and bounds, throwing exceptions on a violation;

- Capability exceptions, in general, are delivered with greater priority than MMU exceptions;

- Permissions on capabilities include the ability to not just control loading and storing of data, but also loading and storing of capabilities;

- Capability-unaware loads, stores, and jump operations via integer pointers are constrained by implied capabilities such as the Default Data Capability and Program Counter Capability, ensuring that legacy code is constrained;

- If present, the Memory Management Unit (MMU), whether through extensions to software-managed Translation Look-aside Buffers (TLBs), or via page-table extensions for hardware-managed TLBs, contains additional permissions controlling the loading and storing of capabilities;

- Strong C-language compatibility is maintained through definitions of NULL to be untagged, zero-filled memory, instructions to convert between capabilities and integer pointers, and instructions providing C-compatible equality operators;

- Reserved capabilities, whether in special registers or within a capability register file, allow a software supervisor to operate with greater rights than non-supervisor code, recovering those rights on exception delivery;

- A simple capability control-flow model to allow the propagation of capabilities to be constrained;

- Sealed capabilities allow software-defined behaviors to be implemented, along with suitable instructions to (with appropriate authorization) seal and unseal capabilities based on permissions and object types;

- By clearing architecture-defined permissions, and utilizing software-defined permissions, capabilities can be used to represent spaces other than the virtual address space;

- For compressed capabilities, pointers can stray well out-of-bounds without becoming unrepresentable;

- For compressed capabilities, alignment requirements do not restrict common object sizes and do not restrict large objects beyond common limitations of allocators and virtual memory mapping; and

- That through inductive properties of the instruction set, from the point of CPU reset, via guarded manipulation, and suitable firmware and software management, it is not possible to "forge" capabilities or otherwise escalate privilege other than as described by this model and explicit exercise of privilege (e.g., via saved exception-handler capabilities, unsealing, etc).

The following design choices are associated with our specific integration of the CHERI model into the 64-bit MIPS ISA, and might be revisited in various forms in integrating CHERI support into other ISAs (or even with MIPS):

- Whether capability registers are in their own register file, or extended versions of existing general-purpose integer registers, as long as tags are used to control dereferencing capabilities;

- The number of capability registers present;

- How capability-related permissions on MMU pages are indicated;

- How capabilities representing escalated privilege for exception handlers are stored;

- Whether specific capability-related failures (in particular, operations violating guarded manipulation) lead to an immediate exception, or simply clearing of the tag and a later exception on use;

- How tags are stored in the memory subsystem – e.g., whether close to the DRAM they protect or in a partition of memory – as long as they are presented with suitable protections and atomicity up the memory hierarchy;

- How the instruction-set opcode space is utilized – e.g., via coprocessor reservations in the opcode space, reuse of existing instructions controlled by a mode, etc;

- What addressing modes are supported by instructions – e.g., whether instructions accept only a capability operand as the base address, perhaps with immediates, or whether they also accept integer operands via non-capability (or untagged) registers; and

- How capabilities are represented microarchitecturally – e.g., compressed or decompressed if compression is used; if the base and offset are stored pre-computed as a cursor rather than requiring additional arithmetic on dereference; or whether an object-type field is present for non-sealed in-memory representations.

3.10.2 Potential Future Changes to the CHERI Architecture

The following changes have been discussed and are targeted for short-term implementation in the CHERI architecture:

- Define the values of base, length, and offset for compressed capabilities with $\mathrm{e} > 43$, where the formulas for decompressing base and top do not make sense due to bit indexes being out of bounds. This is possible for the default capability (defined to have $length = 2^{64}$, although e is unspecified) and untagged data loaded from memory. One proposed behavior is to treat all untagged compressed capabilities as though they have $base = 0$ and $length = 2^{64}$ for the purposes of the instructions where this matters, namely `CGetBase`, `CGetOffset`, `CIncOffset`, `CGetLen`, `CPtrCmp` and `CSub`. However, there is also a desire that `CSetOffset` should preserve the values of T and B for debugging purposes, where possible.

- Consider re-writing pseudocode in terms of absolute addresses rather than offsets, without changing the semantics. This would eliminate repeated use of **base** + **offset** to mean the address field of the capability; it would also potentially reduce ambiguity such as where **base** is not well defined due to $\mathrm{e} > 43$ as above.

- Provide a separate instruction for clearing the *global* bit on a capability. *Global* is currently treated as a permission, but it is really an information flow label rather than a permission. We may want to allow clearing the *global* bit on a sealed capability, which would be easiest to implement with a separate instruction, as permissions cannot be changed on sealed capabilities.

- Provide multiple orthogonal capability "colors", expanding the local-global features to allow multiple consumers. We have considered in particular the use of colors to: (1) prevent kernel pointers from errantly wandering into userspace memory; (2) prevent user pointers from improperly moving between processes sharing some or all of their virtual address spaces; (3) prevent pointers from improperly flowing between intra-process protection domains; and (4) to prevent stack pointers from being improperly shared between threads. Section D.13 elaborates a more efficient representation for this coloring model, requiring one rather than two bits per color, by virtue of utilizing a new capability type to authorize color management.

- Allow clearing of software-defined permission bits for sealed capabilities rather than requiring a domain switch or call to a privileged supervisor to do this. One way to do this would be to provide a separate instruction for clearing the software-defined permission bits on a sealed capability. The other permission bits on a sealed capability can be regarded as the permissions to access memory that the called protected subsystem will gain when `CCall` is invoked on the sealed capability; these should not be modifiable by the caller. On the other hand, the software-defined capability bits can be regarded as application-specific permissions that the caller has for the object that the sealed capability represents, and the caller might want to restrict these permissions before passing the sealed capability to another subsystem.

- Provide a CFromInt instruction that copies a general-purpose integer register into the **offset** field of a capability register, clearing all the other fields of the capability – including the **tag** bit. This is an architecturally cleaner way to implement casting an *int* to an *intcap_t* than the current approach of CFromPtr of the NULL pointer followed by CSetOffset.

- Provide a variant of CSetBounds that sets imprecise bounds suitable for sealing with CSeal. In the 128-bit representation, the bounds of sealed capabilities have stronger alignment requirements than for unsealed capabilities.

- Introduce a CTestSubset instruction, which would allow efficient testing of whether one capability describes rights that are a subset of another, directly exposing the partial order implied by subset tests in CToPtr, the proposed CBuildCap, etc. This is described in more detail in Section D.18.

- Add versions of CSetOffset and CIncOffset that raise an exception, rather than clearing the tag bit, when the result is not representable. This would assist in debugging, by causing an exception to be raised at the point in the program when the capability became unrepresentable, rather than later on when the capability is dereferenced.

  An alternative implementation (rather than having separate trapping and non-trapping instructions) would be to add a status register that enables the trapping behavior. This is similar to floating point, where the FCSR controls whether a floating point overflow results in an IEEE infinity value or an exception being thrown.

  A cheap tag assertion instruction that can trigger a trap when a tag is lost would allow special compilation modes to improve debuggability by detecting unexpected tag loss sooner.

  If MIPS had a user status register, a tag-loss bit could be set implicitly on tag clear, allowing intermittent conditional-branch instructions to detect and handle loss.

- Add a version of CUnseal that returns NULL, rather than raising an exception, if the security checks fail. A common use case for CUnseal is that a protected subsystem is passed a sealed capability by an untrusted (possibly malicious) caller, and the callee uses CUnseal to unseal it. It would be quicker for the callee to use a non-trapping CUnseal and then check that the result is not NULL, rather than either (a) catching the exception in the case that the untrusted caller has passed a bad capability; or (b) checking that the capability is suitable for unsealing before attempting to unseal it.

- Add a CGetPCCIncOffset instruction that is similar to CGetPCCSetOffset, except that it increments the offset instead of setting it. This instruction could be useful (for example) when using the variant of CCall that doesn't push a return capability on to the trusted system stack; CGetPCCIncOffset would provide a convenient way to construct a capability for the return address.

- Add instructions for copying non-capability data from a capability register into a general-purpose integerregister. A use case is when a function is called with a parameter whose type is the union of a pointer and a non-pointer type, such as an int. This parameter must be passed in a capability register, because the tag needs to be preserved when it holds a capability. If the body of the function accesses the non-capability branch of the union, it needs to get the non-capability bits out of the capability register and into a general purpose register. This can be done by spilling the capability register to the stack and then reading it back into a general-purpose integer register, but a register to register copy would be faster. (We need to investigate whether this happens often enough for the optimization to be worthwhile).

With compressed 128-bit capabilities, two instructions are needed (to get the upper and lower 64 bits of the capability register). With uncompressed 256-bit capabilities, 4 instructions are needed.

- Add an instruction that is like `CSetBounds` except that it sets **base** to the current **base** + **offset** and the new length is the old **length** − **offset** (i.e., the upper bound is unchanged). A question that needs to be resolved: what if the requested bounds cannot be represented exactly? The use case for this instruction is when its desired to move up the **base** of the capability, without needing to extra instructions to explicitly calculate the new **length**.

- Add an instruction that returns the alignment requirement for memory regions of a particular size. Having an instruction for this would avoid the need for memory allocators to know the details of the compression scheme.

- Swapping and virtual-machine migration require that tags be stripped from capabilities as memory is serialized, and that tags be reattached to capabilities as memory is restored. Section D.5 describes a set of experimental instructions that improve the efficiency of these operations, but also avoid the need for software to directly inspect and interact with the in-memory representation of capabilities.

- If one wishes to scan memory to *revoke* capabilities, being able to skip over contiguous spans of non-capabilities in mapped memory may greatly accelerate the process and reduce DRAM traffic. An instruction for this purpose is proposed in Section D.4.

- Add instructions for loading/storing floating point registers via a capability. This is not just a performance optimization, but also simplifies register allocation in a compiler: storing a float by moving it to an integer register and then storing the integer register to memory needs an integer register that isn't being used for something else.

The following changes have been discussed for longer-term consideration:

- Allow `CReturn` to accept code/data capability arguments, which might be ignored for the time being – or simply make `CReturn` a variation on `CCall`.

- Introduce support for a userspace exception handler for `CCall` and `CReturn`, allowing more privileged user code (rather than kernel code) to implement the semantics of exception-based domain switching, provide memory for use in trusted stacks (if any), and so on. This would allow application environments to provide their own object models without needing to depend on highly privileged kernel code.

- Introduce finer-grained permissions (or new capability types) to express CPU privileges in a more granular way. For example, to allow management of interrupt-related CPU features without authorizing manipulation of the MMU.

- Introduce a control-flow-focused "immutable" (or, more accurately, "nonmanipulable") permission bit, which would prevent explicit changes to the bounds or offset, while still allowing the offset to be implicitly changed if the capability is placed in execution (i.e., is installed in **PCC**. This would limit the ability of attackers, in the presence of a memory re-use bug, to manipulate the offset of a control-flow capability in order to attempt a code re-use exploit. Some care would be required – e.g., to ensure that it was easy and efficient to update the value in the offset during OS exception handling, where it is common to adjust the value of the **PC** forward after emulating an instruction.

- Introduce further hardware permissions, such as physical-address load and store permissions, which would allow non-virtual-address interpretations of capabilities, bypassing the MMU. These might be appropriate for use by kernels, accelerators, and DMA engines there physical addresses (or perhaps hypervisor-virtualised physical addresses) offer great efficiency or improved semantics.

- Consider whether any further instructions require variants that accept immediate values rather than register operands. Some already exist (e.g., when setting bounds or offsets, to avoid setting up integer register operands) but it may also be worth adding others. For example, if it transpires that permission-masking is a common operation in some workloads, a new `CAndPermImm` could be added.

- Capability linearity, in which the architecture prevents duplication of a capability, might offer stronger invariants around protection-domain crossing. Section D.10 describes an experimental proposal for how this might be implemented.

- Today, a uniform set of capability roots are provided: **PCC**, **DDC**, **KCC**, and possibly other special capability registers, are all preinitialised to grant all permissions across the full address space. This is a simple model that is easy to understand, but implies that certain efficiencies cannot be realized in the in-memory capability representation – for example, although sealing, CIDs, and memory access refer to different namespaces, we cannot efficiently encode the lack of overlap to reduce the number of bits in capability representation.

  Moving to multiple independent roots originating in different special registers would allow these efficiencies to be realized. For example, by having three different capability roots – memory capabilities (with only virtual-address permissions), sealing capabilities (with only sealing and unsealing permissions), and compartment capabilities (with only CID permissions).

  A further root could be achieved by introducing a distinction between **PCC** authorizing use of the privileged ISA (e.g., TLB manipulation) and a special register used for this purpose. If a new "system authorization special register" were to be added, then a further System_Access_Registers-only root could be introduced, and derived capabilities could be installed into the special register when those privileges are required; a NULL capability could be installed when not in order to prevent use.

- Introduce capability-extended versions of virtually indexed cache-management instructions. This is important in order to allow compartmentalized DMA-enabled device drivers to force write-back. Support for invalidate, however, remains challenging, as invalidate instructions could cause memory to "rewind", for example rolling back memory zeroing. This may require some changes around device drivers to avoid the need for direct use of invalidation instructions by unprivileged device drivers, and is a topic for further research.

Chapter 4

The CHERI-MIPS Instruction-Set Architecture

Having considered the software-facing semantics and architecture-neutral aspects of the CHERI protection model in previous chapters, we now turn to elaborating CHERI capabilities within a specific RISC architecture: 64-bit MIPS. Wherever possible, CHERI-MIPS implements architecture-neutral concepts as described in Chapter 3. In addition to the mechanics of defining specific instructions and choices about whether a new register file is used (vs. extending the existing integer register file), MIPS differs substantially from other RISC ISAs in several key areas – especially in its use of a software-managed Translation Look-aside Buffer (TLB), and in the details of its exception mechanism. In those cases, we necessarily take a MIPS-oriented perspective. This chapter specifies the following aspects of CHERI-MIPS:

- Architectural capabilities
- The capability register file
- Special capability registers
- Capability-aware instructions
- Capability state on CPU reset
- Exception handling and capability-related exceptions
- Changes to MIPS ISA processing
- Changes to the Translation Look-aside Buffer (TLB)
- Protection-domain transition
- Capability register conventions and the Application Binary Interface (ABI)

The chapter finishes with a discussion of potential future directions for the CHERI-MIPS ISA. Detailed descriptions of specific capability-aware instructions can be found in Chapter 7.

4.1 The CHERI-MIPS ISA Extension

CHERI-MIPS extends 64-bit MIPS with a new *tagged capability register file* able to hold both valid capabilities and data, and *tagged memory* to distinguish and protect capabilities. It adds new *capability instructions* that inspect, manipulate, and use *capability registers*. CHERI-MIPS also modifies certain existing MIPS architectural behaviors, such as relating to existing MIPS memory accesses, the program counter, exception delivery, and the software-managed TLB. New instructions are added using the coprocessor-2 portion of the MIPS opcode space, which is intended for local vendor extensions. Despite this MIPS-originated nomenclature, microarchitectural implementations of CHERI will be tightly integrated with the main pipeline, rather than as a separate "coprocessor". Wherever possible, CHERI-MIPS inherits its behavior from the architecture-neutral specification found in Chapter 3; however, in some cases must extend it – e.g., by defining MIPS-specific aspects of architectural privilege.

4.2 Capabilities

In CHERI-MIPS, capabilities may be held in a dedicated capability register file, where they can be manipulated or dereferenced using capability coprocessor instructions, in a set of special capability registers, and in tagged memory. Capabilities in the capability register file may be used as operands to capability instructions that retrieve or modify capability contents, namely, load and store instructions, and control-flow instructions. Capability addresses used for load, store, and instruction fetch are always interpreted as virtual addresses. Special capability registers are accessed via new read- and write-register instructions. Guarded manipulation and tagged memory enforce capability unforgeability, capability monotonicity, provenance validity, and capability integrity.

4.2.1 Capability Permissions

Architecture-neutral capability permission bits are described in Section 3.3.1; the following permissions have CHERI-MIPS-specific interpretations:

Access_System_Registers Allow access to **EPCC**, **ErrorEPCC**, **KDC**, **KCC**, **KR1C**, **KR2C** and **capcause** when this permission is set in **PCC**. Also authorize access to kernel features such as the TLB, CP0 registers, and system-call return (see Section 4.8).

4.2.2 Capability Flags

In CHERI-MIPS, the **flags** field has size 0.

4.3 Capability Registers

CHERI supplements the 32 general-purpose per-hardware-thread integer registers provided by the MIPS ISA with 32 additional general-purpose capability registers. Where general-purpose integer registers describe data values operated on by a software thread, capability registers describe its instantaneous rights within an address space. A thread's capabilities potentially imply a larger set of rights (loadable via held capabilities) which may notionally be considered as the protection domain of a thread.

There are also several special capability registers associated with each architectural thread, including a memory capability that corresponds to the instruction pointer, and capabilities used during exception handling. This is structurally congruent to implied registers and system control coprocessor (CP0) registers found in the base MIPS ISA. Special instructions are used to move special capabilities in and out of general-purpose capability registers.

Unlike general-purpose integer registers, capability registers are structured, consisting of a 1-bit tag and a 128-bit or 256-bit set of architectural fields with defined semantics and constrained values. As described in the previous chapter, the in-memory representation of a capability may be far smaller as a result of compression techniques; we define CHERI-MIPS variants with both 128-bit and 256-bit capabilities. Capability instructions retrieve and set these fields by moving values in and out of general-purpose integer registers, enforcing constraints on field manipulation.

4.4 The Capability Register File

In CHERI-MIPS, the general-purpose capability register file is distinct from the general-purpose integer register file. Table 4.4 illustrates capability registers defined by the capability coprocessor. CHERI-MIPS defines 31 general-purpose capability registers, which may be named using most capability register instructions. These registers are intended to hold the working set of rights required by in-execution code, intermediate values used in constructing new capabilities, and copies of capabilities retrieved from **EPCC** and **PCC** as part of the normal flow of code execution, which is congruent with current MIPS-ISA exception handling via coprocessor 0. In addition to the 31 general-purpose capability registers, **C0** is a constant NULL capability[1]. The special capability registers (other than **PCC**) can be read using the `CReadHwr` instruction and set using the `CWriteHwr` instruction, subject to suitable permission.

Each capability register also has an associated tag indicating whether it currently contains a valid capability. Any load, store, or instruction fetch via an invalid capability will trap.

4.5 Capability-Aware Instructions

Per Section 3.6, CHERI-MIPS introduces a number of new capability-related instructions. Many are "portable" CHERI instructions, but others are MIPS-specific either in terms of augmenting the existing instruction set (congruent capability-based jump, load, and store instructions), or to address MIPS-specific interactions with CHERI (e.g., as relates to exception handling). CHERI-MIPS introduces the following control-flow and memory-access instruction classes:

[1] For some instructions, specifying a register operand of 0 will utilize **DDC** rather than **C0**. This can reduce the instruction count for certain sequences such as capability loads and stores in a hybrid compiler mode.

Register(s)	Description
CNULL (**C0**)	A capability register that returns the NULL value when read. Writes to **CNULL** are ignored.
C1...**C25**	General-purpose capability registers referenced explicitly by capability-aware instructions
IDC (**C26**)	Invoked data capability: the capability that was unsealed at the last protected procedure call
C27...**C31**	General-purpose capability registers referenced explicitly by capability-aware instructions

Special Register(s)	Description
PCC	Program counter capability (**PCC**): the capability through which **PC** is indirected by the processor when fetching instructions.
DDC	Capability register through which all non-capability load and store instructions are indirected. This allows legacy MIPS code to be controlled using the capability coprocessor.
KR1C	A capability reserved for use during kernel exception handling.
KR2C	A capability reserved for use during kernel exception handling.
KCC	Kernel code capability: the code capability moved to **PCC** when entering the kernel for exception handling.
KDC	Kernel data capability: the data capability containing the security domain for the kernel exception handler.
EPCC	Capability register associated with the exception program counter (**EPC**) required by exception handlers to save, interpret, and store the value of **PCC** at the time the exception fired.
ErrorEPCC	Capability register associated with the error exception program counter (**ErrorEPC**) that is used on exception return if CP0.Status.ERL is set. The CHERI prototype does not actually support any exception types (e.g. cache error) that require **ErrorEPC** but it is supported for consistency with MIPS.

Table 4.1: Capability registers defined by the capability coprocessor. See `CReadHwr` (7.4) and `CWriteHwr` (7.4) for details on special capability registers.

Load or store via a capability These instructions access memory indirected via an explicitly named capability register, and include a full range of access sizes (byte, half word, word, double word, capability), sign extension for loads, and load-linked/store-conditional variations implementing atomic operations: `CL[BHWDC][U]`, `CS[BHWDC]`, `CLL[BHWDC]`, and `CSC[BHWDC]`.

Capability jumps These instructions jump to an explicitly named capability register, setting the program-counter capability to the value of the capability operand: `CJR` and `CJALR`. These correspond in semantics to the MIPS `JR` jump, used for function returns, and `JALR`, used for function calls, but constrained by the properties of the named capability including permissions, bounds, validity, and so on.

CHERI-MIPS introduces the following further capability-aware instructions to cater to architecture-specific aspects of the MIPS ISA:

Conditional move The `CMovN` and `CMovZ` instructions conditionally move a capability from one register to another, permitting conditional behavior without the use of branches. These support efficient hybrid code, in which use of integer pointers and capabilities are intermixed.

Retrieve program-counter capability These instructions retrieve the architectural program-counter capability, and optionally modify its offset for the purposes of **PCC**-relative addressing: `CGetPCC` and `CGetPCCSetOffset`.

Exception handling The `CGetCause` and `CSetCause` instructions set and get capability-related exception state, such as the cause of the current exception.

Get and set special capability registers The `CReadHWR` and `CWriteHWR` instructions get and set special capability registers such as **DDC**, **EPCC**, **KDC**, and **KCC**.

4.6 Capability State on CPU Reset

Section 3.5 requires that capability root registers be initialized to offer full capability rights; all other registers are initialized to NULL. The capability roots in CHERI-MIPS are **PCC**, **KCC**, **EPCC**, **ErrorEPCC**. These values allow capability-unaware code to load and store data with respect to the full virtual address space, and for exception handling to operate with full rights. All other general-purpose and special capability registers should contain the NULL value.

In our CHERI-MIPS hardware prototype, all tags in physical memory are initialized to 0, ensuring that there are no valid capabilities in memory on reset. This is not strictly required: the firmware, hypervisor, or operating system can in principle ensure that tags are cleared on memory before it is exposed to untrustworthy software, in much the same way that they will normally ensure that memory is cleared to prevent data leaks before memory reuse.

4.7 Exception Handling

As described in Section 3.8.2, CHERI adopts and extends the existing exception model of its host architecture. CHERI-MIPS retains the same general model found in MIPS, with modest extensions to provide exception handlers with both code and data capabilities, as well as allow the interrupted capability-extended context to be saved and restored.

4.7.1 Exception-Related Capabilities

MIPS exception handling saves interrupted state, transfers control to an exception vector, and also grants supervision privilege in the ring model. CHERI-MIPS extends this model so that the exception handler also gains access to additional code and data capabilities (**KCC** and **KDC**) to authorize its execution.

In order to preserve the full interrupted **PCC**, **EPC** has been extended from a special integer register to a special capability register, **EPCC**. Accessing the existing MIPS CP0 **EPC** special register in effect accesses the **offset** field of **EPCC**.

When an exception occurs, the victim **PCC** is copied to **EPCC** so that the exception may return to the correct address, and **KCC**, leaving aside its **offset** field, which will be set to the appropriate MIPS exception-vector address, is moved to **PCC** to grant execution rights for kernel code.

Access to **KDC**, the Kernel Data Capability, is authorized by System_Access_Registers, and can be used to reach exception-handler data. **KCC** will normally be configured to grant this access, if **KDC** is used. Exception handlers making use of legacy loads and stores will most likely install **KDC** in **DDC**. **DDC**, as with other special registers, will need to be saved and restored to implement full context switching.

When an exception handler returns with `ERET`, **EPCC** (or **ErrorEPCC** if CP0.Status.ERL is set), possibly after having been updated by the software exception handler, is moved into **PCC**.

4.7.2 Exception Temporary Special Registers

In the MIPS ABI, two general-purpose integer registers are reserved for use by exception handlers: `$k0` and `$k1`. In earlier CHERI-MIPS revisions, we made a similar design choice for the capability register file, reserving two capabilities for exception-handling use. In more recent revisions, we have defined two special capability registers, **KR1C** and **KR2C**, which are accessible only when Access_System_Registers is present on **PCC**, and can be used to hold values during exception handling – for example, to temporarily save the value of a general-purpose capability register so that it can be used to hold **KDC** during context save.

4.7.3 Capability-Related Exceptions and the Capability Cause Register

CHERI-MIPS implements a **capcause** register that gives additional information about the causes of capability-related exceptions, as described in Section 3.8.2. The `CGetCause` instruction can be used by an exception handler to read the **capcause** register. Software can use `CSetCause` to set *ExcCode* to either an architectural or software-defined value. `CGetCause` and `CSetCause` will raise an exception if **PCC**.**perms**.*Access_System_Registers* is not set, allowing control over whether unprivileged code can access **capcause**. When an attempted operation, prohibited by the capability mechanism, triggers an exception, the *ExcCode* field within the **cause** register of coprocessor 0 are set to 18 (*C2E*, coprocessor 2 exception), and **capcause** will be set to a constant from Table 3.8.2.

4.7.4 Exceptions and Indirect Addressing

If an exception is caused by the combination of the values of a capability register and a general-purpose integer register (e.g., if an expression such as `clb t1, t0(c0)` raises an exception because the offset `t0` is trying to read beyond `c0`'s length), the number of the capability register (not of the general-purpose integer register) will be stored in **capcause**.*RegNum*.

4.7.5 Capability-Related Exception Priority

In CHERI-MIPS, capability exceptions have a lower priority than Reset, Soft Reset, and Non-Maskable Interrupt (NMI) but higher priority than all other exception types. In particular, they have a priority higher than address errors (e.g., alignment exceptions, reported by `AdEL` and `AdES`) and TLB exceptions, as capability processing for addresses occurs logically "before" dereference of a virtual address. With respect to the capability cause register provided via **capcause**, CHERI-MIPS implements the exception priority scheme described in Section 3.8.2.

4.7.6 Implications for Pipelining

MIPS is unusual as an architecture in that the privileged supervisor mode can explicitly see the effects of pipeline hazards. System software must issue suitable NOPs and barriers to ensure that potentially confusing (and even insecure) implications of pipelining are not visible to application software. For example, MIPS normally requires a specific number of NOP instructions follow any writes to TLB-related registers before the effects of those writes are visible to software. In general, CHERI does not change this behavior: general-purpose capability registers experience no visible pipelining effects in normal use; and where exiting pipelining effects exist, such as in accessing **EPC**, similar assumptions should be made about capability-extended registers, such as **EPCC**.

4.8 Changes to MIPS ISA Processing

The following changes are made to the behavior of instructions from the standard MIPS ISA when a capability coprocessor is present:

Instruction fetch The MIPS-ISA program counter (**PC**) is extended to a full program-counter capability (**PCC**), which incorporates the historic **PC** as **PCC.offset**. Instruction fetch is controlled by the *Permit_Execute* permission, as well as bounds checks, tag checks, and a requirement that the capability not be sealed. Failures will cause a coprocessor 2 exception (*C2E*) to be thrown. In general, instructions that set **PCC** (such as CJR) prohibit setting invalid values with the notable exception of ERET. This is because ERET will typically be used in the kernel's exception return routine where triggering an exception could be difficult to recover from. If an ERET occurs with an invalid **EPCC** the relevant exception is raised on the following instruction fetch, just as returning to an invalid **PC** would trigger a TLB fault on the next instruction fetch in MIPS. Note that **EPCC** for the resulting exception will not have changed. This is particularly important if **EPCC** was sealed.

If an address exception occurs during instruction fetch (e.g., AdEL, or a TLB miss) then *BadVAddr* is set equal to **PCC**.**base** + **PCC**.**offset**, providing the absolute virtual address rather than a **PCC**-relative virtual address to the supervisor, avoiding the need for capability awareness in TLB fault handling.

Load and Store instructions Standard MIPS load and store instructions are interposed on by the *default data capability*, **DDC**. Addresses provided for load and store will be transformed and bounds checked by **DDC**.**base**, **DDC**.**offset**, and **DDC**.**length**. **DDC** must have the appropriate permission (*Permit_Store* or *Permit_Load*) set, the full range of addresses covered by the load or store must be in range, **DDC**.**tag** must be set, and **DDC** must not be sealed (i.e., **DDC**.**otype** must be $2^{64}-1$). Failures will cause a coprocessor 2 exception (*C2E*) to be thrown. As with instruction fetch, *BadVAddr* values provided to the supervisor will be absolute virtual addresses, avoiding the need for capability awareness in TLB fault handling.

Standard MIPS load and store instructions will raise an exception if the value loaded or stored is larger than a byte, and the virtual address is not appropriately aligned. With the capability coprocessor present, this alignment check is performed after adding **DDC**.**base**. (**DDC**.**base** will typically be aligned, so the order in which the check is performed will often not be visible. In addition, CHERI1 can be built with an option to allow unaligned loads or stores as long as they do not cross a cache line boundary).

Floating-point Load and Store instructions If the CPU is configured with a floating-point unit, all loads and stores between the floating-point unit and memory are also relative to **DDC**.**base** and **DDC**.**offset**, and are checked against the permissions, bounds, tag, and sealed state of **DDC**.

Jump and branch instructions If the target is out-of-bounds in relation to **PCC**, a coprocessor 2 exception (*C2E*) will be thrown. The *RegNum* field of **capcause** will indicate a **PCC** exception, and *ExcCode* will indicate a Length Violation. **EPC** and **EPCC** will point to the branch instruction and not the branch target (as in previous CHERI revisions). This is better for debugging and removes the need for a special case for handling the situation where EPCC is not representable due to capability compression.

Jump and link register After a JALR instruction, the return address is relative to **PCC**.**base**.

Exceptions The MIPS exception program counter (**EPC**) is extended to a full exception program-counter capability (**EPCC**), which incorporates the historic **EPC** as **EPCC**.**offset**. If an exception occurs while CP0.Status.EXL is false, **PCC** will be saved in **EPCC** and the program counter will be saved in **EPCC.offset** (also visible as **EPC**). If CP0.Status.EXL is true, then **EPCC** and **EPC** are unchanged. (In the MIPS ISA, exceptions leave **EPC** unchanged if CP0.Status.EXL is true). After saving the old **PCC** the contents of the *kernel code capability* (**KCC**), excluding **KCC.offset**, are moved into **PCC**. **PC** (and **PCC.offset**) will be set so that **PCC.base** + **PC** is the exception vector address normally used by MIPS. This allows the exception handler to run with the permissions granted by **KCC**, which may be greater than the permissions granted by **PCC** before the exception occurred.

On return from an exception (`ERET`), **PCC** is restored from **EPCC** (or **ErrorEPCC** if Status.ERL is set in CP0). The program counter is restored from the **offset** field. This allows exception handlers that are not aware of capabilities to continue to work on a CPU with the CHERI-MIPS extensions. If **EPCC** is not appropriate for execution, the target will throw the relevant exception in response to an unusable **PCC**. Similarly, the result of an exception or interrupt is **UNDEFINED** if **KCC.tag** is not set, **KCC** is sealed, or **KCC** does not have the execute permission.

The legacy MIPS instructions `DMFC0` and `DMTC0` can be used to read or write the **offset** of **EPCC** and **ErrorEPCC** as **EPC** and **ErrorEPC** respectively. Note that we must prohibit modification of sealed capabilities and, if capability compression is being used (see Section 3.4.4), cope with the possibility of an unrepresentable result. An attempt to modify a sealed **EPCC** or **ErrorEPCC** using `DMTC0` results in the tag being cleared. If **EPC** is set so far outside the bounds of **EPCC** that the bounds would no longer be representable, then the tag is cleared and other fields are set as per `CSetOffset`. An unrepresentable **EPCC** cannot occur as a result of an exception because the **PC** should always be within the bounds of **PCC** or nearly in bounds: branches outside **PCC** bounds should throw an exception on the branch instruction, and if execution runs off the end of **PCC** then the resulting **EPCC** will have an offset just 4 more than the top of **PCC** so is guaranteed to be representable.

CP0, TLB, CACHE, and ERET privileges The set of MIPS privileges normally reserved for use only in kernel mode, including the ability to read and write CP0 control registers (using `MFC0`, `MTC0`, `DMFC0`, and `DMTC0`), manage the TLB (using `TLBR`, `TLBWI`, `TLBWR`, and `TLBP`), perform `CACHE` operations that could lead to data loss or rollback of stores, and use the `ERET` exception-return instruction, is available only if **PCC** contains the Access_System_Registers permission AND the CPU is running in kernel mode. This permits capability sandboxes to be used in kernel mode by preventing them from being subverted using the TLB.

Other KSU-controlled mechanisms Despite the Access_System_Registers permission controlling use of privileged ISA features, absence of the bit does not change the behavior of the MIPS ISA with respect to other KSU/EXL-related mechanisms. For example, the value present in the bit does not affect any of the following: selection of the TLB miss handler to use; the KSU bits used to select the kernel, supervisor, or user virtual address space used in TLB lookup; the KSU bits reported in the *XContext* register; or the automatic setting and clearing of the EXL flag on exception entry and return. Memory capabilities are used to constrain the use of memory within kernel or supervisor compartments, rather than the ring-based MIPS segmentation mechanism, which is unaffected by the Access_System_Registers permission.

<table>
<tr><td>31</td><td></td><td></td><td></td><td></td><td></td><td>0</td></tr>
<tr><td>S</td><td>L</td><td colspan="5">0</td></tr>
<tr><td colspan="2">0</td><td>PFN</td><td>C</td><td>D</td><td>V</td><td>G</td></tr>
</table>

Table 4.2: EntryLo Register

4.9 Changes to the Translation Look-aside Buffer (TLB)

CHERI-MIPS implements adaptations to virtual-memory support following the model described in Section 3.8.3. As MIPS provides a software-managed TLB, changes are primarily with respect to TLB management instructions (which now require the Access_System_Registers permission on **PCC**) and TLB-entry contents.

Two new permission bits have been added to the MIPS EntryLo portion of each TLB entry to indicate whether a given memory page is configured to have capabilities loaded or stored (see Figure 4.2). This functionality can be used in a variety of ways to control and track capability use within a virtual address space. For example, mapping pages without PTE_SC and/or PTE_LC can be used to prevent sharing of tagged capability values between address spaces where capabilities might have different interpretations. PTE_SC could also be used to implement a "capability dirty bit" in the TLB handler, tracking which pages have been used to store capabilities, perhaps for the purposes of efficient garbage collection or revocation.

Load Capability (PTR_LC) If this bit is set then capability loads are disabled for the page. If the CLC instruction is used on a page with the L bit set, and the load succeeds, the value loaded into the destination register will have its tag bit cleared, even if the tag bit was set in memory.

Store Capability (PTR_SC) IF this bit is set, capability stores are disabled for the page. If the CSC instruction is used on a page with the S bit set, and the capability register to be stored has the tag bit set, then a CP2 exception will be raised, with **capcause** set to 0x9 (MMU prohibits store capability). If the capability register to be stored does not have the tag bit set (i.e., it contains non-capability data), then this exception will not be raised, and the store will proceed.

As with other TLB-related exceptions, BadVAddr will be set to the absolute virtual address that has triggered the fault, and EntryHi will also be set accordingly.

4.10 Protection-Domain Transition with CCall and CReturn

Cross-domain procedure calls are implemented using the CCall instruction, which provides access to controlled non-monotonicity for the purposes of a privileged capability register-file transformation and memory access. The instruction accepts two capability-register operands, which represent the sealed code and data capability describing a target protection domain. CCall checks that the two capabilities are valid, that both are sealed, that the code capability is executable, that the data capability is non-executable, and that they have a matching object type. In addition to a pair of sealed capability-register operands, CCall accepts a *selector* operand that determines which of two domain-transition semantics will be used:

Selector 0 - exception-handler semantics In this semantic, an exception is thrown, with control transferred to the kernel code capability for the purposes of any required capability register-file and memory accesses. For example, the operating-system kernel might implement a “trusted stack” to track the “caller” **PCC** and **IDC** for the purposes of later restoring control, and arrange for the sealed operand capabilities to be installed in **PCC** and **IDC** on exception return via ERET. Other operations might include argument validation (e.g., to ensure that non-global capabilities are not passed across domain transitions), or register clearing (e.g., to ensure that non-argument registers do not leak information from the caller to the callee). A new dedicated exception vector is used, in a style similar to the dedicated TLB miss exception vector on MIPS, so as to avoid overhead arising from adding new code to existing exception vectors (see Section 4.10.1).

Selector 1 - jump-like semantics In this semantic, the sealed code and data capabilities are unsealed by the instruction, and placed in **PCC** and **IDC**, with control transferred directly to the target code capability. A programming-language or concurrent programming-framework runtime might arrange that all sealed code capabilities point to a message-passing implementation that proceeds to check argument registers or clear other registers, switching directly to the target domain via a further CJR, or returning to the caller if the message will be delivered asynchronously.

A further instruction CReturn is provided that triggers an exception in a similar manner to CCall, but without capability operand checks. A different capability cause register value allows software to distinguish CCall from CReturn.

Voluntary protection-domain crossing – i.e., not triggered by an interrupt – will typically be modeled as a form of function invocation or message passing by the operating system. In either case, it is important that function callers/callees, message senders/recipients, and the operating system itself, be constructed to protect themselves from potential confidentiality or integrity problems arising from leaked or improperly consumed general-purpose integer registers or capabilities passed across domain transition. On invocation, callers will wish to ensure that non-argument registers, as well as unused argument registers, are cleared. Callees will wish to receive only expected argument registers. Similarly, on return, callees will wish to ensure that non-return registers, as well as unused return registers, are cleared. Likewise, callers will wish to receive back only expected return values. In practice, responsibility for this clearing lies with multiple of the parties: for example, only the compiler may be aware of which argument registers are unused for a particular function, whereas the operating system or message-passing routine may be able to clear other registers. Work performed by the operating system as a trusted intermediary in a reliable way may be usefully depended on by either party in order to prevent duplication of effort. For example, both caller and callee can rely on the OS to clear non-argument registers on call, and non-return registers on return, allowing that clearing to occur exactly once during in an exception handler (selector 0) or an exception-free message-passing routine (selector 1). Efficient register clearing instructions (e.g., CClearRegs) can also be used to substantially accelerate this process.

In CHERI, the semantics of secure message passing or invocation are software defined, and we anticipate that different operating-system and programming-language security models might handle these, and other behaviors, in different ways. For example, in our prototype CheriBSD implementation, the operating-system kernel maintains a "trusted stack" onto which values are pushed during invocation, and from which values are popped on return. Over time, we anticipate providing multiple sets of semantics, perhaps corresponding to less synchronous domain-transition models, and allowing different userspace runtimes to select (or implement) the specific semantics their programming model requires. This is particularly important in order to provide flexible error handling: if a sandbox suffers a fault, or exceeds its execution-time budget, it is the OS and programming language that will define how recovery takes place, rather than the ISA definition.

4.10.1 CCall Selector 0 and CReturn Exception Handling

`CCall` selector 0 and `CReturn` unconditionally throw exceptions when executed. However, this can happen in one of two ways:

1. One of more checks performed by `CCall` on its sealed capability operands may fail, causing a *C2E* exception to be thrown, a suitable **capcause** value for the error to be set, and the general-purpose exception-handler vector to execute.

2. All checks performed by `CCall` or `CReturn` pass, causing a *C2E* exception to be thrown, **capcause** to be set to *Call Trap* or *Return Trap*, and a dedicated protection-domain transition vector to execute. This vector is 0x100 above the general-purpose exception handler, and as with the similar TLB miss vector, allows performance overhead to be minimized through the use of a specialized fast-path exception handler.

The checks performed automatically by `CCall` allow software to avoid substantial overhead on every transition, and include checking that tag bits and object types of passed code and data capabilities are suitable. If one or more checks fail, then a suitable exception code for the failure, such as *Tag Violation*, *Sealed Violation*, or *Type Violation*, will be set instead. This design balances a desire for a flexible software implementation with the performance benefits of parallel checking in hardware.

4.11 Capability Register Conventions / Application Binary Interface (ABI)

All ABIs implement the following capability register reservations for calls within a protection domain (i.e., ordinary jump-and-link-register / return instructions):

- **C1–C2** are caller-save. During a cross-domain call, these are used to pass the **PCC** and **IDC** values, respectively. In the invoked context, they are always available as temporaries, irrespective of whether the function was invoked as the result of a cross-domain call.

- **C3–C10** are used to pass arguments and are not preserved across calls.
- **C11–C16** and **C25** are caller-save registers.
- **C17–C24** are callee-save registers.

In all ABIs, the following convention also applies:

- **C3** optionally contains a capability returned to a caller (congruent to MIPS **$v0**, **$v1**).

The pure-capability ABI, used within compartments or for pure-capability ("CheriABI“) applications, implements the following further conventions for capability use:

- **C11**, in the pure-capability ABI, contains the stack capability (congruent to MIPS **$sp**).
- **C12**, in the pure-capability ABI, contains the jump register (congruent to MIPS **$t9**).
- **C17**, in the pure-capability ABI, contains the link register (congruent to MIPS **$ra**).

When calling (or being called) across protection domains, there is no guarantee that a non-malicious caller or callee will abide by these conventions. Thus, all registers should be regarded as caller-save, and callees cannot depend on caller-set capabilities for the stack and jump registers. Additionally, all capability registers that are not part of the explicit argument or return-value sets should be cleared via explicit assignment or via the `CClearHi` and `CClearLo` instructions. This will prevent leakage of rights to untrustworthy callers or callees, as well as accidental use (e.g., due to a compiler bug). Where rights are explicitly passed between domains, it may be desirable to clear the global bit that will (in a suitably configured runtime) limit further propagation of the capability. Similar concerns apply to general-purpose integer registers, or capability registers holding data, which should be preserved by the caller if their correct preservation is important, and cleared by the caller or callee if they might leak sensitive data. Optimized clearing instructions `ClearHi` and `ClearLo` are available to efficiently clear general-purpose integer registers.

4.12 Potential Future Changes to the CHERI-MIPS ISA

The following changes have been discussed and are targeted for short-term implementation in the CHERI-MIPS architecture:

- Develop (modest) changes to `CCall` selector 1 so as to ensure that the two sealed capabilities are in expected registers for the callee. With selector 0, our extensions to MIPS CP0 allow the exception handler to check the trapping instruction's encoding to ensure **C1** and **C2** are passed (or other ABI choices), ensuring reliable access to sealed versions of the caller's operands – giving access to the object type, for example. In userspace, we do not have access to MIPS CP0, and it would be preferable to find some other way to reliably pass not just the unsealed versions, but also caller sealed versions, to the callee.

One way to accomplish this would be to complete our shift of reserved capability registers away from the main capability register file into a bank of named system registers, allowing allowing the two input registers and two output sealed registers to be placed in well-known locations for access by the callee. This would also avoid encoding use of **IDC** in the instruction definition, which is similarly undesirable.

- Another feature consideration for `CCall` selector 1 is the need for the caller to construct its own return **PCC** value. In more conventional `CJALR` jump-and-link-register instructions, the call instruction itself saves the current **PCC**. While placing yet more obligation on the architecture to write to registers, this would avoid substantial caller work (typically materialization of a return address provided by the linker, `CGetPCC`, and `CSetOffset` to construct a suitable return capability).
- Add instructions for loading/storing floating point registers via a capability. This is not just a performance optimization, but also simplifies register allocation in a compiler: storing a float by moving it to an integer register and then storing the integer register to memory needs an integer register that isn't being used for something else.

The following changes have been discussed for longer-term consideration:

- Consider further the effects of combining general-purpose integer and capability register files, which would avoid adding a new register file, but make some forms of ABI compatibility more challenging.

Chapter 5

The CHERI-RISC-V Instruction-Set Architecture (Draft)

In this chapter we propose a draft application of the CHERI protection model to the RISC-V ISA. We build on our experience designing CHERI-MIPS, revisiting key design choices and also adapting the model anew to a similar but distinct instruction set with a more contemporary set of architectural features. We choose to design CHERI-RISC-V as a parameterizable instruction set that includes several key design points that allow us to evaluate both microarchitectural and architectural implications via side-by-side experiments. This specification is a draft in that it remains a work-in-progress and has not yet been fully elaborated or realized in microarchitecture. In particular, we are aware that our compiler-targeted instruction set is likely to be largely usable, but the privileged aspects, including exception handling and page-table walking, will likely require further iteration. We anticipate substantial changes in future CHERI ISA revisions as the CHERI-RISC-V ISA matures.

5.1 The RISC-V Instruction-Set Architecture

RISC-V is a contemporary open-source architecture developed at the University of California at Berkeley. RISC-V is intended to be used with a range of microprocessors spanning small 32-bit microcontrollers intended for embedded applications to larger 64-bit superscalar processors intended for use in datacenter computing. The RISC-V ISA is reminiscent of MIPS, with some important differences: a more modular design allows the ISA to be more easily subsetted and extended; a variable-length instruction encoding improves code density; the MMU has a hardware page-table walker rather than relying on software TLB management; the ISA avoids exposing pipelining behaviors to software (e.g., there is no branch-delay slot); and it has a more contemporary approach to atomic memory instructions. Various drafts and standardized extensions add other more contemporary features such as hypervisor support. There is also ongoing work to define broader platform behaviors beyond the architecture, including platform self-description and peripheral-device enumeration. At the time of writing, the RISC-V userspace ISA has been standardized (v2.2) [126], but the privileged ISA remains under development (v1.10) [127][1].

5.2 CHERI-RISC-V Approach

Our application of CHERI to the RISC-V architecture is motivated by several opportunities:

- To gain access to a maturing open-source ISA, hardware, and software ecosystem, for the purposes of a stronger experimental baseline and methodology (such as more mature core variants). At the time of writing, the MIPS software ecosystem remains richer, but we see a substantial community effort at filling gaps for RISC-V.
- To demonstrate the portability of the CHERI approach across multiple architectures, and in particular to illustrate how portable CHERI software stacks can be designed and maintained despite underlying architectural differences.
- To apply lessons learned from CHERI-MIPS in an entirely fresh application of the protection model to a new architecture. Many of our MIPS design choices reflect pragmatic design choices made prior to the development of full compiler and operating-system stacks, and are difficult to change within those stacks.
- To revisit and scientifically explore a design space around CHERI integration into a target architecture – for example, around the use of register files and exceptions.
- To support new CHERI experimentation in the space of microcontrollers, heterogenous cores and accelerators, and DMA, as well as in relation to microarchitectural side channels.
- To lay groundwork for possible open-source transition of the CHERI protection model into the RISC-V architecture.

In the following subsections, we describe our high-level approach before providing a more detailed specification of CHERI-RISC-V.

5.2.1 Target RISC-V ISA Variants

The RISC-V ISA defines both 32-bit (`XLEN=32`) and 64-bit (`XLEN=64`) base integer instruction sets (RV32I, RV64I). Our current proposal would support either mode with few differences beyond capability width, although safe support for both modes in a single processor is not specified at this time. Our definition of CHERI-RISC-V should work with either 32-register or 16-register (RV32E) variants of RISC-V. We specify CHERI as applied to RV-G, which consists of the general-purpose elements of the RISC-V ISA: integer, multiplication and division, atomic, floating-point, and double floating-point instructions. We also describe extensions to RV-S, the draft privileged portion of the ISA.

[1] As v1.11 of the privileged specification remains a work-in-progress, we define CHERI-RISC-V relative to v1.10.

5.2.2 CHERI-RISC-V is an ISA Design Space

A key aim in CHERI-RISC-V is to allow experiments to be run comparing various CHERI-related parameters: Are the general-purpose integer and capability register files separate or merged? Are capabilities with respect to 32-bit or 64-bit virtual addresses? What are the impacts of various instruction-set variations or microarchitectural optimizations? How does greater investment of opcode space affect performance – and what techniques, such as instruction compression or different capability-aware modes, may impact this? How can CHERI interact with other architectural specializations such as DMA and heterogenous compute? To answer these and other questions, we have designed CHERI-RISC-V as an ISA design space, in which several key design dimensions are parameterized:

- Both 32-bit and 64-bit RISC-V are extended, with 64-bit and 128-bit capabilities respectively.
- Both split and merged general-purpose integer and capability register files are supported.
- Optional instruction variations and an optional "capability encoding mode" that invest opcode space differently to reduce instruction count for common instruction sequences – especially with respect to load/store instructions that occupy substantial quantities of opcode space.

With respect to all of these design dimensions, we intend that specific instantiated microarchitectures, compiler targets, compiled operating systems, and compiled software stacks support only one point in the space. However, we hope that carefully parameterized hardware and software designs will be able to target more than one point to allow side-by-side comparison from the perspectives of hardware resource utilization, performance, security, and compatibility.

5.2.3 CHERI-RISC-V Strategy

Our baseline strategy transliterates non-load/store CHERI-specific instruction definitions "as is" to the greatest extent possible, and retains the fundamental CHERI design choices including the use of capabilities in tagged capability registers (protected by guarded manipulation) and tagged memory. As with MIPS, legacy RISC-V integer-relative load/store instructions similarly indirect via **DDC**, and instruction fetch via **PCC**. We utilize the same in-memory capability representation, architectural constants, and compression model in both CHERI-MIPS and CHERI-RISC-V.

Certain necessary divergences arise around RISC-V-specific aspects of the ISA, in particular around privileged features such as exception handling and hardware page-table support. Greater variation also arises around memory load/store instructions, where we attempt to conform to the RISC-V philosophy (supporting only relatively simple addressing modes due to the assumption of micro-op fusion), and for branch instructions (where no branch-delay slot is used).

A key area where CHERI-RISC-V differs from CHERI-MIPS is in allowing the general-purpose integer and capability register files to be "merged", in the style of 64-bit extensions to

32-bit architectures, rather than introducing a new capability register file. This approach offers the potential for reduced microarchitectural overhead due to reduced control logic, as well as reduced disruption to software context management. Both "split" and "merged" register files are supported in CHERI-RISC-V to allow explicit evaluation of their respective compatibility, security, and performance tradeoffs.

Wherever possible, we attempt to conform to the specific aesthetic of RISC-V, such as with respect to opcode layout choices and aligning the semantics of new Special Capability Register access instructions with existing RISC-V CSRs.

5.2.4 Architectural Features Shared with CHERI-MIPS

The following CHERI-MIPS features have been transliterated into CHERI-RISC-V:

- Tagged memory with capability-width tag granularity and alignment.
- An identical architectural capability format (i.e., fields accessed via explicit instructions).
- An identical in-memory capability format, including compression model(s) – except that the format is little endian. We anticipate transitioning CHERI-MIPS to a little-endian in-memory format in due course.
- Registers able to hold capabilities are tagged.
- **PCC** transforms and controls program-counter-relative fetches.
- **DDC** transforms and controls legacy RISC-V load-store instructions, including relocating access addresses using the capability base and offset.
- Requests for non-monotonic capability transformations, capability-related violations (such as loads/stores/fetches via untagged capabilities, out-of-bound accesses, and so on) trigger immediate precise exceptions.
- It is never left ambiguous as to whether a register index operand to a load or store instruction, or the register target of a jump instruction, is a capability and therefore must have a tag set. This both ensures that a split register file can be used (as it is always clear what register file the operand reads from) and also reinforces intentionality.
- The Access_System_Registers permission bit limits privileged ISA operations within privileged rings. While RISC-V's specific privileged operations differ, the intent remains the same: to allow code compartmentalization within the privileged ring.

5.2.5 Architectural Features that Differ from CHERI-MIPS

The following important differences arise between CHERI-MIPS and CHERI-RISC-V:

- CHERI-RISC-V supports a "merged" register-file variant.

- RISC-V exception handling – including register banking, scratch registers, and cause mechanism – is used.

- A new capability exception code, {Interrupt=0, Exception Code=32}, will be reported in the RISC-V mcause, scause, and ucause CSRs when capability-related exceptions (such as tag violations) occur.

- New per-mode capability CSRs are added as {m,s,u}ccsr, which includes additional capability-specific exception cause information, such as more specific cause information and the identity of the faulting register (see Section 5.3.4).

- CHERI-related page permissions are added to RISC-V architectural page-table formats rather than MIPS TLB entries.

- In CHERI-MIPS, capabilities authorizing memory access (i.e., with the Permit_Load, Permit_Store, ... permissions) always have a virtual-address interpretation. This still allows describing physical addresses due to MIPS's architectural physically mapped segments, which directly map portions of the physical address space into the virtual address space.

  In CHERI-RISC-V, the interpretation of addresses in memory capabilities depends on whether virtual addressing is enabled via the RISC-V satp CSR[2]. When satp is set to Bare, capabilities have a physical-address interpretation. When satp enables page-table translation, capabilities have a virtual-address interpretation.

- Both XLEN=32 and XLEN=64 will be supported (albeit not dynamically). In the future, it may be desirable to also support XLEN=128.

- A richer set of atomic instructions is extended with capability support.

- There is support only for compressed capabilities (128 bit for XLEN=64; 64 bit for XLEN=32).

- Floating point is fully supported, including capability-relative floating-point load and store instructions.

- The **flags** field contains a single bit indicating the "capability encoding mode" to use when the capability is installed as **PCC**.

- In the non-compressed RISC-V encoding, the capability encoding mode allows existing opcodes, e.g. for loads, stores, auipc, to be interpreted as expecting capability rather than integer operands (reducing opcode footprint while maintaining intentionality).

[2] This is not a substantially different design choice than in CHERI-MIPS or with MMU addressing enabled: memory capabilities are interpreted relative to the active address space, and control of that address space is delegated to suitably privileged code, whether configuring a simple direct map between virtual and physical memory, or managing multiple more complex address spaces. In all cases, care is required as physical-memory access authorized by a capability is determined by the addressing mode and current translation table contents.

- In the compressed RISC-V encoding, the capability encoding mode allows existing load, store, and jump opcodes to be interpreted as expecting capability rather than integer operands.

5.3 CHERI-RISC-V Specification

In this section, we describe in greater detail the integration of CHERI into the RISC-V instruction set, drawing attention to both similarities and differences from CHERI-MIPS. Draft instruction opcode encodings can be found in Appendix C; these are expected to change as our approach evolves.

5.3.1 Tagged Capabilities and Memory

In CHERI-MIPS, we allow both registers and memory to hold tagged capabilities, allowing capabilities and data to be intermingled. This allows capabilities to be embedded within in-memory data structures, and supports the implementation of capability-oblivious memory copy operations. We recommend that the same approach be taken in CHERI-RISC-V, as this will maintain strong C-language pointer compatibility for capabilities. This implies the use of tagged memory as in CHERI-MIPS, consisting of 1-bit tags protecting capability-aligned, capability-sized words of memory in CHERI-RISC-V, implemented with suitable protection and atomicity properties.

In 64-bit MIPS, we define both 128-bit and 256-bit capability format variants, offering varying degrees of precision and space for additional metadata. Based on the success of the CHERI-128 format in running a full suite of software including the CheriBSD operating system and large applications such as the Postgres database and nginx web server, we choose to define only 128-bit and 64-bit capabilities in CHERI-RISC-V.

While we currently do not define CHERI-RISC-V support for RV128, we anticipate that we will wish to support RV128 in the future. It seems plausible that 256-bit capabilities might incorporate 128-bit addresses along with compressed bounds in a similar manner to our 128-bit capabilities for 64-bit addresses.

5.3.2 Capability Register File

In 64-bit MIPS, we introduce an additional capability register file to hold tagged 128-bit or 256-bit capability registers, rather than extending the general-purpose integer register file. In RISC-V, we are presented with a choice: introduce a new (“split”) register file (e.g., as occurs with the RISC-V F extension for floating point), or extend the existing (“merged”) general-purpose integer registers in the base instruction set.

Both options can be effectively targeted by a CHERI-aware compiler, but offer quite different performance tradeoffs for both the microarchitecture and software code generation. For example, an additional register file may require additional control logic, especially in simpler pipelined designs, and additional registers may impose an additional data-cache footprint due to additional callee/caller register saving and context switching.

In CHERI-RISC-V, we choose to specify the instruction set such that the register-file choice is parameterized, allowing the design space to be evaluated experimentally. Particular CPUs and compiled software stacks will target only one of these two approaches.

Split Register File

In CHERI-RISC-V, the split register file works much as in CHERI-MIPS: CHERI instructions with capability register operands access the capability register file, and integer operands specify access to the integer register file.

Merged Register File

In CHERI-RISC-V, we alternatively allow use of a *merged register file* in which general-purpose integer registers optionally hold full capabilities, along with a tag, reducing the amount of control logic otherwise required (by avoiding an additional register file). This also reduces the size of register context growth, but does require us to avoid a design choice made in earlier version of the CHERI-MIPS ISA in which certain general capability registers had reserved functions, such as **DDC** and **EPCC**. These must instead be accessed via Special Capability Registers accessed via dedicated instructions similar to those accessing conventional RISC-V Control and Status Registers (CSRs), which offer two further advantages: the number of capability registers can more easily be varied (e.g., in RV32E), and the special behavior of those registers with respect to legacy memory access and exception handling is disentangled from the register file's control logic. Clean separation of general-purpose vs. control capability registers is also a design choice present in the CHERI-MIPS ISA as of CHERI ISAv7 for these reasons.

Merging the general-purpose integer and capability register files raises the question of whether and how non-capability-aware instructions should interact with capability values in registers – a concern not dissimilar to the behavior of instructions on 64-bit architectures offering legacy 32-bit support. We specify that individual instructions reading from, or writing to, a register in the register file have fixed integer or capability interpretations based on the opcode encoding – i.e., that new instructions be introduced that explicitly specify whether capability semantics are required for an input or output register, or that the current architectural mode unambiguously specify integer or capability operand interpretation.

A further design choice relates to the specific subset of general-purpose integer registers that are extended to capability width, as it need not be the case that all are. In our baseline specification, we extend all registers, but allow software ABIs to limit specific numbered registers to only integer use. We hope to evaluate different points in this design space to determine whether performance tradeoffs favour a complete set of capability registers, or simply a partial set (which might reduce microarchitectural overhead).

The bottom `XLEN` bits of the register will contain the integer interpretation (which, for a capability, will be its address), and the top `XLEN` bits (plus additional tag bit) will contain any capability metadata. When a register is read as an integer (i.e., using an opcode that dictates an integer interpretation), the register's bottom `XLEN` bits will be utilized, and any other bits ignored. When a register is written as an integer, its bottom `XLEN` bits will hold the new integer value, and the top `XLEN` bits and tag bit will be cleared to match those of the NULL capability.

This both prevents in-register corruption of tagged capabilities by implicitly clearing the tag, and also provides reasonable semantics for integer access to capability values.

Capability Length Architectural Constant (CLEN)

One challenge in introducing CHERI support is that the architectural constant, XLEN, the number of bits in a register, is used to define numerous behaviors throughout the ISA, such as the size of CSRs, the operation of integer operations, the size of addresses, and so on. We choose to leave XLEN as constant as the majority of these operations are intended to be of the natural integer size (e.g., for addition). However, this does mean that in some cases we need to introduce new instructions intended to operate on full capability-wide values. We introduce a new architectural constant, CLEN, which we define as 2×XLEN, which excludes the tag bit. Operations such as capability-width CSR access, capability load, and capability store will operate on CLEN+1 bits including the tag bit.

Specifically, for 32-bit CHERI-RISC-V, CLEN will be 64 bits, and for 64-bit CHERI-RISC-V, CLEN will be 128 bits, affecting a variety of functions including the stride of tag bits in physical memory. Opcode space is reserved in the RISC-V ISA for 64-bit load and store instructions even when XLEN is 32, and we can reuse these opcode reservations and encodings to load 64-bit CLEN words as well as their tag bit. Similarly, when XLEN is 64, we can use 128-bit CLEN load and store opcodes.

We do not currently define support for 32-bit compatibility (with or without capability support) when operating in a 64-bit RISC-V processor, but anticipate that adding non-capability-aware 32-bit support would be straightforward. We also do not yet define an architecture supporting multiple capability widths concurrently, but recognize that there are certain use cases – such as when interoperating between a 64-bit application core and a 32-bit microcontroller within a single System-on-Chip (SoC) – where this would be valuable.

5.3.3 Capability-Aware Instructions

In CHERI-MIPS, two general categories of instructions are added: those that query or manipulate capability fields within registers, and those that utilize registers for the purposes of load, store, or jump operations.

Register-to-register instructions querying and manipulating fields can remain roughly as defined in CHERI-MIPS, allowing integer values to be moved in and out of portions of an in-register capability, subject to guarded manipulation. As such, they are simply new instructions defined in CHERI-RISC-V and added to the opcode space. When using a split register file, those instructions operate as in CHERI-MIPS, accessing the integer or capability register files as specified. With the merged register file, integer and capability values are instead read from, and written back to, the same register file.

In CHERI-RISC-V, assuming that capabilities are stored in the general-purpose integer register file, it is possible to imagine having memory-access and control-flow instructions condition their behavior based on the presence of a tag, selecting a compatible integer behavior if the tag is not set, and a capability behavior if it is set. However, this would violate the principle of intentional use: not only should privilege be minimized, but it should not be unintentionally, implicitly, or ambiguously exercised. Allowing a corrupted capability (i.e., one with its tag stripped due to an overlapping data write) to dereference **DDC** implicitly would violate this design goal. We therefore specify strong *type safety* for all capability-aware instructions: all instructions explicitly encode whether an integer or capability operand is being used, and attempts to use untagged values where tagged ones are expected will lead to an exception.

5.3.4 Control and Status Registers (CSRs)

CHERI-RISC-V extends the behavior of the baseline RISC-V integer CSR set, allowing capability control over access to some CSRs for compartmentalization purposes, as well as adding several new CSRs to control capability-related functionality. These are accessed via existing RISC-V CSR instructions, and their encodings are given in Table 5.1. New Special Capability Registers (SCRs), accessed via new CSR-like instructions, are described in Section 5.3.5.

Encoding	**Register**	Privilege notes
0x8C0	User capability control and status register (`uccsr`)	**PCC**.**perms**.*Access_System_Registers*
0x9C0	Supervisor capability control and status register (`sccsr`)	{S,M}-mode & **PCC**.**perms**.*Access_System_Registers*
0xBC0	Machine capability control and status register (`mccsr`)	M-mode & **PCC**.**perms**.*Access_System_Registers*

Table 5.1: Control and Status Registers (CSRs)

Controlling Access to CSRs

Accessing RISC-V CSRs also requires the PCC.**perms**.*Access_System_Registers* permission to be set for the currently executing code. This allows privileged-level code to be constrained from interfering with key system management functionality (such as exception handling).

Capability Control and Status Registers (CCSRs)

New per HART `{m,s,u}ccsr` `XLEN`-bit RISC-V CSRs are defined as per Figure 5.1 (shown for XLEN=32):

31 16	15 10	9 5	4 2	1	0
WIRI	`cap idx`	`cause`	**WIRI**	`d`	`e`

Figure 5.1: `{m,s,u}ccsr` register format; WIRI bits are Write Ignore Read Ignore.

e The `e` "enable" bit tells whether capability extensions are enabled or disabled.

d The `d` “dirty” bit tells whether a capability register has been written. This is intended to help with memory requirements when implementing context switching.

cause The `cause` field reports the cause of the last capability exception, following the encoding described in Table 3.8.2.

cap idx The `cap idx` field reports the index of the capability register that caused the last exception.

5.3.5 Special Capability Registers (SCRs)

Special Capability Registers (SCRs) are similar to CSRs in that they affect special functions such as exception delivery, rather than being general-purpose registers, but have capability rather than integer types. SCRs are therefore accessed via new capability-aware instructions.

The new `CSpecialRW` instruction allows reading and writing special capability registers. When the destination register is 0, the instruction shall not read the special capability register and shall not cause any of the side-effects that might occur on a special capability register read, similar to the standard `csrrw` RISC-V instruction. When the source register is 0, the instruction will not write to the special capability register at all, and so shall not cause any of the side effects that might otherwise occur on a special capability register write, similarly to the standard `csrrs/c` RISC-V instruction.

Table 5.2 lists the SCRs available via that instruction, as well as their values at CPU reset, which will be set in a manner consistent with the description in Section 3.5. Whether a register is initialized to NULL or the omnipotent capability, its flags field will be initialized to zero (specifying integer encoding mode).

5.3.6 Efficiently Encoding Capability-Relative Operations

The RISC-V instructions that interpret arguments or results as addresses (e.g. loads, stores, jumps, `auipc`) can either act on integer pointers relative to **DDC** or **PCC**, or on explicit capabilities. For example, capability-relative load and store instructions accept (and expect) capability operands that relocate and constrain data accesses, performing tag, bounds, permission, and other checks as required. However, load and store instructions occupy large amounts of instruction encoding space due to having multiple register operands and large immediate values. One consideration in adding CHERI support to RISC-V is the degree to which we are willing to occupy large chunks of remaining encoding space by simply supplementing each address-manipulating instruction with a corresponding capability-relative version, as we did in CHERI-MIPS. Other options that conserve opcode space include utilizing a more limited set of addressing modes or using smaller immediate sizes for capability-relative instructions – with a potentially significant negative impact on performance due to an increase in resulting code size. We therefore consider several points in this design space:

- Introduce a full set of new load and store instructions occupying substantial opcode space, but providing more efficient capability-intensive generated code.

Idx	Register	Modes	ASR	Write	Reset	Extends
0	Program counter cap. (**PCC**)	U, S, M	N	N	∞	PC
1	Default data cap. (**DDC**)	U, S, M	N	Y	∞	-
4	User trap code cap. (**UTCC**)	U, S, M	Y	Y	∞	utvec
5	User trap data cap. (**UTDC**)	U, S, M	Y	Y	$\emptyset$	-
6	User scratch cap. (**UScratchC**)	U, S, M	Y	Y	$\emptyset$	-
7	User exception PC cap. (**UEPCC**)	U, S, M	Y	Y	∞	uepc
12	Supervisor trap code cap. (**STCC**)	S, M	Y	Y	∞	stvec
13	Supervisor trap data cap. (**STDC**)	S, M	Y	Y	$\emptyset$	-
14	Supervisor scratch cap. (**SScratchC**)	S, M	Y	Y	$\emptyset$	-
15	Supervisor exception PC cap. (**SEPCC**)	S, M	Y	Y	∞	sepc
28	Machine trap code cap. (**MTCC**)	M	Y	Y	∞	mtvec
29	Machine trap data cap. (**MTDC**)	M	Y	Y	$\emptyset$	-
30	Machine scratch cap. (**MScratchC**)	M	Y	Y	$\emptyset$	-
31	Machine exception PC cap. (**MEPCC**)	M	Y	Y	∞	mepc

Table 5.2: Special Capability Registers (SCRs). **Modes** shows which RISC-V privilege modes are allowed to access the registers. **ASR** indicates whether **PCC**.**perms**.*Access_System_Registers* must be set to permit access (in addition to being in a permitted mode). **PCC** is not writable via `CSpecialRW`, but is set by `CJALR` and during exceptions. **Reset** indicates whether the register should be initialised to the default root capability (∞) or NULL capability ($\emptyset$). Some special capabilities registers are extensions of existing RISC-V registers, with the capability offset being equal to the original register.

- Introduce only a limited set of new load and store instructions, reducing new opcode utilization, and supportingly less efficient capability-intensive generated code.

- Introduce a new *capability encoding mode* in which existing RISC-V load-store opcode space is reused for capability-relative accesses, allowing a rich set of load-store instructions without substantially occupying available RISC-V opcode space.

  In the conventional (legacy) *integer encoding mode*, a small set of capability-relative loads and stores are added, tuned to limit opcode space utilization – e.g., by having small or no immediates – at the cost of increased code footprint.

  To maintain intentionality, this approach is never ambiguous in either mode as to whether load and store opcodes are intended to access relative to integer or capability operand: address operands are always integer relative in integer encoding mode, and always capability relative in capability encoding mode.

  Pure-capability and hybrid code can be generated against either encoding, but will be most efficient (in terms of instruction footprint) when generated against the corresponding mode. We have specified that the encoding mode will change as a result of jumping to a **PCC** with a different encoding-mode flag. Section 8.26 considers other options for encoding-mode switches.

In the interest of experimentation, we plan to pursue all three approaches in stages, considering tradeoffs around efficiency and instruction-set design across a range of workloads:

1. Begin by adding a conservative set of capability-relative load and store instructions without immediate offsets, which will consume a small amount of opcode space, and be sufficient to allow initial compiler engineering to take place.

2. Introduce an architectural "encoding mode bit" in which RISC-V instruction encodings used for integer-relative **DDC**-constrained loads and stores are instead used for CHERI-RISC-V capability-relative loads and stores. To continue to allow hybrid code when in capability mode, we would introduce a further simple set of **DDC**-indirected integer-relative loads and stores with no immediate, similar to the capability-relative set described above.

3. Introduce a full set of capability-relative loads and stores with immediate offsets to enable full flexibility without switching modes. While this may not be acceptable to the upstream community without increasing the size of instructions, we should understand how much performance is being lost by reducing the flexibility of code generation.

As register-relative jump instructions have relatively light opcode utilization, and because there are many easy-to-imagine uses for protecting control flow using capabilities even in hybrid code, we do not apply semantic changes to those baseline non-compressed RISC-V instructions when in capability encoding mode. The implications for compressed instructions are described in Section 5.3.7.

Encoding Modes

We define two encoding modes, selected using the CHERI-RISC-V-specific encoding-mode flag in the capability **flags** field of **PCC**:

Integer encoding mode (0) Conventional RISC-V execution mode, in which address operands to existing RISC-V load and store opcodes contain *integer addresses*. If using a merged register file, the upper 64 bits and tag bit of the operand register will be ignored. The dereference will implicitly occur relative to **DDC**. The tag bit on **DDC** must indicate that a valid capability is present, and all capability-related checks (such as bounds checks) must be performed in order for a successful load or store to take place.

Capability encoding mode (1) CHERI capability encoding mode, in which address operands to existing RISC-V load and store opcodes contain *capabilities*. The tag bit must indicate a valid capability is present, and all capability-related checks (such as bounds checks) must be performed in order for a successful load or store to take place.

The operating system will automatically save and restore **PCC** on context switches, preserving an execution context's encoding mode. It is essential that changes in encoding mode be properly observed when an exception is processed, as the exception handler must execute with expected semantics or risk insecure behavior. When **{M,S,U}TCC** is set by the operating system, it should contain an appropriate encoding-mode flag to ensure that exception handlers utilize the correct instruction encoding.

Non-Compressed Instructions Affected by Capability Encoding Mode

The following non-compressed RISC-V load and store instructions would be affected by the capability encoding-mode bit (see the following section for further details on compressed instructions):

Integer load	LB	LH	LW	LD	
Integer load (unsigned)	LBU	LHU	LWU	LDU	LQ
Integer store	SB	SH	SW	SD	SQ
Floating-point load	FLW	FLD	FLQ		
Floating-point store	FSW	FSD	FSQ		
Atomic	LR	SC	AMOSWAP	AMOADD	AMOAND
Atomic (cont)	AMOOR	AMOXOR	AMOMAX	AMOMIN	
Address calculation	AUIPC[3]				

5.3.7 Compressed Instructions

The compressed instruction extension (extension `C`) is now routinely used by the RISC-V gcc compiler to improve code density. It seems likely that the `Compressed` extension will become mandatory for the `General` configuration (which is currently `IMAFD`). Two problems arise in adding compressed instruction support for capabilities: the need for additional opcode space for load, store, and jump instructions; and the need to add new instructions to load and store capabilities.

Given the tight encoding space for compressed instructions, some registers (e.g. the stack pointer - `sp`) are implicit for some instructions. Since there appears to be no free encoding space to differentiate between a capability-`sp` and an integer-`sp`, one potential design choice is to use our capability encoding mode to also control the interpretation of compressed instructions. Similarly, for compressed loads and stores that can use only registers `x8`–`x15` as the base address, the encoding mode would allow us to reuse opcode space. As with the baseline compressed instruction set, this imposes ABI-related constraints on the architecture, and would require the compiler to conform to those constraints in order to accomplish the best code density.

In his thesis [125], Waterman gives the following function prologue and epilogue examples to illustrate how compressed instructions improve code density:

```
prologue_legacy:        epilogue_legacy:
  c.addi sp, -16          c.ldsp ra, 8(sp)
```

[3] See Section 5.3.12.

```
  c.sdsp ra, 8(sp)       c.ldsp s0, 0(sp)
  c.sdsp s0, 0(sp)       c.addi sp, 16
  c.jr t0                c.jr ra
```

For capability-aware code, saving and restoring the return address (ra) requires capability store and load instructions. Given the frequency of capability use for the pure-capability code targeted by our capability encoding mode (all pointer loads and stores), one option might be to relieve pressure on the compressed opcode space by removing the less frequently used floating-point double load and store instructions. For RV64, we could replace the compressed load floating-point double (C.FLD) with compressed load capability (C.LC) with the same encoding as compressed load quad (C.LQ) used in RV128. Similarly, replace: C.FSD with C.SC (compressed store capability). For stack-relative memory access, replace floating-point double operations with capability operations: C.FLDSP with C.LCSP and C.FSDSP with C.SCSP.

In the RISC-V I base instruction set (non-compressed instructions), we chose to make capability jump instructions available in both integer and capability encoding modes, as they use relatively little encoding space compared to the amount of free space available. In the RISC-V C extension (compressed instructions), the amount of free space is far smaller, leading us to select a different design choice: when in capability encoding mode, as with load-store instructions, we interpret existing compressed instructions C.J, C.JAL, C.JR, and C.JALR as the capability instructions CJAL, CJR, and CJALR, accepting capability rather than integer register operands for jump target registers and link registers.

There is one large gap in the compressed instruction encoding at 100X_XXXX_XXXX_XX00 (where X= don't care) that could be used to support a CIncOffsetImm (c.cincoff) instruction to allow the stack pointer to be adjusted.

This would result in capability-aware prologue and epilogues:

```
prologue_cap_aware:    epilogue_cap_aware:
  c.cincoff csp, -16     c.lcsp cra, 8(csp)
  c.scsp cra, 8(csp)     c.lcsp s0, 0(csp)
  c.scsp cs0, 0(csp)     c.cincoff csp, 16
  c.cjr ct0              c.cjr cra
```

A further interaction relates to encoding-mode selection. If we pursue a design choice using the lowest bit of a target jump address to set the encoding used following a jump, then no additional opcode pressure is introduced. If we instead choose to use new instructions, such as to get or set the mode explicitly, then additional space might be required. The amount of space required would be modest, but this additional usage might be a further consideration in the encoding-mode management strategy.

If the approach of using a mode bit for compressed instructions is adopted, then it follows that the uncompressed versions of the instructions should follow suit to preserve the design intent that all compressed instructions can be expanded out to uncompressed instructions by the processor's decoder, and that the assembler should be permitted to optimize uncompressed instructions into their compressed form where possible.

Compressed Instructions Affected by Capability Encoding Mode

The following compressed instructions are affected by capability encoding mode:

Control flow	C.JALR	C.JR		
Compressed integer load	C.LW	C.LD	C.LWSP	C.LDSP
Compressed integer store	C.SW	C.SD	C.SWSP	C.SDSP
Compressed floating-point load	C.FLW	C.FLD	C.FLWSP	C.FLDSP
Compressed floating-point store	C.FSW	C.FSD	C.FSWSP	C.FSDSP

5.3.8 Floating Point

The vast majority of floating-point instructions are not impacted by the presence of CHERI-RISC-V. In CHERI-MIPS, we did not define capability-relative load and store instructions for floats and doubles, requiring that they loaded via general-purpose integer registers and then be moved into floating-point registers, leading to reduced code density. Existing RISC-V floating-point load and store instructions, in the integer encoding mode, are relocated and constrained by **DDC**. In CHERI-RISC-V, we define a new set of simple capability-relative load and store instructions, as well as a more rich set via capability encoding mode.

5.3.9 Exception Handling

RISC-V defines several privilege modes, including machine mode, user mode, and supervisor mode, with exceptions allowing controlled transition between those modes. CHERI-RISC-V introduces several new exception-related Special Capability Registers to supplement existing RISC-V exception CSRs with new capability-related functionality. In addition, `{m,s,u}ccsr` will indicate the most recent capability-related exception, as defined in Table 3.8.2, delivered to the corresponding mode.

The 6-bit `cap idx` and 5-bit `cause` fields qualify the last capability exception. `cause` holds the capability exception cause as described in Table 3.8.2. `cap idx` holds the capability register index of the capability associated with the exception. When the most significant bit is set, the 5 least significant bits are used to index the special purpose capability register file described in Table 5.2, otherwise, they index the general-purpose capability register file.

Exceptions to Machine Mode

We define the following new special capability registers that can be read and written only from machine mode:

- **MEPCC** - Machine Mode Exception Program Counter Capability (extends `mepc`)
- **MTDC** - Machine Mode Data Capability
- **MTCC** - Machine Mode Trap Code Capability (extends `mtvec`)
- **MScratchC** - Machine Mode Scratch Capability

Exceptions to Supervisor Mode

We define the following new special capability registers that can be read and written only from supervisor mode and above:

- **SEPCC** - Supervisor Mode Exception Program Counter Capability (extends `sepc`)
- **STDC** - Supervisor Mode Data Capability
- **STCC** - Supervisor Mode Trap Code Capability (extends `stvec`)
- **SScratchC** - Supervisor Mode Scratch Capability

Exceptions to User Mode

We employ the "N" extension (for "User-Level Interrupts") being developed in the newer versions of the RISC-V specifications, and extend it with the following new special capability registers that can be read and written from any mode:

- **UEPCC** - User Mode Exception Program Counter Capability (extends `uepc`)
- **UTDC** - User Mode Data Capability
- **UTCC** - User Mode Trap Code Capability (extends `uvec`)
- **UScratchC** - User Mode Scratch Capability

The extension could be leveraged for user-space-only implementations of `CCall`, as well as routing specific interrupts from suitable devices to user-level compartments for handling by sandboxed device drivers.

Explicit vector and data capabilities replace our definitions of the **KCC** and **KDC** Special Capability Registers in CHERI-MIPS, giving each ring its own code and data capabilities to utilize during exception handling. We define "scratch capabilities" to allow the exception handler to stash a capability register for the purposes of having a working register that corresponding data capabilities can be loaded to in order to begin a full context save. This is consistent with RISC-V's use of scratch registers in various modes to avoid committing general-purpose integer registers to exception handling, as happens in the MIPS ABI with `$k0` and `$k1`. We are therefore able to similarly avoid the need for CHERI-MIPS's **KR1C** and **KR2C**.

5.3.10 Virtual Memory and Page Tables

In CHERI-RISC-V, capability addresses are interpreted with respect to the privilege level of the processor. In Machine Mode, capability addresses, as with integer addresses in RISC-V, are interpreted as physical addresses. In Supervisor and User Modes, capability addresses, again as with RISC-V, are interpreted as virtual addresses.

Unlike in CHERI-MIPS, the page-table walker is implemented by the architecture and not the software stack, and in the absence of further extensions to the page-table format, continues to contain physical addresses loaded and stored on behalf of executing software. In CHERI-RISC-V, we require the Access_System_Registers permission to change the page-table root and other virtual-memory parameters. In the future, it may be desirable to extend the page-table walking mechanism to itself utilize capabilities, allowing the walker to be constrained.

As with CHERI-MIPS, it is desirable to extend the Memory Management Unit to constrain the loading and storing of valid capabilities via specific page mappings. In CHERI-MIPS, this is expressed via two new Translation Look-aside Buffer (TLB) permissions that, if not set, strip tags on load, or trigger an exception when storing a capability with a valid tag bit. The natural translation of this idea to RISC-V, which includes architectural support for page tables, rather than software TLB management, would be to add two effectively identical permission bits to the current Page Table Entry (PTE) format. Unfortunately, there are no remaining spare bits in the RISC-V Sv32 (32-bit) PTE format for additional hardware permissions. For the purposes of prototyping, it may be desirable to utilize the two available software-defined PTE permission bits – but these are likely to be used in current operating systems, requiring a longer-term solution. The Sv39 (39-bit) and Sv48 (48-bit) PTE formats include several reserved bits, which could be allocated for use by CHERI-RISC-V. We define the following new permissions:

- A new page load permission; if not present, it strips tags from loaded capabilities.
- A new page store permission; if not present, it causes attempts to store a capability with the tags set to throw an exception.

The RISC-V architecture defines management of *accessed* and *dirty* bits per page table (and page directory). In the same spirit, we define an additional metadata bit, called *capability-dirty* which is to be set in a leaf PTE whenever that PTE is used as part of a store-capability instruction. Hardware is permitted to set this bit even if the store-capability instruction faults before completing; implementations are also permitted leave setting this bit to software and, so, to trap on capability-store instructions using a leaf PTE in which this bit is clear (using the existing page fault cause `scause` code). Capability-store instructions are, additionally, expected to set the existing dirty bit (or trap if it is clear, again, using the existing `scause` code). Hardware must tolerate a leaf PTE being marked as capability-dirty but not (data) dirty (as well as vice versa) as software may be tracking the two states across different intervals.

5.3.11 The RV-128 LQ, SQ, and Atomic Instructions

The putative 128-bit RISC-V ISA (RV-128) reserves additional quadword load and store instructions, `LQ` and `SQ`, to be used to load and store 128-bit quantities, as well as quad-word atomics. In CHERI-RISC-V for RV-64, we reuse these opcode encodings for our 129-bit capability load and store instructions, `CLC` and `CSC`, to avoid additional opcode commitment. We also introduce corresponding atomics on capabilities reusing the quad-word atomic opcodes.

Should the future RV-128 standard utilize 128-bit addresses, then the most natural course of action would be to utilize compressed 256-bit capabilities, and add new capability load and store opcodes for the broader capability width. However, should an RV-128 be defined that instead uses 64-bit virtual addresses (i.e., one with 128-bit data registers but not a 128-bit address space), our current opcode-space reuse would not be appropriate for a corresponding CHERI-RISC-V variant. Overloaded opcodes might reduce intentionality and in the split register-file configuration we would be unable to distinguish operations intended for the integer register file vs. the capability register file. With respect to intentionality, it remains to be seen how essential this concern is with respect to security: tag-free copies could still be implemented efficiently by stripping Permit_Load_Capability from a source capability during a memory copy. However, the alignment requirements imposed by our capability load, store, and atomic instructions can be beneficial in debugging what is otherwise potential tag loss. Should RV-128 be more fully specified in the future, we will need to revisit whether capability load instructions can be combined with the LQ, SQ, and atomic instructions.

5.3.12 The AUIPC Instruction

The RISC-V AUIPC instruction generates an address derived from **PC** and a 20-bit immediate, typically intended to be used in generating addresses for global variables. Because this instruction occupies a significant amount of opcode space, we choose to implement a capability-based version of the instruction only in the capability encoding mode, where the instruction returns a capability derived from **PCC** rather than an integer virtual address. When using AUIPC to generate an integer in the capability encoding mode, or a capability in the integer encoding mode, an additional, less efficient, instruction sequence must be used instead. Depending on the code linkage model, it might also be desirable to have a further version of the instruction, GAUIPC, which returns a capability derived from a global capability table register.

Chapter 6

The CHERI-x86-64 Instruction-Set Architecture (Sketch)

In this chapter, we explore models for applying CHERI protection to the x86 architecture. The x86 architecture is a widely deployed CPU architecture used in a variety of applications ranging from mobile to high-performance computing. The architecture has evolved over time from 16-bit processors without MMUs to present-day systems with 64-bit processors supporting virtual memory via a combination of segmentation and paging.

The x86 architecture has spanned three register sizes (16, 32, and 64 bits) and multiple memory management models. We choose to define CHERI solely for the 64-bit x86 architecture for a variety of reasons including its more mature virtual-memory model, as well as its larger general-purpose integer register file.

6.1 Capability Registers versus Segments

The x86 architecture first added virtual memory support via relocatable and variable-sized segments. Each segment was assigned a mask of permissions. Memory references were resolved with respect to a specific segment including relocation to a base address, bounds checking, and access checks. Special segment types permitted transitions to and from different protection domains.

These features are similar to features in CHERI capabilities. However, there are also some key differences.

First, x86 addresses are stored as a combination of an offset and a segment spanning two different registers. General-purpose registers are used to hold offsets, and dedicated segment selector registers are used to hold information about a single segment. The x86 architecture provides six segment selector registers – three of which are reserved for code, stack, and general data accesses. A fourth register is typically used to define the location of thread-local storage (TLS). This leaves two segment registers to use for fine-grained segments such as separate segments for individual stack variables. These registers do not load a segment descriptor from arbitrary locations in memory. Instead, each register selects a segment descriptor from a descriptor table with a limited number of entries. One could treat the segment descriptor tables (or portions of these tables) as a cache of active segments.

Second, more fine-grained segments are not derived from existing segments. Instead, each entry in a descriptor table is independent. Write access to a descriptor table permits construction of arbitrary segments (including special segments that permit privilege transitions). Restricting descriptor-table write access to kernel mode does not protect against construction of arbitrary segments in kernel mode due to bugs or vulnerabilities. As a result, segment descriptors are not able to provide the same provenance guarantees as tagged capabilities.

Third, existing segment descriptors do not have available bits for storing types or permissions more expressive than the existing read, write, and execute.

Finally, x86 segmentation is typically not used in modern operating systems. On the 32-bit x86 architecture, systems generally create segments with infinite bounds and use a non-zero base address only for a single segment that provides TLS. The 64-bit x86 architecture codifies this by removing segment bounds entirely and supporting non-zero-base addresses only for two segment registers. Software for x86 systems stores only the offset portion of virtual addresses in pointer variables. Segment registers are set to fixed values at program startup, never change, and are largely ignored.

One approach for providing a similar set of features to CHERI capabilities on x86 would be to extend the existing segment primitives to accommodate some of these differences. For example, descriptor-table entries could be tagged, whereby loading an untagged segment would trigger an exception. However, some other potential changes are broader in scope (e.g., whether segment selectors should contain an index into a table, versus a logical address of a segment descriptor). Extending segments would also result in a very different model compared to CHERI capabilities on other architectures, limiting the ability to share code and algorithms. Instead, we propose to add CHERI capabilities to 64-bit x86 by extending existing general-purpose integer registers.

6.2 Tagged Capabilities and Memory

As with CHERI-MIPS and CHERI-RISC-V, we recommend that both memory and registers contain tagged capabilities. Similar to CHERI-RISC-V, we also recommend a single, 128-bit format for CHERI-x86-64 capabilities.

6.3 Extending Existing Registers

The x86 architecture has expanded its general-purpose integer registers multiple times. Thus, the 16-bit **AX** register has been extended to 32-bit **EAX** and 64-bit **RAX**. We propose extending each general-purpose integer register to a tagged, 128-bit register able to contain a single capability. The capability-sized registers would be named with a 'C' prefix in place of the 'R' prefix used for 64-bit registers (**CAX**, **CBX**, etc.). As with CHERI-RISC-V, we recommend that reads of the general-purpose registers as integers return the cursor value (virtual address). Writes to general-purpose registers using non-capability-aware instructions should clear the tag and upper 128 bits of capability metadata, storing the desired integer value in the register's cursor.

Some x86 instructions have implicit memory operands addressed by a register. When using capabilities to address memory, the instructions would use the full capability register.

The "string" instructions use **RSI** as source address and **RDI** as a destination address. For example, the `STOS` instruction stores the value in **AL/AX/EAX/RAX** to the address in **RDI**, and then either increments or decrements the destination index register (depending on the Direction Flag). When using capabilities, these string instructions should use **CSI** instead of **RSI** and **CDI** instead of **RDI**.

Instructions that work with the stack such as `PUSH` or `CALL` use the stack pointer (**RSP**) as an implicit operand. With capabilities these instructions would use **CSP** instead of **RSP**.

The **RIP** register (which contains the address of the current instruction) would also be extended into a **CIP** capability. This would function as the equivalent of **PCC** for CHERI-MIPS.

6.4 Additional Capability Registers

Additional capability registers beyond those present in the general-purpose integer register set will also be required.

A new register will be required to hold **DDC** for controlling non-capability-aware memory accesses.

The x86 architecture currently uses the **FS** and **GS** segment selector registers to provide thread-local storage (TLS). In the 64-bit x86 architecture, these selectors are mostly reduced to holding an alternate base address that is added as an offset to the virtual address of existing instructions. For CHERI-x86-64 we recommend replacing these segment registers with two new capability registers: **CFS** and **CGS**.

In addition, new capability registers may be required to manage user to kernel transitions as detailed below.

6.5 Using Capabilities with Memory Address Operands

As with CHERI-MIPS, CHERI-x86-64 should support running existing x86-64 code, capability-aware code, and hybrid code. This requires the architecture to support multiple addressing modes. The x86 architecture has implemented this in the past when it was extended to support 32-bit operation. We propose to reuse some of the same infrastructure to support a new capability-based addressing mode.

When x86 was extended from 16 bits to 32 bits, the architecture included the ability to run existing 16-bit code without modification as well as execute individual 16-bit or 32-bit instructions within a 32-bit or 16-bit codebase. The support for 16-bit versus 32-bit operation was split into two categories: operand size and addressing modes. The code segment descriptor contains a single-bit 'D' flag, which sets the default operand size and addressing mode. These attributes can then be toggled to the non-default setting via opcode prefixes. The 0x66 prefix is used to toggle the operand size, and the 0x67 prefix is used to toggle the addressing mode.

In 64-bit ("long") mode, the 'D' flag is currently always set to 0 to indicate 32-bit operands and 64-bit addressing. A value of 1 for 'D' is reserved. The 0x67 opcode prefix is used to toggle between 32-bit and 64-bit addresses, but a few other single-byte opcodes are invalid in 64-bit mode and could be repurposed as a prefix.

We propose a new capability-aware addressing mode that can be toggled via the 'D' flag of the current code segment and a new 0x62 opcode prefix. (In 32-bit x86, the 0x62 opcode is the `BOUND` instruction, which is invalid in 64-bit mode.) If the 'D' flag of a 64-bit code segment is set to 1, then the CPU would execute in "capability mode" – which would include using the capability-aware addressing mode by default. Individual instructions could toggle between capability-aware and "plain" 64-bit addressing via the 0x62 opcode prefix. Addresses using the "plain" 32-bit or 64-bit addressing would always be treated as offsets relative to **DDC**. Instructions using capability-aware addressing would always use 64-bit virtual addresses and ignore any 0x67 opcode prefix.

Note that one can change the value of **CS** in user mode (for example, a user process in FreeBSD/amd64 can switch between 32 and 64-bit by using a far call that loads a different value of **CS**). This would mean that user code could swap into pure-capability mode without requiring a system call. However, this would not alter the contents of capability registers or their enforcement, merely the decoding of instructions. If **DDC** is invalid, then sandboxed code that switched to a non-capability **CS** would still require valid capability registers to access memory.

6.5.1 Capability-Aware Addressing

For instructions with register-based memory operands, capability-aware addressing would use the capability version of the register rather than the virtual address relative to **DDC**.

For example:

```
mov 0x8(%cbp),%rax
```

would read the 64-bit value at offset 8 from the capability described by the **CBP** register.

On the other hand,

```
mov 0x8(%rbp),%rax
```

would read the 64-bit value at an offset of RBP+8 from the **DDC** capability. Both instructions would use the same opcode aside from the addition of an 0x62 opcode prefix. In a code segment with 'D' set to 1, the second instruction would require the prefix. In a code segment with 'D' set to 0, the first instruction would require the prefix.

6.5.2 Scaled-Index Base Addressing

x86 also supports an addressing mode that combines the values of two registers to construct a virtual address known as scaled-index base addressing. These addresses use one register, the *base*, and a second register, the *index*, multiplied by a scaling factor of 1, 2, 4, or 8. For these addresses, capability-aware addresses would select a capability for the base register, but the index register would use the integer value of the register. For example:

```
mov (%rax,%rbx,4),%rcx
```

This computes an effective address of **RAX** + **RBX** * 4 and loads the value at that address into **RCX**, The capability-aware version would be:

```
mov (%cax,%rbx,4),%rcx
```

That is, starting with the **CAX** capability, **RBX** * 4 would be added to the offset, and the resulting address validated against the **CAX** capability.

6.5.3 RIP-Relative Addressing

The 64-bit x86 architecture added a new addressing mode to support more efficient Position-Independent Code (PIC) performance. This addressing mode uses an immediate offset relative to the current value of the instruction pointer. These addresses are known as **RIP**-relative addresses. To support existing code, **RIP**-relative addresses should be resolved relative to **DDC** when executing instructions from a code segment whose segment descriptor has 'D' set to 0. In "capability mode" where 'D' is set to 1, these immediate offset and memory access should instead be validated relative to **CIP** (the equivalent of **PCC** from CHERI-MIPS).

An alternative approach might be to always require **RIP**-relative addresses relative to **CIP** and require the runtime environment to configure a suitable **CIP** capability when executing non-capability-aware code.

6.5.4 Using Additional Capability Registers

The proposed capability-aware addressing mode proposed above allows for the capability versions of existing general-purpose integer registers such as **CAX** or **CBP** to be encoded in existing register instructions. However, it does not permit the direct use of the additional capability registers **DDC**, **CFS**, or **CGS**. **DDC** is not expected to be used as an explicit base address, but **CFS** and **CGS** must be usable in this manner to support TLS with capability-aware addresses.

One option would be to repurpose the existing **FS** and **GS** segment prefixes when used with instructions using capability-aware addresses to select an implicit base register of **CFS** or **CGS**, respectively. However, this approach is potentially confusing. Would an instruction using an existing address of "(%cax)" and an instruction prefix of "GS:" simply use the cursor of **CAX** (value of **RAX**) as an offset relative to **CGS**? In addition, instructions that manipulate capabilities need a way to specify an additional capability register as an operand.

To handle both of these cases, we propose to reuse the existing **FS** and **GS** segment prefixes to extend the capability register selector field in opcodes. This is similar to the use of bits in **REX** prefixes to extend the general-purpose integer register selector fields in other instructions. Instructions with memory addresses will use at most one capability-register, and the **FS** prefix could be used to select capability registers with an index of 32 or higher. For instructions operating on two capability registers, the **FS** prefix would affect the register selected for the first capability register operand, and the **GS** prefix would affect the register selected for the second capability register operand. Additional capability registers such as **DDC**, **CFS**, and **CGS** would be assigned register indices starting at 32 and require a suitable prefix.

6.6 Capability-Aware Instructions

6.6.1 Control-Flow Instructions

Existing control-flow operations such as JMP, CALL, and RET would modify the offset of the **CIP** capability as well as verify that the new offset is valid.

New instructions would be required when performing a control-flow operation that loads a full **CIP** capability. For example, a new CJMP instruction would accept either a capability register or an in-memory capability as its sole argument and load the new capability into **CIP** similar to the CHERI-MIPS CJR instruction. New CCALL and CRET instructions (not to be confused with the CHERI-MIPS protection-domain cross instructions) would be used for function calls that push the full **CIP** capability onto the stack as the return address.

6.6.2 Manipulating Capabilities

New instructions will need to be defined to support capability manipulations similar to CHERI-MIPS.

New MOV variants could handle loading and storing of capabilities similar to CLC and CSC.

A new variant of CMPXCHG will be required to support atomic operations on capabilities. (Note that CMPXCHG16B's existing semantics are not suitable for capabilities as it divides the values into register pairs.) It may also be desirable to define a capability variant of XADD.

Variants of PUSH/POP could be used to save and restore capability registers on the stack – if those operations are common enough to warrant new instructions rather than using capability MOV instructions paired with adjustments to **CSP**.

For other capability operations such as CIncOffset, we propose adopting existing CHERI-MIPS instructions rather than repurposing existing instructions such as ADD. Existing general-purpose x86 instructions support two operands rather than three operands. To avoid introducing a new operand encoding format, we propose to use two-operand variants of CHERI-MIPS instructions when adapting instructions to x86.

The LEA instruction warrants additional consideration. When capability mode addressing is enabled for this instruction, the effective address should be a capability stored in a capability destination register. This would permit the use of LEA to perform some operations such as CIncOffset, without overwriting the source capability register.

6.6.3 Inspecting Tags

CHERI-MIPS provides dedicated CBTU and CBTS instructions for conditional branches. For x86, a more natural model may be to add a new "tagged" flag to the **RFLAGS** register. Any instruction that stored a capability such as capability-aware variants of MOV would set this flag to the tag bit of the stored capability. New conditional jump instructions would be added that tested this flag.

6.7 Capability Violation Faults

For reporting capability violations, we propose reserving a new exception vector. This new exception would report an error code pushed as part of the exception frame similar to GP# and PF# faults. The error code would contain one of the values from Table 3.8.2 to indicate the specific violation. In addition, it may be useful to provide a copy of the relevant capability register via one of the currently-unused but reserved control registers, such as **CR5** or **CR12** – similar to the PF# virtual address stored in **CR2**. This would avoid the need for decoding the faulting instruction to determine the relevant capability.

6.8 Interrupt and Exception Handling

For interrupt and exception handling, we propose a new overall CPU mode that enables the use of capabilities. The availability of this mode would be indicated by a new `CPUID` flag. The mode would be enabled by setting a new bit in **CR4**. When this mode is enabled, exceptions would push a new type of interrupt frame that would replace **RIP** with the full **CIP** capability, and **RSP** with the full **CSP** capability. `IRET` would be modified to unwind this expanded stack frame.

Interrupt and exception handlers require new capabilities for the program counter (**CIP**) and stack pointer (**CSP**) registers. We consider two possible approaches.

6.8.1 Kernel Code and Stack Capabilities

The first approach would add two new control registers: the Kernel Code Capability (**KCC**) and Kernel Stack Capability (**KSC**). Access to these registers would be restricted to supervisor mode. These new registers could be named as instruction operands, using the same approach decribed earlier for **CFS** and **CGS**.

Transitions into supervisor mode would load new offsets relative to **KCC** and **KSC** from existing data structures and tables to construct the new **CIP** and **CSP** register values. For example, the current virtual address stored in each Interrupt Descriptor Table (**IDT**) entry would be used as an offset relative to **KCC** to build **CIP**, and the address stored in the Interrupt Stack Table (**IST**) entry in the current Task-State Segment (**TSS**) would be used as an offset relative to **KSC** to build **CSP**. Transitions via the `SYSCALL` instruction would use the offset from **IA32_LSTAR** to construct the new **CIP**.

This approach does require broad capabilities for **KCC** and **KSC** that can accommodate any desired entry point or stack location. However, it will require minimal changes to existing systems code such as operating-system kernels.

6.8.2 Capabilities in Entry Points

The second approach would be to replace virtual addresses stored in existing entry points with complete capabilities. This is a more invasive change, requiring larger changes to existing systems code, but it enables the use of more fine-grained capabilities for each entry point.

Bit	Name	Description
62	CW	Permits stores of tagged capabilities
61	CR	Preserves tags when reading capabilities
60	CD	Set when a tagged capability is stored to this page

Table 6.1: CHERI-x86-64 Page Table Bits

Setting the desired kernel stack pointers **CSP** would require a new **TSS** layout that expanded the existing **RSP** and **IST** entries to capabilities.

For SYSCALL, a new control register **CSTAR** could be added to hold the target instruction pointer. As with **KCC**, this register would be a privileged register in the same bank as **CFS** and **CGS**.

Entries in the **IDT** would be expanded to 32-bytes, appending a capability code pointer in the last 16 bytes. This would double the size of the **IDT**, and most of the bytes would be unused. However, it would ensure that all of the information currently stored in an **IDT** entry (such as the segment selector, **IST** index, and descriptor type) would be configurable.

6.8.3 SWAPGS and Capabilities

The SWAPGS instruction is used in user-to-kernel transitions for the 64-bit x86 architecture to permit separate TLS pointers for user and kernel mode. One option would be to provide a capability version of SWAPGS, either by extending the **IA32_KERNEL_GS_BASE** MSR to a capability, or adding a new MSR. However, this instruction can be difficult to use. Interrupt and exception handlers must be careful not to invoke SWAPGS if the interrupt or exception is taken while executing the kernel mode **GS**. We recommend avoiding the use of SWAPGS, and instead defining a new privileged control register **KGS**. Operating systems could either choose to use **KGS** to initialize **CGS** in interrupt and exception handlers, or else use **KGS** directly as the kernel mode TLS pointer.

6.9 Page Tables

Similar to CHERI on other architectures, additional page-table permission bits governing loads and stores of capabilities are desirable. In addition, it may be beneficial to have a "capability dirty" bit. At present the 64-bit x86 architecture has reserved bits in a range from bit 52 to bit 62. Similar to the non-execute bit (bit 63), CHERI-x86-64 could use bits starting at bit 62 as described in Table 6.1. Higher bits are preferred, to permit maximal room for growth of the physical address field that currently ends at bit 51.

6.10 Capabilities and Integer Instructions

Throughout this chapter, we have proposed that CHERI-x86-64 should assume that existing integer instructions writing to a capaiblity register should always strip the tag with the exception of `LEA` in capability mode. This approach offers the following benefits:

- It avoids "implicit" operations on capabilities.
- It is consistent with 32-bit instructions zero-extending results when stored in 64-bit registers in long mode.
- Micro-architecturally it permits integer instructions to always assume an untagged output.

However, an alternative approach would be for integer instructions to instead perform the requested integer operation against the virtual address, and strip the tag only if the new virtual address results in unrepresentable bounds. In this model, existing instructions such as `ADD` or `SUB` could be used in place of `CIncOffset`. This would offer the following benefits:

- Pointer manipulations may be able to use shorter instructions, because new capability-specific instructions will all likely be using 2- and 3-byte opcodes.
- Fewer new opcodes may be required, because some capability instructions might be fully implemented via existing instructions.
- Compilers may be able to reuse existing code sequences for function prologues and epilogues.

Chapter 7

The CHERI-MIPS Instruction-Set Reference

CHERI-MIPS's instructions express a variety of operations affecting capability and integer registers as well as memory access and control flow. A key design concern is *guarded manipulation*, which provides a set of constraints across all instructions that ensure monotonic non-increase in rights through capability manipulations. These instructions also assume, and specify, the presence of *tagged memory*, described in the previous chapter, which protects in-memory representations of capability values. Many further behaviors, such as reset state and exception handling (taken for granted in these instruction descriptions), are also described in the previous chapter. A small number of more recently specified experimental instructions are specified in Appendix D rather than in this chapter.

The instructions fall into a number of categories: instructions to copy fields from capability registers into integer registers so that they can be computed on, instructions for refining fields within capabilities, instructions for memory access via capabilities, instructions for jumps via capabilities, instructions for sealing capabilities, and instructions for capability invocation. 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 also described in this chapter.

7.1 Sail language used in instruction descriptions

The instruction descriptions contained in this chapter are accompanied by code in the Sail language [8, 105] taken from the Sail CHERI-MIPS implementation [104]. 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 this document. For a full description see the Sail language documentation.

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 repsect 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. Aritmetic 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 acessed 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 instruction 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.

	128-bit	256-bit	Description
`cap_size`	16	32	Number of bytes used to store a capability.
`max_otype`	$2^{18}-1$	$2^{24}-1$	Maximum **otype** allowed by capability format.

	MIPS	RISC-V	Description
`num_flags`	0	1	Number of capability flags.

Figure 7.1: Constants in Sail code

- Local variables:

  ```
  mutable_var = exp;
  let immutable_var = exp;
  ```

  Mutable variables are introduced by simply assigning to them. 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.

- 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.

7.2 Common Constant Definitions

The constants used in the Sail are show in Table 7.1; their value depends on the capability format in use and architecture specific features such as the RISC-V capability mode flag.

7.3 Common 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 : Int). 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 : Int), 'n >= 1. 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 : Int), 'l >= 0. (int('l), int) -> bits('l)`

`to_bits(l, v)` converts an integer, `v`, to a bit vector of length `l`. If `v` is negative a twos-complement representation is used. If `v` is too large (or too negative) to fit in the requested length then it is truncated to the least significant bits.

`pow2 : forall ('n : Int), 'n >= 0. int('n) -> int(2 ^ 'n)`

`pow2`(n) returns 2 raised to the power n.

`zero_extend`

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

`sign_extend`

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

Functions for ISA exception behavior

`SignalException : forall ('o : Type). Exception -> 'o`

Causes the processor to raise the given exception in the usual manner defined by the processor architecture (as modified for CHERI).

`SignalExceptionBadAddr : forall ('o : Type). (Exception, bits(64)) -> 'o`

causes the processor to raise the given exception as per `SignalException`, but with an associated bad address (on MIPS this is written to the BadVAddr register to aid with exception handling).

`raise_c2_exception : forall ('o : Type). (CapEx, regno) -> 'o` causes the processor to raise a capability exception by writing the given capability exception cause and register number to the CapCause register then signalling an exception using `SignalException` (on CHERI-MIPS this is a C2E exception in most cases, or a special C2Trap for CCall and CReturn).

`raise_c2_exception_noreg : forall ('o : Type). CapEx -> 'o`

is as `raise_c2_exception` except that CapCause.RegNum is written with the special value 0xff indicating PCC or no register.

`checkCP2usable : unit -> unit`

`checkCP2usable` raises a co-processor unusable exception if `CP0Status.CU[2]` is not set. All capability instructions must first check that the capability co-processor is enabled. This allows the operating system to only save and restore the full capability context for processes that use capabilities.

Functions for control flow

`execute_branch : bits(64) -> unit`

`execute_branch` checks the given offset against the bounds of PCC and raises a capability length exception if it is out of bounds, otherwise a branch occurs in the normal manner for the architecture (on MIPS this implies a branch delay slot, so `NextInBranchDelay` is set to true).

`execute_branch_pcc : Capability -> unit`

`execute_branch_pcc` executes a branch to the given capability, replacing PCC and taking the new PC from the offset field. Note that on MIPS the new PCC does not take effect until after the branch delay slot.

`set_next_pcc : Capability -> unit`

`set_next_pcc` sets PCC to the given capability before executing the next instruction. It is used for CCall, which has no branch delay.

Functions for reading and writing register and memory

`rGPR : bits(5) -> bits(64)`

Reads the value of the given general purpose register as a 64-bit vector. Register zero is always zero.

`wGPR : (bits(5), bits(64)) -> unit`

`wGPR`(rd, v) writes the 64-bit value, *v*, to the general purpose register *rd*. Writes to register zero are ignored.

`readCapReg : regno -> Capability`

`readCapReg` reads a given capability register or, the null capabiility if the argument is zero.

`readCapRegDDC : regno -> Capability`

`readCapRegDDC` is the same as `readCapReg` except that when the argument is zero the value of DDC is returned instead of the null capability. This is used for instructions that expect an address, where using null would always generate an exception.

`writeCapReg : (regno, Capability) -> unit` `writeCapReg(cd, cap_val)` writes capability, `cap_val` capability register `cd`. Writes to register zero are ignored.

`memBitsToCapability : (bool, bits(256)) -> Capability`

`memBitsToCapability(tag, capBits)` converts `capBits` from the in-memory capability format to a convenient structure for ease of field access. Note that the bit representation is xored with the bit representation of the null capability to ensure that null is always stored as all-zeros in memory.

`capToMemBits : Capability -> bits(256)`

`capToMemBits` is the reverse of `memBitsToCapability`.

`MEMr_wrapper : forall (’n : Int), (1 <= ’n & ’n <= 8). (bits(64), int(’n)) -> bits(8 * ’n)`

`MEMr_wrapper(addr, size)` reads a vector of `size` bytes of memory from physical address `addr` (big-endian byte order on CHERI-MIPS).

`MEMr_reserve_wrapper : forall (’n : Int), (1 <= ’n & ’n <= 8). (bits(64), int(’n)) -> bits(8 * ’n)`

is the same as `MEMr_wrapper` except that the read is marked as part of a load linked / store conditional pair.

`MEMr_tagged : forall (’size : Int), ’size >= 1. (bits(64), int(’size)) -> (bool, bits(’size * 8))`

reads `size` bytes from the given physical address in memory and the associated tag value.

`MEMr_tagged_reserve : forall (’size : Int), ’size >= 1. (bits(64), int(’size)) -> (bool, bits(’size * 8))`

is as `MEMr_tagged` except that the load is marked as part of a load linked / store conditional.

`MEMw_wrapper : forall (’n : Int), ’n >= 1. (bits(64), int(’n), bits(8 * ’n)) -> unit`

`MEMw_wrapper(addr, size, value)` writes `size` bytes of `value` to physical address `addr`.

`MEMw_conditional_wrapper : forall (’n : Int), ’n >= 1. (bits(64), int(’n), bits(8 * ’n)) -> bool`

`MEMw_conditional_wrapper(addr, size, value)` attempts to write `size` bytes of `value` to physical address `addr` and returns a boolean indicating store conditional success or failure.

`MEMw_tagged : forall (’size : Int), ’size >= 1. (bits(64), int(’size), bool, bits(’size * 8)) -> unit`

`MEMw_tagged(addr, size, t, value)` writes `size` bytes, `value`, to physical address, `addr`, with associated tag value, `t`.

`MEMw_tagged_conditional : forall (’size : Int), ’size >= 1. (bits(64), int(’size), bool, bits(’size * 8)) -> bool`

`MEMw_tagged_conditional(addr, size, t, value)` writes `size` bytes, `value`, to physical address, `addr`, with associated tag value, `t` and returns store conditional success or failure.

`TLBTranslate : (bits(64), MemAccessType) -> bits(64)`

`TLBTranslate(addr, acces_type)` translates the virtual address, `addr`, to a physical address assuming the given `access_type` (load or store). If the TLB lookup fails an ISA exception is raised.

`TLBTranslateC : (bits(64), MemAccessType) -> (bits(64), bool)`

`TLBTranslateC` is the same as `TLBTranslate` except that it also returns a boolean indicating whether capability loads or stores are permitted for the given page.

`wordWidthBytes : WordType -> range(1, 8)`

Returns the width of the given WordType (byte, half, word, double) in bytes.

`isAddressAligned : (bits(64), WordType) -> bool`

`isAddressAligned(address, wordtype)` returns whether `address` is naturally aligned for the given `wordtype`.

`extendLoad : forall ('sz : Int), 'sz <= 64. (bits('sz), bool) -> bits(64)`

`extendLoad(val, signed)` extends `val` to 64-bits in either sign extended or zero extended fashion according to `signed`.

`getAccessLevel : unit -> AccessLevel`

Returns the current effective access level (User, Supervisor or Kernel) determined by accessing the relevant parts of the MIPS status register.

`grantsAccess : (AccessLevel, AccessLevel) -> bool`

Returns whether the first AccessLevel is sufficient to grant access at the second, required, access level.

Functions for manipulating capabilities

The sail code abstracts the capability representation using the following functions for getting and setting fields in the capability. The functions have different implementations for 256-bit and 128-bit capability formats.

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 for 128-bit capabilities top can be up to 2^{64}, meaning the entire 64-bit address space can be addressed, but that the 256-bit capability format has a maximum length of $2^{64} - 1$ so the last byte of the address space is inaccessible. Capability length is defined by the relationship $base + length = top$.

`getCapBase : Capability -> uint64`

returns the base of the given capability as an integer in the range 0 to $2^{64} - 1$.

`getCapTop : Capability -> CapLen`

returns the top of the given capability as an integer in the range 0 to 2^{65}.

`getCapLength : Capability -> CapLen`

returns the length of the given capability as an integer in the range 0 to 2^{65}.

The capability's address (also known as cursor) and offset (relative to base) are related by:

$$base + offset \bmod 2^{64} = cursor$$

The following functions return the cursor and offset of a capability respectively:

`getCapCursor : Capability -> uint64`

returns the address of the capability as an integer in the range 0 to $2^{64} - 1$.

`getCapOffset : Capability -> uint64`

returns the offset of the capability (i.e. address relative to base) as an integer in the range 0 to $2^{64} - 1$

.

The following functions adjust the bounds and offset of capabilities. When using compressed capabilities not all combinations of bounds and offset 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, bits(64), bits(65)) -> (bool, Capability)`

`setCapBounds(cap, base, top)` returns a new capabiltiy value derived from cap with given base and top and the address set to `base`. The returned boolean value indicates whether the requested bounds were exactly representable. When capability compression is in use the base and top of the returned capabiltiy may cover a larger region than requested in order to comply with alignment requirements, however the bounds will never exceed the bounds of the original capabiltiy and the address will still point to the requested base.

`setCapAddr : (Capability, bits(64)) -> (bool, Capability)`

`setCapAddr`(cap, addr) returns a new capability derived from `cap` with the address set to `addr` . If the operation fails due to representability checks then the result will have the expected address but the bounds may be incorrect.

`setCapOffset : (Capability, bits(64)) -> (bool, Capability)`

`setCapOffset`(cap, off) returns a new capability derived from `cap` with the offset set to `off` (i.e. with $address = cap.base + off$). If the operation fails due to representability checks then the result will have the expected address but the bounds may be incorrect. Note that, for performance reasons, an approximate representability check may be used that means the operation could fail even though the result would be representable.

`incCapOffset : (Capability, bits(64)) -> (bool, Capability)`

`incCapOffset` is the same as `setCapOffset` except that the 64-bit value is added to the current capability offset modulo 2^{64} (i.e. signed twos-complement arithemtic).

`sealCap : (Capability, bits(24)) -> (bool, Capability)`

`sealCap`(cap, otype) returns a new capability derived from `cap` but sealed with the given otype. The returned boolean indicates whether the sealed capability is exactly representable (there may be additional restrictions on sealed capabilities depending on the compression used).

`unsealCap : Capability -> Capability`

`unsealCap`(cap) returns a new unsealed capability derived from `cap`. The value of otype is lost.

`int_to_cap : bits(64) -> Capability`

creates a capability with the given 64-bit vector in its offset field. The resulting capability is untagged and has base zero but the other fields are undefined and may vary, depending on the capabilty compression format, to ensure that any offset can be represented.

Capability permissions are accessed using the following functions:

`getCapPerms : Capability -> bits(31)`

gets the permissions of the capabilty as a 31-bit vector. The architecturally defined permissions are in the least significant bits and user defined permissions are in bits 15 and above. Reserved bits are zero.

`setCapPerms : (Capability, bits(31)) -> Capability`

sets the permission of the capabiltiy as a 31-bit vector. Writes to reserved bits are ignored, although this behaviour is subject to change in future revisions.

`getCapFlags : Capability -> CFlags`

Gets the architecture specific capability flags for given capability.

`setCapFlags : (Capability, CFlags) -> Capability`

`setCapFlags(cap, flags)` sets the architecture specific capability flags on `cap` to `flags` and returns the result as new capability.

7.4 Table of CHERI Instructions

Tables 7.2 and 7.3 list available capability coprocessor instructions.

Mnemonic	Description
CGetAddr	Move capability address to an integer register
CGetAndAddr	Move capability address to an integer register, with mask *(experimental)*
CGetBase	Move base to an integer register
CGetFlags	Move flags to an integer register *(experimental)*
CGetLen	Move length to an integer register
CGetOffset	Move offset to an integer register
CGetPerm	Move permissions to an integer register
CGetSealed	Test if a capability is sealed
CGetTag	Move tag bit to an integer register
CGetType	Move object type to an integer register
CPtrCmp	Capability pointer compare
CToPtr	Capability to integer pointer
CAndAddr	Mask address of capability *(experimental)*
CAndPerm	Restrict permissions
CBuildCap	Import a capability *(experimental)*
CClearRegs	Clear multiple registers
CClearTag	Clear the tag bit
CCopyType	Import a capability's **otype** *(experimental)*
CFromPtr	Create capability from pointer
CGetPCC	Move PCC to capability register
CGetPCCSetOffset	Move PCC to capability register with new offset
CIncOffset	Increment offset
CIncOffsetImm	Increment Offset by Immediate
CMove	Move capability
CMOVN	Conditionally move capability on non-zero
CMOVZ	Conditionally move capability on zero
CReadHwr	Read a special-purpose capability register
CSetAddr	Set capability address to value from register
CSetBounds	Set bounds
CSetBoundsExact	Set bounds exactly
CSetBoundsImm	Set bounds (immediate)
CSetFlags	Set flags *(experimental)*
CSetOffset	Set cursor to an offset from base
CSub	Subtract capabilities
CWriteHwr	Write a special-purpose capability register

Figure 7.2: Capability coprocessor instruction summary

Mnemonic	Description
CL[BHWD][U]	Load integer via capability
CLC	Load capability via capability
CLCBI	Load capability via capability (big immediate)
CLL[BHWD][U]	Load linked integer via capability
CLLC	Load linked capability via capability
CSC	Store capability via capability
CS[BHWD]	Store integer via capability
CSC[BHWD]	Store conditional integer via capability
CSCC	Store conditional capability via capability
CBEZ	Branch if capability is NULL
CBNZ	Branch if capability is not NULL
CBTS	Branch if capability tag is set
CBTU	Branch if capability tag is unset
CJALR	Jump and link capability register
CJR	Jump capability register
CCheckPerm	Raise exception on insufficient permission (deprecated)
CCheckTag	Raise exception if capability tag is unset
CCheckType	Raise exception if object types do not match (deprecated)
CSeal	Seal a capability
CCSeal	Conditionally seal a capability *(experimental)*
CUnseal	Unseal a sealed capability
CCall	Call into another security domain
CReturn	Return to the previous security domain
CGetCause	Move the capability exception cause register to an integer register
CGetCID	Move the architectural Compartment ID (CID) to an integer register
CSetCause	Set the capability exception cause register
CSetCID	Set the architectural Compartment ID (CID)

Figure 7.3: Capability coprocessor instruction summary, continued

CAndPerm: Restrict Permissions

Format

CAndPerm cd, cb, rt

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cs	rt	0xd

Description

Capability register *cd* is replaced with the contents of capability register *cb* with the **perms** field set to the bitwise AND of its previous value and bits 0 .. 10 of integer register *rd* and the **uperms** field set to the bitwise and of its previous value and bits *first_uperm* .. *last_uperm* of *rd*.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let rt_val = rGPR(rt);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else
{
  let perms   = getCapPerms(cb_val);
  let newCap  = setCapPerms(cb_val, (perms & rt_val[30..0]));
  writeCapReg(cd, newCap);
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb* is sealed.

CBEZ / CBNZ: Branch if Capability is / is Not Null

Format

CBEZ cb, offset

31 26	25 21	20 16	15 0
0x12	0x11	cb	offset

CBNZ cb, offset

31 26	25 21	20 16	15 0
0x12	0x12	cb	offset

Description

Sets the **PC** to **PC** $+ 4*$*offset* $+ 4$, where *offset* is sign extended, depending on whether *cb* is equal to the NULL capability.

The instruction following the branch, in the delay slot, is executed before branching.

Semantics

```
checkCP2usable();
if InBranchDelay then
   SignalException(ResI);
let cb_val = readCapReg(cb);
if (cb_val == null_cap) ^ notzero then
{
  let offset : bits(64) = sign_extend(imm @ 0b00) + 4;
  execute_branch(PC + offset);
};
NextInBranchDelay = 0b1;
```

Notes

- In the above Sail code `notzero` is false for `CBEZ` and true for `CBNZ` thus inverting the sense of the comparison (via exclusive-or) for the latter.
- Like all MIPS branch instructions, `CBEZ` and `CBNZ` have a branch delay slot. The instruction after it will always be executed, regardless of whether the branch is taken or not.
- This instruction is intended to resemble the conditional branch instructions from the MIPS ISA. In particular, the shift left of the offset by 2 bits and adding 4 is the same as MIPS conditional branches.
- Contrary to previous versions of the CHERI architecture the bounds check on **PCC** is performed during execution of the branch so an out-of-bounds target will result in an exception. In the Sail code this check occurs in the function.

CBTS / CBTU: Branch if Capability Tag is Set / Unset

Format

CBTS cb, offset

31 26	25 21	20 16	15 0
0x12	0x0a	cb	offset

CBTU cb, offset

31 26	25 21	20 16	15 0
0x12	0x09	cb	offset

Description

Sets the **PC** to **PC** $+ 4*$*offset* $+ 4$, where *offset* is sign extended, depending on whether *cb.tag* is set. The instruction following the branch, in the delay slot, is executed before branching.

Semantics

```
checkCP2usable();
if InBranchDelay then
   SignalException(ResI);
let cb_val = readCapReg(cb);
if cb_val.tag ^ notset then
{
  let offset : bits(64) = sign_extend(imm @ 0b00) + 4;
  execute_branch(PC + offset);
};
NextInBranchDelay = 0b1;
```

Notes

- In the above Sail code `notset`, is false for `CBTS` and true for `CBTU` thus inverting the condition (via exclusive-or) for the latter.
- Like all MIPS branch instructions, `CBTS` and `CBTU` have a branch delay slot. The instruction after it will always be executed, regardless of whether the branch is taken or not.
- This instruction is intended to resemble the conditional branch instructions from the MIPS ISA. In particular, the shift left of the offset by 2 bits and adding 4 is the same as MIPS conditional branches.
- Contrary to previous versions of the CHERI architecture the bounds check on **PCC** is performed during execution of the branch so an out-of-bounds target will result in an exception. In the Sail code this check occurs in the function.

CCall: Call into Another Security Domain

Format

CCall cs, cb[, selector]

31 26	25 21	20 16	15 11	10 0
0x12	0x05	cs	cb	selector

Description

CCall is used to make a call between protection domains, unsealing sealed code and data-capability operands, subject to checks on those capabilities. This allows the callee to gain access to a different set of capabilities than its caller, supporting implementation of software encapsulation. The two operand capabilities must be accessible, be valid capabilities, be sealed, have matching types, and have suitable permissions and bounds, or an exception will be thrown. *cs* contains a sealed code capability for the callee subsystem, which will be unsealed and loaded into **PCC**. *cb* contains a sealed data capability for the callee subsystem, which will be unsealed and loaded into **IDC**. In the parlance of object-oriented programming, *cb* is a capability for an *object*'s instance data, and *cs* is a capability for the methods of the object's class. The CCall instruction accepts a *selector* operand that selects between two domain-transition semantics following successful completion of operand capability checks:

0 The protection-domain transition will be implemented by a software exception handler, which will perform any necessary register-file transformation or saving and restoring of state. This mode of operation does not require *Permit_CCall* on the sealed capability operands.

1 The protection-domain transition will be direct, in the style of a jump, without assistance from a software exception handler. The instruction will unseal the sealed operand capabilities and install them as new **PCC** and **IDC** values. This mode of operation requires *Permit_CCall* to be present on both sealed capability operands.

If omitted in assembly, the *selector* field is assumed to be 0. Issuing a CCall instruction with any value other than 0 or 1 is undefined behavior.

With both selectors, a constrained form of non-monotonicity is supported in the architecture. With selector 0, privilege is escalated through a controlled transfer of execution into an exception handler that has additional access to exception-context capability registers (and lower rings). With selector 1, privilege is escalated by virtue of CCall unsealing sealed operand capability registers during a controlled transfer of execution to the callee in a jump-style transfer of control.

Selector 0 - Semantics

Selector 0 implements a software-assisted domain transition via an exception handler:

```
/* Partial implementation of CCall with checks in hardware, but raising a trap to
    perform trusted stack manipulation */
checkCP2usable();
if InBranchDelay then
  SignalException(ResI);
let cs_val = readCapReg(cs);
let cb_val = readCapReg(cb);
let cs_cursor = getCapCursor(cs_val);
if not (cs_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cs)
else if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if not (cs_val.sealed) then
  raise_c2_exception(CapEx_SealViolation, cs)
else if not (cb_val.sealed) then
  raise_c2_exception(CapEx_SealViolation, cb)
else if cs_val.otype != cb_val.otype then
  raise_c2_exception(CapEx_TypeViolation, cs)
else if not (cs_val.permit_execute) then
  raise_c2_exception(CapEx_PermitExecuteViolation, cs)
else if cb_val.permit_execute then
  raise_c2_exception(CapEx_PermitExecuteViolation, cb)
else if cs_cursor < getCapBase(cs_val) then
  raise_c2_exception(CapEx_LengthViolation, cs)
else if cs_cursor >= getCapTop(cs_val) then
  raise_c2_exception(CapEx_LengthViolation, cs)
else
  raise_c2_exception(CapEx_CallTrap, cs);
```

Selector 0 - Exceptions

With selector 0, coprocessor 2 exceptions are raised to both handle failure modes (e.g., that one or both operand capabilities do not have tags set, are not sealed, etc.), and also to implement domain-transition semantics in a trap handler.

Where selector 0 generates the Call Trap exception code, the handler vector 0x100 bytes above the general-purpose exception vector will be used. This alternative vector allows more efficient implementation of fast-path protection-domain switching in a manner similar to the dedicated TLB fill exception vector.

With both selector 0 and selector 1, a coprocessor 2 exception will be raised using the general-purpose exception, rather than dedicated, vector in failure cases:

- *cs* is not sealed.
- *cb* is not sealed.
- cs.**otype** $\neq$ cb.**otype**

- *cs*.**perms**.*Permit_Execute* is not set.
- *cb*.**perms**.*Permit_Execute* is set.
- *cs*.**offset** $\geq$ *cs*.**length**.

Selector 1 - Semantics

Selector 1 implements a jump-like domain transition without using a software exception handler:

```
/* Jump-like implementation of CCall that unseals arguments */
checkCP2usable();
if InBranchDelay then
  SignalException(ResI);
let cs_val = readCapReg(cs);
let cb_val = readCapReg(cb);
let cs_cursor = getCapCursor(cs_val);
if not (cs_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cs)
else if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if not (cs_val.sealed) then
  raise_c2_exception(CapEx_SealViolation, cs)
else if not (cb_val.sealed) then
  raise_c2_exception(CapEx_SealViolation, cb)
else if cs_val.otype != cb_val.otype then
  raise_c2_exception(CapEx_TypeViolation, cs)
else if not (cs_val.permit_ccall) then
  raise_c2_exception(CapEx_PermitCCallViolation, cs)
else if not (cb_val.permit_ccall) then
  raise_c2_exception(CapEx_PermitCCallViolation, cb)
else if not (cs_val.permit_execute) then
  raise_c2_exception(CapEx_PermitExecuteViolation, cs)
else if cb_val.permit_execute then
  raise_c2_exception(CapEx_PermitExecuteViolation, cb)
else if cs_cursor < getCapBase(cs_val) then
  raise_c2_exception(CapEx_LengthViolation, cs)
else if cs_cursor >= getCapTop(cs_val) then
  raise_c2_exception(CapEx_LengthViolation, cs)
else
{
  set_next_pcc(unsealCap(cs_val));
  C26 = unsealCap(cb_val);
  NextPC = to_bits(64, getCapOffset(cs_val));
}
```

CCall with selector 1 executes like a branch in the pipeline, but does not have a branch delay slot. This is due to the difficulty of allowing one instruction from the calling domain to execute in the new domain. See Section 8.24.

Selector 1 - Exceptions

In addition to exceptions that can be thrown by selector 0, selector 1 will raise a coprocessor 2 exception if:

- *cs*.**perms**.*Permit_CCall* is not set
- *cb*.**perms**.*Permit_CCall* is not set

The general-purpose exception vector will be used for these failure-mode exceptions.

Notes

- Selector 0 semantics can be implemented in a number of ways split over hardware and software; we have experimented with several. A simple implementation might have `CCall` throw a software exception, with all other behavior implemented via a software trap handler. A hybrid implementation could perform various checks in hardware, deferring only trusted stack manipulation (or other behaviors, such as asynchronous calling conventions) to the software trap handler. Further defensive coding conventions (beyond instruction semantics) may also sensibly be shifted to the exception handler in order to avoid redundancy – e.g., the clearing of the same registers to prevent leaks in either direction. A significant tension exists in the hardware optimization of this instruction between using a flexible calling convention and semantics versus exploiting hardware optimization opportunities. Authors of compilers or assembly language programs should not rely on `CCall` being implemented in any particular blend of hardware and software.

- From the point of view of security, `CCall` needs to be an atomic operation (i.e. the caller cannot decide to just do some of it, because partial execution could put the system into an insecure state). From a hardware perspective, more complex domain-transition implementations (e.g., to implement function-call semantics or message passing) may need to perform multiple memory reads and writes, which might take multiple cycles and complicate control logic. Supporting both selector 0 and selector 1 semantics for constrained privilege escalation allow software trap handlers or trusted domains to perform those sequences without more complex instructions.

- Implementations may choose to restrict the register numbers that may be passed as *cs* and *cb* in order to avoid the need to decode the instruction and identify the register arguments. The software implementation in CheriBSD at the time of writing requires the *cs* be **C1**, and that *cb* be **C2**, consistent with the CHERI ABI.

- Different microarchitectural tradeoffs exist around exception-like or jump-like semantics for the `CCall` (and corresponding `CReturn`) instructions. For example, exceptions may require greater disruption of speculated instructions in pipeline and superscalar designs. The jump-like semantics may therefore be preferred for this reason, but do require quite different software use of sealed capabilities.

- The 10-bit selector in the CCall instruction allows for the possibility of further semantics being developed – e.g., to model domain transition on hardware multithreading behavior (such as passing values between register files or performing other synchronization), more complex in-hardware sequences including memory access, etc. For example, CCall variations might perform more or less unsealing (e.g., operating only on **PCC**), set up sealed or unsealed link registers for both code and data in the style of a more conventional jump (e.g., by sealing and moving caller **PCC** and **IDC** registers into ABI-reserved registers), or more fully implement models such as the CheriBSD and CheriOS domain transitions as described below (e.g., by pushing return state onto stacks, or implementing message passing).

- In our initial hardware implementation, selector-1 semantics were implemented as selector 42.

- The unsealing of the capabilities stored to PCC and IDC may have implications beyond just the object type of these capabilities. When capability compression is in use, the microarchitectural bit representation of other fields within a capability may depend on the value of the **otype**, so this assignment may have the effect of changing the bit representation of the other fields. i.e., a hardware implementation may need to change the representation of the rest of the capability. (In the deprecated CHERI-128 of Appendix E, for example, which does not have a dedicated **otype** field, CCall clears the sealed bits of the capabilities stored to PCC and IDC but must also zero the bits that held the **otype** values and are now part of the bounds metadata.)

Expected Software Use

Higher-level software protection-domain transitions transform the capability register file to reduce or expand the set of code and data rights available to the executing thread of control. In CHERI-based software, these transitions can be usefully modeled as function invocation or message passing in which data and capability registers are passed as arguments or messsages, and in which callers and callees can be protected from undesired access to internal state from the other party (i.e., encapsulation). Domain transitions may implement symmetric (mutual) or asymmetric distrust between caller and callee, depending on guarantees about limiting callee access to caller state, and vice versa.

Either selector may be used to implement mutual distrust by entering a more privileged “trusted intermediary” able to perform capability and integer register clearing, saving, and restoring, as well as tracking properties of communications such as message passing or implementing a trusted stack for reliable call-return semantics and error recovery. The CCall instruction performs a set of checks on sealed operand capabilities that can be depended on with either selector, allowing domain transition to be more efficient.

With selector 0, the software exception handler will perform any necessary transformation of the register files – e.g., by clearing registers, unsealing and installing a new **PCC** from *cs*, unsealing and installing a new **IDC** from *cb*, or recording a trustworthy return path in a “trusted stack” to implement call-return semantics. This allows implementation of a variety of trust models with varying performance properties; for example, where the caller trusts the callee, less register clearing may be performed. In our CheriBSD software prototype, the `CCall` exception handler implements a strong function-call-like semantic using a trusted stack to support a safe `CReturn`. The sealed code and data capability directly describe the callee protection domain, and so are unsealed and installed in callee capability registers when it starts executing. Returning from the exception prevents further use of privileged exception-handling capabilities.

With selector 1, a number of use cases can be formulated, depending on trust model. To implement mutual distrust, sealed code capabilities must point to an intermediary that is trusted by the callee to implement escalation to callee privilege. With respect to capabilities, the caller can perform its own register clearing and encapsulation of (optional) return state passed via register arguments to the callee. `CCall` selector 1 does not implement a link register, allowing the calling convention to implement semantics not implying a leak **PCC** to the callee. In our CheriOS software prototype, sealed code capabilities refer to one of a set of message-passing implementations, with the sealed data capability describing the message ring and target domain's code and data capabilities. A second `CJR` out of the message-passing implementation into the callee, combined with suitable register clearing, is suitable to deescalate privilege to the callee protection domain without a second use of `CCall`.

Sketch of the CheriBSD CCall Model

The CheriBSD `CCall` model implements domain transition via a short privileged exception handler using selector 0. Modeled on function invocation, the handler depends on hardware-assisted checks (such as of operand register accessibility, validity, sealing, types, and permissions). If the checks pass, the handler will unseal the sealed operand capabilities, installing them in **PCC** and **IDC**. It also clears other non-argument registers to prevent data and capability leakage from caller to callee. In addition, CheriBSD implements a trusted stack that tracks caller **PCC** and **IDC** so that a later `CReturn` can restore control (and security state) one instruction after the original call site. Finally, the CheriBSD handler also implements a form of capability flow control by preventing the passing of non-global capabilities between caller and callee. A corresponding software exception-handler implementation of the `CReturn` instruction will pop an entry from the trusted stack, suitably clear non-return registers, and perform capability flow-control on non-global return capabilities. The CheriBSD `CCall` exception handler operates as follows:

1. **PCC** (with its **offset** field set to the program counter (**PC**) + 4) is pushed onto the trusted system stack.
2. **IDC** is pushed onto the trusted system stack.
3. *cs* is unsealed and the result placed in **PCC**.
4. *cb* is unsealed and the result placed in **IDC**.
5. The program counter is set to *cs*.**offset**. (i.e. control branches to virtual address *cs*.**base** + *cs*.**offset**, because the program counter is relative to **PCC**.**base**).

The CheriBSD `CCall` can be modeled with the following pseudocode:

```
if not cs.tag then
    raise_c2_exception(exceptionTag, cs)
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if not cs.sealed then
    raise_c2_exception(exceptionSealed, cs)
else if not cb.sealed then
    raise_c2_exception(exceptionSealed, cb)
else if cs.otype $\neq$ cb.otype then
    raise_c2_exception(exceptionType, cs)
else if not cs.perms.Permit_Execute then
    raise_c2_exception(exceptionPermitExecute, cs)
else if cb.perms.Permit_Execute then
    raise_c2_exception(exceptionPermitExecute, cb)
else if cs.offset$\geq$ cs.length then
    raise_c2_exception(exceptionLength, cs)
else
    PCC.offset $\leftarrow$ PC $+$ 4
    TSS $\leftarrow$ TSS $-$ capability_size
    mem[TSS .. TSS $+$ capability_size $-$ 1] $\leftarrow$ PCC
    tags[toTag(TSS)] $\leftarrow$ PCC.tag
    TSS $\leftarrow$ TSS $-$ capability_size
    mem[TSS .. TSS $+$ capability_size $-$ 1] $\leftarrow$ IDC
    tags[toTag(TSS)] $\leftarrow$ IDC.tag
    PCC $\leftarrow$ cs
    PCC.sealed $\leftarrow$ false
    PCC.otype $\leftarrow$ 0
    PC $\leftarrow$ cs.offset
    IDC $\leftarrow$ cb
    IDC.sealed $\leftarrow$ false
    IDC.otype $\leftarrow$ 0
end if
```

This software pseudocode may raise a further coprocessor 2 exception if:

- The trusted system stack would overflow (i.e., if **PCC** and **IDC** were pushed onto the system stack, it would overflow the bounds of **TSC**).

The exception handler also clears non-argument capability and integer registers, and prevents the use of argument registers that are valid capabilities but do not have the *global* bit set.

The trusted-stack (TSC) behavior described in the software pseudocode above is not suitable for a RISC-style load-store processor implementation due to its complex combination of control-flow, register-to-register, and memory-access operations.

Sketch of the CheriOS CCall Model

As an alternative to an exception-based implementation, a jump-based interpretation of CCall is also available by setting the selector field to 1. In this case, the architecture allows non-monotonic transformation of the register file when presented with suitable operand capabilities, unsealing the two capabilities into **PCC** and **IDC** without the need for a software exception handler. The "callee" can then use these additional rights to implement domain switching and expansion/reduction of privilege via ordinary loads and register moves.

In the CheriOS model, CCall is used to implement an asynchronous message-passing semantic. The sealed code capability is directed to a software message-passing implementation that acts as a "trusted intermediary", and the sealed data capability refers to a description of the destination domain including message ring. The message-passing implementation adds argument registers to the ring, and will then either return control to the sender context, or continue in to the recipient context. This is accomplished by suitable register-file manipulation to give up any unnecessary privilege, and an ordinary capability jump to pass control to an appropriate unprivileged domain. As with the CheriBSD model, the message-passing routine must perform any necessary saving of caller context, checking and clearing of registers, and installation of callee context to support safe interactions.

CCheckPerm: Raise Exception on Insufficient Permission (DEPRECATED)

Format

CCheckPerm cs, rt

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cs	rt	0x8	0x3f

Description

An exception is raised (and the capability cause set to “user-defined permission violation”) if there is a bit set in *rt* that is not set in *cs*.**perms** (i.e., *rt* describes a set of permissions, and an exception is raised if *cs* does not grant all of those permissions).

Semantics

```
checkCP2usable();
let cs_val = readCapReg(cs);
let cs_perms : bits(64) = zero_extend(getCapPerms(cs_val));
let rt_perms = rGPR(rt);
if not (cs_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cs)
else if (cs_perms & rt_perms) != rt_perms then
  raise_c2_exception(CapEx_UserDefViolation, cs)
```

Exceptions

A coprocessor 2 exception is raised if:

- *cs*.**tag** is not set.
- There is a bit that is set in *rt* and is not set in *cs*.**perms**.
- There is a bit that is set in *rt* and is not set in *cs*.**uperms**.

Notes

- This instruction has been deprecated, and will be removed from future versions of the CHERI architecture. The original use case for which this instruction was imagined, argument checking for object-capability invocation, has not proven to be a fast path requiring architectural acceleration. Further, it is more likely that compartmentalization runtimes and applications will prefer a branching, rather than trapping, instruction.
- If *cs*.**tag** is not set, then *cs* does not contain a capability, *cs*.**perms** might not be meaningful as a permissions field, and so a *tagViolation* exception is raised.
- This instruction can be used to check the permissions field of a sealed capability, so the instruction does not check whether *cs* is sealed.

CCheckTag: Raise Exception if Tag is Unset

Format

CCheckTag cs

0x12	0x0	cs	0x6	0x1f	0x3f

Description

An exception is raised (and the capability cause register set to “tag violation”) if *cs*.**tag** is not set.

Semantics

```
checkCP2usable();
let cs_val = readCapReg(cs);
if not(cs_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cs);
```

Exceptions

A coprocessor 2 exception is raised if:

- *cs*.**tag** is not set.

Notes

- This instruction is intended to support debugging modes for compilers where an untagged capability may result from an attempted non-monotonic operation, rather than an exception.

CCheckType: Raise Exception if Object Types Do Not Match (DEPRECATED)

Format

CCheckType cs, cb

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cs	cb	0x9	0x3f

Description

An exception is raised if *cs*.**otype** is not equal to *cb*.**otype**.

Semantics

```
checkCP2usable();
let cs_val = readCapReg(cs);
let cb_val = readCapReg(cb);
if not (cs_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cs)
else if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if not (cs_val.sealed) then
  raise_c2_exception(CapEx_SealViolation, cs)
else if not (cb_val.sealed) then
  raise_c2_exception(CapEx_SealViolation, cb)
else if cs_val.otype != cb_val.otype then
  raise_c2_exception(CapEx_TypeViolation, cs)
```

Exceptions

A coprocessor 2 exception is raised if:

- *cs*.**tag** is not set.
- *cb*.**tag** is not set.
- *cs* is not sealed.
- *cb* is not sealed.
- *cs*.**otype**$\neq$ *cb*.**otype**.

Notes

- This instruction has been deprecated, and will be removed from future versions of the CHERI architecture. The original use case for which this instruction was imagined, argument checking for object-capability invocation, has not proven to be a fast path requiring architectural acceleration. Further, it is more likely that compartmentalization runtimes and applications will prefer a branching, rather than trapping, instruction.

CClearRegs: Clear Multiple Registers

Format

ClearLo mask
ClearHi mask
CClearLo mask
CClearHi mask
FPClearLo mask
FPClearHi mask

31 26	25 21	20 16	15 0
0x12	0x0f	regset	mask

Description

The registers in the target register set, *regset*, corresponding to the set bits in the immediate *mask* field are cleared. That is, if bit 0 of *mask* is set, then the lowest numbered register in *regset* is cleared, and so on. The following values are defined for the *regset* field:

Mnemonic	*regset*	Affected registers
ClearLo	0	**R0–R15**
ClearHi	1	**R16–R31**
CClearLo	2	**DDC**, **C1–C15**
CClearHi	3	**C16–C31**
FPClearLo	4	**F0–F15**
FPClearHi	5	**F16–F31**

For integer registers, clearing means setting to zero. For capability registers, clearing consists of setting all capability fields such that the in-memory representation will be all zeroes, with a cleared tag bit, granting no rights.

Note: For CClearLo bit 0 in *mask* refers to **DDC** and not **C0** since **C0** is the NULL register.

Semantics

```
if ((regset == CLo) | (regset == CHi)) then
  checkCP2usable();
foreach (i from 0 to 15)
  if (m[i]) then
    match regset {
      GPLo  => wGPR(to_bits(5, i)) = zeros(),
      GPHi  => wGPR(to_bits(5, i+16)) = zeros(),
      CLo   => if i == 0 then
                 DDC = null_cap
```

```
              else
                writeCapReg(to_bits(5, i)) = null_cap,
    CHi    => writeCapReg(to_bits(5, i+16)) = null_cap
  }
```

Exceptions

- A Reserved Instruction exception is raised for unknown or unimplemented values of *regset*.
- CClearLo and CClearHi raise a coprocessor unusable exception if the capability coprocessor is disabled.
- FPClearLo and FPClearHi raise a coprocessor unusable exception if the floating point unit is disabled.

Notes

- These instructions are designed to accelerate the register clearing that is required for secure domain transitions. It is expected that they can be implemented efficiently in hardware using a single 'valid' bit per register that is cleared by the ClearRegs instruction and set on any subsequent write to the register.
- The mnemonic for the integer-register instruction does not make it very clear what the instruction does. It would be preferable to have a more descriptive mnemonic.

CClearTag: Clear the Tag Bit

Format

CClearTag cd, cb

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cb	0xb	0x3f

Description

Capability register *cd* is replaced with the contents of *cb*, with the tag bit cleared.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
writeCapReg(cd, {cb_val with tag=false});
```

CFromPtr: Create Capability from Pointer

Format

CFromPtr cd, cb, rt

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cb	rt	0x13

Note: If the encoded value of *cb* is zero, this instruction will use **DDC** as the *cb* operand

Description

rt is a pointer using the C-language convention that a zero value represents the NULL pointer. If *rt* is zero, then *cd* will be the NULL capability (tag bit not set, all other bits also not set). If *rt* is non-zero, then *cd* will be set to *cb* with the **offset** field set to *rt*.

Semantics

```
checkCP2usable();
let cb_val = readCapRegDDC(cb);
let rt_val = rGPR(rt);
if rt_val == 0x0000000000000000 then
  writeCapReg(cd, null_cap)
else if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else
{
  let (success, newCap) = setCapOffset(cb_val, rt_val);
  if success then
      writeCapReg(cd, newCap)
  else
      writeCapReg(cd, int_to_cap(to_bits(64, getCapBase(cb_val)) + rt_val))
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set and $rt \neq 0$.
- *cb* is sealed and $rt \neq 0$.

Notes

- CSetOffset doesn't raise an exception if the tag bit is unset, so that it can be used to implement the *intcap_t* type. CFromPtr raises an exception if the tag bit is unset: although it would not be a security violation to to allow it, it is an indication that the program is in error.
- The encodings of the NULL capability are chosen so that zeroing memory will set a capability variable to NULL. This holds true for compressed capabilities as well as the 256-bit version.

CGetAddr: Move Capability Address to an Integer Register

Format

CGetAddr rd, cb

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	0xf	0x3f

Description

rd is set to *cb*.**base** + *cb*.**offset**, returning the virtual address of a capability in an integer register. This instruction may be appropriate for use cases in which C is utilizing a virtual-address rather than offset integer interpretation of capabilities, or in which integer values are stored in capability registers.

Semantics

```
checkCP2usable();
let capVal = readCapReg(cb);
wGPR(rd) = capVal.address;
```

Notes

- This differs from `CToPtr` in that it does not perform any range checks with respect to an authorizing capability, nor require that *cb* be a valid and unsealed capability.

CGetBase: Move Base to an Integer Register

Format

CGetBase rd, cb

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	0x2	0x3f

Description

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

Semantics

```
checkCP2usable();
let capVal = readCapReg(cb);
wGPR(rd) = to_bits(64, getCapBase(capVal));
```

CGetCID: Move the Architectural Compartment ID to an Integer Register

Format

CGetCID rd

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	0x4	0x1f	0x3f

Description

Move the architectural Compartment ID (CID) to integer register *rd*, retrieving the last value set by `CSetCID`.

Semantics

```
checkCP2usable();
wGPR(rd) = CID;
```

Exceptions

This instruction does not raise any exceptions.

Notes

- We choose not to require any additional privilege to query the CID. An argument could be made that this is an information leak. If so, one possible design choice would be to require Access_System_Registers to retrieve the value of the CID; however, this would be inconsistent with the access-control model for setting the CID.

- While this instruction has been introduced for debugging purposes, it could also have utility in indexing state – e.g., to implement concepts such as compartment-local storage.

CGetCause: Move the Capability Exception Cause Register to an Integer Register

Format

CGetCause rd

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	0x1	0x1f	0x3f

Description

Integer register *rd* is set equal to the capability cause register.

Semantics

```
checkCP2usable();
if not (pcc_access_system_regs ()) then
  raise_c2_exception_noreg(CapEx_AccessSystemRegsViolation)
else
  wGPR(rd) = zero_extend(CapCause.bits())
```

Exceptions

A coprocessor 2 exception is raised if:

- **PCC**.**perms**.*Access_System_Registers* is not set.

CGetLen: Move Length to an Integer Register

Format

CGetLen rd, cb

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	0x3	0x3f

Description

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

Semantics

```
checkCP2usable();
let capVal = readCapReg(cb);
let len65  = getCapLength(capVal);
wGPR(rd) = to_bits(64, if len65 > MAX_U64 then MAX_U64 else len65);
```

Notes

- With the 256-bit representation of capabilities, **length** is a 64-bit unsigned integer and can never be greater than $2^{64} - 1$. With the 128-bit compressed representation of capabilities, the result of decompressing the length can be 2^{64}; CGetLen will return the maximum value of $2^{64} - 1$ in this case.

CGetOffset: Move Offset to an Integer Register

Format

CGetOffset rd, cb

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	0x6	0x3f

Description

Integer register *rd* is set equal to the **offset** fields of capability register *cb*.

Semantics

```
checkCP2usable();
let capVal = readCapReg(cb);
wGPR(rd) = to_bits(64, getCapOffset(capVal));
```

CGetPCC: Move PCC to Capability Register

Format

CGetPCC cd

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	0x0	0x1f	0x3f

Description

Capability register *cd* is set equal to the **PCC**, with cd.**offset** set equal to **PC**.

Semantics

```
checkCP2usable();
let (success, pcc) = setCapOffset(PCC, PC);
assert (success, "PCC with offset PC should always be representable");
writeCapReg(cd, pcc);
```

CGetPCCSetOffset: Move PCC to Capability Register with New Offset

Format

CGetPCCSetOffset cd, rs

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	rs	0x7	0x3f

Description

Capability register *cd* is set equal to the **PCC**, with cd.**offset** set equal to *rs*.

Semantics

```
checkCP2usable();
let rs_val = rGPR(rs);
let (success, newPCC) = setCapOffset(PCC, rs_val);
if (success) then
  writeCapReg(cd, newPCC)
else
  writeCapReg(cd, int_to_cap(rs_val));
```

Notes

- This instruction is a peformance optimization; a similar effect can be achieved with `CGetPCC` followed by `CSetOffset`.

CGetPerm: Move Permissions to an Integer Register

Format

CGetPerm rd, cb

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	0x0	0x3f

Description

The least significant 11 bits of integer register *rd* are set equal to the **perms** field of capability register *cb*; bits *first_uperm* to *last_uperm* of *rd* are set equal to the **uperms** field of *cb*. The other bits of *rd* are set to zero.

Semantics

```
checkCP2usable();
let capVal = readCapReg(cb);
wGPR(rd) = zero_extend(getCapPerms(capVal));
```

CGetSealed: Test if Capability is Sealed

Format

CGetSealed rd, cb

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	0x5	0x3f

Description

The low-order bit of integer register *rd* is set to 0 if *cb*.**otype** is $2^{64} - 1$ and 1 otherwise. All other bits of *rd* are cleared.

Semantics

```
checkCP2usable();
let capVal = readCapReg(cb);
wGPR(rd) = zero_extend(capVal.sealed);
```

CGetTag: Move Tag Bit to an Integer Register

Format

CGetTag rd, cb

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	0x4	0x3f

Description

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

Semantics

```
checkCP2usable();
let capVal = readCapReg(cb);
wGPR(rd) = zero_extend(capVal.tag);
```

CGetType: Move Object Type to an Integer Register

Format

CGetType rd, cb

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	0x1	0x3f

Description

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

Semantics

```
checkCP2usable();
let capVal = readCapReg(cb);
wGPR(rd) = if capVal.sealed
  then zero_extend(capVal.otype)
  else 0xffffffffffffffff
```

Notes

- If the capability is not sealed, a value of -1 is returned. As the allowed values of **otype** in a sealed capability are non-negative, this makes it easy to tell from the result of CGetType whether the capability was sealed or unsealed. This might be used, for example, in an efficient routine for paging capabilities back into memory from swap.

CIncOffset: Increment Offset

Format

CIncOffset cd, cb, rt

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cb	rt	0x11

Description

Capability register *cd* is set equal to capability register *cb* with its **offset** field replaced with *cb*.**offset** $+$ *rt*.

If capability compression is in use, and the requested **base**, **length** and **offset** cannot be represented exactly, then *cd*.**tag** is cleared, *cd*.**base** and *cd*.**length** are set to zero, *cd*.**perms** is cleared, and *cd*.**offset** is set equal to *cb*.**base** $+$ *cb*.**offset** $+$ *rt*.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let rt_val = rGPR(rt);
if cb_val.tag & cb_val.sealed & (rt_val != 0x0000000000000000) then
  raise_c2_exception(CapEx_SealViolation, cb)
else
{
  let (success, newCap) = incCapOffset(cb_val, rt_val);
  if success then
      writeCapReg(cd, newCap)
  else
      writeCapReg(cd, int_to_cap(cb_val.address + rt_val))
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is set and *cb* is sealed.

Notes

- For security reasons, `CIncOffset` must not change the offset of a sealed capability.
- As a special case, we allow `CIncOffset` with an offset of zero to work on sealed capabilities; this is so that `CIncOffset` can be used as a capability move instruction.
- If the tag bit is not set, and the offset is being used to hold an integer, then `CIncOffset` should still increment the offset. This is so that `CIncOffset` can be used to implement increment of a `intcap_t` type. Because the tag is unset, `CIncOffset` will ignore the decoded

otype and so will not attempt to enforce immutability of detagged sealed capabilities. (The precise effect of `CIncOffset` on such non-capability data will depend on which binary representation of capabilities is being used.)

- If the tag bit is not set, and capability compression is in use, the arbitrary data in *cb* might not decompress to sensible values of the **base** and **length** fields, and there is no guarantee that retaining these values of **base** and **length** while changing **offset** will result in a representable value.

  From a software perspective, the requirement is that incrementing **offset** on an untagged capability will work if **base** and **length** are zero. (This is how integers, and pointers that have lost precision, will be represented). If **base** and **length** have non-zero values (or *cb* cannot be decompressed at all), then the values of **base** and **length** after this instruction are **UNPREDICTABLE**.

- In assembly language, `CMove` *cd*, *cb* is a pseudo-instruction that the assembler converts to `CIncOffset` *cd*, *cb*, *$zero*.

CIncOffsetImm: Increment Offset by Immediate

Format

CIncOffset cd, cb, increment_{imm}

31 26	25 21	20 16	15 11	10 0
0x12	0x13	cd	cb	increment_{imm}

Description

Capability register *cd* is set equal to capability register *cb* with its **offset** field replaced with *cb*.**offset** + $\textit{increment}_{imm}$.

If capability compression is in use, and the requested **base**, **length** and **offset** cannot be represented exactly, then *cd*.**tag** is cleared, *cd*.**base** and *cd*.**length** are set to zero, *cd*.**perms** is cleared, and *cd*.**offset** is set equal to *cb*.**base** + *cb*.**offset** + $\textit{increment}_{imm}$.

Semantics

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is set and *cb* is sealed.

Notes

- For security reasons, `CIncOffsetImm` must not change the offset of a sealed capability.

- If the tag bit is not set, and the offset is being used to hold an integer, then `CIncOffsetImm` should still increment the offset. This is so that `CIncOffsetImm` can be used to implement increment of a `intcap_t` type. Because the tag is unset, `CIncOffset` will ignore the decoded **otype** and so will not attempt to enforce immutability of detagged sealed capabilities. (The precise effect of `CIncOffset` on such non-capability data will depend on which binary representation of capabilities is being used.)

- If the tag bit is not set, and capability compression is in use, the arbitrary data in *cb* might not decompress to sensible values of the **base** and **length** fields, and there is no guarantee that retaining these values of **base** and **length** while changing **offset** will result in a representable value.

  From a software perspective, the requirement is that increasing **offset** on an untagged capability will work if **base** and **length** are zero. (This is how integers, and pointers that have lost precision, will be represented). If **base** and **length** have non-zero values (or *cb* cannot be decompressed at all), then the values of **base** and **length** after this instruction are **UNPREDICTABLE**.

CJR / CJALR: Jump (and Link) Capability Register

Format

CJALR cb, cd

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cb	0xc	0x3f

CJR cb

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cb	0x3	0x1f	0x3f

Description

The current **PCC** (with an offset of the current **PC** + 8) is optionally saved in *cd*. **PCC** is then loaded from capability register *cb* and **PC** is set from its offset.

Semantics

```
checkCP2usable();
if InBranchDelay then
   SignalException(ResI);
let cb_val = readCapReg(cb);
let cb_ptr = getCapCursor(cb_val);
let cb_top = getCapTop(cb_val);
let cb_base= getCapBase(cb_val);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if (cb_val.sealed) then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_execute) then
  raise_c2_exception(CapEx_PermitExecuteViolation, cb)
else if cb_ptr < cb_base then
  raise_c2_exception(CapEx_LengthViolation, cb)
else if (cb_ptr + 4) > cb_top then
  raise_c2_exception(CapEx_LengthViolation, cb)
else if (cb_ptr % 4) != 0 then
  SignalException(AdEL)
else
{
  if link then
  {
    let (success, linkCap) = setCapOffset(PCC, PC+8);
    assert(success, "Link cap should always be representable.");
    writeCapReg(cd, linkCap);
  };
```

```
    execute_branch_pcc(cb_val);
  };
  NextInBranchDelay = 0b1;
```

Exceptions

A coprocessor 2 exception will be raised if:

- *cb*.**tag** is not set.
- *cb* is sealed. (But see Appendix D.12.)
- *cb*.**perms**.*Permit_Execute* is not set.
- *cb*.**offset** + 4 is greater than *cb*.**length**.

An address error exception will be raised if

- *cb*.**base** + *cb*.**offset** is not 4-byte word aligned.

Notes

- CJALR has a branch delay slot.
- The change to **PCC** does not take effect until the instruction in the branch delay slot has been executed.
- *cb*.**base** and *cb*.**length** are treated as unsigned integers, and the result of the addition does not wrap around (i.e., an exception is raised if *cb*.**base**+*cb*.**offset** is greater than maxaddr).

Load Integer via Capability Register

Format

CLB rd, rt, offset(cb)
CLH rd, rt, offset(cb)
CLW rd, rt, offset(cb)
CLD rd, rt, offset(cb)
CLBU rd, rt, offset(cb)
CLHU rd, rt, offset(cb)
CLWU rd, rt, offset(cb)
CLBR rd, rt(cb)
CLHR rd, rt(cb)
CLWR rd, rt(cb)
CLDR rd, rt(cb)
CLBUR rd, rt(cb)
CLHUR rd, rt(cb)
CLWUR rd, rt(cb)
CLBI rd, offset(cb)
CLHI rd, offset(cb)
CLWI rd, offset(cb)
CLDI rd, offset(cb)
CLBUI rd, offset(cb)
CLHUI rd, offset(cb)
CLWUI rd, offset(cb)

31 26	25 21	20 16	15 11	10 3		1 0
0x32	rd	cb	rt	offset	s	t

Note: If the encoded value of *cb* is zero, this instruction will use **DDC** as the *cb* operand

Purpose

Loads a data value via a capability register, and extends the value to fit the target register.

Description

The lower part of integer register *rd* is loaded from the memory location specified by *cb*.**base** + *cb*.**offset** + *rt* + $2^t * offset$. Capability register *cb* must contain a valid capability that grants permission to load data.

The size of the value loaded depends on the value of the *t* field:

0 byte (8 bits)

1 halfword (16 bits)

2 word (32 bits)

3 doubleword (64 bits)

The extension behavior depends on the value of the *s* field: 1 indicates sign extend, 0 indicates zero extend. For example, `CLWU` is encoded by setting *s* to 0 and *t* to 2, `CLB` is encoded by setting *s* to 1 and *t* to 0.

Semantics

```
checkCP2usable();
let cb_val = readCapRegDDC(cb);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_load) then
  raise_c2_exception(CapEx_PermitLoadViolation, cb)
else
{
  let 'size   = wordWidthBytes(width);
  let cursor  = getCapCursor(cb_val);
  let vAddr   = (cursor + unsigned(rGPR(rt)) + size*signed(offset)) % pow2(64);
  let vAddr64 = to_bits(64, vAddr);
  if (vAddr + size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if vAddr < getCapBase(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if not (isAddressAligned(vAddr64, width)) then
    SignalExceptionBadAddr(AdEL, vAddr64)
  else
  {
    let pAddr = TLBTranslate(vAddr64, LoadData);
    memResult : bits(64) = extendLoad(MEMr_wrapper(pAddr, size), signext);
    wGPR(rd) = memResult;
  }
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb* is sealed.
- *cb*.**perms**.*Permit_Load* is not set.
- $addr + size > cb.\textbf{base} + cb.\textbf{length}$

  NB: The check depends on the size of the data loaded.

- $addr < cb$.**base**

An AdEL exception is raised if *addr* is not correctly aligned.

Notes

- This instruction reuses the opcode from the Load Word to Coprocessor 2 (`LWC2`) instruction in the MIPS Specification.
- *rt* is treated as an unsigned integer.
- *offset* is treated as a signed integer.
- BERI1 has a compile-time option to allow unaligned loads and stores. If BERI1 is built with this option, an unaligned load will only raise an exception if it crosses a cache line boundary.

CLC: Load Capability via Capability

Format

CLC cd, rt, offset(cb)
CLCR cd, rt(cb)
CLCI cd, offset(cb)

31 26	25 21	20 16	15 11	10 6 5 0
0x36	cd	cb	rt	offset

Note: If the encoded value of *cb* is zero, this instruction will use **DDC** as the *cb* operand

Description

Capability register *cd* is loaded from the memory location specified by *cb*.**base** + *cb*.**offset** + *rt* + *offset*. Capability register *cb* must contain a capability that grants permission to load capabilities. The virtual address *cb*.**base** + *cb*.**offset** + *rt* + *offset* must be *capability_size* aligned.

The bit in the tag memory corresponding to *cb*.**base** + *cb*.**offset** + *rt* + *offset* is loaded into the tag bit associated with *cd*.

Semantics

```
checkCP2usable();
let cb_val = readCapRegDDC(cb);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_load) then
  raise_c2_exception(CapEx_PermitLoadViolation, cb)
else
{
  let cursor  = getCapCursor(cb_val);
  let vAddr   = (cursor + unsigned(rGPR(rt)) + 16 * signed(offset)) % pow2(64);
  let vAddr64 = to_bits(64, vAddr);
  if (vAddr + cap_size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if vAddr < getCapBase(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if (vAddr % cap_size) != 0 then
    SignalExceptionBadAddr(AdEL, vAddr64)
  else
  {
    let (pAddr, suppressTag) = TLBTranslateC(vAddr64, LoadData);
    let (tag, mem) = MEMr_tagged(pAddr, cap_size);
```

```
      let cap = memBitsToCapability(tag & cb_val.permit_load_cap & not (suppressTag)
          , mem);
      writeCapReg(cd, cap);
    }
  }
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb* is sealed.
- *cb*.**perms**.*Permit_Load* is not set.
- *addr* + *capability_size* $>$ *cb*.**base** $+$ *cb*.**length**.
- *addr* $<$ *cb*.**base**.

An address error during load (AdEL) exception is raised if:

- The virtual address *addr* is not *capability_size* aligned.

Notes

- This instruction reuses the opcode from the Load Doubleword to Coprocessor 2 (LDC2) instruction in the MIPS Specification.
- *offset* is interpreted as a signed integer.
- The CLCI mnemonic is equivalent to CLC with *cb* being the zero register ($zero). The CLCR mnemonic is equivalent to CLC with *offset* set to zero.
- Although the *capability_size* can vary, the offset is always in multiples of 16 bytes (128 bits).

CLCBI: Load Capability via Capability (Big Immediate)

Format

CLCBI cd, offset(cb)

31 26	25 21	20 16	15 11	10 6	5 0
0x1d	cd	cb	offset		

Note: If the encoded value of *cb* is zero, this instruction will use **DDC** as the *cb* operand

Description

Capability register *cd* is loaded from the memory location specified by *cb*.**base** + *cb*.**offset** + *offset*. Capability register *cb* must contain a capability that grants permission to load capabilities. The virtual address *cb*.**base** + *cb*.**offset** + *offset* must be *capability_size* aligned.

The bit in the tag memory corresponding to *cb*.**base** + *cb*.**offset** + *offset* is loaded into the tag bit associated with *cd*.

Semantics

```
checkCP2usable();
let cb_val = readCapRegDDC(cb);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_load) then
  raise_c2_exception(CapEx_PermitLoadViolation, cb)
else
{
  let cursor  = getCapCursor(cb_val);
  let vAddr   = (cursor + 16 * signed(offset)) % pow2(64);
  let vAddr64 = to_bits(64, vAddr);
  if (vAddr + cap_size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if vAddr < getCapBase(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if (vAddr % cap_size) != 0 then
    SignalExceptionBadAddr(AdEL, vAddr64)
  else
  {
    let (pAddr, suppressTag) = TLBTranslateC(vAddr64, LoadData);
    let (tag, mem) = MEMr_tagged(pAddr, cap_size);
    let cap = memBitsToCapability(tag & cb_val.permit_load_cap & not (suppressTag)
        , mem);
    writeCapReg(cd, cap);
    }
  }
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb* is sealed.
- *cb*.**perms**.*Permit_Load* is not set.
- *addr* + *capability_size* $>$ *cb*.**base** $+$ *cb*.**length**.
- *addr* $<$ *cb*.**base**.

An address error during load (AdEL) exception is raised if:

- The virtual address *addr* is not *capability_size* aligned.

Notes

- This instruction reuses the opcode from the Jump and Link Exchange (`JALX`) instruction in the MIPS Specification. Future versions of the architecture may use a different encoding to avoid reusing an opcode with a delay slot for an instruction without a delay slot.
- *offset* is interpreted as a signed integer.
- Although the *capability_size* can vary, the offset is always in multiples of 16 bytes (128 bits).
- The larger immediate of `CLCBI`) enables more efficient code generation in pure-capability programs for accesses of global variables. In many programs and libraries, the 11-bit immediate offset of `CLC` is not sufficient reach all entries in the table of global capabilities and therefore the compiler must to generate a three-instruction sequence instead. By using of `CLCBI`) for accessing globals, the code size of most pure-capability binaries can be reduced by over 10%.
- Architectures with pc-relative loads or instructions to add a large immediate to **PCC** (such as `AUIPC`) can use those instead of adding a capability load with a larger immediate offset.

CLL[BHWD][U]: Load Linked Integer via Capability

Format

CLLB rd, cb
CLLH rd, cb
CLLW rd, cb
CLLD rd, cb
CLLBU rd, cb
CLLHU rd, cb
CLLWU rd, cb

31 26	25 21	20 16	15 11	10 4	3	2	1 0
0x12	0x10	rd	cb	▨	1	s	t

Note: If the encoded value of *cb* is zero, this instruction will use **DDC** as the *cb* operand

Description

`CLL[BHWD][U]` and `CSC[BHWD]` are used to implement safe access to data shared between different threads. The typical usage is that `CLL[BHWD][U]` is followed (an arbitrary number of instructions later) by `CSC[BHWD]` to the same address; the `CSC[BHWD]` will only succeed if the memory location that was loaded by the `CLL[BHWD][U]` has not been modified.

The exact conditions under which `CSC[BHWD]` fails are implementation dependent, particularly in multicore or multiprocessor implementations). The following code is intended to represent the security semantics of the instruction correctly, but should not be taken as a definition of the CPU's memory coherence model.

Semantics

```
checkCP2usable();
let cb_val = readCapRegDDC(cb);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_load) then
  raise_c2_exception(CapEx_PermitLoadViolation, cb)
else
{
  let 'size   = wordWidthBytes(width);
  let vAddr   = getCapCursor(cb_val);
  let vAddr64 = to_bits(64, vAddr);
  if (vAddr + size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if vAddr < getCapBase(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if not (isAddressAligned(vAddr64, width)) then
    SignalExceptionBadAddr(AdEL, vAddr64)
  else
  {
    let pAddr = TLBTranslate(vAddr64, LoadData);
    let memResult : bits(64) = extendLoad(MEMr_reserve_wrapper(pAddr, size),
        signext);
    CP0LLBit  = 0b1;
    CP0LLAddr = pAddr;
    wGPR(rd) = memResult;
  }
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb* is sealed.
- *cb*.**perms**.*Permit_Load* is not set.
- $addr + size > cb.\mathbf{base} + cb.\mathbf{length}$
- $addr < cb.\mathbf{base}$

An AdEL exception is raised if *addr* is not correctly aligned.

CLLC: Load Linked Capability via Capability

Format

CLLC cd, cb

31 26	25 21	20 16	15 11	10 3	2 0
0x12	0x10	cd	cb		1111

Note: If the encoded value of *cb* is zero, this instruction will use **DDC** as the *cb* operand

Semantics

```
checkCP2usable();
let cb_val = readCapRegDDC(cb);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_load) then
  raise_c2_exception(CapEx_PermitLoadViolation, cb)
else
{
  let vAddr   = getCapCursor(cb_val);
  let vAddr64 = to_bits(64, vAddr);
  if (vAddr + cap_size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if vAddr < getCapBase(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if (vAddr % cap_size) != 0 then
    SignalExceptionBadAddr(AdEL, vAddr64)
  else
  {
    let (pAddr, suppressTag) = TLBTranslateC(vAddr64, LoadData);
    let (tag, mem) : (bool, CapBits) = MEMr_tagged_reserve(pAddr, cap_size);
    let cap = memBitsToCapability(tag & cb_val.permit_load_cap & not (suppressTag)
        , mem);
    writeCapReg(cd, cap);
    CP0LLBit  = 0b1;
    CP0LLAddr = pAddr;
  }
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.

- *cb* is sealed.
- *cb*.**perms**.*Permit_Load* is not set.
- *addr* + *capability_size* > *cb*.**base** + *cb*.**length**
- *addr* < *cb*.**base**

An AdEL exception is raised if:

- *addr* is not capability aligned.

CMove: Move Capability to another Register

Format

CMove cd, cb

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cb	0xa	0x3f

Description

`CMove` copies *cb* into *cd*.

Semantics

```
checkCP2usable();
writeCapReg(cd) = readCapReg(cb);
```

Notes

- This instruction currently has a dedicated encoding but it could also be implemented as an alias for `CMOVZ` *$zero*, *cd*, *cb*.
- Originally, `CMove` was an assembler pseudo for `CIncOffset` *cd*, *cb*, *$zero*. However, this requires that `CIncOffset` with a sealed capability succeeds if the increment is zero. A future version of the ISA might no longer support this and require the use of `CMove` for sealed capabilities. This would allow for a simpler implementation of `CIncOffset` where the behavior does not depend on one of the input values.

CMOVZ / CMOVN: Conditionally Move Capability on Zero / Non-Zero

Format

CMOVN cd, cb, rt

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cb	rt	0x1c

CMOVZ cd, cb, rt

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cb	rt	0x1b

Description

CMOVZ copies *cb* into *cd* if $rt = 0$.
CMOVN copies *cb* into *cd* if $rt \neq 0$.

Semantics

```
checkCP2usable();
if (rGPR(rt) == zeros()) ^ ismovn then
  writeCapReg(cd) = readCapReg(cb);
```

Notes

- In the sail code `ismovn` is true for `CMOVN`, thus inverting the condition (via exclusive-or) in that case.
- Some implementations of cryptographic algorithms need a constant-time move operation to avoid revealing secret key material through a timing channel. (An attacker must not be able to determine whether a condition variable inside the cryptographic implementation is true or false from observations of how long the operation took to complete). In the current prototype implementation of CHERI, no guarantees are made about `CMOVN` being constant time.

  If CHERI instructions are to be used in high-security cryptographic processors, consideration should be given to making this operation constant time.

CPtrCmp: CEQ, CNE, CL[TE][U], CEXEQ, CNEXEQ: Capability Pointer Compare

Format

CEQ rd, cb, ct

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	ct	0x14

CNE rd, cb, ct

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	ct	0x15

CLT rd, cb, ct

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	ct	0x16

CLE rd, cb, ct

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	ct	0x17

CLTU rd, cb, ct

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	ct	0x18

CLEU rd, cb, ct

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	ct	0x19

CEXEQ rd, cb, ct

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	ct	0x21

CNEXEQ rd, cb, ct

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	ct	0x1a

Description

Capability registers *cb* and *ct* are compared, and the result of the comparison is placed in integer register *rd*. The rules for comparison are as follows:

- A capability with the **tag** bit unset is less than any capability with the **tag** bit set.
- Otherwise, the result of comparison is the result of comparing (**base** + **offset**) $\bmod 2^{64}$ for the two capabilities. Numerical comparison is signed for `CLT` and `CLE`, and unsigned for `CLTU` and `CLEU`.
- `CExEq` and `CNExEq` compare all the fields of the two capabilities, including **tag** and the bits that are reserved for future use.

This instruction can be used to compare capabilities so that capabilities can replace pointers in C executables.

Mnemonic	*t*	Comparison
`CEQ`	0	$=$
`CNE`	1	$\neq$
`CLT`	2	$<$ (signed)
`CLE`	3	$\leq$ (signed)
`CLTU`	4	$<$ (unsigned)
`CLEU`	5	$\leq$ (unsigned)
`CEXEQ`	6	all fields are equal
`CNEXEQ`	7	not all fields are equal

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let ct_val = readCapReg(ct);
equal : bool = false;
ltu   : bool = false;
lts   : bool = false;
if cb_val.tag != ct_val.tag then
  {
    if not (cb_val.tag) then
      {
        ltu = true;
        lts = true;
      }
  }
else
  {
    cursor1 = getCapCursor(cb_val);
    cursor2 = getCapCursor(ct_val);
    equal   = (cursor1 == cursor2);
    ltu     = (cursor1 < cursor2);
    lts     = to_bits(64, cursor1) <_s to_bits(64, cursor2);
  };
let cmp : bool = match op {
  CEQ    => equal,
  CNE    => not (equal),
  CLT    => lts,
  CLE    => lts | equal,
  CLTU   => ltu,
  CLEU   => ltu | equal,
  CEXEQ  => cb_val == ct_val,
  CNEXEQ => cb_val != ct_val
};
wGPR(rd) = zero_extend (cmp)
```

Exceptions

A reserved instruction exception is raised if

- *t* does not correspond to comparison operation whose meaning has been defined. (All possible values of *t* have now been assigned meanings, so this exception cannot occur).

Notes

- CLTU can be used by a C compiler to compile code that compares two non-NULL pointers (e.g., to detect whether a pointer to a character within a buffer has reached the end of the buffer). When two pointers to addresses within the same object (e.g., to different offsets within an array) are compared, the pointer to the earlier part of the object will be compared as less. (Signed comparison would also work as long as the object did not span address 2^{63}; the MIPS address space layout makes it unlikely that objects spanning 2^{63} will exist in user-space C code).

- Although the ANSI C standard does not specify whether a NULL pointer is less than or greater than a non-NULL pointer (clearly, they must not be equal), the comparison instructions have been designed so that when C pointers are represented by capabilities, NULL will be less than any non-NULL pointer.

- A C compiler may also use these instructions to compare two values of type uintptr_t that have been obtained by casting from an integer value. If the cast is compiled as a CFromPtr of zero followed by CSetOffset to the integer value, the result of CPtrCmp will be the same as comparing the original integer values, because CFromPtr will have set **base** to zero. Signed and unsigned capability comparison operations are provided so that both signed and unsigned integer comparisons can be performed on capability registers.

- A program could use pointer comparison to determine the value of **base**, by setting **offset** to different values and testing which values cause **base** + **offset** to wrap around and be less than **base** + a zero offset. This is not an attack against a security property of the ISA, because **base** is not a secret.

- One possible way in which garbage collection could be implemented is for the garbage collector to move an object and fix up all capabilities that refer to it. If there are appropriate restrictions on which capabilities the program has to start with, the garbage collector can be sure that the program does not have any references to the object stored as integers, and so can know that it is safe to move the object. With this type of garbage collection, comparing pointers by extracting their base and offset with `CGetBase` and `CGetOffset` and comparing the integer values is not guaranteed to work, because the garbage collector might have moved the object part-way through. `CPtrCmp` is atomic, and so will work in this scenario.

- Some compilers may make the optimization that if a check for $(a = b)$ has succeeded, then b can be replaced with a without changing the semantics of the program. This optimization is not valid for the comparison performed by `CEq`, because two capabilities can point to the same place in memory but have different bounds, permissions etc. and so not be interchangeable. The `CExEq` instruction is provided for when a test for semantic equivalence of capabilities is needed; it compares all the fields, even the ones that are reserved for future use.

- Mathematically, `CEq` divides capabilities into *equivalence classes*, and the signed or unsigned comparison operators provide a *total ordering* on these equivalence classes. `CExEq` also divides capabilities into equivalence classes, but these are not totally ordered: two capabilities can be unequal according to `CExEq`, and also neither less or greater according to *CLT* (e.g., if they have the same **base** + **offset**, but different **length**).

- There is an outstanding issue: when capability compression is in use, does `CExEq` compare the compressed representation or the uncompressed capability? There might be a difference between the two if there are multiple compressed representations that decompress to the same thing. If **tag** is false, then then capability register might contain non-capability data (e.g., an integer, or a string) and it might not decompress to anything sensible. Clearly in this case the in-memory compressed representation should be compared bit for bit. Is it also acceptable to compare the compressed representations when **tag** is true? This might lead to two capabilities that are sematically equivalent but have been computed by a different sequence of operations comparing as not equal. The consequence of this for programs that use `CExEq` is for further study.

- If a C compiler compiles pointer equality as `CExEq` (rather than `CEq`), it will catch the following example of undefined behavior. Suppose that a and b are capabilities for different objects, but a has been incremented until its **base** + **offset** points to the same memory location as b. Using `CExEq`, these pointers will not compare as equal because they have different bounds.

- `CNEq` and `CNExEq` are in principle redundant, because a compiler could replace `CNEq` and a conditional branch with `CEq` and a conditional branch with the opposite condition (and if the result of the comparison is assigned to a variable, the compiler could explictly negate the result of `CEq`, at a small performance penalty). We provide explicit tests for not equal in order to simplify the compiler back end.

CReadHwr: Read a Special-Purpose Capability Register

Format

CReadHwr cd, selector

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	selector	0xd	0x3f

Description

Load the value of special-purpose capability register *selector* into capability register *cd*. See Table 7.1 for the possible values of *selector* and the permissions required in order to read the register.

Table 7.1: Access permission required to read special-purpose capability registers

Selector	Register	Required for read access
0	Default data capability (**DDC**)	∅
1	User TLS (**CULR**)	∅
2		∅
8	Privileged TLS (**CPLR**)	**PCC**.**perms**.*Access_System_Registers*
22	Kernel scratch register 1 (**KR1C**)	Supervisor Mode
23	Kernel scratch register 2 (**KR2C**)	Supervisor Mode
28	Error exception program counter (**ErrorEPCC**)	Supervisor Mode and **PCC**.**perms**.*Access_System_Registers*
29	Kernel code capability (**KCC**)	Supervisor Mode and **PCC**.**perms**.*Access_System_Registers*
30	Kernel data capability (**KDC**)	Supervisor Mode and **PCC**.**perms**.*Access_System_Registers*
31	Exception program counter (**EPCC**)	Supervisor Mode and **PCC**.**perms**.*Access_System_Registers*

Semantics

```
checkCP2usable();
let (needSup, needAccessSys) : (bool, bool) = match unsigned(sel) {
  0  => (false, false), /* DDC   -- no access control */
  1  => (false, false), /* CULR -- no access control */
  8  => (false, true),  /* CPLR -- privileged TLS */
  22 => (true, false),  /* KR1C */
```

```
    23 => (true, false),  /* KR2C */
    28 => (true, true),   /* ErrorEPCC */
    29 => (true, true),   /* KCC */
    30 => (true, true),   /* KDC */
    31 => (true, true),   /* EPCC */
    _  => SignalException(ResI)
  };
  if needAccessSys & not(pcc_access_system_regs()) then
     raise_c2_exception(CapEx_AccessSystemRegsViolation, sel)
  else if needSup & not(grantsAccess(getAccessLevel(), Supervisor)) then
     raise_c2_exception(CapEx_AccessSystemRegsViolation, sel)
  else {
    let capVal : Capability = match unsigned(sel) {
      0  => DDC,
      1  => CULR,
      8  => CPLR,
      22 => KR1C,
      23 => KR2C,
      28 => ErrorEPCC,
      29 => KCC,
      30 => KDC,
      31 => EPCC,
      _  => {assert(false, "CReadHwr: should be unreachable code"); undefined}
    };
    writeCapReg(cd, capVal);
  };
```

Exceptions

A reserved Instruction exception is raised for unknown or unimplemented values of *selector*.

A coprocessor 2 exception is raised if:

- the permission checks as specified in Table 7.1 above were not met for *selector*

Notes

- In the future we may decide to make **PCC** accessible via this instruction. This would save opcode space since we would no longer required a dedicated CGetPCC instruction.

CReturn: Return to the Previous Security Domain

Format

CReturn

31 26	25 21	20		0
0x12	0x05	0x0	0x0	0x7ff

Description

CReturn is used to return from a call into a protected subsystem. As defined, the instruction simply triggers a specific CP2 exception via the CCall/CReturn exception vector, allowing a software exception handler to implement any required functionality.

Semantics (hardware)

```
checkCP2usable();
raise_c2_exception_noreg(CapEx_ReturnTrap)
```

Exceptions

A coprocessor 2 exception will be raised so that the desired semantics can be implemented in a trap handler. The capability exception code will be 0x06 and the handler vector will be 0x100 above the general-purpose exception handler.

Notes

- The CReturn instruction may be removed in a future version of the ISA specification (though it might continue to exist as a pseudo-instruction in the assembler), to be replaced by a specific selector in the CCall instruction.

- As with CCall, it is possible to imagine a number of points between this exception-based implementation and a hardware-assisted implementation – e.g., with varying degrees of architectural checking of return values, clearing of registers, etc. In implementing more rich hardware functionalities, software flexibility to support a range of ABIs is reduced.

Expected Software Use

CReturn is designed to complement use of the CCall instruction with selector 0 – i.e., where software implements a function-call-like domain-transition model. It is antipated that CReturn software exception handlers will perform any sanitization of the register file, capability flow control, "undo" actions taken in the CCall exception handler to restore execution to the instruction following CCall in the caller context, unsealing and/or installation of caller capabilities so that it can continue execution in the original caller protection domain.

It is anticipated that software using CCall selector 1 for domain transition may wish to use that same instruction for return, rather than CReturn.

Sketch of the CheriBSD CReturn Model

As with CheriBSD's CCall exception handler, the CheriBSD CReturn is implemented via a short privileged exception handler. A frame is popped off of the trusted stack, allowing the caller **PCC** and **IDC** to be restored, non-return capability and integer registers are cleared, and capability flow control is imposed on return capabilities to prevent non-global capabilities from being propagated across domain transition. The CheriBSD CReturn exception handler operates as follows:

1. **IDC** is popped off the trusted system stack.
2. **PCC** is popped off the trusted system stack.

The CheriBSD CReturn can be modeled with the following pseudocode:

```
IDC $\leftarrow$ mem[TSS .. TSS $+$ capability_size $-$ 1]
IDC.tag $\leftarrow$ tags[toTag(TSS)]
TSS $\leftarrow$ TSS $+$ capability_size
PCC $\leftarrow$ mem[TSS .. TSS $+$ capability_size $-$ 1]
PCC.tag $\leftarrow$ tags[toTag(TSS)]
TSS $\leftarrow$ TSS + capability_size
PC $\leftarrow$ PCC.offset
```

In addition to the coprocessor 2 exceptions listed above, a coprocessor 2 exception may be raised by the software exception handler if:

- The trusted system stack would underflow.
- The tag bits are not set on the memory location that are popped from the stack into **IDC** and **PCC**.

In addition, the CheriBSD CReturn handler checks the *global* bit on capability registers so that CReturn cannot be used to leak local capabilities. It also clears non-return-value capability and integer registers.

CS[BHWD]: Store Integer via Capability

Format

CSB rs, rt, offset(cb)
CSH rs, rt, offset(cb)
CSW rs, rt, offset(cb)
CSD rs, rt, offset(cb)
CSBR rs, rt(cb)
CSHR rs, rt(cb)
CSWR rs, rt(cb)
CSDR rs, rt(cb)
CSBI rs, offset(cb)
CSHI rs, offset(cb)
CSWI rs, offset(cb)
CSDI rs, offset(cb)

31 26	25 21	20 16	15 11	10 3		1 0
0x3A	rs	cb	rt	offset	0	t

Note: If the encoded value of *cb* is zero, this instruction will use **DDC** as the *cb* operand

Purpose

Stores some or all of a register into a memory location.

Description

Part of integer register *rs* is stored to the memory location specified by *cb*.**base** + *cb*.**offset** + *rt* + $2^t * \textit{offset}$. Capability register *cb* must contain a capability that grants permission to store data.

The *t* field determines how many bits of the register are stored to memory:

0 byte (8 bits)

1 halfword (16 bits)

2 word (32 bits)

3 doubleword (64 bits)

If less than 64 bits are stored, they are taken from the least-significant end of the register.

Semantics

```
checkCP2usable();
let cb_val = readCapRegDDC(cb);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_store) then
  raise_c2_exception(CapEx_PermitStoreViolation, cb)
else
{
  let size   = wordWidthBytes(width);
  let cursor = getCapCursor(cb_val);
  let vAddr  = (cursor + unsigned(rGPR(rt)) + size * signed(offset)) % pow2(64);
  let vAddr64= to_bits(64, vAddr);
  if (vAddr + size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if vAddr < getCapBase(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if not (isAddressAligned(vAddr64, width)) then
    SignalExceptionBadAddr(AdES, vAddr64)
  else
  {
    let pAddr  = TLBTranslate(vAddr64, StoreData);
    let rs_val = rGPR(rs);
    match width
    {
      B  => MEMw_wrapper(pAddr, 1) = rs_val[7..0],
      H  => MEMw_wrapper(pAddr, 2) = rs_val[15..0],
      W  => MEMw_wrapper(pAddr, 4) = rs_val[31..0],
      D  => MEMw_wrapper(pAddr, 8) = rs_val
    }
  }
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb* is sealed.
- *cb*.**perms**.*Permit_Store* is not set.
- $addr + size > cb$.**base** + *cb*.**length**.
- $addr < cb$.**base**

An address error during store (AdES) is raised if:

- *addr* is not aligned.

Notes

- This instruction reuses the opcode from the Store Word from Coprocessor 2 (`SWC2`) instruction in the MIPS Specification.
- *rt* is treated as an unsigned integer.
- *offset* is treated as a signed integer.
- BERI1 has a compile-time option to allow unaligned loads and stores. If BERI1 is built with this option, an unaligned store will only raise an exception if it crosses a cache line boundary.

CSC: Store Capability via Capability

Format

CSC cs, rt, offset(cb)
CSCR cs, rt(cb)
CSCI cs, offset(cb)

31 26	25 21	20 16	15 11	10 6 5 0
0x3e	cs	cb	rt	offset

Note: If the encoded value of *cb* is zero, this instruction will use **DDC** as the *cb* operand

Description

Capability register *cs* is stored at the memory location specified by *cb*.**base** + *cb*.**offset** + *rt* + 16 ∗ *offset*, and the bit in the tag memory associated with *cb*.**base** + *cb*.**offset** + *rt* + 16 ∗ *offset* is set to the value of *cs.tag*. Capability register *cb* must contain a capability that grants permission to store capabilities. The virtual address *cb*.**base** + *cb*.**offset** + *rt* + 16 ∗ *offset* must be *capability_size* aligned.

The various capability encoding schemes define bit representations in memory. While a given instantiation of CHERI will use a particular scheme, software should, in general, not attempt to parse capability bit patterns from memory. Instructions for capability interrogation (e.g., CGetAddr, CGetType) do not require that their source registers be holding *tagged* capabilities; software wishing to decode memory bit patterns should rather use CLC and interrogate the result.

Semantics

```
checkCP2usable();
let cs_val = readCapReg(cs);
let cb_val = readCapRegDDC(cb);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_store) then
  raise_c2_exception(CapEx_PermitStoreViolation, cb)
else if not (cb_val.permit_store_cap) then
  raise_c2_exception(CapEx_PermitStoreCapViolation, cb)
else if not (cb_val.permit_store_local_cap) & (cs_val.tag) & not (cs_val.global)
    then
  raise_c2_exception(CapEx_PermitStoreLocalCapViolation, cb)
else
{
  let cursor  = getCapCursor(cb_val);
  let vAddr   = (cursor + unsigned(rGPR(rt)) + 16 * signed(offset)) % pow2(64);
  let vAddr64 = to_bits(64, vAddr);
  if (vAddr + cap_size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if vAddr < getCapBase(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if (vAddr % cap_size) != 0 then
    SignalExceptionBadAddr(AdES, vAddr64)
  else
  {
    let (pAddr, noStoreCap) = TLBTranslateC(vAddr64, StoreData);
    if cs_val.tag & noStoreCap then
      raise_c2_exception(CapEx_TLBNoStoreCap, cs)
    else
      MEMw_tagged(pAddr, cap_size, cs_val.tag, capToMemBits(cs_val));
  }
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb* is sealed.
- *cb*.**perms**.*Permit_Store* is not set.
- *cb*.**perms**.*Permit_Store_Capability* is not set.
- *cb*.**perms**.*Permit_Store_Local* is not set and *cs.tag* is set and *cs*.**perms**.*Global* is not set.
- $\mathit{addr} + \mathit{capability\_size} > \mathit{cb}.\mathbf{base} + \mathit{cb}.\mathbf{length}$.
- $\mathit{addr} < \mathit{cb}.\mathbf{base}$.

A TLB Store exception is raised if:

- *cs*.**tag** is set and the *S* bit in the TLB entry for the page containing *addr* is not set.

An address error during store (AdES) exception is raised if:

- The virtual address *addr* is not *capability_size* aligned.

Notes

- If the address alignment check fails and one of the security checks fails, a coprocessor 2 exception (and not an address error exception) is raised. The priority of the exceptions is security-critical, because otherwise a malicious program could use the type of the exception that is raised to test the bottom bits of a register that it is not permitted to access.

- It is permitted to store a local capability with the tag bit unset even if the permit store local bit is not set in *cb*. This is because if the tag bit is not set then the permissions have no meaning.

- *offset* is interpreted as a signed integer.

- This instruction reuses the opcode from the Store Doubleword from Coprocessor 2 (`SDC2`) instruction in the MIPS Specification.

- The `CSCI` mnemonic is equivalent to `CSC` with *cb* being the zero register ($zero). The `CSCR` mnemonic is equivalent to `CSC` with *offset* set to zero.

- BERI1 has a compile-time option to allow unaligned loads and stores. `CSC` to an unaligned address will raise an exception even if BERI1 has been built with this option, because it would be a security vulnerability if an attacker could construct a corrupted capability with **tag** set by writing it to an unaligned address.

- Although the *capability_size* can vary, the offset is always in multiples of 16 bytes (128 bits).

CSC[BHWD]: Store Conditional Integer via Capability

Format

CSCB rd, rs, cb
CSCH rd, rs, cb
CSCW rd, rs, cb
CSCD rd, rs, cb

31 26	25 21	20 16	15 11	10 6	5 4	3 2	1 0
0x12	0x10	rs	cb	rd		00	t

Note: If the encoded value of *cb* is zero, this instruction will use **DDC** as the *cb* operand

Semantics

```
checkCP2usable();
let cb_val = readCapRegDDC(cb);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_store) then
  raise_c2_exception(CapEx_PermitStoreViolation, cb)
else
{
  let size    = wordWidthBytes(width);
  let vAddr   = getCapCursor(cb_val);
  let vAddr64 = to_bits(64, vAddr);
  if (vAddr + size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if vAddr < getCapBase(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if not (isAddressAligned(vAddr64, width)) then
    SignalExceptionBadAddr(AdES, vAddr64)
  else
  {
    let pAddr  = TLBTranslate(vAddr64, StoreData);
    let rs_val = rGPR(rs);
    let success : bool = if (CP0LLBit[0]) then
        match width
        {
          B  => MEMw_conditional_wrapper(pAddr, 1, rs_val[7..0]),
          H  => MEMw_conditional_wrapper(pAddr, 2, rs_val[15..0]),
          W  => MEMw_conditional_wrapper(pAddr, 4, rs_val[31..0]),
          D  => MEMw_conditional_wrapper(pAddr, 8, rs_val)
        }
      else
        false;
    wGPR(rd) = zero_extend(success);
  }
}
```

CSCC: Store Conditional Capability via Capability

Format

CSCC rd, cs, cb

31 26	25 21	20 16	15 11	10 6	5 3	2 0
0x12	0x10	cs	cb	rd	▨	0111

Note: If the encoded value of *cb* is zero, this instruction will use **DDC** as the *cb* operand

Semantics

```
checkCP2usable();
let cs_val = readCapReg(cs);
let cb_val = readCapRegDDC(cb);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_store) then
  raise_c2_exception(CapEx_PermitStoreViolation, cb)
else if not (cb_val.permit_store_cap) then
  raise_c2_exception(CapEx_PermitStoreCapViolation, cb)
else if not (cb_val.permit_store_local_cap) & (cs_val.tag) & not (cs_val.global)
    then
  raise_c2_exception(CapEx_PermitStoreLocalCapViolation, cb)
else
{
  let vAddr   = getCapCursor(cb_val);
  let vAddr64 = to_bits(64, vAddr);
  if (vAddr + cap_size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if vAddr < getCapBase(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if (vAddr % cap_size) != 0 then
    SignalExceptionBadAddr(AdES, vAddr64)
  else
  {
    let (pAddr, noStoreCap) = TLBTranslateC(vAddr64, StoreData);
    if cs_val.tag & noStoreCap then
      raise_c2_exception(CapEx_TLBNoStoreCap, cs)
    else
    {
      let success = if (CP0LLBit[0]) then
          MEMw_tagged_conditional(pAddr, cap_size, cs_val.tag, capToMemBits(
              cs_val))
        else
```

```
            false;
        wGPR(rd) = zero_extend(success);
      }
    }
  }
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb* is sealed.
- *cb*.**perms**.*Permit_Store* is not set.
- *cb*.**perms**.*Permit_Store_Capability* is not set.
- *cb*.**perms**.*Permit_Store_Local_Capability* is not set and *cs*.**perms**.*Global* is not set.
- $addr + capability\_size > cb.\mathbf{base} + cb.\mathbf{length}$
- $addr < cb.\mathbf{base}$

A TLB Store exception is raised if:

- The *S* bit in the TLB entry corresponding to virtual address *addr* is not set.

An address error during store (AdES) exception is raised if:

- *addr* is not correctly aligned.

CSeal: Seal a Capability

Format

CSeal cd, cs, ct

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cs	ct	0xb

Description

Capability register *cs* is sealed with an **otype** of *ct*.**base** + *ct*.**offset** and the result is placed in *cd*:

- *cd*.**otype** is set to *ct*.**base** + *ct*.**offset**;
- *cd* is sealed;
- and the other fields of *cd* are copied from *cs*.

ct must grant *Permit_Seal* permission, and the new **otype** of *cd* must be between *ct*.**base** and *ct*.**base** + *ct*.**length** − 1.

Semantics

```
checkCP2usable();
let cs_val = readCapReg(cs);
let ct_val = readCapReg(ct);
let ct_cursor = getCapCursor(ct_val);
let ct_top    = getCapTop(ct_val);
let ct_base   = getCapBase(ct_val);
if not (cs_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cs)
else if not (ct_val.tag) then
  raise_c2_exception(CapEx_TagViolation, ct)
else if cs_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cs)
else if ct_val.sealed then
  raise_c2_exception(CapEx_SealViolation, ct)
else if not (ct_val.permit_seal) then
  raise_c2_exception(CapEx_PermitSealViolation, ct)
else if ct_cursor < ct_base then
  raise_c2_exception(CapEx_LengthViolation, ct)
else if ct_cursor >= ct_top then
  raise_c2_exception(CapEx_LengthViolation, ct)
else if ct_cursor > max_otype then
  raise_c2_exception(CapEx_LengthViolation, ct)
else
```

```
{
  let (success, newCap) = sealCap(cs_val, to_bits(24, ct_cursor));
  if not (success) then
      raise_c2_exception(CapEx_InexactBounds, cs)
  else
      writeCapReg(cd, newCap)
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cs*.**tag** is not set.
- *ct*.**tag** is not set.
- *cs* is sealed.
- *ct* is sealed.
- *ct*.**perms**.*Permit_Seal* is not set.
- *ct*.**offset** $\geq$ *ct*.length
- *ct*.**base** $+$ *ct*.**offset** $>$ *max_otype*
- The bounds of *cb* cannot be represented exactly in a sealed capability.

Notes

- If capability compression is in use, the range of possible (**base**, **length**, **offset**) values might be smaller for sealed capabilities than for unsealed capabilities. This means that `CSeal` can fail with an exception in the case where the bounds are no longer precisely representable.

CSetAddr: Set the Address of Capability

Format

CSetAddr cd, cs, rs

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cs	rs	0x22

Description

cd is set to *cb* with *cb*.**a** set to *rs*. If changing the address causes the capability to become unrepresentable, then an untagged capability with the requested address is returned.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let rt_val = rGPR(rt);
if cb_val.tag & cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else
{
  let (representable, newCap) = setCapAddr(cb_val, rt_val);
  if representable then
      writeCapReg(cd, newCap)
  else
      writeCapReg(cd, int_to_cap(rt_val));
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is set and *cb* is sealed.

Notes

- This instruction may be useful, in combination with `CGetAddr`, when C is manipulating pointers in ways that require a round trip through integer registers.
- This instruction is also useful for `uintptr_t` arithmetic when using an address interpretation of capabilities. When interpreting `uintptr_t` as offsets relative to the base, the compiler will use `CGetOffset` and `CSetOffset` instead.

CSetBounds: Set Bounds

Format

CSetBounds cd, cb, rt

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cb	rt	0x8

Description

Capability register *cd* is replaced with a capability that:

- Grants access to a subset of the addresses authorized by *cb*. That is, $cd.\textbf{base} \geq cb.\textbf{base}$ and $cd.\textbf{base} + cd.\textbf{length} \leq cb.\textbf{base} + cb.\textbf{length}$.
- Grants access to at least the addresses $cb.\textbf{base} + cb.\textbf{offset} \ldots cb.\textbf{base} + cb.\textbf{offset} + rt - 1$. That is, $cd.\textbf{base} \leq cb.\textbf{base} + cb.\textbf{offset}$ and $cd.\textbf{base} + cd.\textbf{length} \geq cb.\textbf{base} + cb.\textbf{offset} + rt$.
- Has an **offset** that points to the same memory location as *cb*'s **offset**. That is, *cd*.**offset** = *cb*.**offset** + *cb*.**base** - *cd*.**base**.
- Has the same **perms** as *cb*, that is, *cd*.**perms** = *cb*.**perms**.

When the hardware uses a 256-bit representation for capabilities, the bounds of the destination capability *cd* are exactly as requested. When the hardware uses a smaller (compressed) representation of capabilities in which not all combinations of **base** and **length** are representable, then *cd* may grant access to a range of memory addresses that is wider than requested, but is still guaranteed to be within the bounds of *cb*.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let rt_val = unsigned(rGPR(rt));
let cursor = getCapCursor(cb_val);
let base   = getCapBase(cb_val);
let top    = getCapTop(cb_val);
let newTop = cursor + rt_val;
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if cursor < base then
  raise_c2_exception(CapEx_LengthViolation, cb)
else if newTop > top then
  raise_c2_exception(CapEx_LengthViolation, cb)
```

```
else
{
  let (_, newCap) = setCapBounds(cb_val, to_bits(64, cursor), to_bits(65, newTop)
      );
  writeCapReg(cd, newCap) /* ignore exact */
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb* is sealed.
- $cursor < cb$.**base**
- $cursor + rt > cb$.**base** + cb.**length**

Notes

- In the above Sail code, arithmetic is over the mathematical integers and *rt* is unsigned, so a large value of *rt* cannot cause $cursor + rt$ to wrap around and be less than *cb*.**base**. Implementations (that, for example, will probably use a fixed number of bits to store values) must handle this overflow case correctly.

CSetBoundsExact: Set Bounds Exactly

Format

CSetBoundsExact cd, cb, rt

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cb	rt	0x9

Description

Capability register *cd* is replaced with a capability with **base** *cb*.**base**+ *cb*.**offset**, **length** *rt*, and **offset** zero. When capability compression is in use, an exception is thrown if the requested bounds cannot be represented exactly.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let rt_val = unsigned(rGPR(rt));
let cursor = getCapCursor(cb_val);
let base   = getCapBase(cb_val);
let top    = getCapTop(cb_val);
let newTop = cursor + rt_val;
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if cursor < base then
  raise_c2_exception(CapEx_LengthViolation, cb)
else if newTop > top then
  raise_c2_exception(CapEx_LengthViolation, cb)
else
{
  let (exact, newCap) = setCapBounds(cb_val, to_bits(64, cursor), to_bits(65,
      newTop));
  if not (exact) then
      raise_c2_exception(CapEx_InexactBounds, cb)
  else
      writeCapReg(cd, newCap)
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb* is sealed.

- $\mathit{cursor} < \mathit{cb}.$**base**
- $\mathit{cursor} + \mathit{rt} > \mathit{cb}.$**base** $+$ cb.**length**
- The requested bounds cannot be represented exactly.

Notes

- In the above Sail code, arithmetic is over the mathematical integers and *rt* is unsigned, so a large value of *rt* cannot cause $\mathit{cursor} + \mathit{rt}$ to wrap around and be less than *cb*.**base**. Implementations (that, for example, will probably use a fixed number of bits to store values) must handle this overflow case correctly.

CSetBoundsImm: Set Bounds (Immediate)

Format

CSetBounds cd, cb, length$_{imm}$

31 26	25 21	20 16	15 11	10 0
0x12	0x14	cd	cb	length$_{imm}$

Description

Capability register *cd* is replaced with a capability that:

- Grants access to a subset of the addresses authorized by *cb*. That is, $cd.\mathbf{base} \geq cb.\mathbf{base}$ and $cd.\mathbf{base} + cd.\mathbf{length} \leq cb.\mathbf{base} + cb.\mathbf{length}$.

- Grants access to at least the addresses $cb.\mathbf{base} + cb.\mathbf{offset} \ldots cb.\mathbf{base} + cb.\mathbf{offset} + length_{imm} - 1$. That is, $cd.\mathbf{base} \leq cb.\mathbf{base} + cb.\mathbf{offset}$ and $cd.\mathbf{base} + cd.\mathbf{length} \geq cb.\mathbf{base} + cb.\mathbf{offset} + length_{imm}$.

- Has an **offset** that points to the same memory location as *cb*'s **offset**. That is, *cd*.**offset** = *cb*.**offset** + *cb*.**base** - *cd*.**base**.

- Has the same **perms** as *cb*, that is, *cd*.**perms** = *cb*.**perms**.

When the hardware uses a 256-bit representation for capabilities, the bounds of the destination capability *cd* are exactly as requested. When the hardware uses a smaller (compressed) representation of capabilities in which not all combinations of **base** and **length** are representable, then *cd* may grant access to a range of memory addresses that is wider than requested, but is still guaranteed to be within the bounds of *cb*.

Semantics

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.

- *cb* is sealed.

- $cursor < cb.\mathbf{base}$

- $cursor + length_{imm} > cb.\mathbf{base} + \mathrm{cb}.\mathbf{length}$

Notes

- In the above Sail code, arithmetic is over the mathematical integers and $length_{imm}$ is unsigned, so a large value of $length_{imm}$ cannot cause $cursor + length_{imm}$ to wrap around and be less than *cb*.**base**. Implementations (that, for example, will probably use a fixed number of bits to store values) must handle this overflow case correctly.

- If this instruction is used with **C0** as the destination register, it can be used to assert that a given capability grants access to at least $length_{imm}$ bytes. An assembler pseudo instruction `CAssertInBounds` is supported for this use case.

CSetCause: Set the Capability Exception Cause Register

Format

CSetCause rt

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	0x2	0x1f	0x3f

Description

The capability cause register value is set to the low 16 bits of integer register *rt*.

Semantics

```
checkCP2usable();
if not (pcc_access_system_regs ()) then
  raise_c2_exception_noreg(CapEx_AccessSystemRegsViolation)
else
{
  let rt_val = rGPR(rt);
  CapCause->ExcCode() = rt_val[15..8];
  CapCause->RegNum()  = rt_val[7..0];
}
```

Exceptions

A coprocessor 2 exception is raised if:

- **PCC**.**perms**.*Access_System_Registers* is not set.

Notes

- CSetCause does not cause an exception to be raised (unless the permission check for *Access_System_Registers* fails). CSetCause will typically be used in an exception handler, where the exception handler wants to change the cause code set by the hardware before doing further exception handling. (e.g., when the original cause code was CCall, the CCall handler detects that CCall should fail, and it sets *CapCause* to the reason it failed). In cases like this, it is important that **EPC** (etc.) are not overwritten by CSetCause.

CSetCID: Set the Architectural Compartment ID

Format

CSetCID cb

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cb	0x5	0x1f	0x3f

Description

Set the architectural Compartment ID (CID) to *cb*.**base** + *cb*.**offset** if *cb* has the Permit_Set_CID permission and *cb*.**offset** is in range. The CID can then be used by the microarchitecture to tag microarchitectural state. CIDs can be utilized in a similar style as ASID matching in TLBs to determine in what context microarchitectural state can be used. Typical use will be to prevent sharing where it could otherwise be used as a high-bandwidth microarchitectural side channel between compartments with confidentiality requirements – for example, to limit the impact of Spectre-style attacks [61].

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_set_CID) then
  raise_c2_exception(CapEx_PermitSetCIDViolation, cb)
else
{
  let addr = getCapCursor(cb_val);
  if addr < getCapBase(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if addr >= getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else
    CID = to_bits(64, addr);
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**perms**.*Permit_Set_CID* is not set.
- $\mathit{addr} + 1 > \mathit{cb}.\mathbf{base} + \mathit{cb}.\mathbf{length}$.
- $\mathit{addr} < \mathit{cb}.\mathbf{base}$.

Notes

- The CID can be queried using the CGetCID instruction.
- Although CSetCID has no architectural side effects other than setting an integer register with a compartment ID, the intent is that the microarchitecture can be made aware of boundaries across which microarchitectural side channels are less acceptable. A key design goal for CSetCID is to provide flexible mechanism above which a range of software policies might be implemented.
- For example, the software supervisor might arrange that all compartments have unique CIDs such that branch-predictor state cannot be shared. Other policies might use the same CID for compartments between which strong confidentiality requirements are not present – e.g., where only integrity or availability protection is required.
- We have chosen not to protect the architectural CID using Access_System_Registers in order to support virtualizability of the domain switcher – and, in particular, to not require Access_System_Registers to implement a domain switcher. A new permission is used, together with bounds checks, such that ranges of CIDs can be delegated when multiple domain switchers are in use. For example, a set of CIDs might be reserved for domain-switch implementations themselves, and then subranges delegated to individual language runtimes or processes within the same address space. Note that such a model could obligate two CID operations per domain switch involving mutual distrust: one into the domain switcher, and a second out, in order to not just protect the two endpoint domains from one another but also the switcher.
- How to ensure that CSetCID is not speculated past (e.g., in the case of microarchitectural side-channel attacks such as Spectre) is a critical question. We recommend that CSetCID be considered serialising, and that the CID be set immediately on switcher entry, as well as again on switcher exit.
- An alternative design choice would accept an integer general-purpose register operand, *rt*, as a second argument specifying the CID to switch to. This might be more consistent with the behavior of CGetCID, but also consume more opcode space.

CSetOffset: Set Cursor to an Offset from Base

Format

CSetOffset cd, cs, rt

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cs	rt	0xf

Description

Capability register *cd* is replaced with the contents of capability register *cs* with the **offset** field set to the contents of integer register *rt*.

If capability compression is in use, and the requested **base**, **length** and **offset** cannot be represented exactly, then *cd*.**tag** is cleared, *cd*.**base** and *cd*.**length** are set to zero, *cd*.**perms** is cleared and *cd*.**offset** is set equal to *cs*.**base**+ *rt*.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let rt_val = rGPR(rt);
if cb_val.tag & cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else
{
  let (success, newCap) = setCapOffset(cb_val, rt_val);
  if success then
      writeCapReg(cd, newCap)
  else
      writeCapReg(cd, int_to_cap(to_bits(64, getCapBase(cb_val)) + rt_val))
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cs*.**tag** is set and *cs* is sealed.

Notes

- `CSetOffset` can be used on a capability register whose tag bit is not set. This can be used to store an integer value in a capability register, and is useful when implementing a variable that is a union of a capability and an integer (`intcap_t` in C). The in-memory representation that will be used if the capability register is stored to memory might be surprising to some users (with the 256-bit representation of capabilities, **base** + **offset** is stored in the **cursor** field in memory) and may change if the memory representation of capabilities changes, so compilers should not rely on it.

- When capability compression is in use, and the requested offset is not representable, the result preserves the requested **base** + **offset** (i.e., the cursor) rather than the architectural field **offset**. This field is mainly useful for debugging what went wrong (the capability cannot be dereferenced, as **tag** has been cleared), and for debugging we considered it more useful to know what the requested capability would have referred to rather than its **offset** relative to a **base** that is no longer available. This has the disadvantage that it exposes the value of **base** to a program, but **base** is not a secret and can be accessed by other means. The main reason for not exposing **base** to programs is so that a garbage collector can stop the program, move memory, modify the capabilities and restart the program. A capability with **tag** cleared cannot be dereferenced, and so is not of interest to a garbage collector, and so it doesn't matter if it exposes **base**.

CSub: Subtract Capabilities

Format

CSub rd, cb, ct

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rt	cb	ct	0xa

Description

Register *rd* is set equal to $(cb.\textbf{base} + cb.\textbf{offset} - ct.\textbf{base} - ct.\textbf{offset}) \bmod 2^{64}$.

Semantics

```
checkCP2usable();
let ct_val = readCapReg(ct);
let cb_val = readCapReg(cb);
wGPR(rd) = to_bits(64, getCapCursor(cb_val) - getCapCursor(ct_val))
```

Notes

- `CSub` can be used to implement C-language pointer subtraction, or subtraction of `intcap_t` .
- Like `CIncOffset`, `CSub` can be used on either valid capabilities (**tag** set) or on integer values stored in capability registers (**tag** not set).
- If a copying garbage collector is in use, pointer subtraction must be implemented with an atomic operation (such as `CSub`). Implementing pointer subtraction with a non-atomic sequence of operations such as `CGetOffset` has the risk that the garbage collector will relocate an object part way through, giving incorrect results for the pointer difference. If *cb* and *ct* are both pointers into the same object, then a copying garbage collector will either relocate both of them or neither of them, leaving the difference the same. If *cb* and *ct* are pointers into different objects, the result of the subtraction is not defined by the ANSI C standard, so it doesn't matter if this difference changes as the garbage collector moves objects.

CToPtr: Capability to Integer Pointer

Format

CToPtr rd, cb, ct

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	ct	0x12

Note: If the encoded value of *ct* is zero, this instruction will use **DDC** as the *ct* operand

Description

If *cb* has its tag bit unset (i.e. it is either the NULL capability, or contains some other non-capability data), then *rd* is set to zero. Otherwise, *rd* is set to *cb*.**base** + *cb*.**offset** - *ct*.**base**

This instruction can be used to convert a capability into a pointer that uses the C language convention that a zero value represents the NULL pointer. Note that *rd* will also be zero if *cb*.**base** + *cb*.**offset** = *ct*.**base**; this is similar to the C language not being able to distinguish a NULL pointer from a pointer to a structure at address 0.

Semantics

```
checkCP2usable();
let ct_val = readCapRegDDC(ct);
let cb_val = readCapReg(cb);
if not (ct_val.tag) then
  raise_c2_exception(CapEx_TagViolation, ct)
else if cb_val.tag & cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else
{
  let ctBase = getCapBase(ct_val);
  /* Note: returning zero for untagged values breaks magic constants such as
      SIG_IGN */
  wGPR(rd) = if not (cb_val.tag) then
      zeros()
    else
      to_bits(64, getCapCursor(cb_val) - ctBase)
}
```

Exceptions

A coprocessor 2 exception will be raised if:

- *ct*.**tag** is not set.

Notes

- *ct* being sealed will not cause an exception to be raised. This is for further study.
- This instruction has two different means of returning an error code: raising an exception (if *ct*.**tag** is not set, or the registers are not accessible) and returning a NULL pointer if *cb*.**tag** is not set.
- If the range of *cb* is outside the range of *ct*, a pointer relative to *ct* can't always be used in place of *cb*: some reads or writes will fail because they are outside the range of *ct*. To handle this case, the application can use the `CTestSubset` instruction followed by a conditional move.
- `CGetAddr` similarly allows access to the sum of the base and offset of the operand capability, but without the translation relative to the authorizing capability or validity/sealed checks on the operand.

CUnseal: Unseal a Sealed Capability

Format

CUnseal cd, cs, ct

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cs	ct	0xc

Description

The sealed capability in *cs* is unsealed with *ct* and the result placed in *cd*. The global bit of *cd* is the AND of the global bits of *cs* and *ct*. *ct* must be unsealed, have *Permit_Unseal* permission, and *ct*.**base** + *ct*.**offset** must equal *cs*.**otype**.

Semantics

```
checkCP2usable();
let cs_val = readCapReg(cs);
let ct_val = readCapReg(ct);
let ct_cursor = getCapCursor(ct_val);
if not (cs_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cs)
else if not (ct_val.tag) then
  raise_c2_exception(CapEx_TagViolation, ct)
else if not (cs_val.sealed) then
  raise_c2_exception(CapEx_SealViolation, cs)
else if ct_val.sealed then
  raise_c2_exception(CapEx_SealViolation, ct)
else if ct_cursor != unsigned(cs_val.otype) then
  raise_c2_exception(CapEx_TypeViolation, ct)
else if not (ct_val.permit_unseal) then
  raise_c2_exception(CapEx_PermitUnsealViolation, ct)
else if ct_cursor <  getCapBase(ct_val) then
  raise_c2_exception(CapEx_LengthViolation, ct)
else if ct_cursor >= getCapTop(ct_val) then
  raise_c2_exception(CapEx_LengthViolation, ct)
else
  writeCapReg(cd, {unsealCap(cs_val) with
    global=(cs_val.global & ct_val.global)
  })
```

Exceptions

A coprocessor 2 exception is raised if:

- *cs*.**tag** is not set.
- *ct*.**tag** is not set.

- *cs* is not sealed.
- *ct* is sealed.
- *ct*.**offset** $\geq$ *ct*.**length**
- *ct*.**perms**.*Permit_Unseal* is not set.
- *ct*.**base** $+$ *ct*.**offset** $\neq$ *cs*.**otype**.

Notes

- There is no need to check if *ct*.**base** $+$ *ct*.**offset** $>$ *max_otype*, because this can't happen: *ct*.**base** $+$ *ct*.**offset** must equal *cs*.**otype** for the **otype** check to have suceeded, and there is no way *cs*.**otype** could have been set to a value that is out of range.

CWriteHwr: Write a Special-Purpose Capability Register

Format

CWriteHwr cb, selector

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cb	selector	0xe	0x3f

Description

The value of the capability register *cb* is stored in the special-purpose capability register *selector*. See Table 7.2 for the possible values of *selector* and the permissions required in order to write to the register.

Table 7.2: Access permission required to write special-purpose capability registers

Selector	**Register**	Required for write access
0	Default data capability (**DDC**)	∅
1	User TLS (**CULR**)	∅
2		∅
8	Privileged User TLS (**CPLR**)	**PCC**.**perms**.*Access_System_Registers*
22	Kernel scratch register 1 (**KR1C**)	Supervisor Mode
23	Kernel scratch register 2 (**KR2C**)	Supervisor Mode
28	Error exception program counter (**ErrorEPCC**)	Supervisor Mode and **PCC**.**perms**.*Access_System_Registers*
29	Kernel code capability (**KCC**)	Supervisor Mode and **PCC**.**perms**.*Access_System_Registers*
30	Kernel data capability (**KDC**)	Supervisor Mode and **PCC**.**perms**.*Access_System_Registers*
31	Exception program counter (**EPCC**)	Supervisor Mode and **PCC**.**perms**.*Access_System_Registers*

Semantics

```
checkCP2usable();
let (needSup, needAccessSys) : (bool, bool) = match unsigned(sel) {
  0  => (false, false), /* DDC   -- no access control */
  1  => (false, false), /* CULR -- no access control */
  8  => (false, true),  /* CPLR -- privileged TLS */
  22 => (true, false),  /* KR1C */
```

```
    23 => (true, false),  /* KR2C */
    28 => (true, true),   /* ErrorEPCC */
    29 => (true, true),   /* KCC */
    30 => (true, true),   /* KDC */
    31 => (true, true),   /* EPCC */
    _  => SignalException(ResI)
  };
  if needAccessSys & not(pcc_access_system_regs()) then
     raise_c2_exception(CapEx_AccessSystemRegsViolation, sel)
  else if needSup & not(grantsAccess(getAccessLevel(), Supervisor)) then
     raise_c2_exception(CapEx_AccessSystemRegsViolation, sel)
  else {
    let capVal = readCapReg(cb);
    match unsigned(sel) {
      0  => DDC  = capVal,
      1  => CULR = capVal,
      8  => CPLR = capVal,
      22 => KR1C = capVal,
      23 => KR2C = capVal,
      28 => ErrorEPCC = capVal,
      29 => KCC  = capVal,
      30 => KDC  = capVal,
      31 => EPCC = capVal,
      _  => assert(false, "CWriteHwr: should be unreachable code")
    };
  };
```

Exceptions

A reserved Instruction exception is raised for unknown or unimplemented values of *selector*.

A coprocessor 2 exception is raised if:

- the permission checks as specified in Table 7.2 above were not met for *selector*

Notes

- In the future we may decide to require **PCC**.**perms**.*Access_System_Registers* in order to modify **DDC**
- We may decide to introduce a `CSwapHwr` instruction that swaps special-purpose register *selector* and a general-purpose register

7.5 Assembler Pseudo-Instructions

For convenience, several pseudo-instructions are accepted by the assembler. These expand to either single instructions or short sequences of instructions.

7.5.1 CGetDefault, CSetDefault

Get/Set Default Capability

`CGetDefault` and `CSetDefault` get and set the capability register that is implicitly employed by the legacy MIPS load and store instructions. In the current version of the ISA, this register is special-purpose capability register 0.

In previous versions of the ISA, **DDC** was register **C0** in the main capability register file. In these versions of the architecture, using `CSetDefault` rather than an capability operation with destination **C0** allowd the Clang/LLVM compiler to know that the semantics of subsequent MIPS loads and stores will be affected by the change to **DDC**.

```
# The following are equivalent:
  CGetDDC $c1
  CGetDefault $c1
  CReadHWR $c1, $0
```

```
# The following are equivalent:
  CSetDDC $c1
  CSetDefault $c1
  CWriteHWR $c1, $0
```

7.5.2 CGetEPCC, CSetEPCC

Get/Set Exception Program Counter Capability

Pseudo-operations are provided for getting and setting **EPCC**. In the current ISA, EPCC is special-purpose capability register 31.

```
# The following are equivalent:
  CGetEPCC $c1
  CReadHWR $c1, $31
```

```
# The following are equivalent:
  CSetEPCC $c1
  CWriteHWR $c1, $31
```

7.5.3 CGetKDC, CSetKDC

Get/Set Kernel Data Capability

```
# The following are equivalent:
  CGetKDC $c1
  CReadHWR $30
```

```
# The following are equivalent:
  CSetKDC $c1
  CWriteHWR $30
```

7.5.4 GGetKCC, CSetKCC

Get/Set Kernel Code Capability

```
# The following are equivalent:
  CGetKCC $c1
  CReadHWR $29
```

```
# The following are equivalent:
  CSetKCC $c1
  CWriteHWR $29
```

7.5.5 CAssertInBounds

Assert that a capability in bounds

This pseudo operation can be used to assert that a capability grants access to a given number of bytes (if the size argument is omitted, one byte is assumed). This instruction only checks the bounds of the capability and ignores permissions. Therefore, an access might still be prohibited even if `CAssertInBounds` did not raise an exception.

```
# The following are equivalent (check that at least one byte is accessible):
  CAssertInBounds $c1
  CSetBoundsImm $cnull, $c1, 1
```

```
# The following are equivalent (check that at least 10 bytes are accessible):
  CAssertInBounds $c1, 10
  CSetBoundsImm $cnull, $c1, 10
```

7.5.6 Capability Loads and Stores of Floating-Point Values

The current revision of the CHERI ISA does not have CHERI-MIPS instructions for loading floating point values directly via capabilities. MIPS does provide instructions for moving values between integer and floating point registers, so a load or store of a floating point value via a capability can be implemented in two instructions.

Four pseudo-instructions are defined to implement these patterns. These are `clwc1` and `cldc1` for loading 32-bit and 64-bit floating point values, and `cswc1` and `csdc1` as the equivalent store operations. The load operations expand as follows:

```
  cldc1    $f7, $zero, 0($c2)
  # Expands to:
```

```
cld    $1, $zero, 0($c2)
dmtc1    $1, $f7
```

Note that integer register `$1` (`$at`) is used; this pseudo-op is unavailable if the `noat` directive is used. The 32-bit variant (`clwc1`) has a similar expansion, using `clwu` and `mtc1`.

The store operations are similar:

```
csdc1    $f7, $zero, 0($c2)
# Expands to:
dmfc1    $1, $f7
csd    $1, $zero, 0($c2)
```

The specified floating point value is moved from the floating point register to `$at` and then stored using the correct-sized capability instruction.

Chapter 8

Detailed Design Rationale

During the design of CHERI, we considered many different capability architectures and design approaches. This chapter describes the various design choices; it briefly outlines some possible alternatives, and provides rationales for the selected choices.

8.1 High-Level Design Approach: Capabilities as Pointers

Our goals of providing fine-grained memory protection and compartmentalization led to an early design choice to allow capabilities to be used as C- and C++-language pointers. This rapidly led to a number of conclusions:

- Capabilities exist within virtual address spaces, imposing an ordering in which capability protections are evaluated before virtual-memory protections; this in turn had implications for the hardware composition of the capability coprocessor and conventional interactions with the MMU.

- Capability pointers can be treated by the compiler in much the same way as integer pointers, meaning that they will be loaded, manipulated, dereferenced, and stored via registers and to/from general-purpose memory only by explicit instructions. These instructions were modeled on similar conventional RISC instructions.

- Incremental deployment within programs meant that not all pointers would immediately be converted from integers to capabilities, implying that both forms might coexist in the same virtual memory; also, there was a strong desire to embed capabilities within data structures, rather than store them in separate segments, which in turn required fine-granularity tagging.

- Incremental deployment and compatibility with the UNIX model implied the need to retain the general-purpose memory management unit (MMU) more or less as it then existed, including support for variable page sizes, TLB layout, and so on. The MIPS ISA describes a software-managed TLB rather than hardware page-table walking – as is present in most other ISAs. However, this is not fundamental to our approach, and either model would work.

8.2 Tagged Memory for Non-Probablistic Protection

Introducing tagged memory has the potential to impose a substantial adoption cost for CHERI, due to greater microarchitectural disruption. We have demonstrated that there are efficient implementations of memory tagging, even without integrated tag support within DRAM [54, 55], but even so there is a significant concern as to whether potential adopters will perceive the hurdle of adopting tagged memory as outweighing the benefits that tagged memory brings. In this section, we consider the benefits of tagging, as well as how cryptographic non-tagged approaches might be used. Tagging offers a number of significant potential benefits:

- Tags are a deterministic (non-probabilistic) means of protecting the integrity and provenance validity of pointers in memory. Probabilistic schemes, such as cryptographic hashes, are exposed both to direct brute forcing (especially due to limited bit investment within pointers) and also reinjection if leaked to attackers.

- Tags offer strong atomicity properties that are also well-aligned with current microarchitecture (e.g., in caches), avoiding the need for substantial disruption close to the processor.

- Tags have highly efficient microarchitectural implementations, including being directly embedded in tagged DRAM (an option likely to become increasingly available due to the widespread adoption of error-correcting codes, and also via tag controllers and tag caches that are affine to the DRAM controller. These may be substantially more performance- and energy-efficient than cryptographic techniques that would require hashes to be calculated or checked.

- Tags offer strong C-language compatibility, which has been demonstrated with significant software corpuses including operating-system kernels (FreeBSD), the complete UNIX userspace (FreeBSD), and significant C and C++-language applications (the Postgres database, OpenSSH client and server, and WebKit web-rendering framework).

  Key areas of incompatibility include the need to explicitly preserve tags during memory copies via capability-sized, capability-aligned loads and stores, and stronger alignment requirements for pointers. The operating system must also support maintaining tags in virtual memory, including across operations such as swapping, memory compression, and virtual-machine migration. In general, we have found that the modifications are modestly sized, although some impacts (such as the cost of tag preservation and restoration) are not yet fully quantified – e.g., for memory compression.

- Tags allow pointers to be deterministically identified in memory, a foundation for strong temporal memory-safety techniques such as revocation and garbage collection.

- The choice between tag-preserving and tag-stripping memory copying allows software to impose policies on when it is appropriate and safe for pointers to move between protection domains. For example, a kernel can selectively preserve tags in system-call arguments, preventing data copied into the kernel from an untrustworthy process from being interpreted as a pointer within the kernel, or when received by another process.

As an alternative to tagging, one could imagine making use of probabilistic cryptographic hashing techniques that protect capabilities from corruption, not unlike Cryptographic Control-Flow Integrity (CCFI) [72] or Arm's ARM v8.3 Pointer Authentication Codes (PAC). Some number of bits would be co-opted from either the virtual address (as is the case in CCFI or PAC), or from the metadata portion of a CHERI capability to hold a keyed hash, protecting the contents from corruption in memory or due to mis-manipulation in a register, rather than a tag. With additional capability metadata bits available, consumption of virtual-address bits could be reduced.

Wherever the CHERI architecture requires a tag check, a cryptographic hash check could instead be required architecturally. Wherever the CHERI architecture maintains a tag during pointer manipulation, the cryptographic hash could be updated. While architectural behavior might appear to require frequent checks of, and updates to, the hash (e.g., during loop iteration as a register is successively incremented and then used for loads or stores), it is conceivable that microarchitectural techniques (such as speculation) might both reduce the delay associated with those updates, and perhaps also elide them entirely, updating the hash only during write back. Tags appear to offer the following essential advantages over cryptographic approaches:

- Tags offer deterministic rather than probabilistic protection, and require neither secrecy of a cryptographic key, nor brute-forcing resistance given a bounded number of hash bits. Depending on the OS model, cryptographic keys might also be shared by more than one address space – e.g., if `fork()` is frequently used to generate multiple processes, or if there is a shared memory segment that includes linked pointers.

- Tags do not rely on cryptographic hash generation during capability updates, nor checking during dereference. These could otherwise lead to a performance overhead (e.g., as a result of load-to-use or check-to-use delays), or energy-use overheads (due to frequent cryptographic hash operations).

- Tags prevent reinjection of leaked pointer values, even though the bitwise pattern of the addressable memory contents remain identical. Potential vulnerabilities with hash-based protection include leaking a valid pointer value to a local or remote attacker via socket communications. The attacker could later reinject that value – potentially into a different process if they share keying material (e.g., if they are forked from the same parent).

- Tags ensure provenance validity of capabilities, such that the TCB can deterministically ensure that a pointer value is no longer in memory. As with the previous item, this protects against reinjection, but has the stronger inductive property that the TCB can reliably perform revocation or garbage collection. This is also essential to compartmentalization strength.

However, a hash-based approach also has several appealing properties when compared to tags:

- Cryptographic hashes do not require the implementation of tagged memory, which could reduce memory-subsystem complexity and DRAM-traffic impact.

- Cryptographic hashes do not impose alignment requirements on capabilities, which may improve compatibility.
- Cryptographically protected capabilities can be copied in memory, swapped to disk, or migrated in virtual-machine images, without special support for tags.

  This could entirely avoid the need for special capability load and store instructions, although retaining them might assist with microarchitectural optimization of hash use.

If hashed-based protection were viewed as a stepping stone to a full CHERI implementation, substituting hashing for tags in an initial implementation, there are several steps that could be taken to reduce the further disruption associated with later tag adoption:

- Explicit capability load and store instructions would be maintained and used in future capability-aware memory copying, etc.
- Capability load and store instructions would require strong alignment for values that would later be used for load and store, even though this is not required with hashing.
- Other non-tag-related capability properties, such as monotonicity, would continue to be enforced via guarded manipulation.

However, substantially smaller benefit would arise prior to the introduction of tags: capabilities would be able to provide capability-like spatial memory protection, and probabilistic pointer integrity protection, but not the non-probabilistic protection or enforcement of provenance validity required for stronger policies such as preventing pointer reinjection, supporting temporal memory safety through deterministic pointer identification in memory, or enabling in-address-space compartmentalization that depends on those properties.

8.3 Capability Register File

The decision to separate the capability-register file from the general-purpose integer register file is somewhat arbitrary from a software-facing perspective: we envision capabilities gradually displacing general-purpose integer registers as pointers, but where management of the two register files will remain largely the same, with stack spilling behaving the same way, and so on, as is already the case for disjoint integer and floating-point register files. We selected the separate representation for a few pragmatic reasons:

- Coprocessor interfaces frequently make the assumption of additional register files (a la floating-point registers).
- Capability registers are quite large, and by giving the capability coprocessor its own pipeline for manipulations, we could avoid enforcing a 256-wide path through the main pipeline.
- It is more obvious, given a coprocessor-based interface, how to provide compatibility support in which the capability coprocessor is “disabled,” the default configuration in order to support unmodified MIPS compilers and operating systems.

However, it is entirely possible to imagine a variation on the CHERI design in which (more similar to the manner in which the 32-bit x86 ISA was extended to support 64-bit registers) the two files are *merged* and able to hold both general-purpose integer registers and capability registers. This becomes a more appealing choice in the presence of 128-bit compressed capabilities, as register size doubles rather than quadruples.

Early in our design cycle, capability registers were able to hold only true capabilities (i.e., with tags); later, we weakened this requirement by adding an explicit tag bit to each register, in order to improve support for capability-oblivious code such as memory-copy routines able to copy data structures consisting of both capabilities and ordinary data. This shifts our approach somewhat more towards a merged approach; our view is that efficiency of implementation and compatibility (rather than maintaining a negligible effect on the software model) would be the primary reasons to select one approach or another for a particular starting-point ISA.

Another design variation might have specific capability registers more tightly coupled with general-purpose integer registers – an approach we discussed extensively, especially when comparing with the bounds-checking literature, which has explored techniques based on *sidecar registers* or associative look-aside buffers. Many of these approaches did not adopt tags as a means of strong integrity protection (which we require for the compartmentalization model), which would make associative techniques less suitable. Further, we felt that the working-set properties of the two register files might be quite different; effectively pinning the two to one another would reduce the efficiency of both.

In CHERI-RISC-V (Chapter 5), we parameterize the instruction set to support both split and merged registers files. This will allow us to explore in greater detail the performance and compatibility implications of this design choice.

8.4 The Compiler is Not Part of the TCB for Isolated Code

CHERI is designed to support the isolation of arbitrary untrustworthy code, including code compiled with an incorrect or compromised compiler. The security argument outlined in Chapter 9 starts with the premise that the attacker is able to run arbitrary machine-code. This approach has advantages for high-assurance systems: compilers are often large and complex programs, and proving correctness of their security mechanisms is easier if it does not depend on also proving the correctness of the compiler. This approach also has the advantage that users are not restricted by the security design to programming in just one programming language, and can use any language for which a compiler has been written. In particular, it is a design goal of CHERI that it be able to run legacy code written in C.

Some earlier capability machines, such as the Burroughs B5000, made the compiler a privileged program. We have followed the approach taken in capability machines such as CAP, in which the compiler was not privileged.

8.5 Base and Length Versus Lower and Upper Bounds

The CHERI architecture permits two different interpretations of capabilities: as a virtual address paired with lower and upper bounds; and as a base, length, and current offset. These different interpretations support differing C-language models for pointers. The former, in which pointer casts to integers return their virtual addresses, is more compatible with current software, but risks leaking those virtual addresses (or their implications) out of tagged values where they cannot be found for the purposes of pointer-transformation techniques such as copying garbage collection. The latter, in which pointer casts to integers return their offsets, is less compatible (as comparisons between pointers into different buffers may give surprising equality results), but avoids leakage of virtual address out of tagged values, enabling techniques such as copying garbage collection.

Over time, our thinking on these two approaches has shifted from aiming to support copying garbage collection in C to one focused on revocation and greater compatibility. While some C source code naturally is extremely careful to avoid integer interpretations of pointers, significant amounts of historic code, especially systems code, cannot avoid this idiomatic use. For example, run-time linkers and memory allocators both naturally consider integer virtual addresses as part of their operation. More subtly, techniques such as ordering locks for objects based on object address, or sorting trees based on object address, makes copying garbage collection a difficult prospect. Compressed capabilities further complicate this story, as a precise lower bound may not be possible without padding; this is easy to arrange within memory allocators for new allocations, but when subsetting an existing allocation (e.g., to describe the bounds of an array embedded within another structure), the 0 offset from the bottom of the embedded structure may not carry over to being a 0 offset relative to the base address of a capability.

In recent versions of the CHERI C compiler, we have shifted to preferring a virtual-address interpretation of pointers in all cases except those where specific built-in functions are used to query the offset. We retain an optional compiler mode utilizing an offset interpretation, which will be suitable for future experimentation with copying garbage collection.

8.6 Signed and Unsigned Offsets

In the CHERI instructions that take both a register offset and an immediate offset, the register offset is treated as unsigned integer, whereas the immediate offset is treated as a signed integer.

Register offsets are treated as unsigned so that given a capability to the entire address space (except for the very last byte, as explained above), a register offset can be used to access any byte within it. Signed register offsets would have the disadvantage that negative offsets would fail the capability bounds check, and memory at offsets within the capability greater than 2^{63} would not be accessible.

Immediate offsets, on the other hand, are signed, because the C compiler often refers to items on the stack using the stack pointer as register offset plus a negative immediate offset. We have already encountered observable difficulty due to a reduced number of bits available for immediate offsets in capability-relative memory operations when dealing with larger stack-frame sizes; it is unclear what real performance cost this might have (if any), but it does reemphasize the importance of careful investment of how instruction bits are encoded.

8.7 Address Computation Can Wrap Around

If the target address of a load or store (base + offset + register offset + scaled immediate offset) is greater than *max_addr* or less than zero, it wraps around modulo 2^{64}. The load or store succeeds if this modulo arithmetic address is within the bounds of the capability (and other checks, such as for permissions, also succeed).

An alternative choice would have been for an overflow in the address computation to cause the load or store to fail with a length-violation exception.

The approach of allowing the address to wrap around does not allow malicious code to break out of a sandbox, because a bounds check is still performed on the wrapped-around address.

However, there is a potential problem if a program uses an array offset that comes from a potentially malicious source. For example, suppose that code for parsing packet headers uses an offset within the packet to determine the position of the next header. The threat is that an attacker can put in a very large value for the offset, which will cause wrap-around, and result in the program accessing memory that it is permitted to access, but was not intended to be accessed at this point in the packet processing. This attack is similar to the confused deputy attack. It can be defended against by appropriate use of `CSetBounds`, or by using some explicit range checks in application code in addition to the bounds checks that are performed by the capability hardware.

The advantage of the approach that we have taken is that it fits more naturally with C language semantics, and with optimizations that can occur inside compilers. The following are equivalent in C:

- a[x + y]
- *(a + x + y)
- (a + x)[y]
- (a + y)[x]

They would not be equivalent if they had different behavior on overflow, and the C compiler would not be able to perform optimizations that relied on this kind of reordering.

8.8 Overwriting Capabilities in Memory

In CHERI, if a valid in-memory capability is partly overwritten via an untagged data store, then the tag associated with the in-memory capability is cleared, making it an invalid capability that cannot be dereferenced.

Alternative designs would have been for the capability to be zeroed first before being overwritten; or for the write to raise an exception (with an explicit "clear tag in memory" operation for the case when a program really intends to overwrite a capability with non-capability data).

The chosen approach is simpler to implement in hardware. If store instructions needed to check the tag bit of the memory location that was being written, then they would need a read-modify-write cycle to memory, rather than just a write; in general, the MIPS architecture carefully avoids the need for a read-modify-write cycle within a single instruction. (However, once the memory system needs to deal with cache coherence, a write is not that much simpler than a read-modify-write.)

The CHERI behavior also has the advantage that programs can write to a memory location (e.g., when spilling a register onto the stack) without needing to worry about whether that location previously contained a capability or non-capability data.

A potential disadvantage is that the contents of capabilities cannot be kept secret from a program that uses them. A program can always discover the contents of a capability by overwriting part of it, then reading the result as non-capability data. In CHERI, there are intentionally other, more direct, ways for a program to discover the contents of a capability it owns, and this does not present a security vulnerability.

However, there are ABI concerns: we have tried to design the ISA in such a way that software does not need to be aware of the in-memory layout of capabilities. As it is necessarily exposed, there is a risk that software might become dependent on a specific layout. One noteworthy case is in the operating-system paging code, which must save and restore capabilities and their tags separately; this can be accomplished by using instructions such as `CGetBase` on untagged values loaded from disk and then refining an in-hand capability using `CSetBounds` – an important reason not to limit capability field retrieval instructions to tagged values. We have proposed a new instruction, `CBuildCap`, which would add a tag to an untagged value in a capability-register operand, authorized by a second operand holding a suitably authorized capability, to avoid software awareness of the in-memory layout, as well as to accelerate tag restoration when implementing system services such as swap. This instruction in effect implements rederivation, which is also possible using a sequence of individual instructions refining the authorizing capabilities bounds, permissions, object type, and so on. `CBuildCap` is not intended to change the set of reachable capabilities.

8.9 Reading Capabilities as Bytes

In CHERI, if a data load instruction such as `CLB` is used on a memory location containing a capability, the internal representation of the capability is read. An alternative architecture would have such loads return zero, or raise an exception.

As noted above, because the contents of capabilities are not secret, allowing them to be read as raw data is not a security vulnerability.

8.10 OTypes Are Not Secret

Another consequence of the decision not to make the contents of capabilities secret is that the **otype** field is not secret. It is possible to determine the **otype** of a capability by reading it with `CGetType`, or by reading the capability as bytes. If a program has two pairs of code and data capabilities, (c_1, d_1) and (c_2, d_2) it can check if c_1 and c_2 have the same **otype** by using `CCheckType` on (c_1, d_2), or by invoking `CCall` on (c_1, d_2).

As a result, a program can tell whether it has been passed an object of **otype** O or an interposing object of **otype** I that forwards the `CCall` on to an object of **otype** O (e.g. after having performed some additional access control checks or auditing first).

8.11 Capability Registers are Dynamically Tagged

In CHERI, capability registers and memory locations have a tag bit that indicates whether they hold a capability or non-capability data. (An alternative architecture would give memory locations a tag bit, where capability registers could contain only capabilities – with an exception raised if an attempt were made to load non-capability data into a capability register with `CLC`.)

Giving capability registers and memory locations a tag bit simplifies the implementation of `cmemcpy()`. `cmemcpy()` is a variant of `memcpy()` that copies the tag bit as well as the data, and so can be used to copy structures containing capabilities. As capability registers are dynamically tagged, `cmemcpy()` can copy a structure by loading its constituent words into capability registers and storing them to memory, without needing to know at compile time whether it is copying a capability or non-capability data.

Tag bits on capability registers may also be useful for dynamically typed languages in which a parameter to a function can be (at run time) either a capability or an integer. `cmemcpy()` can be regarded as a function whose parameter (technically a `void *`) is dynamically typed.

8.12 Separate Permissions for Storing Capabilities and Data

CHERI has separate permission bits for storing a capability versus storing non-capability data (and similarly, for loading a capability versus loading non-capability data).

(An alternative design would be just one Permit_Load and just one Permit_Store permission that were used for both capabilities and non-capability data.)

The advantage of separate permission bits for capabilities is that that there can be two protected subsystems that communicate via a memory buffer to which they have Permit_Load and Permit_Store permissions, but do not have Permit_Load_Capability or Permit_Store_Capability. Such communicating subsystems cannot pass capabilities via the shared buffer, even if they collude. (We realized that this was potentially a requirement when trying to formally model the security guarantees provided by CHERI.)

8.13 Capabilities Contain a Cursor

In the C language, pointers can be both incremented and decremented. C pointers are sometimes used as a cursor that points to the current working element of an array, and is moved up and down as the computation progresses.

CHERI capabilities include an offset field, which gives the difference between the base of the capability and the memory address that is currently of interest. The offset can be both incremented and decremented without changing **base**, so that it can be used to implement C pointers.

In the ANSI C standard, the behavior is undefined if a pointer is incremented more than *one* beyond the end of the object to which it points. However, we have found that many existing

C programs rely on being able to increment a pointer beyond the end of an array, decrement it back within range, and then deference it. In particular, network packet processing software often does this. In order to support programs that do this, CHERI offsets are allowed to take on any value.[1] A range check is performed when the capability is dereferenced, so buffer overflows are prevented; thus, the offset can take on intermediate out-of-range values as long as it is not dereferenced.

An alternative architecture would have not included an offset within the capability. This could have been supported by two different capability types in C, one that could not be decremented (but was represented by just a capability) and one that supported decrementing (but was represented by a pair of a capability and a separate integer for the offset). Programming languages that did not have pointer arithmetic could have their pointers compiled as just a capability.

The disadvantage of including offsets within capabilities is that it wastes 64 bits in each capability in cases where offsets are not needed (e.g., when compiling languages that don't have pointer arithmetic, or when compiling C pointers that are statically known to never be decremented).

The alternative (no offset) architecture could have used those 64 bits of the capability for other purposes, and stored an extra offset outside the capability when it was known to be needed. The disadvantage of the no-offset architecture is that C pointers become either unable to support decrementing or enlarging: because capabilities need to be aligned, a pair of a capability and an integer will usually end up being padded to the size of two capabilities, doubling the size of a C pointer, and this is a serious performance consideration.

Another disadvantage of the no-offset alternative is that it makes the seal/unseal mechanism considerably more complicated and hard to explain. A program that has a capability for a range of types has to somehow select which type within its permitted range of types it wishes to use when sealing a particular data capability. The CHERI architecture uses the offset for this purpose; not having an offset field leads to more complex encodings when creating sealed capabilities.

By comparison, the CCured language includes both `FSEQ` and `SEQ` pointers. CHERI capabilities are analogous to CCured's `SEQ` pointers. The alternative (no offset) architecture would have capabilities that acted like CCured's FSEQ, and used an extra offset when implementing SEQ semantics.

8.14 NULL Does Not Have the Tag Bit Set

In some programming languages, pointer variables must always point to a valid object. In C, pointers can either point to an object or be NULL; by convention, NULL is the integer value zero cast to a pointer type.

If hardware capabilities are used to implement a language that has NULL pointers, how is the NULL pointer represented? CHERI capabilities have a **tag** bit; if the **tag** bit is set, a valid

[1] CHERI Concentrate (section 3.4.5) exploits the observation that, in practice, pointers do not wander "far" from their base to reduce the number of bits used to store the base, cursor, and limit addresses. Attempts to move the cursor far out of bounds will, instead, yield an un-tagged result.

capability follows, otherwise the remaining data can be interpreted as (for example) bytes or integers. The representation we have chosen for NULL is that the **tag** bit is not set and the **base** and **length** fields are zero; effectively, NULL is the integer zero stored as a non-capability value in a capability register.

An alternative representation we have could have chosen for NULL would have been with the **tag** bit set, and zero in the **base** field and **length** fields. Effectively, NULL would have been a capability for an array of length zero.

Many CHERI instructions are agnostic as to which of these two conventions for NULL is employed, but the `CFromPtr`, `CToPtr` and `CPtrCmp` operations are aware of the convention.

The advantages of NULL's **tag** bit being unset are:

- Initializing a region of memory by writing zero bytes to it will initialize all capability variables within the region to the NULL capability. Initializing memory by writing zeros is, for example, done by the C `calloc()` function, and by some operating systems.

- It is possible for code to conditionally branch on a capability being NULL by using the `CBTS` or `CBTU` instruction.

8.15 The length of NULL is MAXINT

Given that we have chosen NULL to have its tag bit unset, it isn't semantically meaningful to talk about its length, as NULL is not a reference to a region of memory. But programs can still attempt to query the length of NULL, and the questions arises as to which value is returned.

We have chosen the length of NULL to be $2^{64} - 1$, as this simplifies the implementation of compressed capabilities. To support the semantics of the C language, the capability compression scheme must be able to represent all 2^{64} possible values of **offset** when **tag** is set and **length** is MAXINT. If we make the length of NULL be MAXINT, the compressed capability format can use the same encoding regardless of whether **tag** is set or not: NULL becomes a value whose **offset** is currently zero, but that can be changed (with `CIncOffset`) to any integer value without becoming unrepresentable.

Alternative design choices included:

- Use a capability compression algorithm that also has the property that all values of **offset** are representable when **length** is zero, and make the length of NULL be zero. Versions of the CHERI ISA prior to V7 allowed the length of NULL to be implementation-defined, and used a compression algorithm that had this property, so the length of NULL could be zero. To enable the use of compression algorithms that don't have this property, the V7 ISA defines the length of NULL to be MAXINT.

- Use a different compression algorithm depending on whether **tag** is set or not. This might make the hardware more complex, but there is no reason in principle why valid capabilities (**tag** set) and integers packed into capability registers (**tag** unset) should have to use the same compression algorithm.

8.16 Permission Bits Determine the Type of a Capability

In CHERI, a capability's permission bits together with the **otype** field determine what kind of capability it is. A capability for a region of memory has is unsealed (a **otype** of $2^{64} - 1$) and *Permit_Load* and/or *Permit_Store* set; a capability for an object is sealed and has *Permit_Execute* unset; a capability to call a protected subsystem (a "call gate") is sealed and has *Permit_Execute* set; a capability that allows the owner to create objects whose type identifier (**otype**) falls within a range is unsealed and *Permit_Seal* set.

An alternative architecture would have included a separate *capability type* field, as well as the **perms** field, within each capability; the meaning of the rest of the bits in the capability would have been dependent on the value of the *capability type* field.

A potential disadvantage of not having a *capability type* field is that different kinds of capability cannot use the remaining bits of the capability in different ways.

A consequence of the architecture we have chosen is that it is possible for software receiving the primordial, omnipotent capability to create capabilities with arbitrary permissions. Some of these sets of permissions do not have a clear use case; they just exist as a consequence of the representation chosen for capabilities' permissions. (Other choices are possible; see Appendix D.8 for a less-orthogonal representation.)

8.17 Object Types Are Not Addresses

In CHERI, we make a distinction between the unique identifier for an object type (the **otype** field) and the address of the executable code that implements a method on the type (the **base** + **offset** fields in a sealed executable capability).

An alternative architecture would have been to use the same fields for both, and take the entry address of an object's methods as a convenient unique identifier for the type itself.

The architecture we have chosen is conceptually simpler and easier to explain. It has the disadvantage that the type field is only 24 bits, as there is insufficient space inside the capability for more.

The alternative of treating the set of object type identifiers as being the same as the set of memory addresses enables the saving of some bits within a capability by using the same field for both. It also simplifies assigning type identifiers to protected subsystems: each subsystem can use its start address as the unique identifier for the type it implements. Subsystems that need to implement multiple types, or create new types dynamically can be given a capability with the permission *Permit_Set_Type* set for a range of memory addresses, and they are then able to use types within that range. (The current CHERI ISA does not include the *Permit_Set_Type* permission; it would be needed only for this alternative approach). This avoids the need for some sort of privileged type manager that creates new type identifiers; such a type manager is potentially a source of covert channels. (Suppose that the type manager and allocated type identifiers in numerically ascending order. A subsystem that asks the type manager twice for a new type id and gets back n and $n + 1$ knows that no other subsystem has asked for a new type id in between the two calls; this could in principle be used for covert communication between two subsystems that were supposed to be kept isolated by the capability mechanism.)

8.18 Unseal is an Explicit Operation

In CHERI, it would require an explicit operation to convert an undereferenceable pointer to an object into a pointer that allows the object's contents to be inspected or modified directly. This can be done directly with the `CUnseal` operation, or by using `CCall` to run the result of unsealing the first argument on the result of unsealing the second argument.

An alternative architecture would have been one with "implicit" unsealing, where a sealed capability could be dereferenced without explicitly unsealing it first, provided that the subsystem attempting the dereference had some kind of ambient authority that permitted it to deference sealed capabilities of that type. This ambient authority could have taken the form of a protection ring or the **otype** field of **PCC**.

A disadvantage of an implicit unseal approach such as the one outlined above is that it is potentially vulnerable to the "confused deputy" problem [49]: the attacker calls a protected subsystem, passing a sealed capability in a parameter that the called subsystem expects to be unsealed. If unsealing is implicit, the protected subsystem can be tricked by the attacker into using its privileges to read or write to memory to which the attacker does not have access.

The disadvantage of the architecture we have chosen is that protected subsystems need to be careful not to leak capabilities that they have unsealed, for example by leaving them on the stack when they return to their caller. In an architecture with "implicit unseal", protected subsystems would just need to delete their ambient authority for the type before returning, and would not need to explicitly clean up all the unsealed capabilities that they had created.

8.19 CMove is not Implemented as CIncOffset

`CMove` is an independent instruction to move a capability value from one register to another. In conventional instruction-set design, integer `Move` is frequently an assembler pseudo-operation that expands to an arithmetic operation that does not modify the value (e.g., an add instruction with the zero register as one operand). In an earlier CHERI design, we similarly implemented `CMove` is an assembler pseudo-operation that expanded to `CIncOffset` with an offset of zero. This required that the `CIncOffset` instruction treat a zero offset as a special case, allowing it to be used to move sealed capabilities and values with the tag bit unset. Using a separate opcode for `CMove` has the disadvantage of consuming another opcode, but avoids this special case in the definition of `CIncOffset` in which an exception will not be thrown if a zero operand is used. We have therefore changed to specifying an explicit `CMove` instruction, and removed special casing in `CIncOffset`.

8.20 Instruction-Set Randomization

CHERI does not include features for instruction set randomization [59]; the unforgeability of capabilities in CHERI can be used as an alternative method of providing control flow integrity.

However, instruction set randomization would be easy to add, as long as there are enough spare bits available inside a capability (the 128 bit representation of capabilities does not have many spare bits). Code capabilities could contain a key to be used for instruction set randomization, and capability branches such as CJR could change the current ISR key to the value given in the capability that is branched to.

8.21 System Privilege Permission

In the current version of the CHERI, one of the capability permission bits authorizes access to privileged processor features that would allow bypass of the capability model, if present on **PCC**. This is intended to be used by hybrid operating-system kernels to manage virtual address spaces, exception handling, interrupts, and other necessary architectural features that do not map cleanly into memory-oriented capabilities. It can also be used by stand-alone CHERI-based microkernels to control use of the exception-handling and cache-management mechanisms, and of the MMU on MMU-enabled hardware. Although the permission limits use of features to control the virtual address space (e.g., TLB manipulation), it does not prevent access to kernel-only portions of the virtual address space. This allows kernel code to operate without privileged permission using the capability mechanism to limit which portions of kernel address space are available for use in constrained compartments.

We employ a single permission bit to conserve space (especially in 128-bit capabilities), but also because it offers a coherent view on architectural privilege: many of the privileged architectural instructions allow bypass of in-address-space memory protection in different ways, and using subsets of those operations safely would be quite difficult. In earlier versions of the CHERI ISA, we employed multiple privileged bits, but did not find the differentiation useful in practical software design. In more feature-rich privileged instruction sets (e.g., those with virtualization features), a more fine-grained decomposition might be of greater utility, and could motivate a new capability format intended to authorize use of privilege.

In earlier versions, the privileged permission(s) controlled use of only CP2 privilege (i.e., exception-handling capabilities); in the current version, the bit also controls MIPS privileges available only in kernel mode: TLB, CP0, selected uses of the CACHE instruction, and ERET use. This allows compartmentalization within the kernel address space (e.g., to sandbox untrustworthy components), as well as more general mitigation by limiting use of privileged features to only selected code components, jumped to via code pointers carrying the privileged permission. If virtual-memory and exception-handling features were not controlled by this permission bit, use of those ISA features would allow bypass of in-kernel compartmentalization. Regardless of this bit, extreme care is required to safely compartmentalize within an operating-system kernel.

In our design, absence of the privileged permission denies use of privileged ISA features, but presence does not grant that right unless it is also authorized by kernel mode. Other compositions of the capability permission bit and existing MIPS KSU (kernel/supervisor/user-mode) authorization are imaginable. For example, the permission bit could grant privileged ISA use in userspace regardless of KSU state. While this composition might allow potentially interesting delegation of privilege to user components, the lack of granularity of control appears to offer little benefit when a similar effective delegation can be implemented via the exception model and implied ring transition. In a ring-free design (e.g., one without an MMU or kernel/supervisor/user modes), however, the privileged permission would be the sole means of authorizing privilege.

Another design choice is that we have not extended MIPS with new capability-based privilege instructions; instead, we chose to limit use of existing instructions (such as those used in TLB management). This fails to extend the principle of intentional use to these privileged features; in return we achieve reduced disruption to current software stacks, and avoid introducing new instructions in the opcode space. Despite that slight apparent shortcoming, we observe that fine-grained privilege can still be accomplished – due to use of a permission bit on **PCC**: even within a highly privileged kernel, most functions might operate without the ability to employ privileged instructions, with an explicit use of `CJALR` to jump to a code pointer with the Access_System_Registers permission enabled – which executes only the necessary instructions and reduces the window of opportunity for privilege misuse.

An alternative design would extend the MIPS privileged instruction set to include versions that accept explicit capability operands authorizing use of those instructions, in a manner similar to our extensions to our capability-extended load and store instructions. Another variation on this scheme would authorize setting of a privilege status register, enabling specific instructions (or classes of instructions) based on an offered capability, combining these two approaches to authorize selected (but unmodified) privileged instructions.

Finally, it is conceivable that capabilities could be used to authorize delegation of the right to use privileged instructions to userspace code, rather than simply restricting the right to use privileged instructions in kernel code. We have opted to limit our approach to using capabilities to restrict features in the MIPS model, with a simple and deterministic composition of features.

8.22 Composing CHERI with MIPS Exception Handling

Exception handling is inevitably architecture specific, with the approach taken in MIPS being particularly RISC-esque and quirky. In MIPS, an exception interrupts the instruction flow, sets the **EXL** status flag to transition to kernel mode, disable interrupts, updates the exception cause register to provide information about the exception, saves the current **PC** in **EPC**, and sets **PC** to the appropriate interrupt vector address. Later versions of MIPS allow that address to be configured. Special handling is also provided if an exception fires while **EXL** is set, making use of **ErrorEPC** instead of **EPC**. As only one register is saved by the architecture (**PC**), this makes it difficult to perform a software-based context switch. Other more recent RISC architectures will bank a second register, typically the stack pointer, to allow that to take place. In MIPS, the ABI instead reserves two general-purpose integer registers, `$k0` and `$k1`, for use by exception handlers.

8.22.1 MIPS-centric Exception Handling

A primary goal of our work is to avoid disrupting MIPS exception handling. This has two implications: first, that when a capability-unaware OS is booted and operating, it should work as it did previously; and second, that when a capability-aware OS is used, CHERI extensions to exception handling preserve the existing structure and approach of exception handling.

8.22.2 Capability Extensions to MIPS Special Registers

The most natural extension to MIPS exception handling simply extends the existing **EPC** to be an Exception Program Counter Capability (**EPCC**), and **ErrorEPC** to be an Error Exception Program Counter Capability (**ErrorEPCC**), which can save and restore the full **PCC** rather than just **PC**. However, this is insufficient to authorize or safely protect execution of the exception vector, which the interrupted code may not have access to, and should not be able to influence. As with later MIPS versions, we therefore allow the exception vector to be defined using a special capability register, the Kernel Code Capability (**KCC**), which will be installed in **PCC** when the exception is taken, both authorizing access for the exception handler, and preventing the interrupted code from improperly affecting the exception handler. In the interests of a more object-centered design, in which capabilities for code and data are not unnecessarily, we also provide a special capability register, the Kernel Data Capability (**KDC**), which is readable in kernel mode.

8.22.3 Kernel-Reserved Special Capability Registers

MIPS reserves two general-purpose integer registers for exception-handler bootstrapping, `$k0` and `$k1`. In earlier versions of the CHERI ISA, we similarly reserved two general-purpose capability registers, **KR1C** and **KR2C**, for kernel use. We have since shifted these two the Special Capability Register File, accessed via `CReadHwr` and `CWriteHwr`. We anticipate that they will be used to temporarily save general-purpose capability registers in a similar manner, letting the general-purpose registers be used by the exception handler itself – e.g., to hold a copy of **KDC** for the purposes of memory access to save capability registers for the preempted context. This design choice is especially important when contemplating a merged register file, in which case avoiding further reservations in that file would limit ABI disruption. Avoiding special interpretations of general-purpose capability registers also avoids special access-control rules (e.g., to limit access to those registers), simplifying the ISAs. Finally: these registers are used only very infrequently, and as such take up valuable space that could be available to the compiler, meaning that using up encoding and register-file space is a poorer use of microarchitectural resources.

8.23 Interrupts and CCall Selector 0 Use the Same KCC/KDC

MIPS executes all exception handlers within the same privileged ring. We have inherited that design choice for our `CCall` selector 0 exception handler, with respect to classical ring-based security, and also to the decision to use a single set of **KCC**/**KDC** special registers. Given that domain transitions without user address spaces does not actually require supervisor privilege, it would make substantial sense to shift the software-defined `CCall`/`CReturn` mechanism to a userspace exception handler. These are not supported by MIPS, and substantial prototyping would be required to evaluate this approach. If that were to be implemented, then it would be necessary to differentiate the code and data capabilities for the domain-transition implementation from the kernel's own code and data capabilities – possibly via additional special registers configured and switched by the kernel on behalf of the userspace language runtime.

8.24 CCall Selector 1: Jump-Based Domain Transition

CCall selector 1 offers a non-exception-based mechanism by which non-monotonic capability register-file transformations can be performed, in contrast to the exception-based selector 0. Non-monotonicity is accomplished by virtue of unsealing the sealed operand capabilities to CCall selector 1, whereas selector 0 accomplishes non-monotonicity by virtue of granting access to exception-mode capability registers (**KCC** and **KDC**).

While a standard MIPS pipeline allows a branch delay slot before diverting control flow after a branch, CCall selector 1 does not have a branch-delay slot – as the remaining branch-delay instruction in the calling domain would otherwise have access to data registers written by the branch. Removing the delay slot for this case disturbs the normal control flow of the pipeline, causing a pipeline bubble in our case, and also prevents an important optimization for safe domain crossing – which uses the delay slot to clear the last registers in the calling domain that were needed as operands for the CCall itself. Nevertheless, significant software complexity was necessary to ensure a safe branch-delay slot for CCall selector 1. Rather than removing the delay slot after CCall selector 1, we could have thrown an exception on any instruction that reads or writes **IDC** in the branch-delay slot. The result would be that the newly unsealed **IDC** is available only to code executing at the newly unsealed **PCC**, avoiding premature exposure of **IDC** to the caller before callee code begins executing. However an exception in the branch delay slot would still expose **IDC** to the exception handler, and a creative use of signals could expose **IDC** to the calling domain. Rather than mandate a more complex set of invariants in software, we chose to eliminate the branch delay slot of CCall selector 1.

It is possible to imagine more comprehensive jump-based instructions including:

- A variation that has link-register semantics, saving the caller **PCC** in a manner similar to CJALR. We choose not to implement this to avoid writing two general-purpose registers in one instruction, and because the caller can itself perform a move to a link destination based on CGetPCC.

- A variation that seals caller **PCC** and **IDC** to construct a return-capability pair. We choose not to implement this to multiple register writes in one instruction, because the caller can itself perform any necessary sealing of its own return state, if required. Further, to provide strict call-return semantics, additional more complex behavior is required, which is not well captured by a single RISC instruction.

In general, we anticipate that CCall selector 1 will be used to invoke trusted software routines with similar behavior and tradeoffs to using a software exception handler with selector 0. For example, we expect that microkernel message-passing system calls implemented using selector 0 will clear non-argument capability and general-purpose integer registers, perform global checks, and store any return information required to restore control to the caller before return to userspace. Unlike a return from a system call, the CCall selector 1 trusted routine can jump out of trusted code without any special handling in the ISA, as it will conform to monotonic semantics – i.e., the clearing of registers that should not be passed to the callee, followed by a CJR to transfer control to the callee.

8.25 Compressed Capabilities

256-bit capabilities provide for byte-granularity protection, allowing arbitrary subsets of the address space to be described, as well as providing substantial space for object types, software-defined permissions, and so on. However, they come at a significant performance overhead: the size of 64-bit pointers is quadrupled, increasing cache footprint and utilization of memory bandwidth. Fat-pointer compression techniques exploit information redundancy between the base, pointer, and bounds to reduce the in-memory footprint of fat pointers, reducing the precision of bounds – with substantial space savings. Prior versions of our compression approaches, the CHERI-128 candidates, are described in Appendix E.

8.25.1 Semantic Goals for Compressed Capabilities

Our target for compressed capabilities was 128 bits: the next natural power-of-two pointer size above 64-bit pointers, with an expected one-third of the overhead of the full 256-bit scheme. A key design goal was to allow both 128-bit and 256-bit capabilities to be used with the same instruction set, permitting us to maintain and evaluate both approaches side-by-side. To this end, and in keeping with previously published schemes, the CHERI ISA continues to access fields such as permissions, pointer, base, and bounds via 64-bit general-purpose integer registers. The only visible semantic changes between 256-bit and 128-bit operation should be these: the in-memory footprint when a capability register is loaded or stored, the density of tags (doubled when the size of a capability is halved), potential imprecision effects when adjusting bounds, potential loss of tag if a pointer goes (substantially) out of bounds, a reduced number of permission bits, a reduced object type space, and (should software inspect it) a change in the in-memory format.

The scheme described in our specification is the result of substantial iteration through designs attempting to find a set of semantics that support both off-the-shelf C-language use, as well as providing strong protection. Existing pointer-compression schemes generally provided suitable monotonicity (pointer manipulation cannot lead to an expansion of bounds) and a completely accurate underlying pointer, allowing base and bounds to experience imprecision only during bounds adjustment. However, they did not, for example, allow pointers to go "out of bounds" – a key C-language compatibility requirement identified in our analysis of widely used C programs. The described model is based on a floating-point representation of distances between the pointer and base/bounds, and places a particular focus on fully precise representation bounds for small memory allocations – e.g., as occur on the stack, or when performing string or image processing.

8.25.2 Precision Effects for Compressed Capabilities

Precision effects are primarily visible during the narrowing of bounds on an existing capability. In order to provide the implementation with maximum flexibility in selecting a compression strategy for a particular set of bounds, we have removed the `CIncBase` and `CSetLen` instructions in favor of a single `CSetBounds` instruction that exposes adjustments to both atomically. This allows the implementation to select the best possible parameters with full information about the required bounds, maximizing precision. Precision effects occur in the form of increased alignment requirements for base and bounds: if requested bounds are highly unaligned, then the resulting capability returned by CSetBounds may have broader rights than requested, following stronger alignment rules. CSetBounds maintains full monotonicity; however, bounds on a returned capability will never be broader than the capability passed in. Further, narrowing bounds is itself monotonic: as allocations become smaller, the potential for precision increases due to the narrower range described. Precision effects will generally be visible in two software circumstances: memory allocation and arbitrary subsetting, which have different requirements.

Memory allocation subdivides larger chunks of memory into smaller ones, which are then delegated to consumers – which most frequently are heap and stack allocation, but this can also occur when the operating system inserts new memory mappings into an address space, returning a pointer (now a capability) to that memory. Memory allocators already impose alignment requirements: certainly for word or pointer alignment so that allocated data structures can be stored at natural alignment, but also (for larger allocations) for page or superpage alignment to encourage effective use of virtual memory. Compressed capabilities strengthen these alignment requirements for large allocations, which requires modest changes to heap, stack, and OS memory allocators in order to avoid exposing undesired precision effects. Bounds on memory allocations will be set using CSetBoundsExact, which will throw an exception if precise bounds are not possible due to precision effects.

Arbitrary subsetting occurs when programmers explicitly request that a capability to an existing allocation be narrowed, in order to enforce bounds checks linked to software invariants. For example, an MPEG decoder might subset a larger memory buffer containing many frames into individual frames when processing them, in order to catch misbehavior without permitting (for example) corruption of adjacent frames. Similarly, packet-processing systems frequently embed packet data within other data structures; bugs in protocol parsing or packet construction could affect packet metadata, with security consequences. 128-bit CHERI can provide precise subsetting for smaller subsets, but may experience precision effects for larger subsets. These are accepted in our programmer model, and could permit buffer overflows between subsets, which would be prevented in the 256-bit model. Unless specifically annotated to require full precision, arbitrary subsetting will utilize CSetBounds, which can return monotonically non-increasing – but with potentially imprecise bounds.

Two further cases required careful consideration: object capabilities, and the default data capability, for quite different reasons. Object capabilities require additional capability fields (software-defined permission bits, and the fairly wide object type field). The default data capability is an ordinary 128-bit capability, but has the property that use of a full cursor (base plus offset) introduces a further arithmetic addition in a critical path of MIPS loads and stores. In both cases, we have turned to reduced precision (i.e., increased alignment requirements) to eliminate these problems, looking to minimum page-granularity alignment of bounds while retaining fully precise pointers. By requiring strong alignment for default data capabilities, the extra addition becomes a `logical or` when constructing the final virtual address, assisting with the critical path. As object capabilities are used only by newly implemented software, and provide coarser-grained protection, we accepted the stronger alignment requirement for sealed capabilities, and have not encountered significant problems as a result.

The final way in which imprecision may be visible to software is if the pointer (offset) in a capability goes substantially out of bounds. In this case, the compression scheme may not be able to represent the distances from the pointer to its original bounds accurately. In this scenario, the tag will be cleared on the capability to prevent dereference, and then one of the resulting pointer value or bounds must be cleared due to the unrepresentability of the resulting value. To discourage this from happening in the more common software case of allowing small divergence from the bounds, `CSetBounds` over-provisions bits required to represent the distances during compression; however, that over-provisioning comes at a slight cost to precision: i.e., we accept slightly stronger alignment requirements in return for the ability to allow pointers to be somewhat out of bounds.

8.26 Capability Encoding Mode

As implemented in CHERI-MIPS, CHERI duplicates the full load-store encoding space to provide capability-relative variations on load and store instructions. This approach ensures intentionality: the architecture is always able to perform a **DDC**-relative access with legacy integer-relative load and store instructions, and is always able to assert that the tag bit is set for capability-relative load and store instructions. However, this makes heavy use of remaining unused opcode space in many instruction sets, and so finding alternative encoding models to make less copious use of opcode space is desirable.

One scheme we are deploying in CHERI-RISC-V is the use of legacy vs. capability encoding modes: in the legacy encoding mode, load and store opcodes have their current interpretations, and a small selection of capability-relative loads and stores are added. To get access to the full range of load and store variations, the encoding mode can be switched to one in which existing load and store opcodes are instead interpreted as requiring capability operands, and **DDC**-relative integer-based access is disabled.

There are a variety of mechanisms that could be used to switch between encoding modes, but information on the mode must be available at the time of instruction decode. There are several essential considerations:

How frequently will mode switches take place? There are a range of possibilities, from whole programs or systems operating within a single encoding, to inter-function or sub-function changes in mode depending on ABI and optimization requirements. Given our overall goal in CHERI of avoiding the need for additional exceptions to a privileged supervisor for capability manipulation, we similarly believe that a non-exception-based encoding transition mechanism is desirable to support more tight integrations of integer-relative and capability-relative generated code. As such, the mechanisms we consider will generally support granular transition, at least at library boundaries or individual function call and return.

How will encoding mode be selected and preserved across function calls? Assuming that a more granular approach to encoding is desired – e.g., that there are direct calls between code generated in differing modes – then it will be necessary to switch to the callee encoding during function entry, and restore the caller encoding on function return. This might be supported implicitly through contextual information, such as using page-table properties, or explicitly such as through extended or entirely new compiler- or linker-managed instructions saving and setting encoding modes.

How will encoding mode be preserved across context switches? As with function-call boundaries, this might be implicit (e.g., based on the address or metadata held in **PCC**, or via page-table metadata for the target that **PCC** points to) or explicit (e.g., the saving and restoring of a bit in `{m,s,u}ccsr` when an exception is taken).

What will the performance implications be for microarchitectural optimizations? For example, will the target encoding be accurately predicted alongside the target **PCC**, so that speculative execution can utilize the correct encoding?

How should encoding-mode selection work around protection-domain boundary crossings? When control is transferred across a protection-domain boundary (e.g., by virtue of an exception being thrown, or use of `CCall`), the destination code must be able to ensure that it is being safely executed with its intended interpretation. This might be implied by the mechanism (e.g., by virtue of properties of the virtual page holding the executing code) or explicit (e.g., using dedicated instructions in the callee to switch modes, or assert the mode, before any affected instructions are executed).

Should encoding-mode switches require privilege? One potential fear is that an additional encoding mode increases the gadget space available to control-flow attackers. As long as the effect is only for the current execution context, we currently take the view that changing encoding modes does not require privilege: the set of available capabilities remains the same; the increase in gadget space is small; and attacks on control flow to use gadgets rely on having bypassed control-flow robustness arising from fine-grained code capabilities. See Section 8.27 for further considerations.

Potential encoding mode-switch mechanisms

We are considering the following mechanisms:

New jump instruction sets mode flag in `{m,s,u}ccsr` A mode bit in `{m,s,u}ccsr` would select between the two different instruction encodings. A new jump instruction would allow the target mode to be selected via an immediate operand ("enter integer encoding mode" or "enter capability encoding mode"). This is a simple mechanism allowing dynamic selection of encoding at a fine granularity – e.g., per function. It utilizes existing context switching, as `{m,s,u}ccsr` will already be saved and restored.

This approach has a number of complications from a software perspective: on function call, the caller must be aware of the callee encoding; on function return, the callee must likewise be aware of the caller encoding, so as to ensure that the correct encoding is used when control flow moves between functions. In some usage scenarios, such as dynamically linked libraries, this might require the introduction of thin stubs – already present thanks to PLTs during call, but not presently implemented in current software stacks. Certain more complex control flows, such as those relating to exception delivery, might similarly present obstacles.

Flag in jump-target addresses, maintained in {m,s,u}ccsr Because of the minimum code alignment of 16 bits in RISC-V, the lowest bit in a jump target address is ignored (and cleared when installed in **PC**), leaving it available as a potential flag to instructions such as JALR and CJALR[2]. This bit could be used to select the target ISA encoding, in the style of ARMv7's instruction-set trigger to switch between 32-bit instructions and 16-bit Thumb instructions. {m,s,u}ccsr would contain an architectural mode bit to select between the two instruction codings. A lowest bit of 0 in the target virtual address would select integer encoding mode; a lowest bit of 1 in the target virtual address would select capability encoding mode. JALR and CJALR would similarly adjust the virtual address of a generated return address or capability, so as to restore the correct encoding on function return or exception delivery. This approach would avoid the need for any new instructions being introduced, and would associate the encoding with the callee rather than caller. Branch-predictor targets could also reliably predict encoding to allow speculative fetch and decode.

Software would be relatively easily modified to set the bit as needed during compile-time or run-time linking. However, there may already be software consumers making use of the same bit.

Flag in jump-target addresses, maintained in PCC As with the prior option, the lowest bit in the target virtual address for JALR or CJALR would select the target encoding. However, rather than extending {m,s,u}ccsr, the lowest bit would persist in **PCC** and be ignored as an address for fetching instructions, allowing it to continue to indicate the target encoding. This would require a modest change to the baseline RISC-V ISA to preserve but ignore the bit in **PC**. This approach avoids the need for a new {m,s,u}ccsr bit, and differently addresses the goal of allowing encoding to track executing code, and be saved, set, and restored around function calls and returns.

As the bit would not be cleared, debuggers and other address-aware code, such as code implementing PC-relative GOT access in hybrid mode, would have to be suitably adapted to ignore the bit. It might be desirable to have the bit also ignored for the purposes of AUIPC used for GOT access.

New capability flag to select the target encoding of a jump A new capability flag could be introduced to select the target encoding for capability-relative jump targets (i.e., capabilities authorizing instruction fetch). Changing the flag would not change the rights associated with a capability, allowing us to avoid a new permission bit to authorize changing the flag, and sealing would prevent modification. Target encodings could be saved with corresponding branch-predictor entries to allow speculative fetch and decode. The encoding state would be preserved with **PCC** on call, return, and in exception handling.

Explicit unprivileged instruction to switch modes New instructions could be added to switch explicitly between the two opcode instructions, to be placed either in function prologues/epilogues, or in trampolines inserted by static or dynamic linkage. Standard

[2] The JAL instruction shifts its immediate operand, and could not be used to change mode.

RISC-V `RWI`, `RSI`, and `RCI` CSR manipulation instructions could be used. Dynamic changes of encoding might necessitate invalidating speculative decoding and execution, however.

Page-table flag specifying encoding for executable code On MMU-enabled systems, page-table mappings for pages could themselves contain information on the encoding of instructions stored in the page. As binary pages are typically mapped by a run-time linker that is aware of code properties, this would avoid changes to code generation itself, use of new instructions, flags, etc. However, this would be dependent on having an MMU present, software authors using the MMU, as well as code having page alignment by encoding type. When running without virtual addressing enabled, it would not be possible to switch modes, which would be undesirable for small embedded-class systems.

Of these potential schemes, requesting a target encoding based on a PCC flag seems the most appealing.

8.27 Capability Encoding Mode Switching Can Be Unprivileged

In CHERI-RISC-V, we introduce the concept that existing integer-relative load and store opcodes could be reused in a richer "capability encoding mode", conserving opcode space. We argue above that switching between encodings is a safe operation to be performed without privilege – i.e., by arbitrary untrustworthy code – as long as safe mechanisms exist to switch to a predetermined encoding state when transitioning across trust boundaries. For example, it must be the case that exception handlers can operate reliably in their intended encoding regardless of the encoding mode being used by unprivileged user code triggering an exception. Similarly, a reliable encoding switch must be achieved when using `CCall`.

Our argument for safe unprivileged use is grounded in the belief that the primary concern is one of potential code-reuse attacks, as switching encodings does not change the set of capabilities available to executing code. Instead, the fear is that an attacker able to manipulate control flow now has access to an increased number of gadgets, as executable memory may now be used with multiple interpretations. We agree that the gadget space does modestly increase, and consider the problem from two perspectives:

When the attack is against hybrid code: The attacker may have the ability to influence an integer-based **PC** value, and will gain access to additional gadgets (possibly doubling the gadget space). However, in hybrid code making only limited use of capabilities, CHERI is not intended to provide additional control-flow robustness.

When the attack is against pure-capability code: The attacker must first gain influence over a capability-based **PCC** value, which will not only be protected against a number of common attacks (e.g., by virtue of tagged memory detecting data overwrites), but also will have narrowed bounds significantly limiting available gadget space.

Further, a successful mode switch will have the sole impact of converting capability-relative loads and stores to integer-relative loads and stores against **DDC**, which will hopefully be set to NULL when executing in a pure-capability code environment – meaning that while the interpretation of instructions has changed, the impact of the newly accessible instructions will by default be an exception being thrown.

Neither of these arguments precludes potentially effective manipulations of the run-time environment by the attacker, but many tools currently available to attackers that might benefit from a mode switch are entirely eliminated or significantly mitigated.

Overall, this leads us to the conclusion that unprivileged transition between encodings is permissible. However, significant care must be taken to ensure that when a privilege change does occur, there is a safe mechanism by which exception handlers or domain-transition mechanisms can execute only in the desired mode.

Chapter 9

CHERI in High-Assurance Systems

This chapter considers the roles of formal methods relating to the assurance of CHERI hardware and software. It gives an informal explanation of some features of the CHERI mechanism that may of interest to developers of high-assurance hardware, secure microkernels, and formal models of CHERI, including an initial security argument for a reference monitor. Further work on proofs of properties of the CHERI ISA are now part of the CIVF project, noted in the last section of this chapter.

9.1 Unpredictable Behavior

In the semantics for the CHERI instructions in Chapter 7, we try to avoid defining behavior as "unpredictable". There were several reasons for avoiding unpredictable behavior, including the difficulty it creates for formal verification. Although CHERI is based on the MIPS ISA, the MIPS ISA specification (e.g., for the R4000) makes extensive use of "unpredictable". If "unpredictable" is modeled as "anything could happen", then clearly the system is not secure. As a concrete example, imagine a hypothetical CHERI implementation that contains a Trojan horse such that when a sandboxed program executes an arithmetic instruction whose result is "unpredictable", it also changes the capability registers so that a capability granting access to the entire virtual address space is placed in a capability register. If "unpredictable" means that anything could happen, then this is compliant with the MIPS ISA; it is also obviously insecure. Later versions of the MIPS ISA (e.g., MIPS64 volume I) make it clear that "unpredictable" is more restrictive than this, saying that "*unpredictable* operations must not read, write, or modify the contents of memory or internal state that is inaccessible in the current processor mode". However, that is clearly not strong enough.

For the CHERI mechanism to be secure, we require that programs whose behavior is "unpredictable" according to the MIPS ISA do not modify memory or capability registers in a way that allows the capability mechanism to be bypassed. One easy way to achieve this is that the "unpredictable" case requires that neither memory nor capability registers are modified.

The test suite for our CHERI1 FPGA implementation checks that the CPU follows known CHERI1-specific behavior in the "unpredictable" cases.

9.2 Bypassing the Capability Mechanism Using the TLB

If a program can modify the TLB (the status register has CU0 set, KSU not equal to 2, EXL set or IRL set), then it can bypass the capability mechanism by modifying the TLB. Although composition with the Memory Management Unit and virtual-addressing mechanism in this manner is a critical and intentional part of our design, it is worth considering the implications from the perspective of high-assurance design. The "attack" is as follows: Consider a location in memory whose virtual address is not accessible using the capability mechanism; take its physical address and change the TLB so that its new virtual address is one to which you have a capability, and then access the data through the new virtual address. There are several ways to prevent this attack:

- In CheriBSD, user-space programs are unable to modify the TLB (except through system calls such as `mmap`), and thus cannot carry out this attack. This security argument makes it explicit that the security of the capability mechanism depends on the correctness of the underlying operating system. However, this may not be adequate for high-assurance systems.

- Similarly, a high-assurance microkernel could run untrusted code in user space, with KSU=2, CU0 false, EXL false, and IRL false. A security proof for the combined hardware-software system could verify that untrusted code cannot cause this condition to become false except by reentering the microkernel via a system call or exception.

- A single-address-space microkernel that has no need for the TLB could run on a CHERI-enabled CPU without a TLB. Our CHERI1 FPGA prototype can be synthesized in a version without a TLB, and our formal model in the L3 specification language includes a TLB-less variant. Removing the TLB for applications that don't need it saves chip area, and removes the risk that the TLB could be used as part of an attack.

- We are considering future extensions to CHERI that would allow the capability mechanism to be used for sandboxing in kernel mode; these would allow more control over access to the TLB when in kernel mode. As well as enabling sandboxing of device drivers in monolithic kernels such as that of CheriBSD, the same mechanism could also be used by microkernels.

9.3 Malformed Capabilities

The encoding formats for capabilities can represent values that can never be created using the capability instructions while taking the initial contents of the capability registers as a starting point. For example, in the 256-bit representation, there are bit patterns corresponding to **base** $+$ **length** $> 2^{64}$. The capability registers are initialized on reset, so there will never be malformed capabilities in the initial register contents, and a CHERI instruction will never create malformed capabilities from well-formed ones. However, DRAM is not cleared on system reset, so that it is possible that the initial memory might contain malformed capabilities with the tag bit set.

Operating systems or microkernels are expected to initialize memory before passing references to it to untrusted code. (If you give untrusted code a capability that has the *Load_Capability* permission and refers to uninitialized memory, you don't know what rights you are delegating to it.) This means that untrusted code should not be in a position to make use of malformed capabilities.

There are (at least) two implementation choices. An implementation of the CHERI instructions could perform access-control checks in a way that would work on both well-formed and malformed capabilities. Alternatively, the hardware could be slightly simplified by performing the checks in a way that might behave unexpectedly on malformed capabilities, and then rely on the capability mechanism (plus the operating system initializing memory) to guarantee that they will never become available to untrusted code.

If the hardware is designed to guard against malformed capabilities, this presents special difficulties in testing. No program whose behavior is defined by the ISA specification will ever trigger the case of encountering a malformed capability. (Programs whose behavior is "unpredictable", because they access uninitialized memory, may encounter them). However, some approaches to automatic test generation may have difficulty constructing such tests.

More generally, however, uninitialized memory might also contain highly privileged and yet entirely well-formed capabilities, and hence references to that memory should be given to less trustworthy code only after suitable clearing. This requirement is present today for current hardware, as uncleared memory on boot might contain sensitive data from prior boots, but this requirement is reinforced in a capability-oriented environment.

9.4 Constants in the Formal Model

The L3 language that we used to specify CHERI does not have a notion of a named constant as distinct from a mutable variable. Fully machine-checked security proofs may need to prove that some of these constants are in fact constant. (For example, that it is not possible to bypass the capability mechanism by changing the CPU's endianness and hence the effect of a capability dereference, because there is no way to change the endianness).

9.5 Outline of Security Argument for a Reference Monitor

The CHERI ISA can be used to provide several different security properties (for example, control-flow integrity or sandboxing). This section provides the outline of a security argument for how the CHERI instructions can be used to implement a reference monitor.

The Trusted Computer System Evaluation Criteria ("Orange Book") [88] expressed the requirement for a reference monitor as "The TCB shall maintain a domain for its own execution that protects it from external interference or tampering".

The Common Criteria [51] contain a similar requirement:

> "ADV_ARC.1.1D The developer shall design and implement the [target of evaluation] so that the security features of the [target of evaluation security functionality] cannot be bypassed."

> "ADV_ARC.1.2D The developer shall design and implement the [target of evaluation security functionality] so that it is able to protect itself from tampering by untrusted active entities."

In this section, we we explain how the CHERI mechanism can be used to provide this requirement(s), and provides a semi-formal outline of a proof of its correctness.

We are assuming that the system operates in an environment where the attacker does not have physical access to the hardware, so that hardware-level attacks such as introducing memory errors [43] are not applicable.

In this section, we do not consider covert channels. There are many applications where protection against covert channels is not a requirement. The CHERI1 FPGA implementation has memory caches, which probably could be exploited as a covert channel.

The architecture we use to meet this requirement consists of (a) some trusted code that initializes the CPU and then calls the untrusted code; and (b) some untrusted code. The CHERI capability mechanism is used to restrict which memory locations can be accessed by the untrusted code. Here, "trusted" means that, for the purpose of security analysis, we know what the code does. The "untrusted" code, on the other hand, might do anything.

The reference monitor consists of the trusted code and the CHERI hardware; and the "security domain" provided for the reference monitor consists of a set of memory addresses (S_K) for the data, code, and stack segments of the trusted code, together with the CHERI reserved registers.

Our security requirement of the hardware is that the untrusted code will run for a while, eventually returning control to the trusted code; and when the trusted code is re-entered, (a) it will be reentered at one of a small number of known entry points; (b) its code, data and stack will not have been modified by the untrusted code; and (c) the reserved capability registers will not have been modified by the untrusted code.

This security property provided by the hardware allows us to reason that the trusted code is still trusted when it is reentered. If its code and data have not been modified. we can still know what it will do (to the extent that it is actually trustworthy – not just "trusted"),

The "cannot be bypassed" and "tamperproof" requirements are here interpreted as meaning that there is no way within the ISA to modify the reference monitor's reserved memory or the reserved registers. That is, all memory accesses are checked against a capability register, and do not succeed unless the capability permits them. The untrusted code can access memory without returning control to the trusted code; however, all of its memory access are mediated by the capability hardware, which is considered to be part of the reference monitor. Tampering with the reference monitor by making physical modifications to the hardware is considered to be out of scope; the attacker is assumed not to have physical access.

The proof of this security property proceeds by induction on states. Let the predicate *SecureState* refer to the following set of conditions:

- CP0.Status.KSU $\neq$ 0
- CP0.Status.CU0 = **false**
- CP0.Status.EXL = **false**

- CP0.Status.ERL = **false**

- The TLB is initialized such that every entry has been initialized; every entry has a valid page mask; and there is no (ASID, virtual address) pair that matches multiple entries.

- Let S_U be a set of (virtual) memory addresses allocated for use by the untrusted code, and T_U a set of **otype** values allocated for use by the untrusted code.

- The set of virtual addresses S_U does not contain an address that maps (under the TLB state mentioned above) into any of the memory addresses reserved for use by the trusted code's code, stack or data segments.

- The set of virtual addresses S_U does not contain an address that maps (under the TLB state mentioned above) into the physical address used by a memory-mapped I/O device. (If this property is weakened to allow some I/O devices to be memory-mapped by untrusted code, then the security proof has to show that the I/O device can't be used to break the security property, e.g. by causing the I/O device to DMA into a region of memory outside of S_U).

- The set of virtual addresses S_U are all mapped to cached memory. (A load-linked operation on uncached memory is defined as unpredictable in the MIPS ISA. While this probably can't be used to attack a real system, any unpredictable behavior has to prevent for provable security).

- All capability registers have **base** $+$ **length** $\leq 2^{64}$ **or tag** $=$ **false**.

- The above is also true of all capabilities contained within the set of memory addresses S_U.

- All capability registers are either (a) reserved registers; (b) have **tag** = **false**; (c) are sealed with an **otype** not in T_U; or do not grant *Access_System_Registers* permission.

- The above is also true of all capabilities contained within the set of memory addresses S_U.

- All capability registers are either (a) reserved registers; (b) have **tag** = **false**; (c) are sealed with an **otype** not in T_U; or do not grant access to a region of virtual addresses outside of S_U.

- The above is also true of all capabilities contained within the set of memory addresses S_U.

- All capability registers are either (a) reserved registers; (b) have **tag** = **false**; (c) are sealed with an **otype** not in T_U; or do not grant access to a region of the **otype** space outside of T_U.

- The above is also true of all capabilities contained within the set of memory addresses S_U.

- If the current instruction is in a branch delay slot, then the above restrictions on capability registers also apply to the **PCC** value that is the target of the branch. That is, *SecureState* is not true if the trusted code does a CJR that grants privilege and then runs the first instruction of the untrusted code in the branch delay slot.

Let the predicate *TCBEntryState* refer to a state in which the trusted code has been reentered at one of a small number of known entry points.

We assume that *SecureState* is true initially (i.e., a requirement of the trusted code is that it puts the CPU into this state before calling the untrusted code). We then wish to show that *SecureState* $\Rightarrow$ **X** (*SecureState* **or** *TCBEntryState*) (where **X** is the next operator in linear temporal logic). By induction on states, *SecureState* $\Rightarrow$ *TCBEntryState* **R** *SecureState* (where **R** is the release operator in linear temporal logic).

The argument that *SecureState* $\Rightarrow$ **X** (*SecureState* **or** *TCBEntryState*) can be summarized as:

- Given that CP0.Status.KSU $\neq$ 0, CP0.Status.CU0 = **false**, CP0.Status.EXL = **false** and CP0.Status.ERL = **false**, all instructions will either raise an exception (**X** *TCBEntryState*) or leave CP0 registers unchanged, leaving this part of the *SecureState* invariant unchanged.

- Given that CP0.Status.KSU $\neq$ 0 (etc.), all instructions will either raise an exception or leave the TLB unchanged, preserving the parts of *SecureState* relating to the TLB.

- Given that the TLB is in the state given by *SecureState*, load and store operations will not result in "undefined" or "unpredictable" behavior due to multiple matches in the TLB.

- Given that CP0.Status.KSU $\neq$ 0 (etc.), and the TLB is in the state described above, no instruction can result in behavior that is "undefined" according to the MIPS ISA. (The MIPS ISA specification makes a distinction between "undefined" and "unpredictable", but our model in the L3 language combines the two).

- However, instructions can still result in behavior that is "unpredictable" according to the MIPS ISA. These cases can be dealt with by providing a CHERI-specific refinement of the MIPS ISA (i.e. describing what CHERI does in these cases).

- The capability instructions preserve the part of *SecureState* that relates to the capability registers and to capabilities within S_U.

- Given that the capability registers (apart from reserved registers) do not grant access to any memory addresses outside of S_U, store instructions might raise an exception (**X** *TCBEntryState*), but they will not modify locations outside of S_U; thus, the trusted code's data, code and stack segments will be unmodified.

- Given that the capability registers (apart from the reserved registers) do not grant *Access_System_Registers* permission, the reserved registers will not be modified.

The theorem *SecureState* ⇒ *TCBEntryState* **R** *SecureState* uses the **R** operator, which is a weak form of "until": the system might continue in *SecureState* indefinitely. Sometimes it is desirable to have the stronger property that *TCBEntryState* is guaranteed to be reached eventually. This can be ensured by having the trusted code enable timer interrupts, and use a timer interrupt to force return to *TCBEntryState* if the untrusted code takes too long.

More formally, the following properties are added to *SecureState* to make a new predicate, *SecureStateTimer*:

- CP0.Status.IE = **true**
- CP0.Status.IM(7) = **true**

Given that CP0.Status.KSU $\neq$ 0 (etc.), it follows that these properties are also preserved, i.e. *SecureStateTimer* ⇒ *TCBEntryState* **R** *SecureStateTimer*.

As CP0.Count increases by at least one for every instruction, a timer interrupt will eventually be triggered. (If Compare is 2, for example, and Count increments from 1 to 3 without ever going through the intervening value of 2, a timer interrupt is still triggered). As CP0.KSU $\neq$ 0, CP0.Status.EXL = **false**, CP0.Status.ERL = **false**, CP0.Status.IE = **true** and CP0.Status.IM(7) = **true**, the interrupt will be enabled and return to *TCBEntryState* will occur:

SecureStateTimer ⇒ **F** *TCBEntryState*

It then follows that *SecureStateTimer* ⇒ *SecureStateTimer* **U** *TCBEntryState*, where **U** is the until operator in linear temporal logic.

Illicit Information Flows

Using an argument similar to the one in the preceding section, it ought to be possible to formally prove confidentiality properties of the CHERI ISA. However, proofs of confidentiality suffer from the "refinement paradox": confidentiality properties are not preserved by refinement. If there is any non-determinism in a specification, a refinement of it might leak secret information via values that were originally left unspecified.

In more concrete terms, an implementation of the CHERI ISA might leak secret information due to security problems at the microarchitectural level.

The Common Criteria [52] uses the term "covert channel" (alternatively, "illicit information flow") for cases where it is possible to use features of the implementation to signal information in a way that is prohibited by the security policy.

The most obvious potential source of a covert channel in CHERI is using the memory caches as a timing channel. Meltdown [71] and Spectre [61] are examples of realistic attacks against a CPU's memory protection using the cache as a timing channel. Subsequently, the related Foreshadow attacks have been reported [120, 149].

To reduce the risk of an attack similar to Meltdown, implementations of CHERI should perform MMU and capability permissions checks before a store or load, rather than speculatively executing the load and store before all capability checks have completed: tag violation, bounds check, permissions check, seal check, and so on.

9.6 CIFV

The initial effort begun under CTSRD on applying formal methods to the CHERI instruction-set architecture has been superceded by the I2O-funded MTO CIFV project (CHERI ISA Formal Verification). CIFV is currently scheduled to end in early 2021, in order to track certain CHERI variants being developed under the ECATS project (Extending the CHERI Architecture for Trustworthiness in SSITH) – notably, CHERI-RISC-V. However, early results are expected later in 2019, and during 2020.

Chapter 10

Research Approach

In this chapter, we describe the research approach and methodology, grounded initially in hardware-software co-design and now in hardware-software-formal co-design, used to develop the CHERI protection model and CHERI-MIPS ISA.

10.1 Motivation

The CHERI protection model provides a sound and formally based architectural foundation for the principled development of highly trustworthy systems. The CHERI approach builds on and extends decades of research into hardware and operating-system security.[1] However, some of the historic approaches that CHERI incorporates (especially capability architectures) have not been adopted in commodity hardware designs. In light of these past transition failures, a reasonable question is "Why now?" What has changed that could allow CHERI to succeed where so many previous efforts have failed? Several factors have motivated our decision to begin and carry out this project:

- Dramatic changes in threat models, resulting from ubiquitous connectivity and pervasive uses of computer technology in many diverse and widely used applications such as wireless mobile devices, automobiles, and critical infrastructure. In addition, cloud computing and storage, robotics, software-defined networking. safety of autonomous systems, and the Internet of Things have significantly widened the range of vulnerabilities that can be exploited.

- An extended "arms race" of inevitable vulnerabilities and novel new attack mechanisms has led to a cycle of "patch and pray": systems will be found vulnerable, and have little underlying robustness to attackers should even a single vulnerability be found. Defenders must race to patch systems as vulnerabilities are announced – and vulnerabilities may have long half-lives in the field, especially unpublicized ones. There is a strong need for

[1] Levy's *Capability-Based Computer Systems* [68] provides a detailed history of segment- and capability-based designs through the early 1990s [68]. However, it leaves off just as the transition to microkernel-based capability systems such as Mach [3], L4 [69], and, later, seL4 [60], as well as capability-influenced virtual machines such as the Java Virtual Machine [40], begins. Chapter 11 discuss historical influences on our work in greater detail.

underlying architectures that offer stronger inherent immunity to attacks; when successful attacks occur, robust architectures should yield fewer rights to attackers, minimize gained attack surfaces, and increase the work factor for attackers.

- New opportunities for research into (and possible revisions of) hardware-software interfaces, brought about by programmable hardware (especially FPGA soft cores) and complete open-source software stacks such as FreeBSD [77] and LLVM [66].

- An increasing trend towards exposing inherent hardware parallelism through virtual machines and explicit software multi-programming, and an increasing awareness of information flow for reasons of power and performance that may align well with the requirements of security.

- Emerging advances in programming languages, such as the ability to map language structures into protection parameters to more easily express and implement various policies.

- Reaching the tail end of a "compatibility at all costs" trend in CPU design, due to proximity to physical limits on clock rates and trends towards heterogeneous and distributed computing. While "Wintel" remains entrenched on desktops, mobile systems – such as phones and tablet PCs, as well as appliances and embedded devices – are much more diverse, running on a wide variety of instruction set architectures (especially ARM and MIPS).

- Similarly, new diversity in operating systems has arisen, in which commercial products such as Apple's iOS and Google's Android extend open-source systems such as FreeBSD, Mach [3], and Linux. These new platforms abandon many traditional constraints, requiring that rewritten applications conform to new security models, programming languages, hardware architectures, and user-input modalities.

- Development of *hybrid capability-system models* (notably Capsicum [131]) that integrate capability-system design tenets into current operating-system and language designs. With CHERI, we are transposing this design philosophy into the instruction-set architecture. Hybrid design is a key differentiator from prior capability-system processor designs that have typically required ground-up software-architecture redesign and reimplementation.

- Significant changes in the combination of hardware, software, and formal methods to enhance assurance (such as those noted above) now make possible the development of trustworthy system architectures that previously were simply too far ahead of their times.

10.1.1 C-Language Trusted Computing Bases (TCBs)

Contemporary client-server and cloud computing are based on highly distributed applications, with end-user components executing in rich execution substrates such as POSIX applications on UNIX, or AJAX in web browsers. However, even thin clients are not thin in most practical senses: as with client-server computer systems, they are built from commodity operating-system kernels, hundreds of user-space libraries, window servers, language runtime environments, and web browsers, which themselves include scripting language interpreters, virtual machines, and rendering engines. Both server and embedded systems likewise depend on complex (and quite similar) software stacks. All require confluence of competing interests, representing multiple sites, tasks, and end users in unified computing environments.

Whereas higher-layer applications are able to run on top of type-safe or constrained execution environments, such as JavaScript interpreters, lower layers of the system must provide the link to actual execution on hardware. As a result, almost all such systems are written in the C programming language; collectively, this Trusted Computing Base (TCB) consists of many tens of millions of lines of trusted (but not trustworthy) C and C++ code. Coarse hardware, OS, and language security models mean that much of this code is security-sensitive: a single flaw, such as an errant NULL pointer dereference in the kernel, can expose all rights held by users of a system to an attacker or to malware.

The consequences of compromise are serious, and include loss of data, release of personal or confidential information, damage to system and data integrity, and even total subversion of a user's online presence and experience by the attacker (or even accidentally without any attacker presence!). These problems are compounded by the observation that the end-user systems are also an epicenter for multi-party security composition, where a single web browser or office suite (which manages state, user interface, and code execution for countless different security domains) must simultaneously provide strong isolation and appropriate sharing. The results present not only significant risks of compromise that lead to financial loss or disruption of critical infrastructure, but also frequent occurrences of such events.

Software vulnerabilities appear inevitable; indeed, an arms race has arisen in new (often probabilistic) software-based mitigation techniques and exploit techniques that bypass them. Even if low-level escalation techniques (such as arbitrary code injection and code reuse attacks) could be prevented, logical errors and supply-chain attacks will necessarily persist. Past research has shown that compartmentalizing applications into components executed in isolated sandboxes can mitigate exploited vulnerabilities (sometimes referred to as privilege separation). Only the rights held by a compromised component are accessible to a successful attacker. This technique is effectively applied in Google's Chromium web browser, placing HTML rendering and JavaScript interpretation into sandboxes isolated from the global file system. Compartmentalization exploits the principle of least privilege: if each software element executes with only the rights required to perform its task, then attackers lose access to most all-or-nothing toeholds; vulnerabilities may be significantly or entirely mitigated, and attackers must identify many more vulnerabilities to accomplish their goals.

10.1.2 The Software Compartmentalization Problem

The *compartmentalization problem* arises from attempts to decompose security-critical software into components running in different security domains: the practical application of the principle of least privilege to software. Historically, compartmentalization of TCB components such as operating system kernels and central system services has caused significant difficulty for software developers – which limits its applicability for large-scale, real-world applications, and leads to the abandonment of promising research such as 1990s *microkernel* projects. A recent resurgence of compartmentalization, applied in userspace to system software and applications such as OpenSSH [101] and Chromium [103], and more recently in our own Capsicum project [131], has been motivated by a critical security need; however it has seen success only at very coarse separation granularity due to the challenges involved. A more detailed history of work in this area can be found in Chapter 11.

On current conventional hardware, native applications must be converted to employ message passing between address spaces (or processes) rather than using a unified address space for communication, sacrificing programmability and performance by transforming a local programming problem into a distributed systems problem. As a result, large-scale compartmentalized programs are difficult to design, write, debug, maintain, and extend; this raises serious questions about correctness, performance, and most critically, security.

These problems occur because current hardware provides strong separation only at coarse granularity via rings and virtual address spaces, making the isolation of complete applications (or even multiple operating systems) a simple task, but complicates efficient and easily expressed separation between tightly coupled software components. Three closely related problems arise:

Performance is sacrificed. Creating and switching between process-based security domains is expensive due to reliance on software and hardware address-space infrastructure – such as a quickly overflowed Translation Look-aside Buffer (TLB) and large page-table sizes that can lead to massive performance degradation. Also, above an extremely low threshold, performance overhead from context switching between security domains tends to go from simply expensive to intolerable: each TLB entry is an access-control list, with each object (page) requiring multiple TLB entries, one for each authorized security domain.

High-end server CPUs typically have TLB entries in the low hundreds, and even recent network embedded devices reach the low thousands; the TLB footprint of fine-grained, compartmentalized software increases with the product of in-flight security domains and objects due to TLB aliasing, which may easily require tens or hundreds of thousands of spheres of protection. The transition to CPU multi-threading has not only failed to relieve this burden, but actively made it worse: TLBs are implemented using ternary content-addressable memory (TCAMs) or other expensive hardware lookup functions, and are often shared between hardware threads in a single core due to their expense.

Similar scalability critiques apply to page tables, the tree-oriented in-memory lookup tables used to fill TLB entries. As physical memory sizes increase, and reliance on independent virtual address spaces for separation grows, these tables also grow – competing for cache and memory space.

In comparison, physically indexed general-purpose CPU caches are several orders of magnitude larger than TLBs, scaling instead with the working set of code paths explored or the memory footprint of data actively being used. If the same data is accessed by multiple security domains, it shares data or code cache (but not TLB entries) with current CPU designs.

Programmability is sacrificed. Within a single address space, programmers can easily and efficiently share memory between program elements using pointers from a common namespace. The move to multiple processes frequently requires the adoption of a distributed programming model based on explicit message passing, making development, debugging, and testing more difficult. RPC systems and higher-level languages are able to mask some (although usually not all) of these limitations, but are poorly suited for use in TCBs – RPC systems and programming language runtimes are non-trivial, security-critical, and implemented using weaker lower-level facilities.[2]

Security is sacrificed. Current hardware is intended to provide robust shared memory communication only between mutually trusting parties, or at significant additional expense; granularity of delegation is limited and its primitives expensive, leading to programmer error and extremely limited use of granular separation. Poor programmability contributes directly to poor security properties.

10.2 Methodology

Despite half a century of research into computer systems and software design, it is clear that security remains a challenging problem – and an increasingly critical problem as computer-based technologies find ever expanding deployment in all aspects of contemporary life, from mobile communications devices to self-driving cars and medical equipment. There are many contributing factors to this problem, including the asymmetric advantage held by attackers over defenders (which cause minor engineering mistakes to lead to undue vulnerability), the difficulties in assessing – and comparing – the security of systems, and market pressures to deliver products sooner rather than in a well-engineered state. Perhaps most influential is the pressure for backward compatibility, required to allow current software stacks to run undisturbed on new generations of systems, as well as to move seamlessly across devices (and vendors), locking in least-common-denominator design choices, and preventing the deployment of more disruptive improvements that serve security.

Both the current state, and worse, the current direction, support a view that today's computer architectures (which underlie phenomenal growth of computer-based systems) are fundamentally "unfit for purpose": Rather than providing a firm foundation on which higher-level technologies can rest, they undermine attempts to build secure systems that depend on them. To address this problem, we require designs that mitigate, rather than emphasize, inevitable bugs, and offer strong and well-understood protections on which larger-scale systems can be built. Such technologies can be successful only if transparently adoptable by end users – and, ideally, also many software developers. On the other hand, the resulting improvement must be dramatic to justify adopting substantive architectural change, and while catering to short-term

[2] Through extreme discipline, a programming model can be constructed that maintains synchronized mappings of multiple address spaces, while granting different rights on memory between different processes. This leads to even greater TLB pressure and expensive context switch operations, as the layouts of address spaces must be managed using cross-address-space communication. Bittau has implemented this model via *sthread*s, an OS primitive that tightly couples UNIX processes via shared memory associated with data types – a promising separation approach constrained by the realities of current CPU design [13].

problems, must also offer a longer-term architectural vision able to support further benefit as greater investment is made.

10.2.1 Technical Objectives and Implementation

From a purely technical perspective, the aim of the CHERI project is to introduce architectural support for the principle of least privilege in order to encourage its direct utilization at all levels of the software stack. Current computer architectures make this extremely difficult as they impose substantial performance, robustness, compatibility, and complexity penalties in doing so – strongly disincentivizing adoption of such approaches in off-the-shelf system designs despite the potential to mitigate broad classes of known (and also as-yet unknown) vulnerability classes.

Low-level Trusted Computing Bases (TCBs) are typically written in memory-unsafe languages such as C and C++, which do not offer compatible or performant protection against pointer corruption, buffer overflows, or other vulnerabilities arising from that lack of safety not offered directly by the architecture. Similarly, software compartmentalization, which mitigates both low-level vulnerabilities grounded in program representation and high-level application vulnerabilities grounded in logical bugs, is poorly supported by current MMUs, leading to substantial (crippling) loss of programmability and performance as the technique is deployed.

CHERI also seeks to minimize disruption of current designs, in order to support incremental adoption with significant transparency: Ideally, CHERI could be "slid under" current software stacks (such as Apple's iOS ecosystem, or Google's Android ecosystem), allowing non-disruptive introduction, yet providing an immediate reward for adoption. This requires supporting current low-level languages such as C and C++ more safely, but also cleanly supplementing MMU-based programming models required to support current operating systems and virtualization techniques. These goals have directed many key design choices in the CHERI-MIPS ISA.

10.2.2 Hardware-Software-Formal Co-Design Methodology

Changes to the hardware-software interface are necessarily disruptive. The ISA is a "narrow waist" abstraction that allows hardware designers to pursue sophisticated optimization strategies (e.g., to exploit parallelism), while software developers can simultaneously depend on a (largely unchanging) interface to build successively larger and more complex artifacts. Stable ISAs have allowed the development of operating systems and application suites that can operate successfully on a range of systems, and that outlast the specific platforms on which they were developed.

This structure is inherently predisposed to non-disruption, as platforms that incur lower adoption costs will be preferred to those that have higher costs. However, substantive changes in underlying program representation, such as to support greater memory safety or fine-grained compartmentalization required to dramatically improve security, require changes to the ISA. We therefore aimed to:

- Iteratively explore disruptions to the ISA, projecting changes both up into the software stack including operating systems, compilers, and applications (to assess impact on compatibility and security), as well as down into microarchitecture (assessing impact on performance and viability).

- Start with a conventional and well-established 64-bit RISC ISA, rather than re-invent the wheel for general-purpose computation, to benefit from existing mature software stacks that could then be used for validation.

- Employ realistic open-source software artifacts, including the FreeBSD operating system, Clang/LLVM compiler suite, and an open-source application corpus, to ensure that experiments were run with suitable scale, complexity, performance footprint, and idiomatic use.

- Employ realistic hardware artifacts, developing multiple FPGA soft-core based processor prototypes able to validate key questions about integration with components such as the pipeline and memory hierarchy, as well as support performance validation for the full stack including software.

- Employ formal models of the ISA, to provide an executable gold model for testing, from which tests can be automatically generated, and against which theorem proving can be deployed to ensure that key properties relied on for software security actually hold.

- Pursue the hypothesis that historic capability-system models, designed to support implementation of the principle of least privilege, can be hybridized with current software approaches to support compatible and efficient fine-grained memory protection and compartmentalization.

- Take an initially purist capability-system view, incrementally adapting that model towards one able to efficiently yet safely support the majority of current software use. This approach allowed us to retain well-understood monotonicity and encapsulation properties, as well as pursue capturing notions of explicit valid provenance enforcement and intentional use not well characterized in prior capability-system work. Appropriately but uncompromisingly represented, these properties have proven to align remarkably well with current OS and language designs.

- Aim specifically to cleanly compose with conventional MMUs and MMU-based software designs by providing an in-address-space protection model, as well as be able to represent C-language pointers as capabilities.

- Support incremental adoption, allowing significant benefit to be gained through modest efforts (such as re-compiling) for selected software, while not disrupting binary-compatible execution of legacy applications. Likewise, support incremental deployment of more disruptive compartmentalization into key software through greater (but selective) investment.

- Provide primitives that offer immediate short-term benefit (e.g., invulnerability to common pointer-based exploit techniques, scalable sandboxing of libraries in key software packages), while also offering a longer-term vision for future software structure grounded in strong memory safety and fine-grained compartmentalization.

10.3 Research and Development

Table 10.1: CHERI ISA revisions and major development phases

Year(s)	Version	Description
2010-2012	ISAv1	RISC capability-system model w/64-bit MIPS Capability registers and tagged memory Guarded manipulation of registers
2012	ISAv2	Extended tagging to capability registers Capability-aware exception handling MMU-based OS with CHERI support
2014	ISAv3 [138]	Fat pointers + capabilities, compiler Instructions to optimize hybrid code Sealed capabilities, `CCall`/`CReturn`
2015	ISAv4 [141]	MMU-CHERI integration (TLB permissions) ISA support for compressed capabilities Hardware-accelerated domain switching Multicore instructions: `LL`/`SC` variants
2016	ISAv5 [142]	CHERI-128 compressed capability model Improved generated code efficiency Initial in-kernel privilege limitations
2017	ISAv6 [140]	Mature kernel privilege limitations Further generated code efficiency CHERI-x86 and CHERI-RISC-V sketches Jump-based protection-domain transition
2019	ISAv7 [139]	Architecture-neutral protection model A more complete CHERI-RISC-V elaboration Compartment IDs for side-channel resistance 64-bit capabilities for 32-bit architectures Architectural temporal memory safety CHERI Concentrate compressed capabilities

Between 2010 and 2019, six major versions of the CHERI-MIPS ISA developed a mature hybridization of conventional RISC architecture with a strong (but software-compatible) capability-system model. Key research and development milestones can be found in Figure 10.1 including major publications. The major ISA versions, with their development focuses, are described in Table 10.3. This work occurred in several major overlapping phases as aspects of the approach were proposed, refined, and stabilized through a blend of ISA design, integrated hardware and software prototyping, and validation of the combined stack.

2010–2015: Composing the MMU with a capability-system model

A key early design choice was that the capability-system model would be largely orthogonal

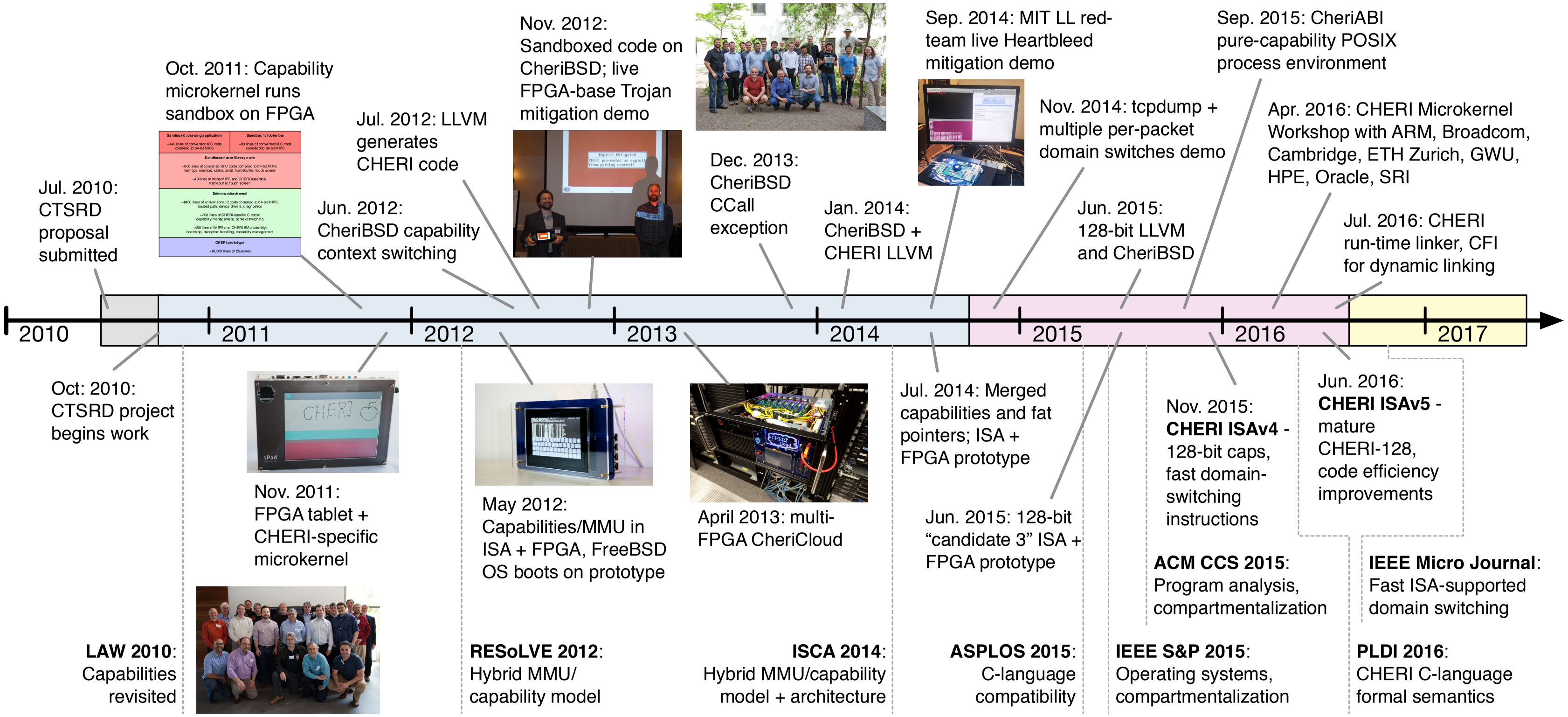
2010
2011
2012
2013
2014
2015
2016
2017
Jul. 2010: CTSRD proposal submitted
Oct. 2010: CTSRD project begins work
Oct. 2011: Capability microkernel runs sandbox on FPGA
Nov. 2011: FPGA tablet + CHERI-specific microkernel
Jul. 2012: LLVM generates CHERI code
Jun. 2012: CheriBSD capability context switching
May 2012: Capabilities/MMU in ISA + FPGA, FreeBSD OS boots on prototype
Nov. 2012: Sandboxed code on CheriBSD; live FPGA-base Trojan mitigation demo
April 2013: multi-FPGA CheriCloud
Dec. 2013: CheriBSD CCall exception
Jan. 2014: CheriBSD + CHERI LLVM
Sep. 2014: MIT LL red-team live Heartbleed mitigation demo
Jul. 2014: Merged capabilities and fat pointers; ISA + FPGA prototype
Nov. 2014: tcpdump + multiple per-packet domain switches demo
Jun. 2015: 128-bit "candidate 3" ISA + FPGA prototype
Jun. 2015: 128-bit LLVM and CheriBSD
Sep. 2015: CheriABI pure-capability POSIX process environment
Nov. 2015: CHERI ISAv4 - 128-bit caps, fast domain-switching instructions
Apr. 2016: CHERI Microkernel Workshop with ARM, Broadcom, Cambridge, ETH Zurich, GWU, HPE, Oracle, SRI
Jun. 2016: CHERI ISAv5 - mature CHERI-128, code efficiency improvements
Jul. 2016: CHERI run-time linker, CFI for dynamic linking
LAW 2010: Capabilities revisited
RESoLVE 2012: Hybrid MMU/ capability model
ISCA 2014: Hybrid MMU/capability model + architecture
ASPLOS 2015: C-language compatibility
IEEE S&P 2015: Operating systems, compartmentalization
ACM CCS 2015: Program analysis, compartmentalization
IEEE Micro Journal: Fast ISA-supported domain switching
PLDI 2016: CHERI C-language formal semantics

Figure 10.1: CHERI research and development timeline, 2010–2017

to the current MMU-based virtual-memory model, yet also compose with it cleanly [153]. We chose to place the capability-system model "before" the MMU, causing capabilities to be interpreted with respect to the virtual, rather than physical, address space. This reflected the goal of providing fine-grained memory protection and compartmentalization within address spaces – i.e., with respect to the application-programmer model of memory.

Capabilities therefore protect and implement virtual addresses dereferenced in much the same way that integer pointers are interpreted in conventional architectures. Exceptions allow controlled escape from the capability model by providing access to privileged capability registers, and execution in privileged rings grants the ability to manipulate the virtual address space, controlling the interpretation of virtual addresses embedded in capabilities.

This approach tightly integrates the capability-system model with the pipeline and register file, requiring that capabilities be first-class primitives managed by the compiler, held in registers, and so on. In order to protect capabilities in the virtual address space, we chose to physically tag them, distinguishing strongly protected pointers from ordinary data, in turn extending the implementation of physical memory, but also making that protection entirely independent from (and non-bypassable by) the MMU mechanism.

2012–2014: Composing C pointers with the capability-system mode

Another key early design choice was the goal of using capabilities to implement C-language pointers – initially discretionarily (i.e., as annotated in the language), and later ubiquitously (i.e., for all virtual addresses in a more-secure program). This required an inevitable negotiation between C-language semantics and the capability-system model, in order to ensure strong compatibility with current software [21, 79].

For example, C embeds a strong notion that pointers point within buffers. This requires that CHERI capabilities distinguish the notion of current virtual address from the bounds of the containing buffer – while also still providing strong integrity protection to the virtual address. This led us to compose fat-pointer [53, 87, 90] and capability semantics as the capability-system model evolved.

Similarly, we wished to allow all pointers to be represented as capabilities – including those embedded within other data structures – leading naturally to a choice to mandatorily tag pointers in memory. A less obvious implication of this approach is that operations such as memory copying must be capability-oblivious, maintaining the tag across pointer-propagating memory operations, requiring that data and capabilities not only be intermingled in memory, but also in register representation. Capability registers are therefore also tagged, allowing them to hold data or capabilities, preserving provenance transparently.

As part of this work, we also assisted with the development of new formal semantics for the C programming language, ensuring that we met the practical requirements of C programs, but also assisting in formalizing the protection properties we offer (e.g., strong protection of provenance validity grounded in an implied pointer provenance model in C).

CHERI should be viewed as providing primitives to support strong C-language pointer protection, rather than as directly implementing that protection: it is the responsibility of the compiler (and also operating system and runtime) to employ capabilities to enforce protections where desired – whether by specific memory type, based on language annotations, or more universally. The compiler can also perform analyses to trade off source-code and binary compatibility, enforcing protection opportunistically in responding to various potential policies on tolerance to disruption.

2014–2015: Fine-grained compartmentalization

A key goal of our approach was to differentiate virtualization (requiring table-based lookups, and already implemented by the MMU) from protection (now implemented as a constant-time extension to the pointer primitive), which would avoid table-oriented overheads being imposed on protection. This applies to C-language protection, but also to the implementation of higher-level security constructs such as compartmentalization [146, 143].

Compartmentalization depends on two underlying elements: strong isolation and controlled communication bridging that isolation. Underlying monotonicity in capabilities – i.e., that a delegated reference to a set of rights cannot be broadened to include additional rights – directly supports the construction of confined components within address spaces. Using this approach, we can place code in execution with only limited access to virtual memory, constructing "sandboxes" (and other more complex structures) within conventional processes. The CHERI exception model permits transition to a more privileged component – e.g., the operating-system kernel or language runtime – allowing the second foundation, controlled communication, to be implemented.

Compartmentalization is facilitated by further extensions to the capability model, including a notion of "sealed" (or encapsulated capabilities). In CHERI, this is implemented as a software-defined capability: one that has no hardware interpretation (i.e., cannot be dereferenced), and also strong encapsulation (i.e., whose fields are immutable). Other aspects of the model include a type mechanism allowing sealed code and data capabilities to be inextricably linked; pairs of sealed code capabilities and data capabilities can then be used to efficiently describe protection domains via an object-capability model. We provide some hardware assistance for protection-domain switching, providing straightforward parallel implementation of key checks, but leave the implementation of higher-level aspects of switching to the software implementation.

Here, as with C-language integration, it is critical that CHERI provide a general-purpose mechanism rather than enforce a specific policy: the sealed capability primitive can be used in a broad variety of ways to implement various compartmentalization models with a range of implied communication and event models for software. We have experimented with several such models, including a protection-domain crossing primitive modeled on a simple (but now strongly protected) function call, and also on asynchronous message passing. Our key performance goal was fixed (low) overhead similar to a function call, avoiding overheads that scale with quantity of memory shared (e.g., as is the case with table-oriented memory sharing configured using the MMU).

2015–2017: Architectural and microarchitectural efficiency

Side-by-side with development of a mature capability-based architectural model, we also explored the implications on performance. This led to iterative refinement of the ISA to improve generated code, but also substantive efforts to ensure that there was an efficient in-memory representation of capabilities, as well as microarchitectural implementations of key instructions.

A key goal was to maintain the principle of a load-store architecture by avoiding combining computations with memory accesses – already embodied by both historic and contemporary RISC architectures. While pointers are no longer conflated with integer values, a natural composition of the capability model and ISA maintains that structural goal without difficulty.

One important effort lay in the reduction from a 256-bit capability (capturing the requirements of software for 64-bit pointer, 64-bit upper bound, and 64-bit lower bound, as well as additional metadata such as permissions) to a 128-bit compressed representation. We took substantial inspiration from published work in pointer compression [62], but found that our C-language compatibility requirements imposed a quite different underlying model and representation. For example, it is strictly necessary to support the common C-language idiom of permitting out-of-bounds pointers (but not dereference), which had been precluded by many proposed schemes [21, 79, 28]. Similarly, the need to support sealed capabilities led to efforts to characterize the tradeoff between the type space (the number of unique classes that can be in execution in a CHERI address space) and bounds precision (the alignment requirements imposed on sealed references).

Another significant effort lay in providing in-memory tags, which are not directly supported by current DRAM layouts [54, 55]. In our initial implementation, we relied on a flat tag table (supported by a dedicated tag cache). This imposed a uniform (and quite high) overhead in additional DRAM accesses across all memory of roughly 10%. We have developed new microarchitectural techniques to improve emulated tag performance, based on a hierarchical table exploiting sparse use of pointers in memory, to reduce this overhead to $< 2\%$ even with very high pointer density (e.g., in language runtimes).

2016–2017: Kernel Compartmentalization

Our initial design focus was on supporting fine-grained memory protection within the user virtual address space, and implicitly, also compartmentalization. Beyond an initial microkernel brought up to validate early capability model variants, kernel prototypes through much of our project have eschewed use of capability-aware code in the kernel due to limitations of the compiler, but also because of a focus on large userspace TCBs such as compression libraries, language runtimes, web browsers, and so on, which are key attack surfaces.

We have more recently returned to in-kernel memory protection and compartmentalization, where the CHERI model in general carries through without change – code executing in the kernel is not fundamentally different from code executing in userspace. The key exception is a set of management instructions available to the kernel, able to manipulate the MMU (and hence the interpretation of capabilities), as well as control features such as interrupt delivery and exception handling. We are now extending CHERI to allow the capability mechanism to control access to these features so that code can be compartmentalized within the kernel. We are also pursuing changes to the exception-based domain-transition mechanism used in earlier ISA revisions that shift towards a jump-based model, which will avoid exception-related overheads in the microarchitecture.

10.3.1 CHERI ISAv7: Beyond MIPS, Temporal Safety, and Efficiency

As we wrap up work on CHERI ISAv7, we are looking beyond the 64-bit MIPS ISA on which we based our hardware-software co-design effort towards further ISAs. These range from the still-developing open-source RISC-V ISA (which strongly resembles the MIPS ISA and hence to which most CHERI ideas will apply with minor translation) to the widely used Intel x86-64 instruction set (which is quite far from the RISC foundations in which we have developed

CHERI). This exploration has allowed us to derive a more general CHERI protection model from our work, rather than seeing CHERI as simply an extension to MIPS. We have focused on developing portable software-facing primitives and abstractions potentially supported by a variety of architectural expressions. We take some inspiration from the diverse range of MMU semantics and interfaces providing a common virtual-memory abstraction, and process model, across a broad range of architectures.

We have also turned our attention to temporal memory safety. CHERI's tagging features have supported deterministic sweeping revocation and garbage collection from early versions of the ISA. However, these require substantial dynamic memory overhead. In CHERI ISAv7, we have introduced instructions to allow more efficient scanning for tagged values in memory using non-temporal tag-loading instructions, as well as subset-testing instructions. These accompany a research effort to explore efficient system-software implementation of capability tracking (e.g., through capability-dirty bits in the page table), further reducing revocation costs. We have included these new instructions as experimental features in this ISA version.

CHERI ISAv7 includes a number of both production and experimental features to improve code density, including immediate-extended versions of instructions that are frequently used with compile-time constants, such as bounds-setting for stack allocations and pointer arithmetic. These have resulted in substantial improvements for a variety of workloads including language runtimes. CHERI Concentrate, our revised capability compression format, makes substantially better use of the available space within capabilities, offering greater precision for a lower bit investment.

New versions of the ISA specification also explore in much greater detail how architecture protection can be exploited by operating systems and compilers to reinforce program structure and mitigate vulnerabilities.

10.4 A Hybrid Capability-System Architecture

Unlike past research into capability systems, CHERI allows traditional address-space separation, implemented using a memory management unit (MMU), to coexist with granular decomposition of software within each address space. Similarly, we have aimed to model CHERI capability behavior not only on strong capability semantics (e.g., monotonicity), but also to be compatible with C-language pointer semantics. As a result, fine-grained memory protection and compartmentalization can be applied selectively throughout existing software stacks to provide an incremental software migration path. We envision early deployment of CHERI extensions in selected components of the TCB's software stack: separation kernels, operating system kernels, programming language runtimes, sensitive libraries such as those involved in data compression or encryption, and network applications such as web browsers and web servers.

CHERI addresses current limitations on memory protection and compartmentalization by extending virtual memory-based separation with hardware-enforced, fine-grained protection within address spaces. Granular memory protection mitigates a broad range of previously exploitable bugs by coercing common memory-related failures into exceptions that can be handled by the application or operating system, rather than yielding control to the attacker. The

CHERI approach also restores a single address-space programming model for compartmentalized (sandboxed) software, facilitating efficient, programmable, and robust separation through the capability model.

We have selected this specific composition of traditional virtual memory with an in-address-space security model to facilitate technology transition: in CHERI, existing C-based software can continue to run within processes, and even integrate with capability-enhanced software within a single process, to provide improved robustness for selected software components – and perhaps over time, all software components. For example, a sensitive library (perhaps used for image processing) might employ capability features while executing as part of a CHERI-unaware web browser. Likewise, a CHERI-enabled application can sandbox and instantiate multiple copies of unmodified libraries, to efficiently and easily gate access to the rest of application memory of the host execution environment.

10.5 A Long-Term Capability-System Vision

While we have modeled CHERI as a hybrid capability-system architecture, and in particular described a well-defined and practical composition with MMU-based designs, CHERI can also support more "pure" capability-oriented hardware and software designs. At one extreme in this spectrum, we have begun early experimentation with an MMU-free processor design offering solely CHERI-based protection for software use. We are able to layer a CHERI-specific microkernel over this design, which executes all programs within a single address-space object-capability model. This approach might be appropriate to microcontroller-scale systems, to avoid the cost of an MMU, and in which conventional operating systems might be inappropriate. The approach might also be appropriate to very large-scale systems, in which an MMU is unable to provide granular protection and isolation due to TLB pressure requiring a shift to very large page sizes.

However, in retaining our primary focus on a hybridization between MMU- and capability-based approaches, software designs can live at a variety of points in a spectrum between pure MMU-based and solely CHERI-based models. A CHERI-based microkernel might be used, for example, within a conventional operating-system kernel to compartmentalize the kernel – while retaining an MMU-based process model. A CHERI-based microkernel might similarly be used within an MMU-based process to compartmentalize a large application. Finally, the CHERI-based microkernel might be used to host solely CHERI-based software, much as in an MMU-less processor design, leaving the MMU dormant, or restricted to specific uses such as full-system virtualization – a task for which the MMU is particularly well suited.

10.6 Threat Model

CHERI protections constrain code "in execution" and allow fine-grained management of privilege within a framework for controlled separation and communication. Code in execution can represent the focus of many potentially malicious parties: subversion of legitimate code in violation of security policies, injection of malicious code via back doors, Trojan horses, and malware, and also denial-of-service attacks. CHERI's fine-grained memory protection mitigates many common attack techniques by implementing bounds and permission checks, reducing opportunities for the conflation of code and data, corruption of control flow, and also catches many common exploitable programmer bugs; compartmentalization constrains successful attacks via pervasive observance of the principle of least privilege.

Physical attacks on CHERI-based systems are explicitly excluded from our threat model, although CHERI CPUs might easily be used in the context of tamper-evident or tamper-resistant systems. Similarly, no special steps have been taken in our design to counter undesired leakage of electromagnetic emanations and certain other side channels such as acoustic inferences: we take for granted the presence of an electronic foundation on which CHERI can run. CHERI will provide a supportive framework for a broad variety of security-sensitive activities; while not itself a distributed system, CHERI could form a sound foundation for various forms of distributed trustworthiness.

CHERI is an ISA-level protection model that does not address increasingly important CPU- or bus-level covert and side-channel attacks, relying on the micro-architecture to limit implicit data flows. In some sense, CHERI in fact increases exposure: the greater the offers of protection within a system, the greater the potential impact of unauthorized communication channels. As such, we hope side-channel attacks are a topic that we will be able to explore in future work. Overall, we believe that our threat model is realistic and will lead to systems that can be substantially more trustworthy than today's commodity systems – while recognizing that ISA-level protections must be used in concert with other protections suitable to different threat models.

10.7 Formal Methodology

Throughout this project, we apply formal semantics and reasoning techniques to help avoid system vulnerabilities. We are (judiciously) applying formal methodology in five areas:

1. Early in the project, we developed a formal semantics for the CHERI-MIPS ISA described in SRI's Prototype Verification System (PVS) – an automated theorem-proving and model-checking toolchain – which can be used to verify the expressibility of the ISA, but also to prove properties of critical code. For example, we are interested in proving the correctness of software-based address-space management and domain transitions. We are likewise able to automatically generate ISA-level test suites from formal descriptions of instructions, which are applied directly to our hardware implementation.

2. We developed extensions to the BSV compiler to export an HDL description to SRI's PVS and SAL model checker. We also developed a new tool (Smten) for efficient SMT (Satisfiability Modulo Theories) modeling of designs (using SRI's Yices), and another tool for automated extraction of key properties from larger designs in the BSV language, both of which greatly simplify formal analysis.

3. We then developed more complete CHERI-MIPS ISA models, incorporating both MIPS and CHERI instructions, first using the L3 and then the Sail instruction-set description languages (both of which support automatic generation of executable emulators from

formal definitions). We have used these as the "golden model" of instruction behavior, against which our test suite is validated, software implementations can be tested in order to generate traces of correct processor execution, and so on. We have used the L3 and Sail models to identify a number of bugs in multiple hardware implementations of CHERI-MIPS, as well as to discover software dependences on undefined instruction-set behavior.

4. We have used these L3 and Sail models also as a basis for mechanised proof of key architectural security properties. L3 and Sail support automatic generation of versions of the models in the definition languages of (variously) the HOL4, Isabelle, and Coq theorem provers. Key architectural verification goals including proving not just low-level properties, such as the monotonicity of each individual instruction and properties of the CHERI capability compression schemes, but also higher-level goals such as compartment monotonicity, in which arbitrary code sequences isolated within a compartment are unable to construct additional rights beyond those reachable either directly via the register file or indirectly via loadable capabilities. We have proven a number of such properties about the CHERI-MIPS ISA, to be documented in future papers and reports.

5. From Sail, we also automatically generate SMT problems, which we have used to check properties of our capability compression schemes.

6. We have explored how CHERI impacts a formal specification of C-language semantics, improving a number of aspects of our C-language compatibility (e.g., as relates to conformant handling of the `intptr_t` type).

10.8 Protection Model and Architecture

As our work on CHERI has proceeded, we have transitioned from a view in which CHERI is an ISA extension to 64-bit MIPS to one in which CHERI is a general protection model that can be expressed through a variety of approaches and mappings into multiple underlying ISAs. This report describes a software-facing protection model (Chapter 2) focused on operating systems and compilers, specific mapping into the 64-bit MIPS ISA for the purposes of experimentation and evaluation (Chapters 3, 4 and 7), and architectural sketches for potential integration into other ISAs (Chapters 5 and 6). However, we have taken a "ground-up" approach utilizing hardware-software co-design to ensure that at least one complete concrete mapping exists that satisfies the practical engineering requirements of architecture, microarchitecture, compiler, operating system, and applications, and hence define a specific CHERI-MIPS ISA that embodies those goals.

Our selection of RISC as a foundation for the CHERI capability extensions is motivated by two factors. First, simple instruction set architectures are easier to reason about, extend, and implement. Second, RISC architectures (such as ARM and MIPS) are widely used in network embedded and mobile device systems such as firewalls, routers, smart phones, and tablets – markets with the perceived flexibility to adopt new CPU facilities, and also an immediate and pressing need for improved security. CHERI's new security primitives would also be useful in workstation and server environments, which face similar security challenges.

In its current incarnation, we have prototyped CHERI as an extension to the 64-bit MIPS ISA. However, our approach – and more generally the CHERI protection model – is intended to easily support other similar ISAs, such as 64-bit ARM and 64-bit RISC-V. The design principles would also apply to other non-RISC ISAs, such as 32-bit and 64-bit Intel and AMD, but require significantly more adaptation work, as well as careful consideration of the implications of the diverse set of CPU features found in more CISC-like architectures.

It is not impossible to imagine pure-software implementations of the CHERI protection model – not least, because we use these daily in our work through both cycle-accurate processor simulations, and a higher-performance but less microarchitecturally realistic Qemu implementation. Further, compiler-oriented approaches employing a blend of static checking and dynamic enforcement could also approximate or implement CHERI protection semantics (e.g., along the lines of software fault isolation techniques [121] or Google Native Client (NaCl) [158]). We do, however, hypothesize that these implementations would be difficult to accomplish without hardware assistance: for example, continuous checking of program-counter and default data capability bounds, as well as atomic clearing of tags for in-memory pointers during arbitrary memory writes might come at substantial expense in software, yet being "free" in supporting hardware.

10.9 Hardware and Software Prototypes

As a central part of this research, we have developed reference prototypes of the CHERI ISA via several CHERI processor designs. These prototypes allow us to explore, validate, evaluate, and demonstrate the CHERI approach through realistic hardware properties and real-world software stacks. A detailed description of the current prototypes, both from architectural and practical use perspectives, may be found in our companion papers and technical reports, described in Section 1.8.

Our first prototype (CHERI1) is based on Cambridge's MAMBA research processor, and is a single-threaded, multi-core implementation intended to allow us to explore ISA design trade-offs with moderate microarchitectural realism. This prototype is implemented in the BSV HDL, a high-level functional programming language for hardware design. CHERI1 is a pipelined baseline processor implementing the 64-bit MIPS ISA, and incorporates an initial prototype of the CHERI-MIPS capability coprocessor that includes capability registers and a basic capability instruction set.

Exploring, and iterating over, a substantial instruction-set design space has been considerably eased by our use of the Bluespec SystemVerilog [14] (BSV) Hardware Description Language (the BSV HDL) in prototyping. BSV has allowed rapid redesigns as our understanding of architectural, microarchitectural, and software requirements evolved – resulting from its use of modular abstractions, encapsulation, and hierarchicalization.

Using the BSV hardware specification language and its Bluespec SystemVerilog, we are able to run the CPU in simulation, and synthesize the CHERI design to execute in field-programmable gate arrays (FPGAs). In our development work, we are targeting an Altera FPGAs on Terasic development boards. However, in our companion MRC2 project we have also targeted CHERI at the second-generation NetFPGA 10G and SUME research and teaching boards, which we hope to use in ongoing research into datacenter network fabrics. That work includes the development of Blueswitch, a BSV language implementation of an OpenFlow switch that can operate as a tightly coupled CHERI coprocessor. In the future, should it become desirable, we will be able to construct an ASIC design from the same BSV specification. We have released the CHERI soft core as *open-source hardware*, making it available for more widespread use in research. This should allow others, especially in the research community, to reproduce and extend our results.

We have also developed a second prototype (CHERI2), which is compatible with CHERI1 but has additional CPU features including fine-grained multi-threading. We have used this as a platform for early exploration of the synergy between compartmentalization and parallelism in multi-threaded processor designs. CHERI2 also employs a more stylized form of the BSV language that is intended to considerably enhance our formal analysis of the hardware architecture.

In addition to the CHERI1 and CHERI2 implementations in BSV, we have implemented an executable model of CHERI in the L3 ISA modeling language [37], and a high-performance emulation in QEMU. The L3 and QEMU implementations support 256-bit capabilities and multiple forms of 128-bit capabilities including compressed capabilities and "magic" uncompressed capabilities, which are identical to 256-bit capabilities except for size. While intended primarily for formal modeling and use as a test oracle, we have also found the L3 ISA modeling language invaluable in practical design-space exploration.

As the CHERI security model is necessarily a hardware-software model, we have also performed substantial experimentation with software stacks targeting the CHERI-MIPS ISA. We have created an adaptation of the commodity open-source FreeBSD operating system, CheriBSD, that supports a wide variety of peripherals on the Terasic tPad and DE4 FPGA development boards; we use these boards in both mobile tablet-style and network configurations. CheriBSD is able to manage the capability coprocessor, maintain additional thread state for capability-aware user applications, expose both hybrid and pure-capability system-call interfaces, and, increasingly, to use capability features for self protection against malicious userspace software. CheriBSD also implements exception-handler support for object-capability invocation, signal delivery when protection faults occur (allowing language runtimes to catch and handle protection violations), and error recovery for in-process sandboxes. We have adapted the Clang and LLVM compiler suite to allow language-level annotations in C to direct capability use in a hybrid ABI. Additionally, we have implemented a pure-capability compilation mode where all C pointers are capabilities. Using a mix of hybrid and pure-capability ABIs, we have developed a number of capability-enhanced applications to demonstrate fine-grained memory protection and in-process compartmentalization – to explore security, performance, and programmability tradeoffs.

Chapter 11

Historical Context and Related Work

As with many aspects of contemporary computer and operating-system design, many of the origins of operating-system security can be found at the world's leading research universities – especially the Massachusetts Institute of Technology (MIT), the University of Cambridge, and Carnegie Mellon University. MIT's Project MAC, which began with MIT's Compatible Time Sharing System (CTSS) [23], and continued over the next decade with MIT's Multics project (joint with Honeywell, and originally Bell Labs), described many central tenets of computer security [24, 44]. Dennis and Van Horn's 1965 *Programming Semantics for Multiprogrammed Computations* [30] laid out principled hardware and software approaches to concurrency, object naming, and security for multi-programmed computer systems – or, as they are known today, multi-tasking and multi-user computer systems. Multics implemented a coherent, unified architecture for processes, virtual memory, and protection, integrating new ideas such as *capabilities*, unforgeable tokens of authority, and *principals*, the end users with whom authentication takes place and to whom resources are accounted [110].

In 1975, Saltzer and Schroeder surveyed the rapidly expanding vocabulary of computer security in *The Protection of Information in Computer Systems* [111]. They enumerated design principles such as the *principle of least privilege* (which demands that computations run with only the privileges they require) and the core security goals of protecting *confidentiality*, *integrity*, and *availability*. The tension between fault tolerance and security (a recurring debate in systems literature) saw its initial analysis in Lampson's 1974 *Redundancy and Robustness in Memory Protection* [63], which considered ways in which hardware memory protection addressed accidental and intentional types of failure: e.g., if it is not reliable, it will not be secure, and if it is not secure, it will not be reliable! Intriguingly, recent work by Nancy Leveson and William Young has unified security and human safety as overarching emergent system properties [67], and allows the threat model to fall out of the top-down analysis, rather than driving it. This work in some sense unifies a long thread of work that considers trustworthiness as a property encompassing security, integrity, reliability, survivability, human safety, and so on (e.g., [91, 93], among others).

The Security Research community also blossomed outside of MIT: Wulf's HYDRA operating system at Carnegie Mellon University (CMU) [155, 22], Needham and Wilkes' CAP Computer at Cambridge [150], SRI's Provably Secure Operating System (PSOS) [36, 93] hardware-software co-design that included strongly typed object capabilities, Rushby's security kernels supported by formal methods at Newcastle [109], and Lampson's work on formal models of security protection at the Berkeley Computer Corporation all explored the structure of operating-system access control, and especially the application of capabilities to the protection problem [64, 65]. Another critical offshoot from the Multics project was Ritchie and Thompson's UNIX operating system at Bell Labs, which simplified concepts from Multics, and became the basis for countless directly and indirectly derived products such as today's Solaris, FreeBSD, Mac OS X, and Linux operating systems [107].

The creation of secure software went hand in hand with analysis of security flaws: Anderson's 1972 US Air Force *Computer Security Technology Planning Study* not only defined new security structures, such as the *reference monitor*, but also analyzed potential attack methodologies such as Trojan horses and inference attacks [5]. Karger and Schell's 1974 report on a security analysis of the Multics system similarly demonstrated a variety of attacks that bypass hardware and OS protection [58]. In 1978, Bisbey and Hollingworth's *Protection Analysis: Project final report* at ISI identified common patterns of security vulnerability in operating system design, such as race conditions and incorrectly validated arguments at security boundaries [12]. Adversarial analysis of system security remains as critical to the success of security research as principled engineering and formal methods.

Almost fifty years of research have explored these and other concepts in great detail, bringing new contributions in hardware, software, language design, and formal methods, as well as networking and cryptography technologies that transform the context of operating system security. However, the themes identified in those early years remain topical and highly influential, structuring current thinking about systems design.

Over the next few sections, we consider three closely related ideas that directly influence our thinking for CTSRD: capability security, microkernel OS design, and language-based constraints. These apparently disparate areas of research are linked by a duality, observed by Jim Morris in 1973, between the enforcement of data types and safety goals in programming languages on one hand, and the hardware and software protection techniques explored in operating systems [85] on the other hand. Each of these approaches blends a combination of limits defined by static analysis (perhaps at compile-time), limits on expression on the execution substrate (such as what programming constructs can even be represented), and dynamically enforced policy that generates runtime exceptions (often driven by the need for configurable policy and labeling not known until the moment of access). Different systems make different uses of these techniques, affecting expressibility, performance, and assurance.

11.1 Capability Systems

Throughout the 1970s and 1980s, high-assurance systems were expected to employ a capability-oriented design that would map program structure and security policy into hardware enforcement; for example, Lampson's BCC design exploited this linkage to approximate least privilege [64, 65].

Systems such as the CAP Computer at Cambridge [150] and Ackerman's DEC PDP-1 architecture at MIT [4] attempted to realize this vision through embedding notions of capabilities in the memory management unit of the CPU, an approach described by Dennis and Van

Horn [30]. Levy provides a detailed exploration of segment- and capability-oriented computer system design through the mid-1980s in *Capability-Based Computer Systems* [68].

Ackerman's architecture [4] in particular seems to have been the first to realize the importance of allowing subsystems to construct multiple, differentiated entry capabilities, to correspond to different permitted requests (e.g., invoking different methods on different logical targets within the same subsystem). A six-bit field, the "transmitted word," was provided within the entry capability, immune from influence of the bearer but made available to the subsystem itself on entry. Similar facilities have been found in almost all subsequent capability systems. CHERI lacks such a field within its capabilities; however, the `CCall` mechanism can be used to similar effect (recall Section 2.3.8).

11.2 Microkernels

Denning has argued that the failures of capability hardware projects were classic failures of large systems projects, an underestimation of the complexity and cost of reworking an entire system design, rather than fundamental failures of the capability model [29]. However, the benefit of hindsight suggests that the earlier demise of hardware capability systems was a result of three related developments in systems research: microkernel OS design, a related interest from the security research community in security kernel design, and Patterson and Sequin's Reduced Instruction-Set Computers (RISC) [99].

However, with a transition from complex instruction set computers (CISC) to reduced instruction set computers (RISC), and a shift away from microcode toward operating system implementation of complex CPU functionality, the attention of security researchers turned to microkernels.

Carnegie Mellon's HYDRA [22, 156] embodied this approach, in which microkernel message passing between separate tasks stood in for hardware-assisted security domain crossings at capability invocation. HYDRA developed a number of ideas, including the relationship between capabilities and object references, refined the *object-capability* paradigm, and further pursued the separation of policy and mechanism.[1] Jones and Wulf argue through the HYDRA design that the capability model allows the representation of a broad range of system policies as a result of integration with the OS object model, which in turn facilitates interposition as a means of imposing policies on object access [56].

Successors to HYDRA at CMU include Accent and Mach [102, 3], both microkernel systems intended to explore the decomposition of a large and decidedly un-robust operating system kernel. In microkernel designs, traditional OS services, such as the file system, are migrated out of ring 0 and into user processes, improving debuggability and independence of failure modes. They are also based on mapping of capabilities as object references into IPC pipes (*ports*), in which messages on ports represent methods on objects. This shift in operating system design went hand in hand with a related analysis in the security community: Lampson's model for capability security was, in fact, based on pure message passing between isolated processes [65]. This further aligned with proposals by Andrews [6] and Rushby [109] for a *security kernel*,

[1] Miller has expanded on the object-capability philosophy in considerable depth in his 2006 PhD dissertation, *Robust composition: towards a unified approach to access control and concurrency control* [81]

whose responsibility lies solely in maintaining isolation, rather than the provision of higher-level services such as file systems. Unfortunately, the shift to message passing also invalidated Fabry's semantic argument for capability systems, namely, that by offering a single namespace shared by all protection domains, the distributed system programming problem could be avoided [35].

A panel at the 1974 National Computer Conference and Exposition (AFIPS) chaired by Lipner brought the design goals and choices for microkernels and security kernels clearly into focus: microkernel developers sought to provide flexible platforms for OS research with an eye towards protection, while security kernel developers aimed for a high assurance platform for separation, supported by hardware, software, and formal methods [70].

The notion that the microkernel, rather than the hardware, is responsible for implementing the protection semantics of capabilities also aligned well with the simultaneous research (and successful technology transfer) of RISC designs, which eschewed microcode by shifting complexity to the compiler and operating system. Without microcode, the complex C-list peregrinations of CAP's capability unit, and protection domain transitions found in other capability-based systems, become less feasible in hardware. Virtual memory designs based on fixed-size pages and simple semantics have since been standardized throughout the industry.

Security kernel designs, which combine a minimal kernel focused entirely on correctly implementing protection, and rigorous application of formal methods, formed the foundation for several secure OS projects during the 1970s. Schiller's security kernel for the PDP-11/45 [112] and Neumann's Provably Secure Operating System [39] design study were ground-up operating system designs based soundly in formal methodology.[2] In contrast, Schroeder's MLS kernel design for Multics [113], the DoD Kernelized Secure Operating System (KSOS) [76], and Bruce Walker's UCLA UNIX Security Kernel [122] attempted to slide MLS kernels underneath existing Multics and UNIX system designs. Steve Walker's 1980 survey of the state of the art in trusted operating systems provides a summary of the goals and designs of these high-assurance security kernel designs [123].

The advent of CMU's Mach microkernel triggered a wave of new research into security kernels. TIS's Trusted Mach (TMach) project extended Mach to include mandatory access control, relying on enforcement in the microkernel and a small number of security-related servers to implement the TCB to accomplish sufficient assurance for a TCSEC B3 evaluation [15]. Secure Computing Corporation (SCC) and the National Security Agency (NSA) adapted PSOS's type enforcement from LoCK (LOgical Coprocessor Kernel) for use in a new Distributed Trusted Mach (DTMach) prototype, which built on the TMach approach while adding new flexibility [114]. DTMach, adopting ideas from HYDRA, separates mechanism (in the microkernel) from policy (implemented in a userspace security server) via a new reference monitor framework, FLASK [119]. A significant focus of the FLASK work was performance: an access vector cache is responsible for caching access control decisions throughout the OS to avoid costly up-calls and message passing (with associated context switches) to the security server. NSA and SCC eventually migrated FLASK to the FLUX microkernel developed by the University of Utah in the search for improved performance. Invigorated by the rise of microkernels and their congruence with security kernels, this flurry of operating system security research

[2] PSOS's ground-up design included ground-up hardware, whereas Schiller's design revised only the software stack.

also faced the limitations (and eventual rejection) of the microkernel approach by the computer industry – which perceived the performance overheads as too great.

Microkernels and mandatory access control have seen another experimental composition in the form of Decentralized Information Flow Control (DIFC). This model, proposed by Myers, allows applications to assign information flow labels to OS-provided objects, such as communication channels, which are propagated and enforced by a blend of static analysis and runtime OS enforcement, implementing policies such as taint tracking [86] – effectively, a composition of mandatory access control and capabilities in service to application security. This approach is embodied by Efstathopoulos et al.'s Asbestos [34] and Zeldovich et al.'s Histar [160] research operating systems.

Despite the decline of both hardware-oriented and microkernel capability system design, capability models continue to interest both research and industry. Inspired by the proprietary KeyKOS system [48], Shapiro's EROS [116] (now CapROS) and Coyotos [115] continued the investigation of higher-assurance software capability designs, and seL4 [60], a formally verified, capability-oriented microkernel, has also continued along this avenue. General-purpose systems also have adopted elements of the microkernel capability design philosophy, such as Apple's Mac OS X [7] (which uses Mach interprocess communication (IPC) objects as capabilities) and Cambridge's Capsicum [131] research project (which attempts to blend capability-oriented design with UNIX).

More influentially, Morris's suggestion of capabilities at the programming language level has seen widespread deployment. Gosling and Gong's Java security model blends language-level type safety with a capability-based virtual machine [42, 41]. Java maps language-level constructs (such as object member and method protections) into execution constraints enforced by a combination of a pre-execution bytecode verification and expression constraints in the bytecode itself. Java has seen extensive deployment in containing potentially (and actually) malicious code in the web browser environment. Miller's development of a capability-oriented E language [81], Wagner's Joe-E capability-safe subset of Java [80], and Miller's Caja capability-safe subset of JavaScript continue a language-level exploration of capability security [82].

11.3 Language and Runtime Approaches

Direct reliance on hardware for enforcement (which is central to both historic and current systems) is not the only approach to isolation enforcement. The notion that limits on expressibility in a programming language can be used to enforce security properties is frequently deployed in contemporary systems to supplement coarse and high-overhead operating-system process models. Two techniques are widely used: virtual-machine instruction sets (or perhaps physical machine instruction subsets) with limited expressibility, and more expressive languages or instruction sets combined with type systems and formal verification techniques.

The Berkeley Packet Filter (BPF) is one of the most frequently cited examples of the virtual machine approach: user processes upload pattern matching programs to the kernel to avoid data copying and context switching when sniffing network packet data [75]. These programs are expressed in a limited packet-filtering virtual-machine instruction set capable of expressing common constructs, such as accumulators, conditional forward jumps, and comparisons, but are incapable of expressing arbitrary pointer arithmetic that could allow escape from confinement, or control structures such as loops that might lead to unbounded execution time. Similar approaches have been used via the type-safe Modula 3 programming language in SPIN [11], and the DTrace instrumentation tool that, like BPF, uses a narrow virtual instruction set to implement the D language [17].

Google's Native Client (NaCl) model edges towards a verification-oriented approach, in which programs must be implemented using a 'safe' (and easily verified) subset of the x86 or ARM instruction sets, which would allow confinement properties to be validated [159]. NaCl is closely related to Software Fault Isolation (SFI) [121], in which safety properties of machine code are enforced through instrumentation to ensure no unsafe access, and Proof-Carrying Code (PCC), in which the safe properties of code are demonstrated through attached and easily verifiable proofs [89]. As mentioned in the previous section, the Java Virtual Machine (JVM) model is similar; it combines runtime execution constraints of a restricted, capability-oriented bytecode with a static verifier run over Java classes before they can be loaded into the execution environment; this ensures that only safe accesses have been expressed. C subsets, such as Cyclone [53], and type-safe languages such as Ruby [108], offer similar safety guarantees, which can be leveraged to provide security confinement of potentially malicious code without hardware support.

These techniques offer a variety of trade-offs relative to CPU enforcement of the process model. For example, some (BPF, D) limit expressibility that may prevent potentially useful constructs from being used, such as loops bounded by invariants rather than instruction limits; in doing so, this can typically impose potentially significant performance overhead. Systems such as FreeBSD often support just-in-time compilers (JITs) that convert less efficient virtual-machine bytecode into native code subject to similar constraints, addressing performance but not expressibility concerns [77].

Systems like PCC that rely on proof techniques have had limited impact in industry, and often align poorly with widely deployed programming languages (such as C) that make formal reasoning difficult. Type-safe languages have gained significant ground over the last decade, with widespread use of JavaScript and increasing use of functional languages such as OCaML [106]; they offer many of the performance benefits with improved expressibility, yet have had little impact on operating system implementations. However, an interesting twist on this view is described by Wong in Gazelle, in which the observation is made that a web browser is effectively an operating system by virtue of hosting significant applications and enforcing confinement between different applications [124]. Web browsers frequently incorporate many of these techniques including Java Virtual Machines and a JavaScript interpreter.

11.4 Bounds Checking and Fat Pointers

In contrast to prior capability systems, a key design goal for CHERI was to support mapping C-language pointers into capabilities. In earlier prototypes, we did this solely through base and bounds fields within capabilities, which worked well but required substantial changes to existing C software that often contained programming idioms that violated monotonic rights decrease for pointers. In later versions of the ISA, we adopt ideas from the C fat-pointer literature, which differentiate the idea of a delegated region from a current pointer: while the base and bounds are subject to guarded manipulation rules, we allow the offset to float within and beyond the delegated region. Only on dereference are protections enforced, allowing a variety of imaginative pointer operations to be supported. Many of these ideas originate with the type-safe C dialect Cyclone [53], and see increasing adaptation to off-the-shelf C programs with work such as Softbound [87], Hardbound [31], and CCured [90]. This flexibility permits a much broader range of common C idiom to be mapped into the capability-based memory-protection model.

11.5 Capabilities In Hardware

Capability systems manifested as computing hardware must have some mechanism to distinguish capabilities from non-capability data or, equivalently, for determining the semantic type assigned to bits being accessed by the instruction stream. Broadly speaking, two approaches have emerged: making the type distinction *intrinsically associated* with the bits in question or associating the type with the *accees path* taken to those bits.

Systems choosing the former option are generally said to be "tagged architectures" or to have "tagged memory:" at least one bit is associated with a granule of memory no larger than a capability, which indicates whether the associated granule contains capability-typed bits or data-typed bits. CHERI is such a design, with one bit per capability-sized and suitably-aligned piece of memory. The IBM System/38 uses four bits per capability-sized piece of memory and requires that they all be set when attempting to decode a suitably-aligned bit pattern as a capability.

The second variety of systems seem to lack a similarly punchy moniker, but we may, at the risk of further overloading an already burdened term, call them "segmented architectures." In these systems, it is usually the (memory-referencing) capabilities themselves that describe the type of the bits to be found therein; integrity of the capability representations is ensured by software's careful avoidance of overlapping capabilities. In simplest manifestation, a capability to memory designates, in addition to bounds and permissions, the type of all bits found therein. Such capabilities are often described with terms such as "C-type" or "D-type" (as in the Cambridge CAP family), emphasising the homogeneous nature of the segment of memory referenced. Some other segmented architectures have bifurcated segments, wherein each segment is effectively two: one containing capabilities and one containing data; capabilities specify the midpoint and length of both segments.

11.5.1 Tagged-Memory Architectures

Perhaps the most well-known tagged machine design these days is that of the Burroughs Large systems, starting with the B5000, designed in 1961. Both contemporaneous [26, 25, 98] and retrospective [73, 9] material about this family of machines is available for the curious reader, as is an interesting report of a concerted penetration test against Burroughs' operating system [151]. For present purposes, however, we focus on its memory model and, in particular, its use of tags and descriptors. In the B5000, each word was equipped with a bit distinguishing its intended use as either data or instructions. The later B6500 moved to a three-bit tag; we may (very) roughly summarize this latter taxonomy as differentiating between data words, program instructions, and pointers of various sorts. In several cases, the tags were used to convey *type* information to the CPU, so that, for example, the unique addition instruction would operate on single-precision words or double-precision word pairs depending on the data tag of its operands [98, p. 97], or the processor's "step and branch" instruction can manipulate a "step index word" containing all of the current value, increment, and limit of iteration [25, p. 7-5]. More naturally (to a CHERI-minded reader, at least), loads and stores and indirect transfers of control required their operands to be properly tagged, and subroutine entry generates tagged return addresses on the stack [25, ch. 7].

While concerned mostly with detection of software bugs, rather than any consideration of system security, Gumpertz's *Error Detection with Memory Tags* [47] deserves mention. The tags in this work are not used to determine operations (as they might have been in the Burroughs B5000) but rather as additional checks on tag-independent operations. Gumpertz's design focuses on light-weight checks, performed in parallel with CPU operations and makes it "possible to check an arbitrary number of assertions using only a small tag of fixed size" [47, p. i], which is similar to CHERI's imbuing of an arbitrary number of bits (i.e., the width of a capability) with architectural semantics using a single, external tag bit.

11.5.2 Segmented Architectures

Cambridge CAP Computer

The family of Cambridge CAP computer designs[3] [150] are, at their core, capability-based refinements of earlier, base-and-length memory segmentation schemes. In these earlier schemes, the CPU computes offsets within a segment and then dereferences memory as a pair of an offset and an *index* into a segment table; on dereference, the offset is checked to be in bounds, and the indirection between segment and memory—usually just an addition operation—is performed to yield the "linear" (or "physical") address used to communicate with the memory subsystem. Programs running on the CAP computer similarly have a virtual address space consisting of pairs of indices into a *capability table* and offsets within those capabilities. While the exact interpretation and mechanisms of the capabilities of each CAP design differed, there are commonalities across the family.

The CAP computers interpreted virtual addresses, held in arithmetic registers, as pairs of a capability specifier and a 16-bit index to a word within that capability. On the CAP computers, capabilities are interpreted only after a virtual address has been dispatched from the CPU. This separation of construction and interpretation violates our principle of intensional use and enables certain kinds of confusion. To wit, overflowing the offset results in a potentially in-bound offset *within a different capability*. This is in stark contrast to a pure-capability CHERI design, wherein capabilities *supplant* virtual addresses and are directly manipulated while in registers, making it impossible for operations on a capability's offset to change to *which* capability the offset is relative.[4]

[3] The CAP experiment seems to have produced one physical, heavily microprogrammed CPU design and at least three different microcode programs.

[4] Though CHERI does have its IDC mechanism for compatibility with non-capability programs. Similar con-

11.6 Influences of Our Own Past Projects

Our CHERI capability hardware design responds to all these design trends – and their problems. Reliance on traditional paged virtual memory for strong address-space separation, as used in Mach, EROS, and UNIX, comes at significant cost: attempts to compartmentalize system software and applications sacrifice the programmability benefits of a language-based capability design (a point made convincingly by Fabry [35]), and introduce significant performance overhead to cross-domain security boundaries. However, running these existing software designs is critical to improve the odds of technology transfer, and to allow us to incrementally apply ideas in CHERI to large-scale contemporary applications such as office suites. CHERI's hybrid approach allows a gradual transition from virtual address separation to capability-based separation within a single address space, thus restoring programmability and performance so as to facilitate fine-grained compartmentalization throughout the system and its applications.

We consider some of our own past system designs in greater detail, especially as they relate to CTSRD:

Multics The Multics system incorporated many new concepts in hardware, software, and programming [97, 27]. The Multics hardware provided independent virtual memory segments, paging, interprocess and intra-process separation, and cleanly separated address spaces. The Multics software provided symbolically named files that were dynamically linked for efficient execution, rings of protection providing layers of security and system integrity, hierarchical directories, and access-control lists. Input-output was also symbolically named and dynamically linked, with separation of policy and mechanism, and separation of device independence and device dependence. A subsequent redevelopment of the two inner-most rings enabled Multics to support multilevel security in the commercial product [113]. Multics was implemented in a stark subset (EPL) of PL/I that considerably diminished the likelihood of many common programming errors. In addition, the stack discipline inherently avoided buffer overflows.

PSOS SRI's Provably Secure Operating System hardware-software design was formally specified in a single language (SPECIAL), with encapsulated modular abstraction, interlayer state mappings, and abstract programs relating each layer to those on which it depended [93, 94]. The hardware design provided tagged, typed, unforgeable capabilities required for every operation, with identifiers that were unique for the lifetime of the system. In addition to a few primitive types, application-specific object types could be defined and their properties enforced with the hardware assistance provided by the capability-based access controls. The design allowed application layers to efficiently execute instructions, with object-oriented capability-based addressing directly to the hardware – despite appearing at a much higher layer of abstraction in the design specifications.

fusion is possible in hybrid applications if an offset intended to be relative to one capability is instead used with another, for example, due to improper management of IDC. Historically, similar confusion can arise in the more common segmentation models, as seen in, for example, Intel's X86 CPUs, in which segment table indices ("segment selectors") are held in dedicated registers and only combined with offsets (held in arithmetic registers) by the instruction stream.

MAC Framework The MAC Framework is an OS reference-monitor framework used in FreeBSD, also adopted in Mac OS X and iOS, as well as other FreeBSD-descended operating systems such as Juniper Junos and McAfee Sidewinder [130]. Developed in the DARPA CHATS program, the MAC Framework allows static and dynamic extension of the kernel's access-control model, supporting implementation of *security localization* – that is, the adaptation of the OS security to product and deployment-specific requirements. The MAC Framework (although originally targeted at classical mandatory access control models) found significant use in application sandboxing, especially in Junos, Mac OS X, and iOS. One key lesson from this work is the importance of longer-term thinking about security-interface design, including interface stability and support for multiple policy models; these are especially important in instruction-set design. Another important lesson is the increasing criticality of extensibility of not just the access-control model, but also the means by which remote principals are identified and execute within local systems: not only is consideration of classical UNIX users inadequate, but also there is a need to allow widely varying policies and notions of remote users executing local code across systems. These lessons are taken to heart in capability systems, which carefully separate policy and enforcement, but also support extensible policy through executable code.

Capsicum Capsicum is a lightweight OS capability and sandbox framework included in FreeBSD 9.x and later [131, 128]. Capsicum extends (rather than replaces) UNIX APIs, and provides new kernel primitives (sandboxed capability mode and capabilities) and a userspace sandbox API. These tools support compartmentalization of monolithic UNIX applications into logical applications, an increasingly common goal supported poorly by discretionary and mandatory access controls. This approach was demonstrated by adapting core FreeBSD utilities and Google's Chromium web browser to use Capsicum primitives; it showed significant simplicity and robustness benefits to Capsicum over other confinement techniques. Capsicum provides both inspiration and motivation for CHERI: its hybrid capability-system model is transposed into the ISA to provide compatibility with current software designs, and its demand for finer-grained compartmentalization motivates CHERI's exploration of more scalable approaches.

11.7 A Fresh Opportunity for Capabilities

Despite an extensive research literature exploring the potential of capability-system approaches, and limited transition to date, we believe that the current decade has been the time to revisit these ideas, albeit through the lens of contemporary problems and with insight gained through decades of research into security and systems design. As described in Chapter 1, a transformed threat environment deriving from ubiquitous computing and networking, and the practical reality of widespread exploitation of software vulnerabilities, both provide a strong motivation to investigate improved processor foundations for software security. This change in environment has coincided with improved and more rapid hardware prototyping techniques and higher-level hardware-definition languages that facilitate academic hardware-software system research at larger scales; without them we would have been unable to explore the CHERI approach in such detail. Simultaneously, our understanding of operating-system and programming-language security has been vastly enhanced by several decades of research; in particular, recent development of the hybrid capability-system Capsicum model suggests a strong alignment between capability-based techniques and successful mitigation approaches that can be translated into processor design choices.

Chapter 12

Conclusion

The CTSRD project (of which CHERI is just one element) is now in the final stages of completed its ninth year – an evolution described in detail in Chapter 10. The final few years of CTSRD included the CSTT task for software technology transfer. CTSRD's productivity has also been extended by our MRC2 (now completed), CIFV (in progress), and ECATS (in progress) projects to further consider topics such as multiprocessing, software-stack maturity, formal modeling and verification, and full System-on-Chip (SoC) implications for CHERI. Our focuses to date have been in several areas:

1. Develop the CHERI protection model and reference CHERI-MIPS Instruction-Set Architecture that offer low-overhead fine-grained memory protection and support scalable software compartmentalization based on a hybrid capability model. Over several generations of the ISA, refine integration with a conventional RISC ISA, compose the capability-system model with the MMU, pursue strong C-language compatibility, develop compartmentalization features based on an object-capability model, refine the architecture to improve performance and adoptability through features such as compressed 128-bit capabilities, and develop the notion of a portable protection model that can be applied to further ISAs (such as RISC-V and x86-64). Explore the implications of CHERI on 32-bit microcontroller architectures that do not have MMUs, giving capabilities a physical interpretation, and also taking into account common microcontroller microarchitectural choices.

2. Employ increasingly complete formal models of the protection model and ISA semantics. We began by using PVS/SAL formal models of the ISA to analyze expressivity and security. Subsequently, and in close collaboration with the University of Cambridge's EPSRC-funded Rigorous Engineering of Mainstream Systems (REMS) Project and DARPA-funded CHERI Instruction-set Formal Verification (CIFV), we developed L3 and Sail formal models suitable to act as a gold model for testing, to use in automated test generation, and as inputs to formal verification tools to prove ISA-level security properties. We have also used formal modeling to explore how CHERI interacts with C-language semantics. In the future, we hope to employ these models in support of hardware and software verification.

3. Elaborate the ISA feature set in CHERI to support a real-world operating system – primarily, this has consisted of adding support for the MIPS system management coprocessor, CP0, which includes the MMU and exception model, but also features such as a programmable interrupt controller (PIC). We have also spent considerable time refining successive versions of the ISA intended to better support high levels of MMU-based operating-system and C-language compatibility, as well as automatic use by compilers. This work has incorporated ideas from, but also gone substantially beyond, the C-language fat-pointer and software compartmentalization research literature.

4. Port the FreeBSD operating system first to a capability-free version of CHERI, known as BERI. This is known as FreeBSD/BERI, and this support has been upstreamed such that new releases of FreeBSD support the BERI processor and its peripheral devices.

5. Prototype, test, and refine CHERI-MIPS ISA extensions, which are incorporated via a new capability coprocessor, CP2. We have open-sourced the reference BERI and CHERI processor designs, and Qemu ISA-level emulator, in order to allow reproducible experimentation with our approach, as well as to act as an open-source platform for other future hardware-software research projects.

6. Adapt FreeBSD to make use of CHERI features. Key areas of work included adapting the kernel and userspace runtime (including system library and runtime linker) to support tagged memory, capability state, strongly enforced valid pointer provenance, and bounds/permissions reduction. This is known as CheriBSD. We developed a hardware-software in-address-space object-capability model rested on architectural capabilities. We have also developed a pure-capability system-call ABI and process environment known as CheriABI, which pushes to an extreme point the use of capabilities to represent all pointers (and implied virtual addresses, such as return addresses) in user code generation and in interaction with a conventional kernel. We are currently pursuing a pure-capability CheriBSD kernel. While open sourced, these changes remain outside of the upstream FreeBSD repository, due to their experimental nature.

7. Adapt the Clang/LLVM compiler suite to be able to generate CHERI ISA instructions as directed by C-language annotations, exploring a variety of language models, code-generation models, and ABIs. We have explored two new C-language models and associated code generation: a hybrid in which explicitly annotated or automatically inferred pointers are compiled as capabilities; and a pure-capability model in which all pointers and implied virtual addresses are compiled as capabilities. Similarly, we have begun an exploration of how CHERI affects program linkage, with early prototype integration with the compile-time and run-time linkers. These collectively provide strong spatial and pointer protection for both data and code pointers. We have upstreamed substantial improvements to Clang/LLVM MIPS support, as well as changes making it easier to support ISA features such as extra-wide pointers utilized in the CHERI ISA. We have also begun to explore how CHERI can support higher-level language protection, such as by using it to reinforce memory safety and security for native code running under the Java Native Interface (JNI).

8. Begin to develop semi-automated techniques to assist software developers in compartmentalizing applications using Capsicum and CHERI features. This is a subproject known as Security-Oriented Analysis of Application Programs (SOAAP), and performed in collaboration with Google.

9. Develop FPGA-based demonstration platforms, including an early prototype on the Terasic tPad, and more mature server-style and tablet-style prototypes based on the Terasic DE4 board. We have also made use of CHERI on the NetFGPA 10G board.

Collectively, these accomplishments have validated our research hypotheses: that a hybrid capability-system architecture and viable supporting microarchitecture can support low-overhead memory protection and fine-grained software compartmentalization while maintaining strong compatibility with current RISC, MMU-based, and C-language software stacks, as well as an incremental software adoption path to additional trustworthiness. Further, the resulting protection model, co-designed around a specific ISA and concrete extensions, is in fact a generalizable and portable protection model that can be applied to other ISAs; it is suitable for a multitude of implementations in architecture and microarchitecture. Formal methodology deployed judiciously throughout the design and implementation process has increased our confidence that the resulting design can support robust and resilient software designs.

12.1 Future Work

We have made a strong beginning, but clearly there is still much to do in our remaining CTSRD efforts. Our ongoing key areas of research include:

- Continuing to refine performance with respect to both the architecture (e.g., models for capability compression) and microarchitecture (e.g., as relates to efficient implementations of compression and tagged memory).

- Exploring how CHERI's features might be scaled up (e.g., to superscalar processor designs), down (e.g., to 32-bit microcontrollers without MMUs), and to other compute types (e.g., DMA engines, GPUs, and so on). Also, looking at how CHERI interacts with other emerging hardware technologies such as non-volatile memory, where CHERI may support more rapid, robust, and secure adoption.

- Continuing to elaborate how CHERI should affect the design of operating systems (whether hybrid systems such as CheriBSD, or clean-slate designs), languages (e.g., C, C++, Java, and so on), and runtimes (e.g., system libraries, run-time linking, and higher-level language runtimes).

- Continuing to explore how CHERI affects software tracing and debugging; for example, through capability-aware software debuggers.

- Continuing to explore potential models for software compartmentalization, such as clean-slate microkernel-style message passing grounded in CHERI's object-capability features, but not hybridized with conventional OS designs. In addition, continuing to investigate potential approaches to semi- or fully automated software compartmentalization.

- Continuing our efforts to develop and utilize formal models of the microarchitecture, architecture, operating system, linkage model, language properties, compilation, and higher-level applications. This will help us understand (and ensure) the protection benefits of CHERI up and down the hardware-software stack.

Appendix A

CHERI ISA Version History

This appendix contains a detailed version history of the CHERI Instruction-Set Architecture. This report was previously made available as the *CHERI Architecture Document*, but is now the *CHERI Instruction-Set Architecture*.

1.0 This first version of the CHERI architecture document was prepared for a six-month deliverable to DARPA. It included a high-level architectural description of CHERI, motivations for our design choices, and an early version of the capability instruction set.

1.1 The second version was prepared in preparation for a meeting of the CTSRD External Oversight Group (EOG) in Cambridge during May 2011. The update followed a week-long meeting in Cambridge, UK, in which many aspects of the CHERI architecture were formalized, including details of the capability instruction set.

1.2 The third version of the architecture document came as the first annual reports from the CTSRD project were in preparation, including a decision to break out formal-methods appendices into their own *CHERI Formal Methods Report* for the first time. With an in-progress prototype of the CHERI capability unit, we significantly refined the CHERI ISA with respect to object capabilities, and matured notions such as a trusted stack and the role of an operating system supervisor. The formal methods portions of the document was dramatically expanded, with proofs of correctness for many basic security properties. Satisfyingly, many 'future work' items in earlier versions of the report were becoming completed work in this version!

1.3 The fourth version of the architecture document was released while the first functional CHERI prototype was in testing. It reflects on initial experiences adapting a microkernel to exploit CHERI capability features. This led to minor architectural refinements, such as improvements to instruction opcode layout, some additional instructions (such as allowing `CGetPerm` retrieve the unsealed bit), and automated generation of opcode descriptions based on our work in creating a CHERI-enhanced MIPS assembler.

1.4 This version updated and clarified a number of aspects of CHERI following a prototype implementation used to demonstrate CHERI in November 2011. Changes include updates to the CHERI architecture diagram; replacement of the `CDecLen` instruction with

CSetLen, addition of a CMove instruction; improved descriptions of exception generation; clarification of the in-memory representation of capabilities and byte order of permissions; modified instruction encodings for CGetLen, CMove, and CSetLen; specification of reset state for capability registers; and clarification of the CIncBase instruction.

1.5 This version of the document was produced almost two years into the CTSRD project. It documented a significant revision (version 2) to the CHERI ISA, which was motivated by our efforts to introduce C-language extensions and compiler support for CHERI, with improvements resulting from operating system-level work and restructuring the BSV hardware specification to be more amenable to formal analysis. The ISA, programming language, and operating system sections were significantly updated.

1.6 This version made incremental refinements to version 2 of the CHERI ISA, and also introduced early discussion of the CHERI2 prototype.

1.7 Roughly two and a half years into the project, this version clarified and extended documentation of CHERI ISA features such as CCall/CReturn and its software emulation, Permit_Set_Type, the CMove pseudo-op, new load-linked and instructions for store-conditional relative to capabilities, and several bug fixes such as corrections to sign extension for several instructions. A new capability-coprocessor cause register, retrieved using a new CGetCause, was added to allow querying information on the most recent CP2 exception (e.g., bounds-check vs type-check violations); priorities were provided, and also clarified with respect to coprocessor exceptions vs. other MIPS ISA exceptions (e.g., unaligned access). This was the first version of the *CHERI Architecture Document* released to early adopters.

1.8 Less than three and a half years into the project, this version refined the CHERI ISA based on experience with compiler, OS, and userspace development using the CHERI model. To improve C-language compatibility, new instructions CToPtr and CFromPtr were defined. The capability permissions mask was extended to add user-defined permissions. Clarifications were made to the behavior of jump/branch instructions relating to branch-delay slots and the program counter. CClearTag simply cleared a register's tag, not its value. A software-defined capability-cause register range was made available, with a new CSetCause instruction letting software set the cause for testing or control-flow reasons. New CCheckPerm and CCheckType instructions were added, letting software object methods explicitly test for permissions and the types of arguments. TLB permission bits were added to authorize use of loading and storing tagged values from pages. New CGetDefault and CSetDefault pseudo-ops have become the preferred way to control MIPS ISA memory access. CCall/CReturn calling conventions were clarified; CCall now pushes the incremented version of the program counter, as well as stack pointer, to the trusted stack.

1.9 - UCAM-CL-TR-850 The document was renamed from the *CHERI Architecture Document* to the *CHERI Instruction-Set Architecture*. This version of the document was made available as a University of Cambridge Technical Report. The high-level ISA description and ISA reference were broken out into separate chapters. A new rationale chapter was added, along with more detailed explanations throughout about design choices. Notes were added in a number of places regarding non-MIPS adaptations of CHERI and 128-bit variants. Potential future directions, such as capability cursors, are discussed in more detail. Further descriptions of the memory-protection model and its use by operating systems and compilers was added. Throughout, content has been updated to reflect more recent work on compiler and operating-system support for CHERI. Bugs have been fixed in the specification of the CJR and CJALR instructions. Definitions and behavior for user-defined permission bits and OS exception handling have been clarified.

1.10 This version of the Instruction-Set Architecture is timed for delivery at the end of the fourth year of the CTSRD Project. It reflects a significant further revision to the ISA (version 3) focused on C-language compatibility, better exception-handling semantics, and reworking of the object-capability mechanism.

The definition of the NULL capability has been revised such that the memory representation is now all zeroes, and with a zeroed tag. This allows zeroed memory (e.g., ELF BSS segments) to be interpreted as being filled with NULL capabilities. To this end, the tag is now defined as unset, and the Unsealed bit has now been inverted to be a Sealed bit; the CGetUnsealed instruction has been renamed to CGetSealed.

A new **offset** field has been added to the capability, which converts CHERI from a simple base/length capability to blending capabilities and fat pointers that associate a base and bounds with an offset. This approach learns from the extensive fat-pointer research literature to improve C-language compatibility. The offset can take on any 64-bit value, and is added to the base on dereference; if the resulting pointer does not fall within the base and length, then an exception will be thrown. New instructions are added to read (CGetOffset) and write (CSetOffset) the field, and the semantics of memory access and other CHERI instructions (e.g., CIncBase) are updated for this new behavior.

A new CPtrCmp instruction has been added, which provides C-friendly comparison of capabilities; the instruction encoding supports various types of comparisons including ‘equal to’, ‘not equal to’, and both signed and unsigned ‘less than’ and ‘less than or equal to’ operators.

CGetPCC now returns **PC** as the **offset** field of the returned **PCC** rather than storing it to a general-purpose integer register. CJR and CJALR now accept target **PC** values via the offsets of their jump-target capability arguments rather than via explicit general-purpose integer registers. CJALR now allows specification of the return-program-counter capability register in a manner similar to return-address arguments to the MIPS JALR instruction.

CCall and CReturn are updated to save and restore the saved **PC** in the **offset** field of the saved **EPCC** rather than separately. **EPCC** now incorporates the saved exception **PC** in its **offset** field. The behavior of **EPCC** and expectations about software-supervisor behavior are described in greater detail. The security implications of exception cause-code precedence as relates to alignment and the emulation of unaligned loads and stores are clarified. The behavior of CSetCause has been clarified to indicate that the instruction should not raise an exception unless the check for *Access_EPCC* fails. When an exception is raised due to the state of an argument register for an instruction, it is now defined which register will be named as the source of the exception in the capability cause register.

The object-capability type field is now 24-bit; while a relationship to addresses is maintained in order to allow delegation of type allocation, that relationship is deemphasized. It is assumed that the software type manager will impose any required semantics on the field, including any necessary uniqueness for the software security model. The `CSetType` instruction has been removed, and a single `CSeal` instruction replaces the previous separate `CSealCode` and `CSealData` instructions.

The validity of capability fields accessed via the ISA is now defined for untagged capabilities; the undefinedness of the in-memory representation of capabilities is now explicit in order to permit 'non-portable' micro-architectural optimizations.

There is now a structured description of the pseudocode language used in defining instructions. Format numbers have now been removed from instruction descriptions.

Ephemeral capabilities are renamed to 'local capabilities,' and non-ephemeral capabilities are renamed to 'global capabilities'; the semantics are unchanged.

1.11 - UCAM-CL-TR-864 This version of the CHERI ISA has been prepared for publication as a University of Cambridge technical report. It includes a number of refinements to CHERI ISA version 3 based on further practical implementation experience with both C-language memory protection and software compartmentalization.

There are a number of updates to the specification reflecting introduction of the **offset** field, including discussion of its semantics. A new `CIncOffset` instruction has been added, which avoids the need to read the offset into a general-purpose integer register for frequent arithmetic operations on pointers.

Interactions between **EPC** and **EPCC** are now better specified, including that use of untagged capabilities has undefined behavior. `CBTS` and `CBTU` are now defined to use branch-delay slots, matching other MIPS-ISA branch instructions. `CJALR` is defined as suitably incrementing the returned program counter, along with branch-delay slot semantics. Additional software-path pseudocode is present for `CCall` and `CReturn`.

`CAndPerm` and `CGetPerm` use of argument-register or return-register permission bits has been clarified. Exception priorities and cause-code register values have been defined, clarified, or corrected for `CClearTag`, `CGetPCC`, `CSC`, and `CSeal`. Sign or zero extension for immediates and offsets are now defined `CL`, `CS`, and other instructions.

Exceptions caused due to TLB bits controlling loading and storing of capabilities are now CP2 rather than TLB exceptions, reducing code-path changes for MIPS exception handlers. These TLB bits now have modified semantics: **LC** now discards tag bits on the underlying line rather than throwing an exception; **SC** will throw an exception only if a tagged store would result, rather than whenever a write occurs from a capability register. These affect `CLC` and `CSC`.

Pseudocode definitions now appear earlier in the chapter, and have now been extended to describe **EPCC** behavior. The ISA reference has been sorted alphabetically by instruction name.

1.12 This is an interim release as we begin to grapple with 128-bit capabilities. This requires us to better document architectural assumptions, but also start to propose changes to the instruction set to reflect differing semantics (e.g., exposing more information to potential capability compression). A new CSetBounds instruction is proposed, which allows both the base and length of a capability to be set in a single instruction, which may allow the micro-architecture to reduce potential loss of precision. Pseudocode is now provided for both the pure-exception version of the CCall instruction, and also hardware-accelerated permission checking.

1.13 This is an interim release as our 128-bit capability format (and general awareness of imprecision) evolves; this release also makes early infrastructural changes to support an optional converging of capability and general-purpose integer register files.

Named constants, rather than specific sizes (e.g., 256-bit vs. 128-bit) are now used throughout the specification. Reset state for permissions is now relative to available permissions. Two variations on 128-bit capabilities are defined, employing two variations on capability compression. Throughout the specification, the notion of "representable" is now explicitly defined, and non-representable values must now be handled.

The definitions of CIncOffset, CSetOffset, and CSeal have been modified to reflect the potential for imprecision. In the event of a loss of precision, the capability base, rather than offset, will be preserved, allowing the underlying memory object to continue to be accurately represented.

Saturating behavior is now defined when a compressed capability's length could represent a value greater than the maximum value for a 64-bit MIPS integer register.

EPCC behavior is now defined when a jump or branch target might push the offset of PCC outside of the representable range for EPCC.

CIncBase and CSetLen are deprecated in favor of CSetBounds, which presents changes to base and bounds to the hardware atomically. The CMove pseudo-operation is now implemented using CIncOffset rather than CIncBase. CFromPtr has been modified to behave more like CSetOffset: only the offset, not the base, is modified. Bug fixes have been applied to the definitions of CSetBounds and CUnseal.

Several bugs in the specification of CLC, CLLD, CSC, and CSD, relating to omissions during the update to capability offsets, have been fixed. CLC's description has been updated to properly reflect its immediate argument.

New instructions CClearHi and CClearLo have been added to accelerate register clearing during protection-domain switches.

New pseudo-ops CGetEPCC, CSetEPCC, CGetKCC, CSetKCC, CGetKDC, and CSetKDC have been defined, in the interests of better supporting a migration of 'special' registers out of the capability register file – which facilitates a convergence of capability and general-purpose integer register files.

1.14 Two new chapters have been added, one describing the abstract CHERI protection model in greater detail (and independent from concrete ISA changes), and the second exploring the composition of CHERI's ISA-level features in supporting higher-level software protection models.

The value of the NULL capability is now centrally defined (all fields zero; untagged).

ClearLo and ClearHi instructions are now defined for clearing general-purpose integer registers, supplementing CClearHi and CClearLo. All four instructions are described together under CClearRegs.

A new CSetBoundsExact instruction is defined, allowing an exception to be thrown if an attempt to narrow bounds cannot occur precisely. This is intended for use in memory allocators where it is a software invariant that bounds are always exact. A new exception code is defined for this case.

A full range of data widths are now support for capability-relative load-linked, store conditional: CLLB, CLLH, CLLW, CLLD, CSCB, CSCH, CSCW, and CSCD (as well as unsigned load-linked variations). Previously, only a doubleword variation was defined, but cannot be used to emulate the narrower widths as fine-grained bounds around a narrow type would throw a bounds-check exception. Existing load-linked, store-conditional variations for capabilities (CLLC, CSCC) have been updated, including with respect to opcode assignments.

A new 'candidate three' variation on compressed capabilities has been defined, which differentiates sealed and unsealed formats. The unsealed variation invests greater numbers of bits in bounds accuracy, and has a full 64-bit cursor, but does not contain a broader set of software-defined permissions or an object-type field. The sealed variation also has a full 64-bit cursor, but has reduced bounds accuracy in return for a 20-bit object-type field and a set of software-defined permissions.

'Candidate two' of compressed capabilities has been updated to reflect changes in the hardware prototype by reducing toBase and toBound precision by one bit each.

Explicit equations have been added explaining how bounds are calculated from each of the 128-bit compressed capability candidates, as well as their alignment requirements.

Exception priorities have been documented (or clarified) for a number of instructions including CJALR, CLC, CLLD, CSC, CSCC, CSetLen, CSeal, CUnSeal, and CSetBounds.

The behavior of CPtrCmp is now defined when an undefined comparison type is used.

It is clarified that capability store failures due to TLB-enforced limitations on capability stores trigger a TLB, rather than a CP2, exception.

A new capability comparison instruction, CEXEQ, checks whether all fields in the capability are equal; the previous CEQ instruction checked only that their offsets pointed at the same location.

A new capability instruction, CSUB, allows the implementation of C-language pointer subtraction semantics with the atomicity properties required for garbage collection.

The list of BERI- and CHERI-related publications, including peer-reviewed conference publications and technical reports, has been updated.

1.15 - UCAM-CL-TR-876 This version of the CHERI ISA, *CHERI ISAv4*, has been prepared for publication as a University of Cambridge technical report.

The instructions CIncBase and CSetLen (deprecated in version 1.13 of the CHERI ISA) have now been removed in favor of CSetBounds (added in version 1.12 of the CHERI ISA). The new instruction was introduced in order to atomically expose changes to both upper and lower bounds of a capability, rather than requiring them to be updated separately, required to implement compressed capabilities.

The design rationale has been updated to better describe our ongoing exploration of whether special registers (such as **KCC**) should be in the capability register file, and the potential implications of shifting to a userspace exception handler for CCall/CReturn.

1.16 This is an interim update of the instruction-set specification in which aspects of the 128-bit capability model are clarified and extended.

The "candidate 3" unsealed 128-bit compressed capability representation has been to increase the exponent field (**e**) to 6 bits from 4, and the **baseBits** and **topBits** fields have been reduced to 20 bits each from the 22 bits. **perms** has been increased from 11 to 15 to allow for a larger set of software-defined permissions. The sealed representation has also been updated similarly, with a total of 10 bits for **otype** (split over **otypeLow** and **otypeHigh**), 10 bits each for **baseBits** and **topBits**, and a 6-bit exponent. The algorithm for decompressing a compressed capability has been changed to better utilize the encoding space, and to more clearly differentiate representable from in-bounds values. A variety of improvements and clarifications have been made to the compression model and its description.

Differences between, and representations of, permissions for 128-bit and 256-bit capability are now better described.

Capability unrepresentable exceptions will now be thrown in various situations where the result of a capability manipulation or operation cannot be represented. For manipulations such as CSeal and CFromPtr, an exception will be thrown. For operations such as CBTU and CBTS, the exception will be thrown on the first instruction fetch following a branch to an unrepresentable target, rather than on the branch instruction itself. CHERI1 and CHERI2 no longer differ on how out-of-bounds exceptions are thrown for capability branches: it uniformly occurs on fetching the target instruction.

The ISA specification makes it more clear that CEQ, CNE, CL[TE]U, and CEXEQ are forms of the CPtrCmp instruction.

The ISA todo list has been updated to recommend a capability conditional-move (CCMove) instruction.

There is now more explicit discussion of the MIPS n64 ABI, Hybrid ABI, and Pure-Capability ABI. Conventions for capability-register have been updated and clarified – for example, register assignments for the stack capability, jump register, and link register. The definition that **RCC**, the return code capability, is register **C24** has been updated to reflect our use of **C17** in actual code generation.

Erroneous references to an undefined instruction `CSetBase`, introduced during removal of the `CIncBase` instruction, have been corrected to refer to `CSetBounds`.

1.17 This is an interim update of the instruction-set architecture enhancing (and specifying in more detail) the CHERI-128 "compressed" 128-bit capability format, better aligning the 128-bit and 256-bit models, and adding capability-related instructions required for more efficient code generation. This is a draft release of what will be considered *CHERI ISAv5*.

The chapter on ISA design now includes a section describing "deep" versus "surface" aspects of the CHERI model as mapped into the ISA. For example, use of tagged capabilities is a core aspect of the model, but the particular choice to have a separate capability register file, rather than extending general-purpose integer registers to optionally hold capabilities, is a surface design choice in that the operating system and compiler can target the same software-visible protection model against both. Likewise, although CHERI-128 specifies a concrete compression model, a range of compression approaches are accepted by the CHERI model.

A new chapter has been added describing some of our assumptions about how capabilities will be used to build secure systems, for example, that untrusted code will not be permitted to modify TLB state – which permits changing the interpretation of capabilities relative to virtual addresses.

The rationale chapter has been updated to more thoroughly describe our capability compression design space.

A new CHERI ISA quick-reference appendix has been added to the specification, documenting both current and proposed instruction encodings.

Sections of the introduction on historical context have been shifted to a stand-alone chapter.

Descriptions in the introduction have been updated relating to our hardware and software prototypes.

References to PhD dissertations on CHERI have been added to the publications section of the introduction.

A clarification has been added: the use of the term "capability coprocessor" relates to CHERI's utilization of the MIPS ISA coprocessor opcode space, and is not intended to suggest substantial decoupling of capability-related processing from the processor design.

Compressed capability "candidate 3" is now CHERI-128. The **baseBits**, **topBits** and **cursor** fields have been renamed respectively **B**, **T** and **a** (following the terminology used in the micro paper). When sealed, only the top 8 bits of the **B** and **T** fields are preserved, and the bottom 12 bits are zeroes, which implies stronger alignment requirements for sealed capabilities. The exponent **e** field remains a 6-bit field, but its bottom 2 bits are ignored, as it is believed that coarser granularity is acceptable, and making the hardware simpler. The **otype** field benefits from the shorter **B** and **T** fields and is now 24 bits – which is the same as the **otype** for 256-bit CHERI. Finally, the representable region associated with a capability has changed from being centred around the described object to an asymmetric region with more space above the object than below. The full description is available in Section 3.4.4.

Alignment requirements for software allocators (such as stack and heap allocators) in the presence of capability compression are now more concisely described.

The immediate operands to various load and store instructions, including CLC, CSC, CL[BHWD][U], and CS[BHWD] are now "scaled" by the width of the data being stored (with the exception of capability stores, where scaling is by 16 bytes regardless of in-memory capability size). This extends the range of capability-relative loads and stores, permitting a far greater proportion of stack spills to be expressed without additional stack-pointer modification. This is a binary-incompatible change to the ISA.

The textual description of the CSeal instruction has been updated to match the pseudocode in using $>=$ rather than $>$ in selecting an exception code.

A redundant check has been removed in the definition of the CUnseal instruction, and an explanation added.

Opcodes have now been specified for the CSetBoundsExact and CSub instructions.

To improve code generation when constructing a **PCC**-relative capability as a jump target, a new CGetPCCSetOffset instruction has been added. This instruction has the combined effects of performing sequential CGetPCC and CSetOffset operations.

A broader set of opcode rationalizations and cleanups have been applied across the ISA, to facilitate efficient decoding and future use of the opcode space. This includes changes to CGetPCC.

C25 is no longer reserved for exception-handler use, as **C27** and **C28** are already reserved for this purpose. It is therefore available for ABI use.

The 256-bit architectural capability model has been updated to use a single system permission, Access_System_Registers, to control access to exception-handling and privileged ISA state, rather than splitting it over multiple permissions. This brings the permission models in 128-bit and 256-bit representations back into full alignment from a software perspective. This also simplifies permission checking for instructions such as CClearRegs. The permission numbering space has been rationalized as part of this change. Similarly, the set of exceptions has been updated to reflect a single system permission. The descriptions of various instructions (such as CClearRegs have been updated with respect to revised protections for special registers and exception handling.

The descriptions of CCall and CReturn now include an explanation of additional software-defined behavior such as capability control-flow based on the local/global model.

The common definition of privileged registers (included in the definitions of instructions) has been updated to explicitly include **EPCC**.

Future ISA additions are proposed to add testing of branch instructions for NULL and non-NULL capabilities.

1.18 - UCAM-CL-TR-891 This version of the CHERI ISA, *CHERI ISAv5*, has been prepared for publication as a University of Cambridge technical report.

The chapter on the CHERI protection model has been refined and extending, including adding more information on sealed capabilities, the link between memory allocation and the setting of bounds and permissions, more detailed coverage of capability flow control, and interactions with MMU-based models.

A new chapter has been added exploring assumptions that must be made when building high-assurance software for CHERI.

The detailed ISA version history has shifted from the introduction to a new appendix; a summary of key versions is maintained in the introduction, along with changes in the current document version.

A glossary of key terms has been added.

The term "coprocessor" is deemphasized, as, while it refers correctly to CHERI's use of the MIPS opcode extension space, some readers found it suggestive of an independent hardware unit rather than tight integration into the processor pipeline and memory subsystem.

A reference has been added to Robert Norton's PhD dissertation on optimized CHERI domain switching.

A reference has been added to our PLDI 2016 paper on C-language semantics and their interaction with the CHERI model.

The object-type field in both 128-bit and 256-bit capabilities is now 24 bits, with Top and Bottom fields reduced to 8 bits for sealed capabilities. This reflects a survey of current object-oriented software systems, suggesting that 24 bits is a more reasonable upper bound than 20 bits.

The assembly arguments to CJALR have been swapped for greater consistency with jump-and-link register instructions in the MIPS ISA.

We have reduced the number of privileged permissions in the 256-bit capability model to a single privileged permission, Access_System_Registers, to match 128-bit CHERI. This is a binary-incompatible change.

We have improved the description of the CHERI-128 model in a number of ways, including a new section on the CHERI-128 representable bounds check.

The architecture chapter contains a more detailed discussion of potential ways to reduce the overhead of CHERI by reducing the number of capability registers, converging the general-purpose integer and capability register files, capability compression, and so on.

We have extended our discussion of "deep" vs "shallow" aspects of the CHERI model.

New sections describe potential non-pointer uses of capabilities, as well as possible uses as primitives supporting higher-level languages.

Instructions that convert from integers to capabilities now share common `int_to_cap` pseudocode.

The notes on CBTS have been synchronized to those on CBTU.

Use of language has generally been improved to differentiate the architectural 256-bit capability model (e.g., in which its fields are 64-bit) from the 128-bit and 256-bit in-memory representations. This includes consideration of differing representations of capability permissions in the architectural interface (via instructions) and the microarchitectural implementation.

A number of descriptions of features of, and motivations for, the CHERI design have been clarified, extended, or otherwise improved.

It is clarified that when combining immediate and register operands with the base and offset, 64-bit wrap-around is permitted in capability-relative load and store instructions – rather than throwing an exception. This is required to support sound optimizations in frequent compiler-generated load/store sequences for C-language programs.

1.19 This release of the *CHERI Instruction-Set Architecture (ISA) Specification* is an interim version intended for submission to DARPA/AFRL to meet the requirements of CTSRD deliverable A015.

The behavior of `CToPtr` in the event that the pointer of one capability is to the base of the containing capability has been clarified.

The Access_System_Registers permission is extended to cover non-CHERI ISA privileges, such as use of MIPS TLB-management, interrupt-control, exception-handling, and cache-control instructions available in the kernel ring. The aim of these in-progress changes is to allow the compartmentalization of kernel code.

1.20 - UCAM-CL-TR-907 This version of the CHERI ISA, *CHERI ISAv6*, has been prepared for publication as University of Cambridge technical report UCAM-CL-TR-907.

Chapter 1 has been substantially reformulated, providing brief introductions to both the CHERI protection model and CHERI-MIPS ISA, with much remaining content on our research methodology now shifted to its own new chapter, Chapter 10. Our architectural and application-level least-privilege motivations are now more clearly described, as well as hybrid aspects of the CHERI approach. Throughout, better distinction is made between the CHERI protection model and the CHERI-MIPS ISA, which is a specific instantiation of the model with respect to 64-bit MIPS. The research methodology chapter now provides a discussion of our overall approach, more detailed descriptions of various phases of our research and development cycle, and describes major transitions in our approach as the project proceeded.

Chapter 2 on the software-facing CHERI protection model has been improved to provide more clear explanations of our approach as well as additional illustrations. The chapter now more clearly enunciates two guiding principles underlying the CHERI ISA design: the *principle of least privilege*, and the *principle of intentional use*. The former has been widely considered in the security literature, and motivates privilege reduction in the CHERI ISA. The latter has not previously described, and is supports the use of explicitly named rights, rather than implicitly selected ones, wherever possible in order to avoid 'confused deputy' problems. Both contribute to vulnerability mitigation effects.

New sections have been added on revocation and garbage collection. The role and implementation of monotonicity (and also non-monotonicity) in the ISA are more clearly described.

A chapter on architectural sketches has been added, describing how the CHERI protection model might be introduced in the RISC-V and x86-64 ISAs. In doing so, we identify a number of key aspects of the CHERI model that are required regardless of the underlying ISA. We argue that the CHERI protection model is a *portable* model that can be implemented consistently across a broad range of underlying ISAs and concrete integrations with those ISAs. One implication of this argument is that portable CHERI-aware software can be implemented across underlying architectural implementations.

Chapter 3 now describes, at a high level, CHERI's expectations for tagged memory.

We in general now prefer the phrase "control-flow robustness" to "control-flow integrity" when talking about capability protection for code pointers, in order to avoid confusion with conventional CFI.

The descriptions of software-defined aspects of the `CCall` and `CReturn` instructions have been removed from the description and pseudocode of each instruction. They are instead part of an expanded set of notes on potential software use for these instructions.

A new `CCall` selector 1 has been added that provides a jump-like domain transition without use of an architectural exception. In this mode of operation, `CCall` unseals the sealed code and data capabilities to enter the new domain, offering a different set of hardware and software tradeoffs from the existing selector-0 semantics. For example, complex exception-related mechanism is avoided in hardware for domain switches, with the potential to substantially improve performance. Software would most likely use this mechanism to branch into a trusted intermediary capability of supporting safe and controlled switching to a new object.

To support the new `CCall` selector 1, a new permission, *Permit_CCall* is defined authorizing use of the selector on sealed capabilities. The permission must be present on both sealed code and data capabilities.

To support the new `CCall` selector 1, a new CP2 exception cause code, Permit_CCall Violation is defined to report a lack of the *Permit_CCall* permission on sealed code or data capabilities passed to `CCall`.

New experimental instructions `CBuildCap` (import a capability), `CCopyType` (import the **otype** field of a capability), and `CCSeal` (conditionally seal a capability) have been added to the ISA to be used when re-internalizing capabilities that have been written to non-capability-aware memory or storage. This instruction is intended to satisfy use cases such as swapping to disk, migrating processes, migrating virtual machines, and run-time linking. A suitable authorizing capability is required in order to restore the tag. As these instructions are considered experimental, they are documented in Appendix D rather than the main specification.

The `CGetType` instruction now returns -1 when used on an unsealed capability, in order to allow it to be more easily used with `CCSeal`.

Two new conditional-move instructions are added to the CHERI-MIPS ISA: CMOVN (conditionally move capability on non-zero), and CMOVZ (conditionally move capability on zero). These complement existing conditional-move instructions in the 64-bit MIPS ISA, allowing more efficient generated code.

The CJR (capability jump register) and CJALR (capability jump and link register) have been changed to accept non-global capability jump targets.

The CLC (capability load capability) and CLLC (capability load-linked conditional) instructions will now strip loaded tags, rather than throwing an exception, if the Permit_Load_Capability permission is not present.

The CToPtr (capability to pointer) instruction now checks that the source register is not sealed, and performs comparative range checks of the two source capabilities. More detailed rationale has been provided for the design of the CToPtr instruction in Chapter 8.

The pseudocode for the CCheckType (capability check type) instruction has been corrected to test uperm as well as perm. The pseudocode for CCheckType has been corrected to test the sealed bit on both source capabilities. An encoding error for CCheckType in the ISA quick reference has been corrected.

The pseudocode for the CGetPerm (capability get permissions) instruction has been updated to match syntax used in the CGetType and CGetCause instructions.

The pseudocode for the CUnseal (capability unseal) instruction has been corrected to avoid an aliasing problem when the source and destination register are the same.

The description of the CSeal (capability seal) instruction has been clarified to explain that precision cannot be lost in the case where bounds are no longer precisely representable, as an exception will be thrown.

The description of the fast representability check for compressed capabilities has been improved.

CHERI-related exception handling behavior is now clarified with respect to the MIPS EXL status bit, with the aim of ensuring consistent behavior. Regardless of bounds set on **KCC**, a suitable offset is selected so that the standard MIPS exception vector will be executed via the exception **PCC**.

The section on CHERI control in Chapter 4 has been clarified to more specifically identify 64-bit MIPS privileged instructions, KSU bits, and general operation modified by the Access_System_Registers permission. The section now also more specifically described privileged behaviors not controlled by the permission, such as use of specific exception vectors. A corresponding rationale section has been added to Chapter 8.

A number of potential future instruction-set improvements relating to capability compression, control flow, and instruction variants with immediates have been added to the future ISA changes list in Chapter 3.

Opcode-space reservations for the previously removed CIncBase and CSetLen instructions have also been removed.

C25, which had its hard-coded ISA use removed in CHERI ISAv5, has now been made a caller-save capability register in the ABI.

Citations to further CHERI research publications have been added.

1.21 This release of the *CHERI Instruction-Set Architecture* is an interim version intended for submission to DARPA/AFRL to meet the requirements of CTSRD deliverable A001, and contains the following changes relative to CHERI ISAv6:

The ISA encoding reference has been updated for new experimental instructions.

A new `CNExEq` instruction has been added, which provides a more efficient implementation of a test for negative exact inequality than utilizing `CExEq` and inverting the result.

Specify that when a TLB exception results from attempting to store a tagged capability via a TLB entry that does not authorize tagged store, the MIPS EntryHi register will be set correspondingly.

7.0-ALPHA1 This release of the *CHERI Instruction-Set Architecture* is an interim version intended for submission to DARPA/AFRL to meet the requirements of CTSRD deliverable A001:

- The CHERI ISA specification version numbering scheme has changed to include the target major version in the draft version number.
- A significant refactoring of early chapters in the report has taken place: there is now a more clear distinction between architecture-neutral aspects of CHERI, and those that are architecture specific. The CHERI-MIPS ISA is now its own chapter distinct from architecture-neutral material. We have aimed to maximize architecture-neutral content – e.g., capability semantics and contents, in-memory representation, compression, etc. – using the architecture-specific chapters to address only architecture-specific aspects of the mapping of CHERI into the specific architecture – e.g., as relates to register-file integration, exception handling, and the Memory Management Unit (MMU). In some areas, content must be split between architecture-neutral and architecture-specific chapters, such as behavior on reset, handling of the System_Access_Registers permission and its role in controlling architecture-specific behavior, and the integration of CHERI with virtual memory, where the goals are largely architecture neutral but mechanism is architecture specific.
- There are now dedicated chapters for each of our applications of CHERI to each of three ISAs: 64-bit MIPS (Chapter 4), 64-bit RISC-V (Chapter 5), and x86-64 (Chapter 6).
- Our CHERI-RISC-V prototype has been substantially elaborated, and now includes an experimental encoding in Appendix C. We have somewhat further elaborated our x86-64 model, including addressing topics such as new page-table bits for CHERI, including a hardware-managed capability dirty bit. We also consider potential implications for RISC-V compressed instructions.

- We have completed an opcode renumbering for CHERI-MIPS. The “proposed new encoding” from CHERI ISAv6 has now become the established encodings; the prior encodings are now documented as “deprecated encodings”.
- Substantial improvements have been made to descriptive text around memory protection, with the concept of “pointer protection” – i.e., as implemented via tags – more clearly differentiated from memory protection.
- We now more clearly describe how terms like “lower bound” and “upper bound” relate to the base, offset, and length fields.
- We now more clearly differentiate language-level capability semantics from capability use in code generation and the ABI, considering pure-capability and hybrid C as distinct from pure-capability and hybrid code generation. We explain that different language-level integer interpretations of capabilities are supportable by the architecture, depending on compiler code-generation choices.
- Potential software policies for revocation, garbage collection, and capability flow control based on CHERI primitives are described in greater detail.
- Monotonicity is more clearly described, as are the explicit opportunities for non-monotonicity around exception handling and `CCall` Selector 1. Handling of disallowed requests for non-monotonicity or bypass of guarded manipulation by software is more explicitly discussed, including the opportunities for both exception throwing and tag stripping to maintain CHERI’s invariants.
- Further notes have been added regarding the in-memory representation of capabilities, including the storage of NULL capabilities, virtual addresses for non-NULL capabilities, and how to store integer values in untagged capability registers. These values now appear in the bottom 64 bits of the in-memory representation. Topics such as endianness are also considered.
- NULL capabilities are now defined as having a base of 0x0, the maximum length supported in a particular representation (2^{64} for 128-bit capabilities, and $2^{64} - 1$ for 256-bit capabilities), and no granted permissions. NULL capabilities continue to have an all zeros in-memory representation. This allows integers to be stored in the offset of an untagged capability without concern that they may hold values that are unrepresentable with respect to capability bounds.
- New instructions `CReadHwr` and `CWriteHwr` have been added. These have allowed us to migrate special capability registers (SCRs) out of the general-purpose capability register file, including **DDC**, the new user TLS register (**CULR**), the new privileged TLS register (**CPLR**), **KR1C**, **KR2C**, **KCC**, **KDC**, and **EPCC**. Access to privileged special registers continues to be authorized by the Access_System_Registers permission on **PCC**.
- With this migration, **C0** is now available to use as a NULL capability register, which is more consistent with the baseline MIPS ISA in which **R0** is the zero register. The only exception to this is in capability-relative load and store instructions, and the `CTestSubset` instruction, in which an operand of **C0** specifies that **DDC** should be used.

- Various instruction pseudo-ops to access special registers, such as `CGetDefault`, now expand to special capability register access instructions instead of capability move instructions.

- With consideration of merged rather than split integer and capability register files for RISC-V and x86-64, and a separation between general-purpose capability registers and special capability registers (SCRs) on 64-bit MIPS, we avoid describing the integer register file as the "general-purpose register file". We describe a number of tradeoffs around ISA design relating to using a split vs. merged register file; avoiding the use of specific capability registers as special registers assists in supporting both register-file approaches.

- The CPU reset state of various capability registers is now more clearly defined. Most capability registers are initialized to NULL on reset, with the exception of **DDC**, **PCC**, **KCC**, and **EPCC**. These defaults authorize initial access to memory for the boot process, and are designed to allow CHERI-unaware code to operate oblivious to the capability-system feature set.

- We more clearly describe design choices around failure-mode choices, including throwing exceptions and clearing tag bits. Here, concerns in conclude stylistic consistency with the host architecture, potential use cases, and interactions with the compiler and operating system.

- In general, we now refer to software-defined permissions rather than user-defined permissions, as these permissions without an architectural interpretation may be used in any ring.

- Permission numbering has been rationalized so that 128-bit and 256-bit microarchitectural permission numbers consistently start at 15.

- The existing permission Permit_Seal, which authorized sealing and explicit unsealing of sealed capabilities, has now been broken out into two separate permissions: Permit_Seal, which authorizes sealing, and Permit_Unseal, which authorizes explicit unsealing. This will allow privilege to be reduced where unsealing is desirable (e.g., within object implementations, or in C++ vtable use) by not requiring that permission to seal for the object type is also granted.

- The ISA quick reference has been updated to reflect new instructions, as well as to more correctly reflect endianness.

- We have added a reference to a new technical report, *Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks* [147], which considers the potential interactions between CHERI and the recently announced Spectre and Meltdown microarchitectural side-channel attacks. CHERI offers substantial potential to assist in mitigating aspects of these attacks, as long as the microarchitecture performs required capability checks before performing any speculative memory accesses.

- We have added two new instructions, Get the architectural Compartment ID (`CGetCID`) and Set the architectural Compartment ID (`CSetCID`), which allow information on compartments to be passed to via architecture to microarchitecture in order to support mitigation of side-channel attacks. This could be used to tag branch-predictor entries to control the compartments in which they can be used, for example. A new Permit_Set_CID permission allows capabilities to delegate use of ranges of CIDs.

- Bugs have been fixed in the definitions of capability-relative load and store instructions: permission checks involving the Permit_Load, Permit_Load_Cap, Permit_Store, and Permit_Store_Cap permissions were not properly updated from our shift from an untagged capability register file to a tagged register file. All loads now require Permit_Load. If Permit_Load_Cap is also present, then tags will not be stripped when loading into a capability register. All stores now require Permit_Store. If Permit_Store_Cap is also present, then storing a tagged capability will not generate an exception.
- New Capability Set Bounds From Immediate (`CSetBoundsImm`) and Capability Increment Offset From Immediate (`CIncOffsetImm`) instructions have been added. These instructions optimize global-variable setup and stack allocations by reducing the number of instructions and registers required to adjust pointer values and set bounds.
- New Capability Branch if Not NULL (`CBNZ`) and Capability Branch if NULL (`CBEZ`) instructions have been added, which optimize pointer comparisons to NULL.
- A new Capability to Address (`CGetAddr`) instruction allows the direct retrieval of a capability's virtual address, rather than requiring the base and offset to be separately retrieved and added together. This facilitates efficient implementation of a CHERI C variant in which all casts of capabilities to integers have virtual-address rather than offset interpretation. A capability's virtual address is now more directly defined when we specify capability fields.
- We more clearly describe `CCall` Selector 1 as "exception-free domain transition" rather than "userspace domain transition", as it is also intended to be used in more privileged rings.
- We have shifted to more consistently throwing an exception at jump instructions (e.g., `CJR`) that go out of bounds, rather than throwing the exception when fetching the first instruction at the target address. This provides more debugging information when using compressed capabilities, as otherwise **EPCC** might have unrepresentable bounds in the event that the jump target is outside of the representable region.
- The exception vectors use during failures of Selector 0 and Selector 1 `CCall` have been clarified. The general-purpose exception vector is used for all failure modes with `CCall` Selector 1.
- New experimental instruction Test that Capability is a Subset of Another (`CTestSubset`) has been added. This instruction is intended to be used by garbage collectors that need to rapidly test whether a capability points into the range of another capability.
- A new experimental 64-bit capability format for 32-bit virtual addresses has been added (Section D.7).

- A description of an experimental *linear capability* model has been added (Section D.10). This model introduces the concept that a capability may be linear – i.e., that it can only be moved rather copied in memory-to-register, register-to-register, and register-to-memory operations. This introduces two new instructions, Linear Load Capability Register (`LLCR`) and Linear Store Capability Register (`LSCR`). This functionality has not yet been fully specified.
- An experimental appendix considers possible implementations of *indirect capabilities*, in which a capability value points at an actual capability to utilize, allowing table-based capability lookups (Section D.11).
- An experimental appendix considering potential forms of compression for capability permissions has been added (Section D.8).
- We have added a reference to our ICCD 2017 paper, *Efficient Tagged Memory*, which describes how to efficiently implement tagged memory in memory subsystems not supporting inline tags directly in DRAM [54].

7.0-ALPHA2 This version of the *CHERI Instruction-Set Architecture* is an interim version distributed for review by DARPA and our collaborators:

- We have removed the range check from the `CToPtr` specification, as this has proven microarchitecturally challenging. We anticipate that current consumers requiring this range check can use the new `CTestSubset` instruction alongside `CToPtr`.
- Use of a branch-delay slot with `CCall` Selector 1 has been removed.
- With the addition of `CReadHwr` and `CWriteHwr` and shifting of special capability registers out of the general-purpose capability register file, we have now removed the check for the Access_System_Registers permission for all registers in the general-purpose capability register file.
- A new `CCheckTag` instruction is added, which throws an exception if the tag is not set on the operand capability. This instruction could be used by a compiler to shift capability-related exception behavior from invalid dereference to calculation of an invalid capability via a non-exception-throwing manipulation.
- We have added a new `CLCBI` instruction that allows capability-relative loads of capabilities to be performed using a substantially larger immediate (but without a general-purpose integer-register operand). This substantially accelerates performance in the presence of CHERI-aware linkage by avoiding multi-instruction sequences to load capabilities for global variables.
- We have added new discussion relating to microarchitectural side channels such as Spectre and Meltdown (Section 2.5).
- We have added a reference to our ASPLOS 2019 paper, *CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment*, which describes how to adapt a full MMU-based OS design to support ubiquitous use of capabilities to implement C and C++ pointers in userspace [28].

- We have added a reference to our POPL 2019 paper, *ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS*, which describes a formal modeling approach for instruction-set architectures, as well as a formal model of the CHERI-MIPS ISA [8].
- We have added a reference to our POPL 2019 paper, *Exploring C Semantics and Pointer Provenance*, which describes a formal model for C pointer provenance, and is evaluated in part using pure-capability CHERI code [78].
- We have added a description of an experimental compact capability coloring scheme, a possible candidate to replace our Local-Global capability flow-control model (Section D.13). In the proposed scheme, a series of orthogonal "colors" can be set or cleared on capabilities, authorized by a color space implemented in a style similar to the sealed-capability object-type space using a single permission. For a single color implementing the Local-Global model, two bits are still used. However, for further colors, only a single bit is used. This could make available further colors to use for kernel-user separation, inter-process isolation, and so on.
- An experimental Permit_Recursive_Mutable_Load permission is described, which, if not present, causes further capabilities loaded via that capability to be loaded without store permissions (see Section D.6).
- We have added a new experimental `CLoadTags` instruction that allows tags to be loaded for a cache line without pulling data into the cache.
- A new experimental *sealed entry capability* feature is described, which permits entry via jump but otherwise do not allow dereferencing (Section D.12). These are similar to enter capabilities from the M-Machine [18], and could provide utility in providing further constraints on capability use for the purposes of memory protection – e.g., in the implementation of C++ v-tables.
- A new experimental *memory type token* feature is described, which provides an alternative mechanism to object types within pairs of sealed capabilities (Section D.14).

7.0-ALPHA3 This version of the *CHERI Instruction-Set Architecture* is an interim version distributed for review by DARPA and our collaborators:

- The CHERI Concentrate capability compression format is now documented, with a more detailed rationale section than the prior CHERI-128 section.
- The `CLCBI` (Capability Load Capability with Big Immediate) instruction, which accelerates position-independent access to global variables, is no longer considered experimental.
- The architecture-neutral description of tagged memory has been clarified.
- The maximum supported lengths for both compressed and uncompressed capabilities has been updated: 2^{64} for 128-bit +capabilities, and $2^{64}-1$ for 256-bit capabilities.
- It is clarified that `CLoadTags` instruction must provide cache coherency consistent with other load instructions. We recommend "non-temporal" behavior, in which unnecessary cache-line fills are avoided to limit cache pollution during revocation.

- We now define the object type for unsealed capabilities, returned by the CGetType instruction, as $2^{64} - 1$ rather than 0.
- An experimental section has been added on how CHERI capabilities might compose with memory-versioning schemes such as Sparc ADI and Arm MTE (see Section D.9).
- Pseudocode throughout the CHERI ISA specification is now generated from our Sail formal model of the CHERI-MIPS ISA [8].
- The Glossary has been updated for CHERI ISAv7 changes including CHERI-RISC-V, split vs. merged register files, capabilities for physical addresses, and special capability registers.
- Capability exception codes are now shared across architectures.
- CHERI-RISC-V now includes capability-relative floating-point load and store instructions. We have clarified that existing RISC-V floating-point load and store instructions are constrained by **DDC**.
- CHERI-RISC-V now throws exceptions, rather than clearing tags, when non-monotonic register-to-register capability operations are attempted.
- While a specific encoding-mode transition mechanism is not yet specified for CHERI-RISC-V, candidate schemes are described and compared in greater detail.
- CHERI-RISC-V's "capability encoding mode" now has different impacts for uncompressed instructions vs. compressed instructions: In the compressed ISA, jump instructions also become capability relative.
- CHERI-RISC-V page-table entries now contain a "capability dirty bit" to assist with tracking the propagation of capabilities.
- Throwing an exception on an out-of-bounds capability-relative jump rather than on the target fetch is now more clearly explained: This improves debuggability by maintaining precise information about context state on jump, whereas after the jump, bounds may not be representable due to capability compression. When an inappropriate **EPCC** is installed, the exception will still be thrown on instruction fetch.
- A new **ErrorEPCC** special register has been defined, to assist with exceptions thrown within exception handlers; its behavior is modeled on the existing MIPS **ErrorEPC** special register.

7.0-ALPHA4 This version of the *CHERI Instruction-Set Architecture* is an interim version distributed for review by DARPA and our collaborators:

- We have added new instructions CSetAddr (Set capability address to value from register), CAndAddr (Mask address of capability – experimental), and CGetAndAddr (Move capability address to an integer register, with mask – experimental), which optimize common virtual-address-related operations in language runtimes such as WebKit's Javascript engine. These instructions cater better to a language mapping from C's intptr_t type to the virtual address, rather than offset, of a capability, which has been our focus previously. These complement the previously added CGetAddr that allows easier compiler access to a capability's virtual address.

- We have added two new experimental instructions, CRAM (Retrieve Mask to Align Capabilities to Precisely Representable Address) and CRRL (Round to Next Precisely Representable Value), which allow software to retrieve alignment information for the base and length for a proposed set of bounds.
- CMove, which was previously an assembler pseudo-operation for CIncOffset, is now a stand-alone instruction. This avoids the need to special case sealed capabilities when CIncOffset is used solely to move, not to modify, a capability.
- The names of the instructions CSetBoundsImmediate and CIncOffsetImmediate have been shortened to CSetBoundsImm and CIncOffsetImm.
- The instructions CCheckType and CCheckPerm have been deprecated, as they have not proven to be particularly useful in implementing multi-protection-domain systems.
- We have added a new pseudo-operation, CAssertInBounds, described in Section 7.5.5, allows an exception to be thrown if the address of a capability is not within bounds.
- The instruction CCheckTag has now been assigned an opcode.
- We have revised the encodings of many instructions in our draft CHERI-RISC-V specification in Appendix C.
- We more clearly specify that when a special register write occurs to **EPC**, the result is similar to CSetOffset but with the tag bit stripped, in the event of a failure, rather than an exception being thrown.
- We have added a reference to our TaPP 2018 paper, *Pointer Provenance in a Capability Architecture*, which describes how architectural traces of pointer behavior, visible through the CHERI instruction set, can be analyzed to understand software and structure.
- We have added a reference to our ICCD 2018 paper, *CheriRTOS: A Capability Model for Embedded Devices*, which describes an embedded variant of CHERI using 64-bit capabilities for 32-bit addresses, and how embedded real-time operating systems might utilize CHERI features.
- We have revised our description of conventions for capability values, including when used as pointers, to hold integers, and for NULL value, to more clearly describe their use. We more clearly describe the requirements for the in-memory representation of capabilities, such as a zeroed NULL capability so that BSS behaves as desired. We provide more clear architecture-neutral explanations of pointer dereferencing, capability permissions and their composition, the namespaces protected by capability permissions, exception handling, exception priorities, virtual memory, and system reset. These definitions appear in Chapter 3. Chapter 4, which describes CHERI-MIPS, has been shortened as a variety of content has been made architectural neutral.

- More detailed rationale is provided for our composition of CHERI with MIPS exception handling.
- We are more careful to use the term "pointer" to refer to the C-language type, verses integer or capability values that maybe used by the compiler to implement pointers.
- With the advent of ISA variations utilizing a merged register file, we are more careful to differentiate integer registers from general-purpose registers, as general-purpose registers may also hold capabilities.
- We more clearly define the terms "upper bound" and "lower bound".
- We now more clearly describe the effects of our *principle of intentionality* on capability-aware instruction design in Section 3.6.
- We better describe the rationale for tagged capabilities in registers and memory, in contrast to cryptographic and probabilistic protections, in Section 8.2.
- We have made a number of improvements to the CHERI-x86-64 sketch, described in Chapter 6, to improve realism around trap handling and instruction design.
- We have rewritten our description of the interaction between CHERI and Direct Memory Access (DMA) in Section 3.8.4. to more clearly describe tag-stripping and capability-aware DMA options.

7.0 This version of the *CHERI Instruction-Set Architecture* is a full release of the Version 7 specification:

- We have now deprecated the CHERI-128 capability compression format, in favor of CHERI Concentrate.
- The RISC-V `AUIPC` instruction now returns a **PCC**-relative capability in the capability encoding mode.
- Capabilities now contain a **flags** field, which will hold state that can be changed without affecting privilege. Corresponding experimental `CGetFlags` and `CSetFlags` instructions have been added. These are described in greater detail in Section D.1.
- The capability encoding-mode bit in CHERI-RISC-V is specified as a bit in the **flags** field of a capability. The current mode is defined as the flag bit in the currently installed **PCC**. Design considerations and other potential options are described in Chapter 8.
- We now more explicitly describe the reset states of special and general-purpose capability registers for CHERI-MIPS and CHERI-RISC-V.
- Compressed capabilities now contain a dedicated **otype** field that always holds an object type (see sections 2.3.7 and 3.3.1), rather than stealing bounds bits for object type when sealing. Now, any representable capability may be sealed. Several object type values are reserved for architectural experimentation (see table 3.2).
- More detail is provided regarding the integration of CHERI Concentrate with special registers, its alignment requirements, and so on.

- Initial discussion of a disjoint capability tree for physical addresses and hardware facilities using these has been added to the experimental appendix, in appendix D.16.
- Initial discussion of a hybrid 64/128-bit capability design has been added to the experimental appendix, in appendix D.15.
- We have added formal Sail instruction semantics for CHERI-RISC-V; this is currently in Appendix C.
- We have added a reference to our IEEE TC 2019 paper, *CHERI Concentrate: Practical Compressed Capabilities*, which describes our current approach to capability compression.
- We have added a reference to Alexandre Joannou's PhD dissertation, *High-performance memory safety: optimizing the CHERI capability machine*, which describes approaches to improving the efficiency of capability compression and tagged memory.

Appendix B

CHERI-MIPS ISA Quick Reference

This appendix provides a quick reference for CHERI-MIPS instruction encodings excluding experimental instructions (see Appendix D).

B.1 Current Encodings

The following encodings are correct for implementations that exist at the time of this document's publication.

B.1.1 Capability-Inspection Instructions

31 26	25 21	20 16	15 11	10 6	5 0	
0x12	0x0	rd	cb	0x0	0x3f	CGetPerm rd, cb
0x12	0x0	rd	cb	0x1	0x3f	CGetType rd, cb
0x12	0x0	rd	cb	0x2	0x3f	CGetBase rd, cb
0x12	0x0	rd	cb	0x3	0x3f	CGetLen rd, cb
0x12	0x0	rd	cb	0x4	0x3f	CGetTag rd, cb
0x12	0x0	rd	cb	0x5	0x3f	CGetSealed rd, cb
0x12	0x0	rd	cb	0x6	0x3f	CGetOffset rd, cb
0x12	0x0	cd	0x0	0x1f	0x3f	CGetPCC cd
0x12	0x0	cd	rs	0x7	0x3f	CGetPCCSetOffset cd, rs
0x12	0x0	rd	cb	0xf	0x3f	CGetAddr rd, cb
0x12	0x0	rd	cb	rs	0x23	CGetAndAddr rd, cb, rs
31 26	25 21	20 16	15 11	10 6	5 0	
0x12	0x0	rd	cb	0x12	0x3f	CGetFlags rd, cb

B.1.2 Capability-Modification Instructions

31 26	25 21	20 16	15 11	10 6	5 0	
0x12	0x0	cd	cs	ct	0xb	CSeal cd, cs, ct
0x12	0x0	cd	cs	ct	0xc	CUnseal cd, cs, ct

0x12	0x0	cd	cs	rt	0xd	CAndPerm cd, cs, rt
0x12	0x0	cd	cs	rt	0xe	CSetFlags cd, cs, rt
0x12	0x0	cd	cs	rt	0xf	CSetOffset cd, cs, rt
0x12	0x0	cd	cs	rt	0x8	CSetBounds cd, cs, rt
0x12	0x0	cd	cs	rt	0x9	CSetBoundsExact cd, cs, rt
0x12	0x14	cd	cb	length		CSetBoundsImm cd, cb, length
0x12	0x0	cd	cb	0xb	0x3f	CClearTag cd, cb
0x12	0x0	cd	cb	rt	0x11	CIncOffset cd, cb, rt
0x12	0x13	cd	cb	increment		CIncOffsetImm cd, cb, increment
0x12	0x0	cd	cb	ct	0x1d	CBuildCap cd, cb, ct
0x12	0x0	cd	cb	ct	0x1e	CCopyType cd, cb, ct
0x12	0x0	cd	cs	ct	0x1f	CCSeal cd, cs, ct
0x12	0x0	cd	cs	rs	0x22	CSetAddr cd, cs, rs
0x12	0x0	cd	cb	rs	0x24	CAndAddr cd, cb, rs

B.1.3 Pointer-Arithmetic Instructions

31 26	25 21	20 16	15 11	10 6	5 0	
0x12	0x0	rd	cb	cs	0x12	CToPtr rd, cb, cs
0x12	0x0	cd	cb	rs	0x13	CFromPtr cd, cb, rs
0x12	0x0	rt	cb	cs	0xa	CSub rt, cb, cs
0x12	0x0	cd	cs	0xa	0x3f	CMove cd, cs
0x12	0x0	cd	cs	rs	0x1b	CMOVZ cd, cs, rs
0x12	0x0	cd	cs	rs	0x1c	CMOVN cd, cs, rs

B.1.4 Pointer-Comparison Instructions

31 26	25 21	20 16	15 11	10 6	5 0	
0x12	0x0	rd	cb	cs	0x14	CEQ rd, cb, cs

0x12	0x0	rd	cb	cs	0x15	CNE rd, cb, cs
0x12	0x0	rd	cb	cs	0x16	CLT rd, cb, cs
0x12	0x0	rd	cb	cs	0x17	CLE rd, cb, cs
0x12	0x0	rd	cb	cs	0x18	CLTU rd, cb, cs
0x12	0x0	rd	cb	cs	0x19	CLEU rd, cb, cs
0x12	0x0	rd	cb	cs	0x21	CNEXEQ rd, cb, cs
0x12	0x0	rd	cb	cs	0x1a	CEXEQ rd, cb, cs

B.1.5 Exception-Handling Instructions

31 26	25 21	20 16	15 11	10 6	5 0	
0x12	0x0	rd	0x1	0x1f	0x3f	CGetCause rd
0x12	0x0	rs	0x2	0x1f	0x3f	CSetCause rs

B.1.6 Control-Flow Instructions

31 26	25 21	20 16	15 0	
0x12	0x9	cd	offset	CBTU cd, offset
0x12	0xa	cd	offset	CBTS cd, offset
0x12	0x11	cd	offset	CBEZ cd, offset
0x12	0x12	cd	offset	CBNZ cd, offset

31 26	25 21	20 16	15 11	10 6	5 0	
0x12	0x0	cb	0x3	0x1f	0x3f	CJR cb
0x12	0x0	cd	cb	0xc	0x3f	CJALR cd, cb

31 26	25 21	20 16	15 11	10 0	
0x12	0x05	cs	cb	selector	CCall cs, cb[, selector]
0x12	0x05	0x0	0x0	0x7ff	CReturn ; pseudo

B.1.7 Assertion Instructions

31 26	25 21	20 16	15 11	10 6	5 0	
0x12	0x0	cs	rt	0x8	0x3f	CCheckPerm cs, rt
0x12	0x0	cs	cb	0x9	0x3f	CCheckType cs, cb
0x12	0x0	cs	0x6	0x1f	0x3f	CCheckTag cs
0x12	0x0	rd	cb	ct	0x20	CTestSubset rd, cb, ct

B.1.8 Special-Purpose Register access Instructions

0x12	0x0	cd	sel	0xd	0x3f	CReadHwr cd, selector
0x12	0x0	cb	sel	0xe	0x3f	CWriteHwr cb, selector

B.1.9 Fast Register-Clearing Instructions

31 26	25 21	20 16	15 0	
0x12	0xf	0x0	mask	ClearLo mask
0x12	0xf	0x1	mask	ClearHi mask
0x12	0xf	0x2	mask	CClearLo mask
0x12	0xf	0x3	mask	CClearHi mask
0x12	0xf	0x4	mask	FPClearLo mask
0x12	0xf	0x5	mask	FPClearHi mask

B.1.10 Adjusting to Compressed Capability Precision Instructions

31 26	25 21	20 16	15 11	10 6	5 0	
0x12	0x0	rt	rs	0x10	0x3f	CRoundRepresentableLength rs, rd
0x12	0x0	rt	rs	0x11	0x3f	CRepresentableAlignmentMask rs, rd

B.1.11 Memory-Access Instructions

31 26	25 21	20 16	15 11	10 0	
0x3e	cs	cb	rt	offset	CSC cs, rt, offset(cb)
0x36	cs	cb	rt	offset	CLC cd, rt, offset(cb)
0x1d	cs	cb	offset		CLCBI cd, offset(cb)

31 26	25 21	20 16	15 11	10 3	2	1 0	
0x32	rd	cb	rt	offset	*s*	*t*	CL*x* rd, rt, offset(cb)
0x32	rd	cb	rt	offset	1	0	CLB rd, rt, offset(cb)
0x32	rd	cb	rt	offset	1	1	CLH rd, rt, offset(cb)
0x32	rd	cb	rt	offset	1	2	CLW rd, rt, offset(cb)
0x32	rd	cb	rt	offset	0	0	CLBU rd, rt, offset(cb)
0x32	rd	cb	rt	offset	0	1	CLHU rd, rt, offset(cb)
0x32	rd	cb	rt	offset	0	2	CLWU rd, rt, offset(cb)

0x32	rd	cb	rt	offset	0	3	CLD rd, rt, offset(cb)
0x3a	rs	cb	rt	offset	0	*t*	CS*x* rs, rt, offset(cb)
0x3a	rs	cb	rt	offset	0	0	CSB rs, rt, offset(cb)
0x3a	rs	cb	rt	offset	0	1	CSH rs, rt, offset(cb)
0x3a	rs	cb	rt	offset	0	2	CSW rs, rt, offset(cb)
0x3a	rs	cb	rt	offset	0	3	CSD rs, rt, offset(cb)

B.1.12 Atomic Memory-Access Instructions

31 26	25 21	20 16	15 11	10 6		3	2	1 0	
0x12	0x10	cd	cb			0xf			CLLC cd, cb
0x12	0x10	cs	cb	rd		0x7			CSCC rd, cs, cb
0x12	0x10	rd	cb			1	*s*	*t*	CLL*x* rd, cb
0x12	0x10	rd	cb			1	1	0	CLLB rd, cb
0x12	0x10	rd	cb			1	1	1	CLLH rd, cb
0x12	0x10	rd	cb			1	1	2	CLLW rd, cb
0x12	0x10	rd	cb			1	0	0	CLLBU rd, cb
0x12	0x10	rd	cb			1	0	1	CLLHU rd, cb
0x12	0x10	rd	cb			1	0	2	CLLWU rd, cb
0x12	0x10	rd	cb			1	0	3	CLLD rd, cb
0x12	0x10	rs	cb	rd		0		*t*	CSC*x* rd, cb
0x12	0x10	rs	cb	rd		0		0	CSCB rd, cb
0x12	0x10	rs	cb	rd		0		1	CSCH rd, cb
0x12	0x10	rs	cb	rd		0		2	CSCW rd, cb
0x12	0x10	rs	cb	rd		0		3	CSCD rd, cb

B.1.13 Encoding Summary

All three-register-operand, non-memory-accessing CHERI-MIPS instructions use the following encoding:

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	r1	r2	r3	func

	000	001	010	011	100	101	110	111
000	CGetPerm*	CGetType*	CGetBase*	CGetLen*	CGetCause*	CGetTag*	CGetSealed*	CGetPCC*
001	CSetBounds	CSetBoundsExact	CSub	CSeal	CUnseal	CAndPerm	CSetFlags	CSetOffset
010	UNUSED	CIncOffset	CToPtr	CFromPtr	CEQ	CNE	CLT	CLE
011	CLTU	CLEU	CEXEQ	CMovN**	CMovZ**	CBuildCap	CCopyType	CCSeal
100	CTestSubset	CNEXEQ	CSetAddr**	CGetAndAddr**	CAndAddr**	UNUSED	UNUSED	UNUSED
101	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
110	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
111	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	Two Op†

* Deprecated encoding for instruction

** Reserved instruction slot for future opcode

† This value is used for two-operand instructions.

All two-operand instructions are of the following form:

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	r1	r2	func	0x3f

	000	001	010	011	100	101	110	111
00	CGetPerm	CGetType	CGetBase	CGetLen	CGetTag	CGetSealed	CGetOffset	CGetPCCSetOffset
01	CCheckPerm	CCheckType	CMove	CClearTag	CJALR	CReadHwr	CWriteHwr	CGetAddr
10	CRRL	CRAM	CGetFlags	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
11	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	CLoadTags*	One Op†

* This instruction accesses tag memory.

† This value is used for one-operand instructions.

All one-operand instructions are of the following form:

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	r1	func	0x1f	0x3f

	000	001	010	011	100	101	110	111
00	CGetPCC	CGetCause	CSetCause	CJR	CGetCID†	CSetCID†	CCheckTag	UNUSED
01	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
10	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
11	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED

† Opcode may change

B.2 Deprecated Encodings

The following encodings were present in prior CHERI ISA versions, but have been deprecated.

B.2.1 Capability-Inspection Instructions

31 26	25 21	20 16	15 11	10 3	2 0	
0x12	0x0	rd	cb		0x0	CGetPerm rd, cb
0x12	0x0	rd	cb		0x1	CGetType rd, cb

0x12	0x0	rd	cb			0x2	CGetBase rd, cb
0x12	0x0	rd	cb			0x3	CGetLen rd, cb
0x12	0x0	rd	cb			0x5	CGetTag rd, cb
0x12	0x0	rd	cb			0x6	CGetSealed rd, cb
0x12	0x0d	rd	cb			0x2	CGetOffset rd, cb
0x12	0x0	cd	0x0	0x1f	0x3f		CGetPCC cd
0x12	0x0	cd	rs	0x7	0x3f		CGetPCCSetOffset cd, rs

B.2.2 Capability-Modification Instructions

31 26	25 21	20 16	15 11	10 6	5 0	
0x12	0x02	cd	cs	ct		CSeal cd, cs, ct
0x12	0x03	cd	cs	ct		CUnseal cd, cs, ct

31 26	25 21	20 16	15 11	10 6	5 3	2 0	
0x12	0x04	cd	cb	rt		0x0	CAndPerm cd, cb, rt
0x12	0x04	cd	cb			0x5	CClearTag cd, cb
0x12	0x0d	cd	cb	rt		0x0	CIncOffset cd, cb, rt
0x12	0x13	cd	cb	increment			CIncOffsetImm cd, cb, increment
0x12	0x0d	cd	cb	rt		0x1	CSetOffset cd, cb, rt
0x12	0x01	cd	cb	rt			CSetBounds cd, cb, rt
0x12	0x0	cd	cb	rt	0x9		CSetBoundsExact cd, cb, rt
0x12	0x14	cd	cb	length			CSetBoundsImm cd, cb, length

B.2.3 Pointer-Arithmetic Instructions

31 26	25 21	20 16	15 11	10 6	5 0		
0x12	0x0c	rd	cb	ct			CToPtr rd, cb, ct
0x12	0x04	cd	cb	rt		0x7	CFromPtr cd, cb, rt
0x12	0x0	rt	cb	ct	0xa		CSub rt, cb, ct

B.2.4 Pointer-Comparison Instructions

31 26	25 21	20 16	15 11	10 6	5 3	2 0	
0x12	0x0e	rd	cb	ct		0	CEQ rd, cb, ct

0x12	0x0e	rd	cb	ct		1	CNE rd, cb, ct
0x12	0x0e	rd	cb	ct		2	CLT rd, cb, ct
0x12	0x0e	rd	cb	ct		3	CLE rd, cb, ct
0x12	0x0e	rd	cb	ct		4	CLTU rd, cb, ct
0x12	0x0e	rd	cb	ct		5	CLEU rd, cb, ct
0x12	0x0e	rd	cb	ct		6	CEXEQ rd, cb, ct

B.2.5 Exception-Handling Instructions

31 26	25 21	20 16	15 11	10 6	5 3	2 0	
0x12	0x0	rd	0x0			0x4	CGetCause rd
0x12	0x04	0x0	0x0	rt		0x4	CSetCause rd

B.2.6 Control-Flow Instructions

31 26	25 21	20 16	15 0	
0x12	0x09	cd	offset	CBTU cd, offset
0x12	0x0a	cd	offset	CBTS cd, offset
0x12	0x11	cd	offset	CBEZ cd, offset
0x12	0x12	cd	offset	CBNZ cd, offset

31 26	25 21	20 16	15 11	10 6	5 0	
0x12	0x08		cb			CJR cb
0x12	0x07	cd	cb			CJALR cd, cb

31 26	25 21	20 16	15 11	10 0	
0x12	0x05	cs	cb	selector	CCall cs, cb[, selector]
0x12	0x06				CReturn

B.2.7 Assertion Instructions

31 26	25 21	20 16	15 11	10 6	5 3	2 0	
0x12	0x0b	cs		rt		0x0	CCheckPerm cs, rt
0x12	0x0b	cs	cb			0x1	CCheckType cs, cb

B.2.8 Fast Register-Clearing Instructions

31 26	25 21	20 16	15 0	
0x12	0xf	0x0	mask	ClearLo mask

0x12	0xf	0x1	mask	ClearHi mask
0x12	0xf	0x2	mask	CClearLo mask
0x12	0xf	0x3	mask	CClearHi mask
0x12	0xf	0x4	mask	FPClearLo mask
0x12	0xf	0x5	mask	FPClearHi mask

B.2.9 Deprecated and Removed Instructions

31 26	25 21	20 16	15			0	
0x32	rd	cb	rt	offset	1	3	CLLD rd, rt, offset(cb)
0x3a	rs	cb	rt	offset	1	3	CSCD rs, rt, offset(cb)

Appendix C

CHERI-RISC-V ISA Quick Reference (Draft)

C.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 format.

C.1.1 Capability-Inspection Instructions

31 25	24 20	19 15	14 12	11 7	6 0	
0x7f	0x0	cs1	0	rd	0x5b	CGetPerm rd, cs1
0x7f	0x1	cs1	0	rd	0x5b	CGetType rd, cs1
0x7f	0x2	cs1	0	rd	0x5b	CGetBase rd, cs1
0x7f	0x3	cs1	0	rd	0x5b	CGetLen rd, cs1
0x7f	0x4	cs1	0	rd	0x5b	CGetTag rd, cs1
0x7f	0x5	cs1	0	rd	0x5b	CGetSealed rd, cs1
0x7f	0x6	cs1	0	rd	0x5b	CGetOffset rd, cs1
0x7f	0x7	cs1	0	rd	0x5b	CGetFlags rd, cs1
0x7f	0xf	cs1	0	rd	0x5b	CGetAddr rd, cs1

C.1.2 Capability-Modification Instructions

31 25	24 20	19 15	14 12	11 7	6 0	
0xb	cs2	cs1	0	cd	0x5b	CSeal cd, cs1, cs2
0xc	rs2	cs1	0	cd	0x5b	CUnseal cd, cs1, rs2

0xd	rs2	cs1	0	cd	0x5b	CAndPerm cd, cs1, rs2
0xe	rs2	cs1	0	cd	0x5b	CSetFlags cd, cs1, rs2
0xf	rs2	cs1	0	cd	0x5b	CSetOffset cd, cs1, rs2
0x11	rs2	cs1	0	cd	0x5b	CIncOffset cd, cs1, rs2
increment		cs1	1	cd	0x5b	CIncOffsetImm cd, cs1, increment
0x8	rs2	cs1	0	cd	0x5b	CSetBounds cd, cs1, rs2
0x9	rs2	cs1	0	cd	0x5b	CSetBoundsExact cd, cs1, rs2
length		cs1	2	cd	0x5b	CSetBoundsImm cd, cs1, length
0x7f	0xb	cs1	0	cd	0x5b	CClearTag cd, cs1
0x1d	cs2	cs1	0	cd	0x5b	CBuildCap cd, cs1, cs2
0x1e	cs2	cs1	0	cd	0x5b	CCopyType cd, cs1, cs2
0x1f	cs2	cs1	0	cd	0x5b	CCSeal cd, cs1, cs2

C.1.3 Pointer-Arithmetic Instructions

31 25	24 20	19 15	14 12	11 7	6 0	
0x12	cs2	cs1	0	rd	0x5b	CToPtr rd, cs1, cs2
0x13	rs2	cs1	0	cd	0x5b	CFromPtr cd, cs1, rs2
0x14	cs2	cs1	0	rd	0x5b	CSub rd, cs1, cs2
0x7f	0xa	cs1	0	cd	0x5b	CMove cd, cs1
0x1	idx	cs1	0	cd	0x5b	CSpecialRW cd, cs1, idx

C.1.4 Control-Flow Instructions

0x7f	0xc	cs1	0	cd	0x5b	CJALR cd, cs1

0x7e	cs2	cs1	0	selector	0x5b	CCall cs1, cs2[, selector]
0x7e	cs2	cs1	0	0x1F	0x5b	CReturn ; pseudo

C.1.5 Assertion Instructions

0x20	cs2	cs1	0	rd	0x5b	CTestSubset rd, cs1, cs2

C.1.6 Fast Register-Clearing Instructions

31 25	24 20	19 18	17 15	14 12	11 7	6 0	
0x7f	0xd	q	m$_{[7:5]}$	0	m$_{[4:0]}$	0x5b	`Clear` q(uarter), m(ask)
0x7f	0x10	q	m$_{[7:5]}$	0	m$_{[4:0]}$	0x5b	`FPClear` q(uarter), m(ask)

C.1.7 Memory Loads with Explicit Address Type Instructions

These memory load instructions explicitly expect either capability addresses or integer offsets to **DDC**, with bounds coming either from cb or **DDC** respectively. For non-reserved loads, the encoding of bits 24 to 20 tries to follow the standard RISC-V mapping for the width and signedness of the memory access:

bit 24 0 to indicate non-reserved load.

bit 23 When 0, the load is DDC relative. Explicit capability is provided otherwise.

bit 22 When 0, the result of the load is sign-extended, and zero-extended otherwise.

bit 21-20 00 loads a byte, 01 loads a half-word, 10 loads a word, 11 loads a double-word.

For reserved loads (which require the A extension), the encoding of bits 24 to 20 tries to follow the standard RISC-V mapping for the width of the memory access:

bit 24 1 to indicate LR version of the load.

bit 23 When 0, the load is DDC relative. Explicit capability is provided otherwise.

bit 22-20 000 loads a byte, 001 loads a half-word, 010 loads a word, 011 loads a double-word, 100 loads a quad-word/capability.

Note that the RISC-V A extension (atomic) does not add unsigned versions of the LR instruction.
Note that the LQ{ddc, cap} instructions do not strictly follow this pattern.

31 25	24 20	19 15	14 12	11 7	6 0	
0x7d	0x00	rs1	0	rd	0x5b	`LBddc` rd, rs1
0x7d	0x01	rs1	0	rd	0x5b	`LHddc` rd, rs1
0x7d	0x02	rs1	0	rd	0x5b	`LWddc` rd, rs1
0x7d	0x03	rs1	0	rd	0x5b	`LDddc` rd, rs1
0x7d	0x17	rs1	0	{rd, cd}	0x5b	`LQddc` {rd, cd}, rs1
0x7d	0x04	rs1	0	rd	0x5b	`LBUddc` rd, rs1
0x7d	0x05	rs1	0	rd	0x5b	`LHUddc` rd, rs1

0x7d	0x06	rs1	0	rd	0x5b	LWUddc rd, rs1
0x7d	0x07	rs1	0	rd	0x5b	LDUddc rd, rs1
0x7d	0x08	cb	0	rd	0x5b	LBcap rd, cb
0x7d	0x09	cb	0	rd	0x5b	LHcap rd, cb
0x7d	0x0a	cb	0	rd	0x5b	LWcap rd, cb
0x7d	0x0b	cb	0	rd	0x5b	LDcap rd, cb
0x7d	0x1f	cb	0	{rd, cd}	0x5b	LQcap {rd, cd}, cb
0x7d	0x0c	cb	0	rd	0x5b	LBUcap rd, cb
0x7d	0x0d	cb	0	rd	0x5b	LHUcap rd, cb
0x7d	0x0e	cb	0	rd	0x5b	LWUcap rd, cb
0x7d	0x0f	cb	0	rd	0x5b	LDUcap rd, cb
0x7d	0x10	rs1	0	rd	0x5b	LRddc.B rd, rs1
0x7d	0x11	rs1	0	rd	0x5b	LRddc.H rd, rs1
0x7d	0x12	rs1	0	rd	0x5b	LRddc.W rd, rs1
0x7d	0x13	rs1	0	rd	0x5b	LRddc.D rd, rs1
0x7d	0x14	rs1	0	{rd, cd}	0x5b	LRddc.Q {rd, cd}, rs1
0x7d	0x18	cb	0	rd	0x5b	LRcap.B rd, cb
0x7d	0x19	cb	0	rd	0x5b	LRcap.H rd, cb
0x7d	0x1a	cb	0	rd	0x5b	LRcap.W rd, cb
0x7d	0x1b	cb	0	rd	0x5b	LRcap.D rd, cb
0x7d	0x1c	cb	0	{rd, cd}	0x5b	LRcap.Q {rd, cd}, cb

C.1.8 Memory Stores with Explicit Address Type Instructions

These memory store instructions explicitly expect either capability addresses or integer offsets to **DDC**, with bounds coming either from cb or **DDC** respectively. The encoding of bits 11 to 7 tries to follow the standard RISC-V mapping for the width of the memory access:

bit 11 When 1 with the A extension, SC version of the store.

bit 10 When 0, the store is DDC relative. Explicit capability is provided otherwise.

bit 9-7 000 stores a byte, 001 stores a half-word, 010 stores a word, 011 stores a double-word, 100 stores a quad-word/capability.

31 25	24 20	19 15	14 12	11 7	6 0	
0x7c	rs2	rs1	0	0x00	0x5b	SBddc rs2, rs1
0x7c	rs2	rs1	0	0x01	0x5b	SHddc rs2, rs1
0x7c	rs2	rs1	0	0x02	0x5b	SWddc rs2, rs1
0x7c	rs2	rs1	0	0x03	0x5b	SDddc rs2, rs1
0x7c	{rs2, cs}	rs1	0	0x04	0x5b	SQddc {rs2, cs}, rs1
0x7c	rs2	cb	0	0x08	0x5b	SBcap rs2, cb
0x7c	rs2	cb	0	0x09	0x5b	SHcap rs2, cb
0x7c	rs2	cb	0	0x0a	0x5b	SWcap rs2, cb
0x7c	rs2	cb	0	0x0b	0x5b	SDcap rs2, cb
0x7c	{rs2, cs}	cb	0	0x0c	0x5b	SQcap {rs2, cs}, cb
0x7c	rs2	rs1	0	0x10	0x5b	SCddc.B rs2, rs1
0x7c	rs2	rs1	0	0x11	0x5b	SCddc.H rs2, rs1
0x7c	rs2	rs1	0	0x12	0x5b	SCddc.W rs2, rs1
0x7c	rs2	rs1	0	0x13	0x5b	SCddc.D rs2, rs1
0x7c	{rs2, cs}	rs1	0	0x14	0x5b	SCddc.Q {rs2, cs}, rs1
0x7c	rs2	cb	0	0x18	0x5b	SCcap.B rs2, cb
0x7c	rs2	cb	0	0x19	0x5b	SCcap.H rs2, cb
0x7c	rs2	cb	0	0x1a	0x5b	SCcap.W rs2, cb
0x7c	rs2	cb	0	0x1b	0x5b	SCcap.D rs2, cb
0x7c	{rs2, cs}	cb	0	0x1c	0x5b	SCcap.Q {rs2, cs}, cb

C.2 Memory-Access via Capability with Offset Instructions

C.2.1 Memory-Access Instructions

31 25	24 20	19 15	14 12	11 7	6 0	
offset		rs1	0x2	cd	0xf	LC cd, rs1, offset
off[11:5]	cs2	rs1	0x4	off[0:4]	0x23	SC cs2, rs1, offset

C.2.2 Atomic Memory-Access Instructions

31 27	26	25	24 20	19 16	15	14 12	11	10 7	6 0	
0	aq	rl	0	cb	0	0x3	0	cd	0xb	CLR.C cd, cb
0	aq	rl	cs	cb	1	0x3	0	rd	0xb	CSC.C rd, cs, cb

C.3 Encoding Summary

CHERI-RISC-V general-purpose instructions use the 0x5b major opcode and most use the R instruction format. 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)

000	001	010	011	100	101	110	111
Three Op	CIncOffsetImm	CSetBoundsImm	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED

Three-operand encoding allocation (funct7 field)

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

0xfunc	rs1,cs2	cs1	0	cd	0x5b

	000	001	010	011	100	101	110	111
0000	UNUSED	CSpecialRW	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
0001	CSetBounds	CSetBoundsExact	UNUSED	CSeal	CUnseal	CAndPerm	CSetFlags	CSetOffset
0010	UNUSED	CIncOffset	CToPtr	CFromPtr	CSub	UNUSED	UNUSED	UNUSED
0011	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	CBuildCap	CCopyType	CCSeal
0100	CTestSubset	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
0101	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
0110	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
0111	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
1000	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
1001	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
1010	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
1011	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
1100	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
1101	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
1110	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
1111	UNUSED	UNUSED	UNUSED	UNUSED	Stores	Loads	CCall	Source & Dest

Stores encoding allocation (rd field)

Store instructions are of the following form:

31 25	24 20	19 15	14 12	11 7	6 0
0x7c	{rs2, cs}	{rs1, cb}	0	func	0x5b

	000	001	010	011	100	101	110	111
00	SBddc	SHddc	SWddc	SDddc	SQddc	UNUSED	UNUSED	UNUSED
01	SBcap	SHcap	SWcap	SDcap	SQcap	UNUSED	UNUSED	UNUSED
10	SCddc.B†	SCddc.H†	SCddc.W†	SCddc.D†	SCddc.Q†	UNUSED	UNUSED	UNUSED
11	SCcap.B†	SCcap.H†	SCcap.W†	SCcap.D†	SCcap.Q†	UNUSED	UNUSED	UNUSED

† The SC{ddc, cap}.{B, H, W, D, Q} instructions are only available when the RISC-V A extension (atomic) is present.

Loads encoding allocation (rs2 field)

Load instructions are of the following form:

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	func	{rs1, cb}	0	{rd, cd}	0x5b

	000	001	010	011	100	101	110	111
00	LBddc	LHddc	LWddc	LDddc	LBUddc	LHUddc	LWUddc	LDUddc‡
01	LBcap	LHcap	LWcap	LDcap	LBUcap	LHUcap	LWUcap	LDUcap‡
10	LRddc.B†	LRddc.H†	LRddc.W†	LRddc.D†	LRddc.Q†	UNUSED	UNUSED	LQddc
11	LRcap.B†	LRcap.H†	LRcap.W†	LRcap.D†	LRcap.Q†	UNUSED	UNUSED	LQcap

† The LR{ddc, cap}.{B, H, W, D, Q} instructions are only available when the RISC-V A extension (atomic) is present.

‡ LDU{ddc, cap} instructions are only available in RV128.

Source & Dest encoding allocation (rs2 field)

Source & Dest instructions are of the following form:

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0xfunc	cs1	0	rd	0x5b

	000	001	010	011	100	101	110	111
00	CGetPerm	CGetType	CGetBase	CGetLen	CGetTag	CGetSealed	CGetOffset	CGetFlags
01	UNUSED	UNUSED	CMove	CClearTag	CJALR	Clear	UNUSED	CGetAddr
10	FPClear	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
11	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	One Op†

† This value is used for one-operand instructions.

One-operand encoding allocation (rs1 field)

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

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x1f	0xfunc	0	rd	0x5b

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

C.4 CHERI-RISC-V Sail definitions

This section contains Sail definitions for the CHERI-RISC-V instructions. It is mostly identical to the CHERI-MIPS definitions except for some differences in style and naming to accommodate the existing RISC-V model.

C.4.1 Capability Inspection

CGetPerm

```
let capVal = readCapReg(cb);
X(rd) = EXTZ(getCapPerms(capVal));
RETIRE_SUCCESS
```

CGetType

```
let capVal = readCapReg(cb);
X(rd) = if   capVal.sealed
        then EXTZ(capVal.otype)
        else 0xffffffffffffffff;
RETIRE_SUCCESS
```

CGetBase

```
let capVal = readCapReg(cb);
X(rd) = to_bits(64, getCapBase(capVal));
RETIRE_SUCCESS
```

CGetLen

```
let capVal = readCapReg(cb);
let len65  = getCapLength(capVal);
X(rd) = to_bits(64, if len65 > MAX_U64 then MAX_U64 else len65);
RETIRE_SUCCESS
```

CGetTag

```
let capVal = readCapReg(cb);
X(rd) = EXTZ(capVal.tag);
RETIRE_SUCCESS
```

CGetSealed

```
let capVal = readCapReg(cb);
X(rd) = EXTZ(capVal.sealed);
RETIRE_SUCCESS
```

CGetOffset

```
let capVal = readCapReg(cb);
X(rd) = to_bits(64, getCapOffset(capVal));
RETIRE_SUCCESS
```

CGetFlags

```
let capVal = readCapReg(cb);
X(rd) = EXTZ(getCapFlags(capVal));
RETIRE_SUCCESS
```

CGetAddr

```
let capVal = readCapReg(cb);
X(rd) = to_bits(64, getCapCursor(capVal));
RETIRE_SUCCESS
```

C.4.2 Capability Modification

CAndPerm

```
let cb_val = readCapReg(cb);
let rt_val = X(rt);
if not (cb_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cb);
  RETIRE_FAIL
} else if cb_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, cb);
  RETIRE_FAIL
} else {
  let perms   = getCapPerms(cb_val);
  let newCap  = setCapPerms(cb_val, (perms & rt_val[30..0]));
  writeCapReg(cd, newCap);
  RETIRE_SUCCESS
}
```

CClearTag

```
let cb_val = readCapReg(cb);
writeCapReg(cd, {cb_val with tag=false});
RETIRE_SUCCESS
```

CSetFlags

```
let cb_val = readCapReg(cb);
let rt_val = X(rt);
if cb_val.tag & cb_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, cb);
  RETIRE_FAIL
} else {
  let newCap  = setCapFlags(cb_val, truncate(rt_val, num_flags));
  writeCapReg(cd, newCap);
  RETIRE_SUCCESS
}
```

CSetOffset

```
let cb_val = readCapReg(cb);
let rt_val = X(rt);
if cb_val.tag & cb_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, cb);
  RETIRE_FAIL
} else {
  let (success, newCap) = setCapOffset(cb_val, rt_val);
  if success then
      writeCapReg(cd, newCap)
  else
      writeCapReg(cd, int_to_cap(to_bits(64, getCapBase(cb_val)) + rt_val));
  RETIRE_SUCCESS
}
```

CIncOffset

```
let cb_val = readCapReg(cb);
let rt_val = X(rt);
if cb_val.tag & cb_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, cb);
  RETIRE_FAIL
} else {
  let (success, newCap) = incCapOffset(cb_val, rt_val);
  if success then
      writeCapReg(cd, newCap)
  else
      writeCapReg(cd, int_to_cap(cb_val.address + rt_val));
  RETIRE_SUCCESS
}
```

CIncOffsetImmediate

```
let cb_val = readCapReg(cb);
let imm64 : bits(64) = EXTS(imm);
if cb_val.tag & cb_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, cb);
  RETIRE_FAIL
} else {
  let (success, newCap) = incCapOffset(cb_val, imm64);
  if success then
      writeCapReg(cd, newCap)
  else
      writeCapReg(cd, int_to_cap(cb_val.address + imm64));
  RETIRE_SUCCESS
}
```

CSetBounds

```
let cb_val = readCapReg(cb);
let rt_val = unsigned(X(rt));
let cursor = getCapCursor(cb_val);
let base   = getCapBase(cb_val);
let top    = getCapTop(cb_val);
let newTop = cursor + rt_val;
if not (cb_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cb);
  RETIRE_FAIL
} else if cb_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, cb);
  RETIRE_FAIL
} else if cursor < base then {
  handle_cheri_reg_exception(CapEx_LengthViolation, cb);
  RETIRE_FAIL
} else if newTop > top then {
  handle_cheri_reg_exception(CapEx_LengthViolation, cb);
  RETIRE_FAIL
} else {
  let (_, newCap) = setCapBounds(cb_val, to_bits(64, cursor), to_bits(65, newTop));
  writeCapReg(cd, newCap); /* ignore exact */
  RETIRE_SUCCESS
}
```

CSetBoundsExact

```
let cb_val = readCapReg(cb);
let rt_val = unsigned(X(rt));
let cursor = getCapCursor(cb_val);
let base   = getCapBase(cb_val);
let top    = getCapTop(cb_val);
let newTop = cursor + rt_val;
if not (cb_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cb);
  RETIRE_FAIL
} else if cb_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, cb);
  RETIRE_FAIL
} else if cursor < base then {
  handle_cheri_reg_exception(CapEx_LengthViolation, cb);
  RETIRE_FAIL
} else if newTop > top then {
  handle_cheri_reg_exception(CapEx_LengthViolation, cb);
  RETIRE_FAIL
} else {
  let (exact, newCap) = setCapBounds(cb_val, to_bits(64, cursor), to_bits(65,
      newTop));
  if not (exact) then {
    handle_cheri_reg_exception(CapEx_InexactBounds, cb);
    RETIRE_FAIL
  } else {
    writeCapReg(cd, newCap);
    RETIRE_SUCCESS
  }
}
```

CSetBoundsImmediate

```
let cb_val = readCapReg(cb);
let immU   = unsigned(imm);
let cursor = getCapCursor(cb_val);
let base   = getCapBase(cb_val);
let top    = getCapTop(cb_val);
let newTop = cursor + immU;
if not (cb_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cb);
  RETIRE_FAIL
} else if cb_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, cb);
  RETIRE_FAIL
} else if cursor < base then {
  handle_cheri_reg_exception(CapEx_LengthViolation, cb);
  RETIRE_FAIL
} else if newTop > top then {
  handle_cheri_reg_exception(CapEx_LengthViolation, cb);
  RETIRE_FAIL
} else {
  let (_, newCap) = setCapBounds(cb_val, to_bits(64, cursor), to_bits(65, newTop));
  writeCapReg(cd, newCap); /* ignore exact */
  RETIRE_SUCCESS
}
```

CSeal

```
let cs_val = readCapReg(cs);
let ct_val = readCapReg(ct);
let ct_cursor = getCapCursor(ct_val);
let ct_top    = getCapTop(ct_val);
let ct_base   = getCapBase(ct_val);
if not (cs_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cs);
  RETIRE_FAIL
} else if not (ct_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, ct);
  RETIRE_FAIL
} else if cs_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, cs);
  RETIRE_FAIL
} else if ct_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, ct);
  RETIRE_FAIL
} else if not (ct_val.permit_seal) then {
  handle_cheri_reg_exception(CapEx_PermitSealViolation, ct);
  RETIRE_FAIL
} else if ct_cursor < ct_base then {
  handle_cheri_reg_exception(CapEx_LengthViolation, ct);
  RETIRE_FAIL
} else if ct_cursor >= ct_top then {
  handle_cheri_reg_exception(CapEx_LengthViolation, ct);
  RETIRE_FAIL
} else if ct_cursor > max_otype then {
  handle_cheri_reg_exception(CapEx_LengthViolation, ct);
  RETIRE_FAIL
} else {
  let (success, newCap) = sealCap(cs_val, to_bits(24, ct_cursor));
  if not (success) then {
    handle_cheri_reg_exception(CapEx_InexactBounds, cs);
    RETIRE_FAIL
  } else {
    writeCapReg(cd, newCap);
    RETIRE_SUCCESS
  }
}
```

CUnseal

```
let cs_val = readCapReg(cs);
let ct_val = readCapReg(ct);
let ct_cursor = getCapCursor(ct_val);
if not (cs_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cs);
  RETIRE_FAIL
} else if not (ct_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, ct);
  RETIRE_FAIL
} else if not (cs_val.sealed) then {
  handle_cheri_reg_exception(CapEx_SealViolation, cs);
  RETIRE_FAIL
} else if ct_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, ct);
  RETIRE_FAIL
} else if ct_cursor != unsigned(cs_val.otype) then {
  handle_cheri_reg_exception(CapEx_TypeViolation, ct);
  RETIRE_FAIL
} else if not (ct_val.permit_unseal) then {
  handle_cheri_reg_exception(CapEx_PermitUnsealViolation, ct);
  RETIRE_FAIL
} else if ct_cursor <  getCapBase(ct_val) then {
  handle_cheri_reg_exception(CapEx_LengthViolation, ct);
  RETIRE_FAIL
} else if ct_cursor >= getCapTop(ct_val) then {
  handle_cheri_reg_exception(CapEx_LengthViolation, ct);
  RETIRE_FAIL
} else {
  writeCapReg(cd, {unsealCap(cs_val) with
    global=(cs_val.global & ct_val.global)
  });
  RETIRE_SUCCESS
}
```

CBuildCap

```
let cb_val = readCapRegDDC(cb);
let ct_val = readCapReg(ct);
let cb_base = getCapBase(cb_val);
let ct_base = getCapBase(ct_val);
let cb_top = getCapTop(cb_val);
let ct_top = getCapTop(ct_val);
let cb_perms = getCapPerms(cb_val);
let ct_perms = getCapPerms(ct_val);
let ct_offset = getCapOffset(ct_val);
if not (cb_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cb);
  RETIRE_FAIL
} else if cb_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, cb);
  RETIRE_FAIL
} else if ct_base < cb_base then {
  handle_cheri_reg_exception(CapEx_LengthViolation, cb);
  RETIRE_FAIL
} else if ct_top > cb_top then {
  handle_cheri_reg_exception(CapEx_LengthViolation, cb);
  RETIRE_FAIL
} else if ct_base > ct_top then { /* check for length < 0 - possible because ct
    might be untagged */
  handle_cheri_reg_exception(CapEx_LengthViolation, ct);
  RETIRE_FAIL
} else if (ct_perms & cb_perms) != ct_perms then {
  handle_cheri_reg_exception(CapEx_UserDefViolation, cb);
  RETIRE_FAIL
} else {
  let (exact, cd1) = setCapBounds(cb_val, to_bits(64, ct_base), to_bits(65, ct_top)
      );
  let (representable, cd2) = setCapOffset(cd1, to_bits(64, ct_offset));
  let cd3 = setCapPerms(cd2, ct_perms);
  {
    assert(exact, "CBuildCap: setCapBounds was not exact"); /* base and top came
        from ct originally so will be exact */
    assert(representable, "CBuildCap: offset was not representable"); /* similarly
        offset should be representable XXX except for fastRepCheck */
    writeCapReg(cd, cd3);
    RETIRE_SUCCESS
  }
}
```

CCopyType

```
let cb_val = readCapReg(cb);
let ct_val = readCapReg(ct);
let cb_base = getCapBase(cb_val);
let cb_top  = getCapTop(cb_val);
let ct_otype = unsigned(ct_val.otype);
if not (cb_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cb);
  RETIRE_FAIL
} else if cb_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, cb);
  RETIRE_FAIL
} else if ct_val.sealed then {
  if ct_otype < cb_base then {
    handle_cheri_reg_exception(CapEx_LengthViolation, cb);
    RETIRE_FAIL
  } else if ct_otype >= cb_top then {
    handle_cheri_reg_exception(CapEx_LengthViolation, cb);
    RETIRE_FAIL
  } else {
    let (success, cap) = setCapOffset(cb_val, to_bits(64, ct_otype - cb_base));
    assert(success, "CopyType: offset is in bounds so should be representable");
    writeCapReg(cd, cap);
    RETIRE_SUCCESS
  }
} else {
  writeCapReg(cd, int_to_cap(0xffffffffffffffff));
  RETIRE_SUCCESS
}
```

CCSeal

```
let cs_val = readCapReg(cs);
let ct_val = readCapReg(ct);
let ct_cursor = getCapCursor(ct_val);
let ct_top    = getCapTop(ct_val);
let ct_base   = getCapBase(ct_val);
if not (cs_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cs);
  RETIRE_FAIL
} else if not (ct_val.tag) | (getCapCursor(ct_val) == MAX_U64) then {
  writeCapReg(cd, cs_val);
  RETIRE_SUCCESS
} else if cs_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, cs);
  RETIRE_FAIL
} else if ct_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, ct);
  RETIRE_FAIL
} else if not (ct_val.permit_seal) then {
  handle_cheri_reg_exception(CapEx_PermitSealViolation, ct);
  RETIRE_FAIL
} else if ct_cursor < ct_base then {
  handle_cheri_reg_exception(CapEx_LengthViolation, ct);
  RETIRE_FAIL
} else if ct_cursor >= ct_top then {
  handle_cheri_reg_exception(CapEx_LengthViolation, ct);
  RETIRE_FAIL
} else if ct_cursor > max_otype then {
  handle_cheri_reg_exception(CapEx_LengthViolation, ct);
  RETIRE_FAIL
} else {
  let (success, newCap) = sealCap(cs_val, to_bits(24, ct_cursor));
  if not (success) then {
    handle_cheri_reg_exception(CapEx_InexactBounds, cs);
    RETIRE_FAIL
  } else {
    writeCapReg(cd, newCap);
    RETIRE_SUCCESS
  }
}
```

C.4.3 Pointer Arithmetic

CToPtr

```
let ct_val = readCapRegDDC(ct);
let cb_val = readCapReg(cb);
if not (ct_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, ct);
  RETIRE_FAIL
} else if cb_val.tag & cb_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, cb);
  RETIRE_FAIL
} else {
  let ctBase = getCapBase(ct_val);
  /* Note: returning zero for untagged values breaks magic constants such as SIG_IGN
      */
  X(rd) = if not (cb_val.tag) then
            zeros()
          else
            to_bits(64, getCapCursor(cb_val) - ctBase);
  RETIRE_SUCCESS
}
```

CFromPtr

```
let cb_val = readCapRegDDC(cb);
let rt_val = X(rt);
if rt_val == 0x0000000000000000 then {
  writeCapReg(cd, null_cap);
  RETIRE_SUCCESS
} else if not (cb_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cb);
  RETIRE_FAIL
} else if cb_val.sealed then {
  handle_cheri_reg_exception(CapEx_SealViolation, cb);
  RETIRE_FAIL
} else {
  let (success, newCap) = setCapOffset(cb_val, rt_val);
  if success then
      writeCapReg(cd, newCap)
  else
      writeCapReg(cd, int_to_cap(to_bits(64, getCapBase(cb_val)) + rt_val));
  RETIRE_SUCCESS
}
```

CSub

```
let ct_val = readCapReg(ct);
let cb_val = readCapReg(cb);
X(rd) = to_bits(64, getCapCursor(cb_val) - getCapCursor(ct_val));
RETIRE_SUCCESS
```

CMove

```
writeCapReg(cd) = readCapReg(cb);
RETIRE_SUCCESS
```

AUIPCC (AUIPC in capability mode)

```
let ret = setCapAddrOrNull(PCC, PC + off);
writeCapReg(rd, ret);
RETIRE_SUCCESS
```

C.4.4 Control-Flow

CJALR

```
let cb_val = readCapReg(cb);
let newPC = [cb_val.address with 0 = bitzero]; /* clear bit zero as for RISCV JALR
    */
let newPC_int = unsigned(newPC);
let (cb_base, cb_top)  = getCapBounds(cb_val);
if not (cb_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cb);
  RETIRE_FAIL
} else if (cb_val.sealed) then {
  handle_cheri_reg_exception(CapEx_SealViolation, cb);
  RETIRE_FAIL
} else if not (cb_val.permit_execute) then {
  handle_cheri_reg_exception(CapEx_PermitExecuteViolation, cb);
  RETIRE_FAIL
} else if newPC_int < cb_base then {
  handle_cheri_reg_exception(CapEx_LengthViolation, cb);
  RETIRE_FAIL
} else if (newPC_int + min_instruction_bytes ()) > cb_top then {
  handle_cheri_reg_exception(CapEx_LengthViolation, cb);
  RETIRE_FAIL
} else if newPC[1] & ~(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 instrucitons */
  assert(success, "Link cap should always be representable.");
  writeCapReg(cd, linkCap);
  nextPC  = newPC;
  nextPCC = cb_val;
  RETIRE_SUCCESS
};
```

CCall

```
/* Partial implementation of CCall with checks in hardware, but raising a trap to
   perform trusted stack manipulation */
let cs_val = readCapReg(cs);
let cb_val = readCapReg(cb);
let cs_cursor = getCapCursor(cs_val);
if not (cs_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cs);
  RETIRE_FAIL
} else if not (cb_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cb);
  RETIRE_FAIL
} else if not (cs_val.sealed) then {
  handle_cheri_reg_exception(CapEx_SealViolation, cs);
  RETIRE_FAIL
} else if not (cb_val.sealed) then {
  handle_cheri_reg_exception(CapEx_SealViolation, cb);
  RETIRE_FAIL
} else if cs_val.otype != cb_val.otype then {
  handle_cheri_reg_exception(CapEx_TypeViolation, cs);
  RETIRE_FAIL
} else if not (cs_val.permit_execute) then {
  handle_cheri_reg_exception(CapEx_PermitExecuteViolation, cs);
  RETIRE_FAIL
} else if cb_val.permit_execute then {
  handle_cheri_reg_exception(CapEx_PermitExecuteViolation, cb);
  RETIRE_FAIL
} else if cs_cursor < getCapBase(cs_val) then {
  handle_cheri_reg_exception(CapEx_LengthViolation, cs);
  RETIRE_FAIL
} else if cs_cursor >= getCapTop(cs_val) then {
  handle_cheri_reg_exception(CapEx_LengthViolation, cs);
  RETIRE_FAIL
} else {
  handle_cheri_reg_exception(CapEx_CallTrap, cs);
  RETIRE_FAIL
}
```

CCallFast

```
/* Jump-like implementation of CCall that unseals arguments */
let cs_val = readCapReg(cs);
let cb_val = readCapReg(cb);
let newPC = [cs_val.address with 0 = bitzero]; /* clear bit zero as for RISCV JALR
    */
let newPC_int = unsigned(newPC);
let (cs_base, cs_top) = getCapBounds(cs_val);
if not (cs_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cs);
  RETIRE_FAIL
} else if not (cb_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cb);
  RETIRE_FAIL
} else if not (cs_val.sealed) then {
  handle_cheri_reg_exception(CapEx_SealViolation, cs);
  RETIRE_FAIL
} else if not (cb_val.sealed) then {
  handle_cheri_reg_exception(CapEx_SealViolation, cb);
  RETIRE_FAIL
} else if cs_val.otype != cb_val.otype then {
  handle_cheri_reg_exception(CapEx_TypeViolation, cs);
  RETIRE_FAIL
} else if not (cs_val.permit_ccall) then {
  handle_cheri_reg_exception(CapEx_PermitCCallViolation, cs);
  RETIRE_FAIL
} else if not (cb_val.permit_ccall) then {
  handle_cheri_reg_exception(CapEx_PermitCCallViolation, cb);
  RETIRE_FAIL
} else if not (cs_val.permit_execute) then {
  handle_cheri_reg_exception(CapEx_PermitExecuteViolation, cs);
  RETIRE_FAIL
} else if cb_val.permit_execute then {
  handle_cheri_reg_exception(CapEx_PermitExecuteViolation, cb);
  RETIRE_FAIL
} else if newPC_int < cs_base then {
  handle_cheri_reg_exception(CapEx_LengthViolation, cs);
  RETIRE_FAIL
} else if (newPC_int + min_instruction_bytes ()) > cs_top then {
  handle_cheri_reg_exception(CapEx_LengthViolation, cs);
  RETIRE_FAIL
} else if newPC[1] & ~(haveRVC()) then {
  handle_mem_exception(newPC,  E_Fetch_Addr_Align);
  RETIRE_FAIL
} else {
  C26 = unsealCap(cb_val);
  nextPC = newPC;
  nextPCC = unsealCap(cs_val);
  RETIRE_SUCCESS
}
```

CReturn

```
handle_cheri_reg_exception(CapEx_ReturnTrap, 0b11111); /* XXX what should correct
    reg number be? */
RETIRE_FAIL
```

C.4.5 Miscellaneous

CTestSubset

```
let cb_val = readCapRegDDC(cb);
let ct_val = readCapReg(ct);
let ct_top   = getCapTop(ct_val);
let ct_base  = getCapBase(ct_val);
let ct_perms = getCapPerms(ct_val);
let cb_top   = getCapTop(cb_val);
let cb_base  = getCapBase(cb_val);
let cb_perms = getCapPerms(cb_val);
let result = if cb_val.tag != ct_val.tag then
                0b0
             else if ct_base < cb_base then
                0b0
             else if ct_top > cb_top then
                0b0
             else if (ct_perms & cb_perms) != ct_perms then
                0b0
             else
                0b1;
X(rd) = EXTZ(result);
RETIRE_SUCCESS
```

CSpecialRW

```
let (specialExists, ro, priv, needASR) : (bool, bool, Privilege, bool) = match idx {
  0  => (true, true,  User, false),
  1  => (true, false, User, false),
  4  => (true, false, User, true),
  5  => (true, false, User, true),
  6  => (true, false, User, true),
  7  => (true, false, User, true),
  12 => (true, false, Supervisor, true),
  13 => (true, false, Supervisor, true),
  14 => (true, false, Supervisor, true),
  15 => (true, false, Supervisor, true),
  28 => (true, false, Machine, true),
  29 => (true, false, Machine, true),
  30 => (true, false, Machine, true),
  31 => (true, false, Machine, true),
  _  => (false, true, Machine, true)
};
if (not(specialExists)) then {
  handle_illegal();
  RETIRE_FAIL
} else if (ro & cs != 0) |
   (cur_privilege <_u priv) |
   (needASR & not(pcc_access_system_regs())) then {
  handle_cheri_cap_exception(CapEx_AccessSystemRegsViolation, 0b1 @ idx);
  RETIRE_FAIL
} else {
  let cs_val = readCapReg(cs);
  if (cd != 0) then {
    // read special cap
    let special_val : Capability = match idx {
      0  => {
        let (success, pcc) = setCapAddr(PCC, PC);
        assert (success, "PCC with offset PC should always be representable");
        pcc
      },
      1  => DDC,
      4  => UTCC,
      5  => UTDC,
      6  => UScratchC,
      7  => UEPCC, /* XXX should mask offset as per uepc etc? */
      12 => STCC,
      13 => STDC,
      14 => SScratchC,
      15 => SEPCC,
      28 => MTCC,
      29 => MTDC,
      30 => MScratchC,
      31 => MEPCC,
      _  => {assert(false, "unreachable"); undefined}
    };
    writeCapReg(cd, special_val);
  };
  if (cs != 0) then {
    // write special cap
    match idx {
      1  => DDC = cs_val,
      4  => UTCC = cs_val,   /* XXX should legalize mode? */
      5  => UTDC = cs_val,
      6  => UScratchC = cs_val,
      7  => UEPCC = cs_val, /* XXX should legalize offset as per uepc etc? */
      12 => STCC = cs_val,
      13 => STDC = cs_val,
      14 => SScratchC = cs_val,
      15 => SEPCC = cs_val,
      28 => MTCC = cs_val,
      29 => MTDC = cs_val,
      30 => MScratchC = cs_val,
      31 => MEPCC = cs_val,
      _  => assert(false, "unreachable")
    }
  };
  RETIRE_SUCCESS
}
```

ClearRegs

```
/*
if ((regset == CLo) | (regset == CHi)) then
  checkCP2usable();
*/
foreach (i from 0 to 7)
  if (m[i]) then
    match regset {
      GPRegs => X(8 * unsigned(q) + i) = zeros(),
      FPRegs => () /* XXX no F regs yet */
    };
RETIRE_SUCCESS
```

C.4.6 Loads

Loads of data share the following common logic:

```
function handle_load_data_via_cap(rd, cs, cap_val, vaddrBits, is_unsigned, width) =
     {
  let (base, top) = getCapBounds(cap_val);
  let vaddr = unsigned(vaddrBits);
  let size = word_width_bytes(width);
  let aq : bool = false;
  let rl : bool = false;
  if not(cap_val.tag) then {
    handle_cheri_cap_exception(CapEx_TagViolation, cs);
    RETIRE_FAIL
  } else if cap_val.sealed then {
    handle_cheri_cap_exception(CapEx_SealViolation, cs);
    RETIRE_FAIL
  } else if not (cap_val.permit_load) then {
    handle_cheri_cap_exception(CapEx_PermitLoadViolation, cs);
    RETIRE_FAIL
  } else if (vaddr + size) > top then {
    handle_cheri_cap_exception(CapEx_LengthViolation, cs);
    RETIRE_FAIL
  } else if vaddr < base then {
    handle_cheri_cap_exception(CapEx_LengthViolation, cs);
    RETIRE_FAIL
  } else if check_misaligned(vaddrBits, width) then {
    handle_mem_exception(vaddrBits, E_Load_Addr_Align);
    RETIRE_FAIL
  } else match translateAddr(vaddrBits, Read, Data) {
    TR_Failure(e) => { handle_mem_exception(vaddrBits, e); RETIRE_FAIL },
    TR_Address(addr) => process_load(rd, vaddrBits, mem_read(addr, size, aq, rl,
        false), is_unsigned)
  }
}
```

L[BHWD][U]ddc

```
let ddc_val = DDC;
let vaddr = ddc_val.address + X(rs);
handle_load_data_via_cap(rd, DDC_IDX, ddc_val, vaddr, is_unsigned, width)
```

L[BHWD][U]cap

```
let cap_val = readCapReg(cs);
let vaddr = cap_val.address;
handle_load_data_via_cap(rd, 0b0 @ cs, cap_val, vaddr, is_unsigned, width)
```

Loads of capabilities share the following common logic:

```
function handle_load_cap_via_cap(rd, cs, cap_val, vaddrBits) = {
  let (base, top) = getCapBounds(cap_val);
  let vaddr = unsigned(vaddrBits);
  let aq : bool = false;
  let rl : bool = false;
  if not(cap_val.tag) then {
    handle_cheri_cap_exception(CapEx_TagViolation, cs);
    RETIRE_FAIL
  } else if cap_val.sealed then {
    handle_cheri_cap_exception(CapEx_SealViolation, cs);
    RETIRE_FAIL
  } else if not (cap_val.permit_load) then {
    handle_cheri_cap_exception(CapEx_PermitLoadViolation, cs);
    RETIRE_FAIL
  } else if (vaddr + cap_size) > top then {
    handle_cheri_cap_exception(CapEx_LengthViolation, cs);
    RETIRE_FAIL
  } else if vaddr < base then {
    handle_cheri_cap_exception(CapEx_LengthViolation, cs);
    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, Data) {
    TR_Failure(e) => { handle_mem_exception(vaddrBits, e); RETIRE_FAIL },
    TR_Address(addr) => {
      let c = mem_read_cap(addr, aq, rl, false);
      match c {
        MemValue(v) => {writeCapReg(rd, v); RETIRE_SUCCESS},
        MemException(e) => {handle_mem_exception(vaddrBits, e); RETIRE_FAIL }
      }
    }
  }
}
```

LQddc

```
let ddc_val = DDC;
let vaddr = ddc_val.address + X(rs);
handle_load_cap_via_cap(rd, DDC_IDX, ddc_val, vaddr)
```

LQcap

```
let cap_val = readCapReg(cs);
let vaddr = cap_val.address;
handle_load_cap_via_cap(rd, 0b0 @ cs, cap_val, vaddr)
```

LC

```
let offset : xlenbits = EXTS(off12);
let (cap_val, vaddr, cause_regno) = get_cheri_cap_addr(rs1, offset);
handle_load_cap_via_cap(cd, cause_regno, cap_val, vaddr)
```

The following function selects between capability and DDC relative addressing in above, according to the capability mode flag:

```
function get_cheri_cap_addr (base_reg : regidx, offset : xlenbits) = {
  if (PCC.flag_cap_mode) then
    let base_cap = readCapReg(base_reg) in
    (base_cap, base_cap.address + offset, 0b0 @ base_reg)
  else
    let ddc = DDC in
    (ddc, ddc.address + X(base_reg) + offset, DDC_IDX);
}
```

C.4.7 Stores

Stores of data use the following common logic:

```
function handle_store_data_via_cap(rs, cs, cap_val, vaddrBits, width) = {
  let (base, top) = getCapBounds(cap_val);
  let vaddr = unsigned(vaddrBits);
  let size = word_width_bytes(width);
  let aq : bool = false;
  let rl : bool = false;
  if not(cap_val.tag) then {
    handle_cheri_cap_exception(CapEx_TagViolation, cs);
    RETIRE_FAIL
  } else if cap_val.sealed then {
    handle_cheri_cap_exception(CapEx_SealViolation, cs);
    RETIRE_FAIL
  } else if not (cap_val.permit_store) then {
    handle_cheri_cap_exception(CapEx_PermitStoreViolation, cs);
    RETIRE_FAIL
  } else if (vaddr + size) > top then {
    handle_cheri_cap_exception(CapEx_LengthViolation, cs);
    RETIRE_FAIL
  } else if vaddr < base then {
    handle_cheri_cap_exception(CapEx_LengthViolation, cs);
    RETIRE_FAIL
  } else if check_misaligned(vaddrBits, width) then {
    handle_mem_exception(vaddrBits, E_SAMO_Addr_Align);
    RETIRE_FAIL
  } else match translateAddr(vaddrBits, Write, Data) {
    TR_Failure(e) => { handle_mem_exception(vaddrBits, e); RETIRE_FAIL },
    TR_Address(addr) => {
      let eares : MemoryOpResult(unit) = mem_write_ea(addr, size, aq, rl, false);
      match (eares) {
        MemException(e) => { handle_mem_exception(addr, e); RETIRE_FAIL },
        MemValue(_) => {
          let rs_val = X(rs);
          let res : MemoryOpResult(bool) = match width {
            BYTE   => mem_write_value(addr, 1, rs_val[7..0],  aq, rl, false),
            HALF   => mem_write_value(addr, 2, rs_val[15..0], aq, rl, false),
            WORD   => mem_write_value(addr, 4, rs_val[31..0], aq, rl, false),
            DOUBLE => mem_write_value(addr, 8, rs_val,        aq, rl, false)
          };
          match (res) {
            MemValue(true)  => RETIRE_SUCCESS,
            MemValue(false) => internal_error("store got false from mem_write_value"
                ),
            MemException(e) => { handle_mem_exception(addr, e); RETIRE_FAIL }
          }
        }
      }
    }
  }
}
```

S[BHWD]ddc

```
let ddc_val = DDC;
let vaddr = ddc_val.address + X(rs);
handle_store_data_via_cap(rd, DDC_IDX, ddc_val, vaddr, width)
```

S[BHWD]cap

```
let cap_val = readCapReg(cs);
let vaddr = cap_val.address;
handle_store_data_via_cap(rs, 0b0 @ cs, cap_val, vaddr, width)
```

Stores of capabilities use the following common logic:

```
function handle_store_cap_via_cap(rs, cs, cap_val, vaddrBits) = {
  let (base, top) = getCapBounds(cap_val);
  let vaddr = unsigned(vaddrBits);
  let aq : bool = false;
  let rl : bool = false;
  if not(cap_val.tag) then {
    handle_cheri_cap_exception(CapEx_TagViolation, cs);
    RETIRE_FAIL
  } else if cap_val.sealed then {
    handle_cheri_cap_exception(CapEx_SealViolation, cs);
    RETIRE_FAIL
  } else if not (cap_val.permit_store) then {
    handle_cheri_cap_exception(CapEx_PermitStoreViolation, cs);
    RETIRE_FAIL
  } else if (vaddr + cap_size) > top then {
    handle_cheri_cap_exception(CapEx_LengthViolation, cs);
    RETIRE_FAIL
  } else if vaddr < base then {
    handle_cheri_cap_exception(CapEx_LengthViolation, cs);
    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, Data) {
    TR_Failure(e) => { handle_mem_exception(vaddrBits, e); RETIRE_FAIL },
    TR_Address(addr) => {
      let eares : MemoryOpResult(unit) = mem_write_ea_cap(addr, aq, rl, false);
      match (eares) {
        MemException(e) => { handle_mem_exception(addr, e); RETIRE_FAIL },
        MemValue(_) => {
          let rs_val = readCapReg(rs);
          let res : MemoryOpResult(bool) = mem_write_cap(addr, rs_val, aq, rl,
              false);
          match (res) {
            MemValue(true)  => RETIRE_SUCCESS,
            MemValue(false) => internal_error("store got false from mem_write_value"
                ),
            MemException(e) => { handle_mem_exception(addr, e); RETIRE_FAIL }
          }
        }
      }
    }
  }
}
```

SQddc

```
let ddc_val = DDC;
let vaddr = ddc_val.address + X(rs);
handle_store_cap_via_cap(rd, DDC_IDX, ddc_val, vaddr)
```

SQcap

```
let cap_val = readCapReg(cs);
let vaddr = cap_val.address;
handle_store_cap_via_cap(rs, 0b0 @ cs, cap_val, vaddr)
```

SC

```
let offset : xlenbits = EXTS(off12);
let (cap_val, vaddr, cause_regno) = get_cheri_cap_addr(rs1, offset);
handle_store_cap_via_cap(cs2, cause_regno, cap_val, vaddr)
```

Appendix D

Experimental Features and Instructions

This appendix describes experimental features and instructions proposed for possible inclusion in later versions of the CHERI ISA. These items for consideration include optimizations, new permissions, new compression formats, and overhauls of existing CHERI mechanisms. Some are relatively mature, and we anticipate their achieving a non-experimental status in the next version of the CHERI ISA specification (e.g., capability flags and temporal memory safety). Others arose as part of our more general design-space exploration, and we document these alternative approaches (e.g., indirect capabilities) or potential future avenues of investigation (e.g., linear capabilities). We present them here in roughly increasing order of complexity. The body of the appendix describes the rationale and approach for each experimental feature; specific instruction encodings and semantics may be found in Section D.18.

D.1 Capability Flags

We define two new experimental instructions, `cgetflags` and `csetflags`, to get and set bitwise flags on a capability. These flags are intended to affect the semantics of access, rather than impose access control, and thus (unlike bounds and permissions) do not have monotonicity properties. Currently, only one flag is defined: the capability encoding-mode flag for CHERI-RISC-V, which controls what interpretation opcodes are given when fetched via **PCC**. In the future, we may wish to use these flag fields for other purposes, such as to hint as to cache interactions for shared-memory rings, or to control the behavior of operations such as capability equality testing.

D.2 Capability Address and Length Rounding

Capability compression requires stronger alignment as allocation sizes increase. For infrequent allocations of large memory mappings, the software cost of calculating suitable alignment is small. However, stack allocations occur frequently and have less tolerance for arithmetic overheads. Further, it may be desirable for an architecture to support a range of compression parameters – for example, the bits invested in exponents, top, and bottom fields. In this case, having the architecture calculate requirements based on its specific parameterization would be beneficial. We propose two new instructions that allow the architecture to provide information to memory allocators regarding precision effects:

CRepresentableAlignmentMask (CRAM) The `CRAM` instruction accepts a proposed bounds length, and returns a mask suitable for use in aligning down the address of an allocation.

CRoundRepresentableLength (CRRL) The `CRRL` instruction accepts a proposed bounds length, and returns a rounded-up size that will be accepted by `CSetBoundsExact` without throwing an exception.

Collectively, these instructions can be used to efficiently calculate suitable base and length alignment, to permit exception-free bounds setting using `CSetBoundsExact`. They are intended to be well suited for use with dynamic stack allocation – e.g., using `alloca`, but also other types of allocation.

D.3 Fast Capability Subset Testing

When implementing revocation or garbage collection requiring fast scanning of memory for matching capabilities, significant numbers of instructions would be used to check whether a tested capability is a subset of a reference capability. We propose a new `CTestSubset` instruction that reduces this instruction count substantially; see page 458 for details.

D.4 Loading Multiple Tags Without Corresponding Data

Occasionally, one may wish to have access to tags without, or before, loading capabilities to registers. This would be potentially useful when paging to disk, for example, where one may wish to use DMA to transfer memory contents to the disk, but yet one must separately store the corresponding tags. In the absence of direct (i.e., read) access to the tags, the only alternative would be to involve the CPU in the bulk data copy and `CLC` all of the memory to be paged. Separately, when sweeping memory for revocation or garbage collection, being able to skip contiguous spans of non-capabilities in memory could dramatically reduce the DRAM traffic involved in sweeping.

Towards these ends, we introduce a `CLoadTags` instruction, which takes a capability to memory and loads several tag bits into a target register. The least-significant bit corresponds to the tag for the memory at the capability cursor; more significant bits correspond to tags of memory at larger addresses. The instruction requires that the capability bears `Permit_Load_Capability` and `Permit_Load` rights, that its cursor be suitably aligned, and that its bounds include all of the memory whose tags are to be read.

The design of our cache fabric allows us to instantiate this instruction with an efficient load of the tag bits from one cache-line worth of memory, or, for CHERI Concentrate, 8 tags at once. However, the width of the load is not architecturally specified – save that it must be a power of two, at least 1, and no more than the width of the registers. Software can easily discover the width used by any implementation by constructing an aligned array of capabilities in memory and observing the result of CLoadTags;[1] such probes need be done only rarely, at system or allocator startup. For multi-core or multi-processor systems with cache fabrics wherein cache lines are of different sizes, CLoadTags must behave as if all cores view the memory subsystem through the *smallest* cache line in the system.

Full details of the CLoadTags instruction may be found on page 452.

D.5 Capability Reconstruction

These additional experimental instructions can be used to efficiently reconstruct capabilities (e.g., when a program has been paged out to disk and then paged back in, and the operating system needs to reconstruct the capabilities that were originally in its address space). They also reduce the need for software to inspect the in-memory representation of capabilities, making software more robust to format changes.

Software should store or transit tags separately from the corresponding capability-sized and capability-aligned memory via a trustworthy medium. The ISA requires that tags be restored using a suitable authorizing capability through which it should have been possible to derive the same resulting tagged capability – that is, without violating capability monotonicity. A security review of these instructions is still in progress, and so they should not yet be considered part of the ISA or safe to implement. These instructions serve two purposes:

1. They allow efficient internalization of capabilities that have been stored or transferred via media that do not preserve tags. This functionality might be utilized when tags must be restored by the kernel's swap or compressed-memory pager, when migrating the memory of a virtual machine, when restoring a process snapshot, or by an in-address-space run-time linker.

2. They allow tags to be restored on capabilities in a manner that maintains architectural abstraction: software restoring tags need not encode the specifics of the in-memory capability representation, making that software less fragile in the presence of future use of reserved fields or changed semantics.

Capabilities can also be reconstructed using the current CGetBase, CGetLen, etc., instructions, examining those fields and then recreating them utilizing the corresponding CSetBounds instruction, and so on, but with reduced abstraction and substantially less efficiency.

Details of the proposed CBuildCap, CCopyType, and CCSeal instructions are deferred to appendix D.18.

D.6 Recursive Mutable Load Permission

Several software capability systems have exploited the use of immutable data structured to facilitate safe sharing (e.g., Joe-E [80]). CHERI capabilities can provide references through

[1] For a CHERI instantiation with 256-bit capability representations and 64-bit integer registers, the maximal alignment requirement for these probes is 512 bytes.

which stores are not permitted; however, because they can be refined and distributed throughout the system, simply holding a read-only reference is not sufficient to allow a consumer to ensure that no simultaneous access can occur to the same memory via another capability. Further, passing a read-only reference to memory does not ensure that further loads of capabilities from within that memory provide only read-only access to 'deep' data structures – e.g., linked lists.

Various software-level invariants could be used to improve confidence for both callers and callees. For example, the software runtime might make use of read-only MMU mappings for immutable data, and provide capabilities that clearly provide an indication that they refer to those read-only mappings – e.g., via use of a software-defined permission bit set only for such references, via use of reserved portions of the address space, sealed via a certain type, or checkable via a dynamic service operating in a trustworthy protection domain. In addition, memory could be allocated as mutable and its MMU mapping later modified to 'freeze' the contents, or by performing a revocation-like sweep to convert any extant store-enabled capabilities into load-only capabilities.

However, providing strong architectural invariants to software offers significant value. One idea we have considered is a new permission, Permit_Recursive_Mutable_Load, which if not present, clears store permissions and the recursive mutable load permission, on any capability loaded via a capability with this permission present.[2] A module may clear the store permissions and also clear Permit_Recursive_Mutable_Load on a capability before passing it to another module. Having done so, the originator is guaranteed that this passed capability could not then be used to mutate memory it directly describes (lacking store permissions) or memory transitively referenced therefrom, even if the latter capabilities, authorizing transitive access, bear some store permissions. This would not prevent temporal vulnerabilities associated with reallocation of the memory; subject to other invariants and safety properties, it might make it easier to construct safe references. In particular, this mechanism is likely to be of great utility to systems wishing to enforce the '*-property' ('no write down') of the model of Bell and La Padula [10].[3]

D.7 CHERI-64

Modern OSes and large-scale systems adopt 64-bit virtual address space to accommodate large applications, and to ease address space management and address randomization. However, 32-bit address space is still predominant for embedded systems, IoT devices, peripherals, DMA engines, etc., and 128-bit capabilities are without doubt not acceptable. To deploy CHERI

[2] The concept of such *transitively* read-only capabilities appears to have been first developed in KeyKOS, where such capabilities were termed 'sensory keys' [48]. While sensory keys were necessarily read-only, the descendent notion of the 'weak' access modifier in EROS could be applied to both read and write operations. When modifying reads, it behaves as described so far; attempts to store some input capability through a weak write-permitting capability resulted in a weakened version of the input capability being stored [116]. In the successor system Coyotos, 'weak' was once again made to imply read-only access [32, 117].

[3] Readers may be familiar with the infamous proof of Boebert [33] that "an unmodified capability machine" is unable to enforce this property. As CHERI distinguishes between capabilities and data, the proof is not directly applicable [84], and, indeed, one could imagine using trusted intermediate software to emulate the effects of Permit_Recursive_Mutable_Load, as proposed by Miller [83]. Despite that, Permit_Recursive_Mutable_Load is still of practical utility, as it is a light-weight, architecturally enforceable mechanism that avoids indirection.

in such systems, we have prototyped and developed the CHERI-64 format. In order to reuse some of the existing toolchain and software stack and to maintain compatibility, the encoding of CHERI-64 now is mostly a low-precision version for CHERI-128. In the future when the usage models change, the encoding might also change correspondingly.

D.7.1 CHERI-64 Encoding

Below are the encodings for 64-bit unsealed and sealed capabilities.

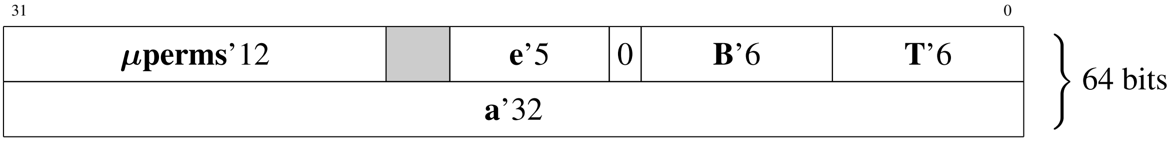

Figure D.1: Unsealed CHERI-64 memory representation of a capability

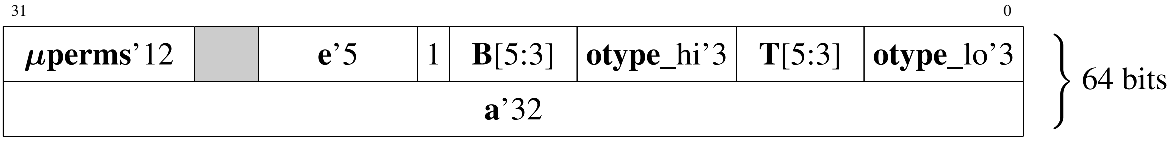

Figure D.2: Sealed CHERI-64 memory representation of a capability

For unsealed capabilities, the precision of T&B drops to 6 bits, and sealed capabilities further lose 3 bits for **otype**, making a total **otype** count of 64. There are still 2 unused bits available for expansion. To maintain compatibility, the hardware permission fields are the same with CHERI-128, while the software permission field is reduced to only one bit.

D.7.2 CHERI-64/MIPS-n32 ABI

CHERI-64 does not reuse the MIPS-n64 ABI since ABIs with 64-bit address space do not function well with the 32-bit address field in CHERI-64; for smaller devices and embedded systems, there is no need to move on to 64-bit. Instead, MIPS-n32 is the base ABI for CHERI-64 as n32 reduces the pointer size to 32 bits. However, please note that n32 is not the original o32 ABI, and the difference is not trivial. The major difference is that o32 is a native 32-bit ABI, whereas n32 runs on 64-bit MIPS but with limited address space. Except for the difference in the size of the address space, n32 shares more similarities with n64, both having the same calling convention, size of registers, stack alignment, etc. The handbook [38] provides a detailed tutorial of the n32 ABI.

Although n32 works within 32-bit address space, it still has access to 64-bit arithmetic instructions and 64-bit general-purpose integer registers. Therefore, CHERI-64 with n32 has a better opportunity to implement a merged register file, because merging the two register files does not increase the size; under CHERI-128 the width of the general-purpose integer register file needs to be doubled to accommodate capabilities if merged, and non-CHERI instructions cannot access the top half of the bits in the register file – as there is currently no 128-bit integer types in the 64-bit ISA.

D.8 Compressed Permission Representations

The model of Section 3.3.1 describes each permission as a separate bit. This has certain advantages, including the ability to describe *the* all-powerful capability, a uniform presentation, wherein the monotonic non-increase of rights is directly encoded by the monotonic operation of bitwise *and*, and a fast operational test for a given permission. However, in use and interpretation, the permission bits are not orthogonal, so one could aim for a compressed representation, freeing up bits for use as user permissions, or reserving them for future expansion of the ISA. We do not fully develop this story; instead, we merely indicate examples of redundancy in the abstract model, which may be useful to architects wishing to squeeze every last bit out of any particular representation.

The Global attribute, despite being enumerated as a permission, does not describe permissions to the memory or objects designated by a capability. Instead, it interacts with data storage permissions of other capabilities (via Permit_Store_Local_Capability). As such, it truly is orthogonal to the rest of the permission bits (though it remains 'monotonic' in the sense that clearing the Global permission results in a capability capable of participating in fewer operations).

Broadly speaking, there are three spaces of identifiers described within the CHERI capability system: virtual addresses, object types, and compartment identifiers. Rights concerning executability, loads, and stores apply only to capabilities describing virtual addresses, while the rights to (un)seal an object apply only to capabilities describing object types. The Permit_Set_CID permission applies only to capabilities describing compartment identifiers. This permits some reduction of encoding space.

Similar reduction in encoding space may be realized if one mandates that certain *user* permission bits are similarly applicable only to novel non-architectural spaces of identifiers (e.g., UNIX file descriptors). However, at present we consider the sealing mechanism more useful and flexible for the construction of such spaces of identifiers, as typically such identifiers are ultimately given meaning by some bytes in virtual memory, to which one may gain access by unsealing an object capability used as a reference.[4] However, the notion of other spaces is not entirely out of the question; *physical* addresses may prove to be a compelling example on some systems.

While Permit_CCall is *checked* only as part of `CCall`'s operation on sealed (i.e., object) capabilities, it is inherited from these sealed capabilities' precursors. That is, the present CHERI architecture permits the creation of regions of virtual address space that can be (subdivided and) sealed, but for which these derived object capabilities are not useful with `CCall` (just with `CUnseal`). The utility of such regions is perhaps not readily apparent, but any shift to make Permit_CCall apply only to object capabilities would require modification of the `CSeal` instruction

[4] Sadly, while sealed capabilities are almost exactly what one wants for file descriptors, because UNIX chose to type file descriptors as `int`, the conversion to use sealed capabilities will be broadly invasive, even if most of the changes will simply be to change the types.

and would slightly change the capability ontology.

Within the virtual-address-specific permissions, one finds several opportunities for compressing representations. First, many architectures consider writable-and-executable to be too dangerous to permit; applying this to CHERI's taxonomy would mean that the presence of Permit_Execute implied the absence of Permit_Store, Permit_Store_Capability, and Permit_Store_Local_Capability (see below). Further, granting Permit_Load_Capability effectively implies granting Permit_Load: CLC and CLLC would trap without the latter, but more substantially, a capability load of an untagged (in memory or via the paging hardware) 'should' result in a load of data transferred in to a capability register, albeit with the tag cleared. On the store side, Permit_Store_Local_Capability implies Permit_Store_Capability, which, in turn, implies Permit_Store. Taking all of these implications into consideration, one finds that there are 15 consistent states of the six virtual-address-space rights (Permit_Execute, Permit_Load, Permit_Load_Capability, Permit_Store, Permit_Store_Capability, Permit_Store_Local_Capability) considered, enabling a four-bit compressed representation.

Consider the powerful Access_System_Registers permission. Because this bit is meaningful only on capabilities used as a program counter, at the very least its presence rather directly implies Permit_Execute. Moreover, because this bit gates access to other architectural protection mechanisms, including those, such as the paging hardware, involved in *interpreting* (other) capabilities, it seems likely that this bit implies the ability to at least read, and likely mutate (or cause the mutation of), any other capability present in the system. (Admittedly, perhaps the ability to synthesize new capabilities from whole cloth would remain beyond the reach of code executing with Access_System_Registers, but given the far-reaching powers potentially conveyed, this hardly seems worth nitpicking.) As such, one may be justified in considering Access_System_Registers to be a single value in one's encoding of capability permissions, rather than an orthogonal bit.

D.8.1 A Worked Example of Type Segregation

Pushing a bit further on the 'spaces of identifiers' concept above, we can describe an alternative use of the 15 bits of μ**perms** available in the 128-bit encoding scheme of Section E.3.1. We continue to leave the 18-bit **otype** field where it stands, and we claim no new use of any reserved bits. Diagrams of the bit representations may be found in Figure D.3.

In all capabilities, we reserve three bits for uninterpreted user permissions, and four bits for the flow control detailed in Section D.13. One more bit distinguishes between virtual-address capabilities and all other types. We have thus far consumed 8 of the 15 permission bits.

For virtual-address capabilities (subsequently to be abbreviated as 'VA capabilities'), the remaining seven bits correspond one-to-one with memory-specific permissions. Specifically, they are: Permit_Execute (Ex), Permit_Load (L), Permit_Store (St), Permit_Load_Capability (LC), Permit_Store_Capability (SC), Permit_CCall (CC),[5] and Access_System_Registers (ASR). We have made no effort to eliminate redundancy in this particular segment of the encoding, but all the observations made above about these bits continue to hold.

[5] While any capability type can, in principle, be sealed and could be unsealed at CCall time, the fast CCall mechanism unseals only two capabilities, installing them as PCC and IDC. As such, it seems sensible to restrict CCall to operating only on VA capabilities, and so Permit_CCall is defined only therein.

Type	**Bit layout**								
Virtual Address	1	ASR	CC	SC	LC	St	L	Ex	user perms’3
Architectural Control	0	1				CID	Se	U	user perms’3
Guarded word	0	0	user perms’9						

Figure D.3: Bit-level representations of a type-segregated metadata-bit-packing scheme.

For non-virtual-address capabilities, we take one bit to distinguish *architectural control* capabilities from *guarded-word* capabilities. The latter are as might be expected: they are simply bounded (as per usual with CHERI capabilities) *integers*, protected by architectural provenance, monotonicity, and nonforgeability. Guarded-word capabilities confer no architectural authority, but may be of use to system software (e.g., for describing file descriptors). The remaining six bits are all permission-like (and are subject to manipulation via CAndPerm), but are otherwise uninterpreted by the hardware.[6]

Architectural control capabilities include the ability to seal and unseal particular object types, set the compartment identifier, and manipulate colors (again, as detailed in Section D.13). The remaining six bits are, again, all permission-like. Three are reserved for future use (not currently interpreted), while the other three correspond to the current Permit_Unseal (U), Permit_Seal (Se), and Permit_Set_CID (CID). No attempt has been made to further refine the type space, so we continue to architecturally conflate object types and compartment identifiers and rely on system software to maintain proper partitioning.

In this scheme, three primordial architectural roots should be created at system reset: one for virtual addresses, one for architectural control, and one for guarded words. All primordial capabilities should be unsealed, have all defined and user permission bits asserted, and cover the full space of their respective identifiers.

D.8.2 Type-segregation and Multiple Sealed Forms

Experiments with CheriOS have found that the increased alignment requirements for sealed capabilities induced by the original 128-bit compressed format are awkward (recall Section E.3.1). In particular, there is a desire to pass small sealed memory objects, with size (and so, ideal alignment) well below the requisite alignment size for sealing. Subsequent work has defined a different CHERI Concentrate form with a dedicated **otype** field, no need of a sealed bit, and no increased alignment requirements to make room or the **otype** bits. And so, the remainder of this subsection is largely mooted: all capabilities may be sealed in the new CHERI Concentrate format. We retain it in this document for interest and its possible applicability to implementers considering different capability encoding options.

[6] It may seem odd to deliberately create architecturally ‘useless’ tagged integers; it may seem as though they could simply be VA capabilities with all permission bits cleared. However, just because an agent has some rights to memory address 0x1234 does not imply that they have rights to the *integer* 0x1234, but monotonic action on a capability authorizing the former could result in one authorizing the latter in this hypothetical ‘all-permission-bits-zero’ encoding. The *separate provenance tree* of guarded-word capabilities distinguishes these: there is no monotonic mechanism to transmute one into the other.

Type	**Bit layout**										
Unsealed VA	1	0	0	ASR	CC	SC	LC	St	L	Ex	user perms'3
Sealed VA	1	1	SV	ASR	CC	SC	LC	St	L	Ex	user perms'3
Architectural Control	1	0	1					CID	Se	U	user perms'3
Unsealed guarded word	0	0	0	user perms'10							
Sealed guarded word	0	1	0	user perms'10							
Reserved	0		1								

Figure D.4: A variant of packed metadata including multiple sealed forms.

The small objects passed by CheriOS are never sealed as interior pointers. That is, the sealed forms are guaranteed to have offset zero (i.e., equal cursor and base addresses). This permits 10 bits of the B field to be transferred to the T field, offering much smaller alignment requirements. (Byte alignment remains possible until objects approach 1 mibibyte in length. Offsets need not be zero, but must be small, in the sense that they must be below 2^{e}.) The experimental architectural encoding presently requires stealing one of the two bits described in this document as reserved within a capability representation. Given the possible utility of this additional sealed form to the other provenance trees discussed above, it seems worthwhile to present a possible unified story.

For this example, we drop the ability to seal architectural control capabilities, as we do not think these will be passed as tokens; instead, we believe, should system programmers desire similar policies, they are free to indirect, i.e., to place architectural control capabilities into small regions of memory, seal the rights thereto, and pass that sealed capability instead of a sealed architectural control capability. This further removes concerns around the encoding of **otype**s and capability color changing permissions (to be discussed).

This illustrative encoding uses 17 bits: 15 from the former **μperms**, 1 from the former sealed flag, and 1 formerly reserved. Bit-field representations are shown in Figure D.4. For VA capabilities, the new 'Sealed Variant' (SV) flag, which is not a permission bit (and so not subject to manipulation by `CAndPerm`), distinguishes between the form with both T and B specified and the form with only T specified. We expect an architecture using this form to have two `CSeal`-like instructions, each generating one of the variants. For sealed guarded-word capabilities, we permit only the latter form, as we believe sealed guarded words are more likely to be used as tokens than as regions of integers. One-fourth of our type encoding values are reserved for future expansion.

D.8.3 `W^X` Saves A Bit

`W^X` ('W xor X') is a shorthand for the notion that no block of memory should be, at the same time, both writable and executable. Most implementations in hardware work within the MMU, and rely on the operating system to enforce the exclusivity of write and execute permissions.

From the view of application software, this means that a given pointer value has additional hidden state beyond its being mapped or unmapped. Applications on CHERI could, instead, structure the permissions within capabilities to enforce exclusivity of write and execute permissions, trading the stateful MMU protection for having multiple capabilities representing the two different rights.

Were we to push `W^X` on CHERI to an extreme, it could become a property of the capability encoding itself and, thereby, allow for more compact encoding of permissions. The existing eight-bit architectural permission field,

ASR	CC	SLC	SC	LC	St	L	Ex

could instead be re-coded as a 7-bit field, making the `W^X` explicit:

RX capability:	0	CC	▨	ASR	LC	Ex	L
RW capability:	1	CC	SLC	SC	LC	St	L

As in the type-segregation proposals, this design creates yet another split of architectural provenance roots: there must be two capabilities present at system startup, granting separate read-write and read-execute regions. Similarly, a single capability then could not express the total set of permissions that may be granted by, e.g., the *nix `mmap()` call; the API and consumers must be revised. (One hopes that relatively few consumers initially request (or later transition, via `mprotect()`, to having) both write and execute permissions.) It is not yet clear what additional challenges this split imposes on our goal of C compatibility.

There is some redundancy yet in this encoding, in that either RX or RW capabilities can be monotonically turned into read-only capabilities. One could imagine further segregation into a `R^W^X` taxonomy, but this seems especially likely to complicate C compatibility. Moreover, the obvious utility of RW capabilities and popularity of data constants adjacent to executable code (and thereby reachable using relative offsets from the instruction pointer) argue for permitting read permissions in both write and execute forms.

When and if combined with the compact coloring proposal below, the Permit_Store_Local_Cap (SLC) bit and its unused slot in the RX form would vanish.

D.9 Memory-Capability Versioning

Several existing architectures have responded to temporal safety issues in software by proposing to 'version' memory, embed versions into pointers, and require that the versions of the pointer and target match on each dereference. Two prominent examples are Oracle's SPARC's ADI/SSM [2] and ARM's MTE [1]. We conjecture that the combination of these ideas with CHERI would enhance both and continue to have reasonable performance overheads. Between these mechanisms, we can offer an attractive secure mitigation of temporal safety violations in untrusted code.

Specifically, we propose to use approximately four of the reserved bits in the capability metadata word in each memory-authorizing capability, together with an equal number of bits per ‘granule’ of physical memory, which we suggest to be roughly 64 bytes. (The proposed values give a spatial overhead equivalent to CHERI’s capability tags: one bit per 16 bytes.) This scheme leaves all virtual address bits in memory-authorizing capabilities intact, and thus does not reduce the application’s address space. To ensure that untrusted code cannot inappropriately re-version memory granules, we provide a simple model of authorization that does not require the intervention of supervisor software.

We divide memory-authorizing capabilities into two classes, versioned and unversioned, and introduce an instruction that derives a versioned capability from an unversioned one. The core of this protection mechanism is this: if a versioned capability is used to access a granule, the access succeeds only if (in addition to the existing CHERI permissions and bounds checks’ passing) the granule and intra-capability versions are equal. In the case of mismatch, an implementation must, at a minimum, cause data fetches to return 0, capability fetches to return untagged NULLs, stores to fail silently, and instruction fetches to trap. To improve the debugging experience, implementations may provide optional or mandatory traps on these fetch and stores as well.

Only unversioned capabilities can authorize the re-versioning of memory granules. Additionally, unversioned capabilities potentially authorize access regardless of the version of the granule being accessed. We expect that these will become closely held within subsystems that then exchange derived versioned capabilities with other subsystems; in general, the only subsystems holding unversioned capabilities are likely to appear to be allocators.

Versions are ‘sticky,’ in that any capability monotonically derived from a versioned progenitor will have the same version. Dually, derivations from unversioned capabilities are unversioned, unless the version is explicitly branded into the progeny.

D.9.1 Instructions

- `CStoreVersion` sets the version bits of a memory granule to the value given in a register operand; the authorizing capability must be unversioned and must authorize stores of both data and capabilities to the entire target granule. Setting the granule’s version to 0 will cause it to be accessible only to unversioned capabilities.

- `CFetchVersion` fetches the version bits of a memory granule; the authorizing capability must be unversioned, and must authorize data fetches from the entire target granule. A return of 0 indicates that the granule is accessible only via unversioned capabilities.

- `CGetVersion` copies the version field of a capability into a register. It is useful mostly for debugging and for maintaining an abstract interface to capabilities despite the encoded form bits’ being accessible to software.

- `CSetVersion` derives a versioned capability from an unversioned capability and a version value from a register operand. Attempting to set the version to 0 will trap. No other fields are modified in the derived copy. Attempting to make a versioned capability from a versioned one may succeed only if the desired and existing versions are equal, otherwise the result will have its tag cleared.[7]

[7] It may be sensible to always clear the tag or always trap, as well. We do not have a use case for the tagged

D.9.2 Use With System Software

Because there are only finitely many versions available, we envision that the *system software* will provide a *revocation* mechanism to de-tag all capabilities with mismatching versions. To minimize the testing required by this facility, it will test only the granule containing the *base* of each versioned capability it encounters; software engaging in version-based revocation should, nevertheless, re-version all (partially) contained granules so that derived capabilities with offset bases are also revoked. In a sense, granules exist because they are a sufficient and straightforward mechanism to capture spans of version information, not because we expect individual granules within a single segment authorized by a capability to be changed. Dually, objects with different lifetimes should not share granules; this results in much stronger alignment requirements for allocators, but the practical impact remains to be measured.

We do not specify the shape of the interface exposed for this facility; a traditional system call to the (privileged) kernel is one possibility for implementation, but more 'autonomic' approaches are feasible as well. We envision a global 'epoch' counter maintained by the kernel, stepping after every revocation pass. If software remembered the counter's value at the time each allocation came to have its current version, that software would know when all capabilities with their base in that allocation and of the wrong version had necessarily been destroyed: in the second epoch after re-versioning. Such a scheme would permit sharing work across many allocators desiring revocation within the same address space.

Because revocation may be done in the background, versions are intended to be used once between revocations. That is, software should not assume that it can restore an earlier version to re-authorize an existing capability, because at any moment the mismatched capability may have become de-tagged.

Whereas we conjecture that the minimum requirements given above for mismatched versions for loads and stores are sufficient to eliminate temporal safety issues, there remains the possibility of apparently *inducing* bugs in programs running under our new semantics. For example, if software attempts to (re)initialize an object using a stale capability, the memory will not be updated and may be reused in inconsistent state. Trapping on version mismatch would better expose such issues.

D.9.3 Microarchitectural Impact

The cache fabric must now store the version of each granule in each cache line (which, in the proposal above, is one, given 64-byte cache lines). Dereference operations must forward the capability's version field down to the cache fabric as well. The minimum requirements for version mismatch are, however, intended to remove the need to track store requests through the memory hierarchy. While precise traps on stores would require essentially a full read-modify-write cycle, the cache fabric may be able to raise *imprecise* traps well after accepting a store by tracking the tentative version bits until they can be checked against the authoritative version table.

result when-equal case.

D.10 Linear Capabilities

Linear capabilities are intended to support the implementation of operating-system and language-level linearity features, which ensure that at most one reference to an object is held at a time. This feature might be used to help support efficient memory reuse – e.g., by requiring that a reference to stack memory be 'returned' before a caller is able to reuse the memory. Architectural linearity does not prevent destruction of the reference, which may require slow-path behavior such as garbage collection, but can support strong invariants that would help avoid that behavior in the presence of compliant software. This architectural proposal has not yet been validated through implementation in architecture, microarchitecture, or software.

D.10.1 Capability Linearity in Architecture

We propose to add a new bit to the capability format marking a capability as *linear*. It could be that this is a permission (e.g., Permit_Non_Linear). However, as this feature changes a number of other aspects of capability behavior, we recommend not conflating this behavior with the permission mechanism, instead adding a new field.
Two new *linear move* instructions would be added:

Linear Load Capability Register (LLCR) This instruction loads a capability from memory into a register, atomically clearing the memory location [regardless of whether it loaded a linear capability?].

Linear Store Capability Register (LSCR) This instruction stores a capability from a register into memory, atomically clearing the register when a successful store takes place (e.g., if it does not trigger a page fault) [regardless of whether it stored a linear capability?].

The reason to introduce an explicit linear load is to avoid taking the cost of an atomic operation for every capability load dependent on whether the loaded capability is linear. A separate linear store instruction is not motivated by this concern, but would add symmetry, avoiding the need for store instructions to vary their behavior based on capability type.
A new Permit_Linear_Override permission is added, which controls how existing capability load and store instructions (e.g., CLC and CSC) interact with linear capabilities. If the permission is not present, then loaded linear capabilities will have their tag cleared when written into a register, and stored linear capabilities will have their tag cleared when written to memory. This behavior maintains linearity without changing the register or memory write-back behaviors of these instructions.
If Permit_Linear_Override is present on the capability being used to load or store non-linear capabilities, then linearity is violated, allowing both the in-register and in-memory capabilities to continue to be valid and marked as linear. This permission allows for privileged system software to violate linearity when, for example, implementing mechanisms such as Copy-on-Write (COW) in the the OS virtual-memory subsystem or debugging features.
To save instruction encoding space, we might limit these memory access instructions to be R-type with only a register-specified offset. This may be adequate if the instructions are infrequently used.

For register-to-register instructions, there are several options – in particular, when implementing capability-manipulation instructions such as CIncOffset and CSetOffset:

- We might make existing instructions remove the tag in register write back for linear capabilities, enforcing linearity by preventing duplication of linear capabilities.

- We might require that, when existing instructions operate on linear capabilities, they write back to their source register, enforcing linearity by avoiding duplication to a second register. This might be simplest microarchitecturally.

- We might add new explicitly linear variants of some existing instructions, which would enforce linearity by clearing the source register, preventing duplication.

In general, ensuring write-back to the same register is easy and cheap to check dynamically; it avoids the need to introduce a large number of new instructions offering near-identical behavior. It also avoids increasing the number of registers that must be written back by instructions. Additional concerns exist around the implementation of **PCC** as relates to CGetPCC, which normally duplicates a capability. Although undesirable, the natural design choice is to strip the tag when writing to the target register, if **PCC** is linear.

D.10.2 Capability Linearity in Software

The above architectural behavior means that, on the whole, software must be aware when handling linear capabilities; code must be generated specifically to use new linear load and store instructions, and to utilize other register-to-register instructions in a manner consistent with linearity. There are several specific implications that must be taken into account when writing system software or compilers:

- Linear capabilities must be explicitly identified via the source language – e.g., via types or qualifiers – so as to guide code generation. It might be desirable to utilize techniques such as symbol mangling to prevent accidents.

- Linear values cannot be properly preserved by ordinary stack loads and spills, so the compiler must take explicit action to prevent this from being necessary. This might also require static limitations on use of capabilities in the language.

- When linear capabilities are used and manipulated as pointers, it may be necessary to generate code quite differently, or to limit expressiveness. For example, implied pointer arithmetic when iterating using a pointer requires that the original pointer be destroyed, or that the pointer be left unmodified but accessed using an integer-register index. It is not yet clear to what extent this would interact with common C-language idioms.

- Some systems code must be linearity-oblivious, such as context-switching or VM code, and can employ Permit_Linear_Override to load and store ordinary and linear capabilities using non-linear loads and stores. However, it must assuredly not violate invariants of affected software, or else linearity may not be enforced.

- Many current C-language OS and library APIs may be linearity-unfriendly, as they frequently accept an existing pointer as an argument, but do not ‘return’ it to the caller. It may be desirable to have a specific set of extended APIs that are linearity-friendly – e.g., variants of `memcpy` that copy data into and out of linearly referenced memory. It is unclear whether this would extend to a broader suite of APIs, such as OS `read` and `write` system calls – and perhaps would imply polyinstantiation.

- Debugging tools would need to become aware of linearity so as to accurately display information about linear capabilities found in registers or memory. They might use Permit_Linear_Override to gain access to the full contents of the register with tag, but must still inspect capability fields suitably, and avoid the need to spill values. It is not clear how this would interact with current debugger internals.

In general, when linearity is violated, it will lead to loss of tags, preventing dereferences that violate invariants. It is not clear to what extent this would be easily debuggable. We can imagine having non-linear sequences generate an exception, but in some cases this may be microarchitecturally awkward.

Overall, it is not clear to what extent this proposal can interact well with real-world software designs, or to what extent it usefully supports new software behaviors. Key use cases motivating this design typically involve garbage collection avoidance: e.g., passing an stack pointer across protection-domain boundaries and checking that it is ‘returned’ before continuing, avoiding the need for a GC to sweep the recipient domain. But this does not necessarily alleviate the need to implement more complex behaviors such as GC in the event that the invariant is violated.

D.10.3 Related Work in Linear Capabilities

Skorstengaard et al. have concurrently developed ideas about linear capabilities [118], which focus on how to produce a memory-safe execution substrate over a CHERI-derived abstract capability instruction set. They are able to use linear capabilities to construct a temporally safe stack calling convention against the model. This allows formal proof of well-bracketed control flow and stack-frame encapsulation. However, their approach also relies on two further instructions not present in our current sketch: capability split and splice instructions allowing linear capabilities for stack subsets to be separated, delegated, returned, and rejoined. It is not yet clear to us whether these additional instructions are microarchitecturally realistic, especially in the presence of compressed capabilities.

The creators of the SAFE architecture [19] also propose that *linear pointers* could contribute to reasoning about concurrent memory use.

D.11 Indirect Capabilities

Indirect capabilities could support revocable or relocatable objects without modification of application executables. An indirect capability would be identified by the hardware as a pointer to the pointer to the data. That is, a load that takes as an address a capability that is marked as an indirect capability would load a capability from the base address of the indirect capability, and then would apply any offset to the loaded capability before dereferencing and placing the returned data in the destination register. Therefore, a single load that finds an indirect capability as its address would perform two loads, a pointer access, and then a data access.

D.11.1 Indirect Capabilities in Architecture

We propose to add a new bit to the capability format, marking a capability as *indirect*. We recommend not conflating this behavior with the permission mechanism, instead adding a new field.

One new instruction would be added:

Make Indirect (`CMI`) This instruction makes an ordinary capability into an indirect capability such that any future dereference will effectively dereference the capability pointed to by this indirect capability. The bounds of the capability must be at least the size of one capability, and will be effectively truncated to this length by `CMI`, though the original bounds will be preserved and applied to the pointer on data access.

The `CMI` instruction makes a capability indirect, but no instruction can make an indirect capability direct again. As a result, delegating an indirect capability does not delegate access to the pointer that is dereferenced, but only to the data being pointed to.

Capability-manipulation instructions such as `CIncOffset` and `CSetOffset` would transform the offset of the indirect capability, but this offset would be applied to the pointer on data access. The pointer access will always use the base of the indirect capability. In addition, `CSetBounds` will transform the bounds of the indirect capability, but these bounds will be applied to the pointer on data access. The final access must be both within the length of the indirect capability, which may contain program-narrowed bounds, and the bounds of the object pointer. The bounds of the indirect capability would be implicitly the size of one capability, and would not need to be stored. This behavior allows pointer arithmetic to work as expected on indirect capabilities, to allow programs expecting standard capabilities to work unmodified.

D.11.2 Indirect Capabilities in Software

The above architectural behavior means that, on the whole, that software need not be aware when handling indirect capabilities, but only code that performs allocation or delegation would construct indirect capabilities, maintaining pointer tables.

Indirect capabilities might be used for general revocation between compartments. A buffer passed to another compartment could be passed as an indirect capability, with a word allocated by the caller to hold the pointer. On return, this pointer capability will be invalidated, and no further use of the indirect capability will succeed.

Indirect capabilities might be used to achieve memory safety for the heap in C. Every allocation could return an indirect capability, and generate a new entry in a pointer table. A call to free would invalidate the entry in the pointer table, and memory could be reused immediately with a new allocation in the pointer table. Sweeping revocation may eventually be necessary to free virtual memory space consumed by freed segments of the pointer table.

Indirect capabilities might be used for a copying garbage collector. Relocation of allocated objects would be facilitated by all references being indirected through a single pointer. When an object is moved, a single pointer could be updated. While an object is being moved, the pointer could be made invalid, with any use causing a trap that could be caught and handled appropriately.

D.12 Sealed Enter Capabilities

The existing sealing mechanism has three ultimate consumers of sealed capabilities: software may

- compare sealed capabilities for equality, using them as tokens created by components holding capabilities bearing Permit_Seal of the tokens' type;
- unseal a sealed capability, provided access to a capability bearing Permit_Unseal for the correct type;
- CCall a type-matched pair of code and data capabilities.

Should one wish to grant the right to invoke a sequence of instructions *at a particular entry point*, the mechanisms available are to pass a 'dummy' sealed data capability for use with CCall, or to use a trusted intermediate (probably also entered with CCall) to unseal the instruction pointer and jump.

Other capability architectures, notably the M-machine [18], have *enter* capabilities, residing somewhere between CHERI's unsealed and sealed Permit_Execute-bearing capabilities.[8] Similar to sealed capabilities, enter capabilities are immutable by their bearer and do not authorize memory loads or stores. Like unsealed capabilities, the bearer may directly jump to the enter capability to begin executing the instructions it references. The jump instruction (e.g., CJR or CJALR) atomically makes the enter capability into an unsealed capability and installs it to the program counter capability register. The dual-purposing of the architecture's indirect transfer instructions means that code can be oblivious to whether it is jumping through an ordinary code capability or an enter capability.

In this system, sealing to create enter capabilities is taken to be an ambient monotonic action. It should require no additional permission to construct an enter capability than to have a capability bearing Permit_Execute in the first place. We propose a CSealEnter instruction that derives an enter capability from any Permit_Execute-bearing VA capability, otherwise preserving permissions, bounds, and cursor. Enter capabilities take **otype** of $2^{64} - 2$ (truncated as required by the implementation; recall table 3.2) but are not intended to be unsealable within general system software[9] except by entry of control flow.[10]

[8] Because they act in tandem with CHERI's sealing mechanism and describe function entry points, we sometimes refer to CHERI's enter capabilities as 'sealed entry' or 'sentry' capabilities. We use 'enter capabilities' herein to emphasize similarity with extant literature.

[9] While it would be ideal if the permission to unseal **otype** $2^{64} - 2$ (and $2^{64} - 1$) were excluded from the primordial capability set, instead we imagine that early boot code can enforce this when it partitions its boot capabilities into the provenance roots it uses in the steady state.

[10] Of course, one could create a 'self-unsealing enter capability' that transferred PCC to the return value (ca-

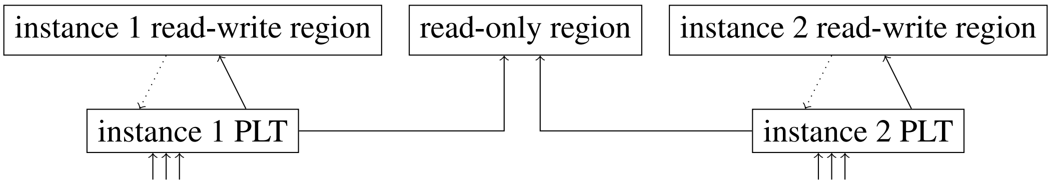

Figure D.5: PLT-style multiple instantiation showing capability reachability. The RO region is referenced with a subset of execute and load (data and capability) permissions by the PLT. The PLT references its corresponding RW region with any desired set of permissions. The PLT is referenced using enter capabilities by the outside world. The RW instance region may also hold references to the corresponding PLT with additional permissions (dotted lines); such references are required when the object's methods are not leafs of the control graph.

D.12.1 Per-Library Globals Pointers

Such capabilities are useful for multiply instantiated objects (e.g., shared libraries), as schematically shown in figure D.5. We wish to guarantee that any transfer of control into the read-only region is guaranteed to have a capability to some instance's read-write section in a register. In the case of a shared library, this may be a capability to the library instance's global `.data` and `.bss` segments, and so one sometimes hears the name 'globals register' for this register use. More generally, the capability may be likened to C++'s `this`.

In order to achieve this effect, the loader should, at instantiation time, create a Procedure Linkage Table (PLT) per instance; the PLT contains dedicated trampoline code, together with capabilities to the read-only and per-instance read-write regions. For efficiency, we would like the caller to affect as direct a transfer of control as possible, yet we wish to guard against frame-shifted entry to the trampoline code. Moreover, the trampoline must arrange for the invoked code to have the correct state capability (e.g., to a library's global variables), and yet the caller of the library must not directly hold this capability. The atomic unseal-and-jump behavior of enter capabilities is ideal: the PLT may contain the capability to the state, and the enter capability can authorize its (PC-relative) load once it has been entered, yet the user can neither fetch or manipulate capabilities through the enter capability nor enter the instruction stream at an incorrect offset.

In order to continue to ensure that the code runs with the correct capability in the globals register after return from a transfer of control outside the library, re-entry must also be gated by similar PLT stubs. That is, the return addresses must themselves be given PLT entries and direct control transfers must not be used to call out from the library. Instead, return addresses (in addition to the usual function entry points) should be given appropriate PLT stubs and enter capabilities to those stubs must be used as the return address given to the callee.

The contents of the stack and register file are otherwise shared with the callee; the stack may still be visible to the caller, as well. This mechanism is therefore not suitable for distrust-

pability) register and then returned control to the caller. While this particular gadget is unlikely to be more than a niche party trick, it demonstrates the need to manage, and (in particular) clear, capabilities derived from the unsealed PCC before yielding control.

ing inter-domain calls, but we believe it affords a reasonable amount of control flow integrity assurance within a domain, acting as a defense against return- or jump-oriented techniques.
This technique relies very little on architectural mechanism: PC-relative loads of capabilities and enter capabilities. Moreover, it is likely simple to explain to a traditional dynamic linker. However, it requires dedicated trampolines per instance of the object (library) under study, and does not completely guarantee control flow: for example, code called by our enter-capability-guarded library instance may engage in non-stack-discipline control flow and skip its return.

D.12.2 Environment Calls via Enter Capabilities

Enter capabilities are also useful for sandboxing. While sandboxed code can be made to look like a library to the calling environment, a more interesting observation is that the reverse is also possible and that enter capabilities are also viable for calls *from* the sandbox back to a single-threaded supervisor environment. On sandbox construction, the environment allocates space for its state closure (a `longjmp` buffer and (pointers to) other state) and builds a set of PLT-like stubs for this new sandbox that will ensure that a capability to this closure is passed to the functions invoked, just as the PLT stubs above ensured that the global pointer is passed. Whenever the environment calls into the sandbox, it must update its state closure as part of preparing the register file for entry to the sandbox. The return address given to the sandbox should, as discussed above, also be an enter capability pointing to one of the constructed PLT-like stubs.
In the case of multiple threads calling into the sandbox, the environment must demultiplex its closure pointers, as it cannot necessarily depend on the sandbox to not use the return enter capability from one thread within another thread's execution. The trampoline code for invoking or returning to the environment will, ultimately, involve asking the *environment's environment* for the notion of 'current thread' and using that information to retrieve the appropriate closure state. In the case that the environment is running under a kernel, demultiplexing may avail itself of a system call or fetch from VDSO to retrieve the current thread identifier or thread local storage capability. In the case that the environment *is* the kernel, it must use privileged architectural state (e.g., a saved stack pointer) to distinguish threads (and so the enter capability itself must bear Permit_Access_System_Registers or have access to another capability that does).

D.12.3 Bit Representation

In the past, enter capabilities used a now-again-reserved bit to indicate that a sealed capability could be unsealed by entry of control flow. We believe the current proposal, which uses a dedicated **otype** value, to be superior.

D.13 Compact Capability Coloring

As noted above, the Global permission described in the model of Section 3.3.1 is semantically not parallel to the other permissions. It is a one-bit attribute of the capability itself, a concept we term a *color*, borrowing from the information-flow analysis community [100]. Capabilities without the Global color (called Local) have their *flow* constrained, in that they can be stored only through a capability (of any color) bearing the Permit_Store_Local_Capability permission (as well as Permit_Store_Capability and Permit_Store). These two bits, one color and one permission, are leveraged by the existing runtime system to ensure that pointers to the stack can be stored only to the stack (and not the heap). That is, excepting capabilities within the TCB, all capabilities authorizing access to stack memory are colored Local, and all capabilities bearing the Permit_Store_Local_Capability permission authorize access only to stack memory. While the model permits a capability to stack memory (which must, per the above restriction, be Local) to be without the Permit_Store_Local_Capability permission, such capabilities are not deliberately constructed (unless they lack Permit_Store_Capability and/or Permit_Store as well, i.e., as part of a read-only view).

To recapitulate, then, we have the following four states of being for capabilities:

Color	**Permit_Store_Local_Capability**	**Use**
Global	Yes	TCB only
Global	No	Heap memory
Local	Yes	Stack memory
Local	No	Unused

The last configuration may be created (even outside its read-only utility) by monotonic action from any of the other configurations. These colorings and permissions capture the following intended flow policy:

Capability type...	**Stored through type...**	**Permitted**
Stack	Stack	Yes
Heap	Stack	Yes
Stack	Heap	No
Heap	Heap	Yes

In this policy, stack-type capabilities are universal authorizers of stores (‘universal recipients’, if you will) and heap-type capabilities are universally authorized to be stored (‘universal donors’). (The TCB-only, Global capabilities with Permit_Store_Local_Capability may be stored to and may authorize any capability store; the unused state can be stored only to TCB- or stack-state capabilities, and may authorize storage only of TCB- or heap-state capabilities.)

Neglecting the TCB state for a moment, we see that a single bit should be sufficient to encode our desired policy, using a material conditional: *if* the capability being stored is stack-type, then the capability authorizing this store must also be stack-stated (or, equivalently, phrased as the contrapositive, *if* the capability authorizing the store is heap-stated, the capability being stored must also be heap-stated). Similar flow policies also exist for flows across permission rings (the kernel may hold its own and user capabilities, but user programs may hold only user capabilities) and for flows through garbage-collector-managed memory regions (capabilities to managed memory may be stored only in managed memory, so that the collector must be notified of roots escaping). This suggests that we are justified in carving out several bits for orthogonal colorations; we suggest at least three, for the cases just considered, and perhaps no more than six, for reasons we will discuss below.

To abstract over the several colors, we adopt the terms ‘positively colored’ and ‘negatively colored’ to refer to the two possible states of a color. The flow policy is the logical *and* of the conditional for each color: “if the capability being stored is positively colored, then the capability authorizing the store must also be positively colored” or, equivalently, “if the capability authorizing the store is negatively colored, the capability being stored must be negatively colored.” Positively colored capabilities are the ‘universal recipients’, and negatively colored capabilities are the ‘universal donors’.[11]

The two-bit color-and-permission scheme described at the start of the section has a simple answer to the ‘primordial’ coloring of capabilities, and to the recoloring of capabilities into target states: the maximally permissive TCB state may be monotonically transformed with `CAndPerm` into any other state. Subsequent (monotonic) actions will never convert a heap-type capability into a stack-type one, or vice-versa. Given only a single bit for our color, any primordial capability must have *some* color, not a dedicated TCB-only ‘colorless’ choice. Further, our one-bit scheme must not ambiently permit conversion, in either direction, between the two states. We therefore propose that color bits are separate from permissions, immune to the action of the ambiently available `CAndPerm` instruction. We suggest that, primordially, capabilities be positively colored in all colors, so that, having explicitly changed the color of some memory capabilities, the software may not accidentally store into these now negatively colored regions.

What remains to be spelled out, then, is the *selective* authority to alter colors. Towards this end, we conceptually introduce yet another ‘space’ of identifiers guarded by capabilities and introduce a ‘color-change authority’ capability, which moves about the system as any other (and itself bears colors). The primordial capability authorizes any change to any color of any capability anywhere in memory. Such authority may be monotonically shed, coming to authorize only some changes (e.g., creating stacks from heap memory, but not the reverse) to some colors (e.g., changing only the stack/heap color but not the kernel/user color).[12]

Variant 1 We introduce a new instruction, `CChangeColor`, which takes a capability register containing the source capability, another for the destination, and a third for the authority capability. This instruction carries out *all authorized transitions* to produce a target that differs from the source only in its colors. We might have preferred a four-parameter instruction, which additionally specified *which* color to change from the authorized set, but this would likely require too many bits; in practice, we believe that color-change-authorizing capabilities would be few and relatively static, so the cost of tailoring to uses would be small.

An initial encoding of such color-change authority capabilities, backwards-compatible with the existing capability encoding described in this document, is to use a capability that

[11] Another dimension of generalization would be to have *load*-side color checking. That is, we could imagine enforcing policies of the form “if the capability authorizing a load is positively colored, then the capability loaded must also be positively colored (and if not, the result is not a capability).” We have no immediate use for such policies, but for somewhat related considerations, see Section D.6.

[12] In principle, one could also monotonically confine color changes to capabilities located in particular parts of memory or, perhaps more usefully, to memory capabilities *referencing* particular parts of memory. Encoding a restricted notion of change authority for non-memory capabilities such as sealing, compartment, or color-change capabilities is less obvious. We are not yet sure how to proceed in this dimension of monotonicity, and do not so here. Our color-change capabilities will always authorize changes to any capability anywhere, but, of course, the would-be authorized agent needs access to the source capability in the first place.

- Bears no permissions other than a new Permit_Change_Color permission. (Ideally, this would be encoded as the *type* of the capability, and not consume an entire permission bit.)
- Has a base of zero and a limit of the top of the address space.
- Stores in its offset a bitmask authorizing color changes as follows: color n may be transitioned from its current value c_n to its negation if bit $2n + c_n$ is set.

It is immaterial which of '0' or '1' one assigns to the different color choices. However, the system must pick one; we suggest using '1', commonly read as 'true', for the 'positively colored' choice, in keeping with the presentation above. In this encoding, the offset-adjusting instructions must be modified to permit only bitwise *and* operations on the offsets of these capabilities. (If one is conflating capability types, as we do at present, the appropriate guard is that *only* Permit_Change_Color is set.) This is perhaps the most awkward feature of this design, though we believe the checks can be added without impacting timing. (In a world where capability types were explicit and separate from permission bits, we could reuse the permission bits, already subject to manipulation only by CAndPerm to carry our permission bitmask, assuming there are at most half as many colors as permission bits.)

Variant 2 Perhaps a more natural encoding would instead have capabilities that enact exactly one color change when cited (but may *authorize* more than one). Here, we propose that the space of integers from 0 to $2C$, with C being the number of color bits available in the system, be another 'identifier space' for capabilities. A color-change capability holding value $2n + c_n$ requests toggling color $n < C$ from c_n to its negation when used as the authorizing capability with the CChangeColor instruction. In this scheme, there would be no need for any fiddly bit manipulations of capability offsets, but at the cost of more capabilities held by agents authorized to perform some, but not all, color changes.

Variant 3 In fact, there is no need to introduce an entirely new capability type, permission bit, or instruction. Because sealing object types (**otype**), in practice, are only at most 24 bits wide, and there are very few colors, we could reuse invalid encoding space for sealing capabilities to also authorize color changes: values x in the range of 2^{24} to $2^{24} + C$ could be defined as colors rather than invalid **otype**s and the existing use of Permit_Seal and Permit_Unseal bits could control setting the target capability's color number $x - 2^{24}$ to become positively or negatively colored. The existing CSeal and CUnseal instructions could be used in lieu of any new CChangeColor. This shares with variant 2 the need to have many capabilities held by agents authorized to change multiple colors if they are not contiguous or authorize different transition directions.

D.14 Sealing With In-Memory Tokens

Deciding on the number of **otype** bits within a sealed capability has been challenging, because the bits come at the expense of bits for precision of bounds, permissions, and colors. In this section, we propose that *virtual addresses* can play double-duty as *type identifiers*, either supplanting or reducing the need for in-capability **otype** bits. The design of this section is a somewhat invasive change to CHERI, but appears promising.

D.14.1 Mechanism Overview

We propose that sealed objects have their type not in the referring capability, but rather in a tagged capability-sized structure at the *base* of the object in memory. This structure is termed a 'type token' and it contains a virtual address (and metadata) but does not confer any permissions, to its contained address or otherwise, to its bearer; in fact, as a defensive posture, we do not permit tagged type tokens to be loaded into registers unless PCC has Permit_Access_System_Registers.[13] In addition to creating a sealed reference capability, sealing an object would *store* a suitable type token to memory, derived from the capability used to authorize the seal. Unsealing *fetches* and verifies this type token against the capability authorizing the unsealing.

D.14.2 Shared VTables with Enter Capabilities and Type Tokens

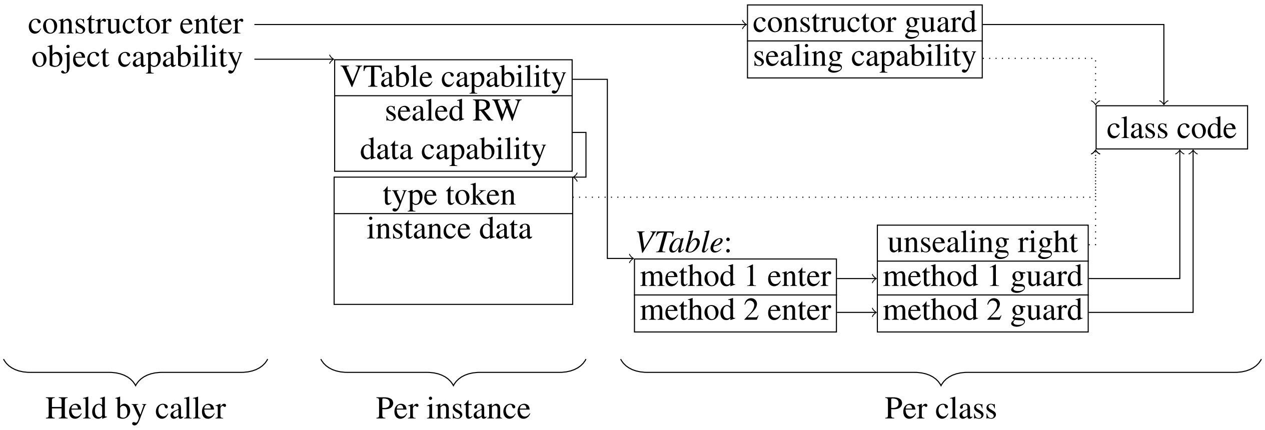

Figure D.6: Schematic representation of a shared VTable design for a base class. The user directly holds an enter capability to the object constructor guard, which uses the adjacent Permit_Create_Type_Token-bearing capability to stamp object instances. Each object instance is held by the user through a Permit_Load_Capability-bearing capability and has a two-capability header, consisting of a Permit_Load_Capability-bearing capability to the VTable and a sealed capability bearing load and store permissions to the object instance data. The VTable itself is an array of enter capabilities pointing at method guards, which in turn verify the object instance's type token against their unsealing right before invoking the actual class method handler.

[13] This means that a sealed object cannot simply be copied via `memmove`; a copy or move constructor must be invoked to reconstruct the type tag on the target memory. This does not seem to be an especially high burden. In fact, even the Permit_Access_System_Registers caveat can be removed if an alternative mechanism for tag reconstruction is made available to the kernel; for example, capability reconstruction as per appendix D.5 could gain the ability to reconstruct tags given the sealing authority.

Enter capabilities (recall appendix D.12) give software the ability to ensure that control flow can enter a given region at a particular address: the bearer of an enter capability can jump to it but cannot adjust its offset. However, unlike the existing `CCall` mechanism, enter capabilities when invoked transition only the PCC register. To transition other registers as a function of the instance, we propose a PLT-like scheme using dedicated trampolines to load *unsealed* capabilities that were nevertheless beyond the reach of the caller, due to the sealed nature of the enter capability held.

In-memory type tokens allow software the ability to mimic the existing CHERI sealing mechanism, trading one capability in memory to not need the **otype** bits in referring capabilities. (This does come with the additional cost that sealing a region of memory under multiple seals will require the use of several tokens in memory with successively larger bounds in the referring capabilities.) In figure D.6 we show a schematic representation of using in-memory type tokens to guard method invocation of a multiply instantiated (C++) object.

Combined with enter capabilities, an object's shared code can now securely verify that its first argument is indeed a sealed capability to a data region resulting from this object's constructor. The constructor is made available as an enter capability to a region containing a capability bearing Permit_Seal. The non-constructor capabilities in the VTable are enter capabilities pointing within a region bearing corresponding Permit_Unseal rights. These three regions (the constructor guard code, the method guard code, and the VTable) are created once, when the object class is loaded, and will never be written to thereafter. Conveniently, the object-class code location can be used as its own type token value, there is no need for a separate pool of virtual addresses for type token values. The separation of unsealing rights is not essential and is another defense in depth: the non-constructor methods will not necessarily come to hold, even transitively, a capability bearing Permit_Seal for this object type.

D.14.3 The Mechanism in More Detail

Type tokens are created directly into memory with a new `CSealTyT` instruction, stored at the *base address* of the capability being sealed, which must be capability-aligned (and the to-be-sealed capability must authorize an at-least-one-capability-sized segment of memory). `CSealTyT` requires that the capability to be sealed bear Permit_Load and Permit_Store and that the invocation reference an in-bounds Permit_Seal-bearing[14] capability whose cursor will form the type tag.[15] Software must ensure that the store done as part of sealing is visible to other processors before publishing the sealed capability anywhere it may be read by another core. Immediate fencing is not always required, and so we suggest it not be intrinsic to the `CSealTyT` instruction. The sealed capability resulting from `CSealTyT` will have its **otype** set to $2^{64} - 3$, truncated as required by the implementation.

Attempting to load a type token via `CLC` will succeed, but will strip the tag. The resulting register contents need not be particularly well specified; in particular, we should no more ex-

[14] For compatibility with CHERI-MIPS, we exclude from `CSealTyT`'s domain sealing capabilities referencing the bottom of memory, from 0 and to the maximum **otype** value, interpreted as an unsigned integer, available to the implementation, inclusive. These are reserved for use with the existing `CSeal` instruction.

[15] It is not clear whether `CSealTyT` should permit the clearing of Permit_Load and/or Permit_Store in the resulting sealed capability, despite requiring them on input.

pect sensible results from the capability-observing instructions here than if we had loaded an arbitrary untagged region of memory.

Token-mediated unsealing is done by a new CUnsealTyT that takes a sealed capability (with **otype** of $2^{64} - 3$) and an in-bounds authorizing capability bearing Permit_Unseal. If the cursor of the authorizing capability matches the virtual address stored in the type token at the base of the sealed object,[16] then CUnsealTyT produces an unsealed version of the sealed capability. Microarchitecturally, CTyTUnseal is somewhat akin to a compare-and-swap whose store-back is into the register file rather than memory.

It might be helpful to software to add a CGetTypeTyT instruction that somewhat mirrors the CGetType instruction. CGetTypeTyT would fetch from the base address of a sealed capability (of the right **otype**) and store the virtual address from the type token back to a general-purpose integer register. We propose that, if an exception is not desirable, that the value $2^{64} - 1$ be used if the memory at the base is not a type token.

D.14.4 Unseal-Once Type Tokens

It is likely useful to have a version of unsealing that atomically prevents any future attempts. Rather than merely *fetch* the type token, this instruction would carry out a CAS-like update of the type token in memory.

D.14.5 User Permissions For Type-Sealed VA Capabilities

Because type tokens are capability-sized structures used only for their contained virtual addresses, there are many spare bits in the structure (in fact, a few type-tagging bits shy of an entire machine word's worth). One especially attractive possibility, if it can be demonstrated to be sufficiently secure, is to push the architecturally defined permission bits within the sealed capability into the type token. This would permit the use of the intra-capability permission bits as user permissions, subject to the action of CAndPerm despite the sealed nature of the capability. We would then be able to use capability permission bits to help arbitrate permissions to methods within an object, as is typical of other capability systems, rather than, as suggested by the design in appendix D.14.2 above, having one enter capability per procedure and gating permission by possession of the procedure's guard's enter capability. CUnsealTyT would use the bits from the type token in its output capability, and software would be able to inspect the permission bits of the input object reference (i.e., there would be no need for a second register storeback in CUnsealTyT).

In this scheme, should an object wish to be able to grant sealed references with one of several sets of architectural permissions, it suffices to place an array of type tokens at the beginning of instance memory and adjust the base of the (to be sealed) capability, while leaving the cursor to point at the start of the object's data. Any type tokens within reach confer no authority, even after we have moved architectural permission bits into them. Further, because type tokens cannot be created in memory except by CSealTyT or highly privileged software, aliasing of the memory containing the type token cannot *de novo* amplify architectural access (but may be vulnerable to confusion within suitably authorized control flow).

[16] This load is why CSealTyT required Permit_Load of its to-be-sealed capability.

D.14.6 Token-mediated CCall

`CCall` poses something of a challenge for in-memory type tags: a single instruction must, seemingly, perform *two* fetches from memory and then do a comparison on the loaded values. However, because the instruction cares only about the equality, it seems that we can turn this into a fetch from one capability's base and then a CAS-style *comparison* against the other's. In fact, this combines nicely with unseal-once type tokens: if `CCall` fetches from the sealed code capability first, it is then in a position to issue the appropriate CAS against the sealed data capability. In CHERI-MIPS, `CCall` is already a two-cycle instruction, occupying two successive stages of the pipeline, and so we conjecture that the changes requisite to support token-mediation are small.

D.14.7 Hybridization

This scheme uses one **otype** value for its sealed capabilities; the remaining values are still available for the rest of the system's use. It is our hope that most users of **otype** values can be rearchitected to use this in-memory scheme and that the **otype** field can be reduced in size. However, the **otype** field should not be entirely eliminated: its existence allows us avoid some of the overhead of this design in the innermost ring of the system.[17] Such **otype** bits would also let software create sealed objects other than enter capabilities without memory footprint.

D.15 Chaperoned Short Capabilities

An frequent initial objection to CHERI is that even the 128-bit compressed form of capabilities occupies too much space, especially for pointer-heavy workloads. However, when discussing a 64-bit virtual address space, it seems plausible that 128 bits is the best we can do: the metadata CHERI requires vastly outstrips any 'spare' bits in the address, and any size that was not a power of two bits would be awkward, at best. One way out, would be to imagine that one could mix 128-bit and 64-bit capabilities within an address space, with the caveat that the 64-bit capabilities could address only a 32-bit address space (i.e., they would have a 4 GiB reach) and would have a smaller set of permission bits, fewer flag bits, and fewer bits for object types. While we could limit all 64-bit capabilities to referencing a particular, fixed 4 GiB region of the larger addess space (e.g., the first 4 GiB), a better design, if we could get it, would be to allow the 4 GiB window to be chosen by a 128-bit capability.

The design we detail here treats these 64-bit capabilities as specialized representations of 128-bit capabilities. Importantly, this design does not modify the representation or semantics of capabilities within the register file: the bulk of the system's operation is unimpacted. We introduce new, purpose-made instructions for loading and storing these short representations of capabilities; stores especially may fail if translation is not possible.

[17] Because the innermost ring is presumably the kernel's TCB, a hypervisor, or 'nanokernel'– effectively microcode – the resulting system has some similarities to the Intel 432 / BiiN / i960MX lineage, which had a few architecturally understood special types of capabilities – but relied on software interpretation for the rest.

D.15.1 Chaperoning Capabilities

Because 64-bit capabilities operate only within a 4 GiB window of the address space, when fetching a 64-bit capability from memory, we fill in the implied upper 32 bits of the full 64-bit address from the *cursor* of the *capability authorizing the fetch*. This straightforward operation is provided by the CLShC instruction.

When attempting to (encode and) store a capability to a short form in memory, the store will fail unless all three of the following addresses agree on their top 32 bits: the computed destination address of the store and the base and limit of the capability being stored; the cursor of the capability to be stored is permitted to be within either adjacent 4 GiB window (but must still be representable).[18] [19] All of this is provided by the CSShC instruction.

A consequence of this design is that short capabilities (transitively reached through short capabilities) are always interpreted within the 4 GiB window specified by the initial reference through a full capability. These capabilities may be stored as short capabilities anywhere within this window (or as full capabilities anywhere in the address space). Because capabilities in registers always have their full 64-bit virtual address cursor and bounds, it is impossible to use a short capability in one 4 GiB window to derive a capability to any part of a different window: the dereferencable region is always contained within the original window whence the capability was loaded, and so attempted stores to another window will fail.[20]

D.15.2 Restrictions Within Short Capabilities

In order to reduce the space required for metadata within short capabilities, we suggest several restrictions.

Within the permissions field, we suggest that short capabilities be limited to expressing virtual address space, so that Permit_Seal, Permit_Unseal, and Permit_SetCID are implicitly false for any short capability. This seems reasonable, as these gate fundamentally new facilities offered by CHERI and seem like they will be relatively rare even in fully CHERI-fied software stacks, so the requirement to use a 128-bit capability should not be onerous. Further, because we intend short capabilities to be used mostly for sandboxes within a larger ecosystem, we think it reasonable to imply that Permit_Access_System_Registers is also false. Similarly, we do not foresee the utility of the Local/Global distinction for short capabilities, and so propose implying Permit_Store_Local_Capability to be false.[21] All told, these implications eliminate

[18] To expand on the meaning of "fail" here: it would be sufficient to store a de-tagged word, but trapping is more likely programmer friendly. However, this is a data-dependent action, as it requires a comparison between the (untranslated, virtual address) and the address from the register file. However, this is not the only data dependence in the short capability store instruction.

[19] Alternatively, it would suffice to ensure that, on decoding, any access beyond the limits of the 4-GiB-aligned region had been shed. Because short capabilities are never used directly, there is some flexibility in enforcement here.

[20] If ever direct memory-to-memory capability copies become possible, it would be necessary to explicitly check that copied short capabilities are not being replicated in ways that would change their decoding.

[21] We could also imply the Global permission bit to be *true*, but then we would need to fail attempts to encode local capabilities into short forms. While we do not anticipate the use of capabilities bearing Permit_Store_Local_Capability outside trusted software, it nevertheless seems simpler to leave Global within the short capability encoding.

five existing permission bits from short capabilities' representations.

We suggest a reduced object type range for short capabilities, as well. This will have implications in the software stack: 'small' object types will be somewhat precious, and so may need to have special handling in the allocator(s) thereof. The utility of sealed short capabilities, and especially of architecturally defined sealing object types to short capabilities, remains an open question.

Bound metadata may also be subject to pressure, and so short capabilities may face stricter alignment requirements for large objects than full, 128-bit capabilities. While this would not be great, it may be that references to large objects are relatively sparse, and so software may find it easier to fall back to full capabilities rather than insist that all capabilities should be short whenever possible.

D.15.3 Tag Bits and Representation

Given such a system with mixed capability widths, we require more bits for distinguishing capabilities from data. In a 128-bit-sized and -aligned region of memory, there are five possible options: ① One 128-bit capability. ② Two 64-bit capabilities. ③ One 64-bit capability, followed by data. ④ One 64-bit capability, preceded by data. ⑤ Only data. There are several ways that we could arrange to distinguish these possibilities, but two seem especially attractive. Perhaps the simplest approach is to use three out-of-band tag bits rather than the one per 128-bit granule of memory that CHERI now imposes; this would leave us with three values reserved for future expansion. One could slightly tamp down on the need for tag bits by tagging entire *cache lines* instead: eight sets of 5-way discrimination, corresponding to 128-bit cache lines, takes only 19 bits rather than the more straightforward 24, at the cost of more complex decoding logic (likely in the LLC).

However, we may be better served by the use of two out-of-band tags and one bit in the capability encodings themselves, effectively giving us somewhere between two and four bits of metadata, depending on the scenario. One possible encoding is shown in table D.1. Forbidden states should trigger machine check exceptions or something similarly indicative of catastrophe. This scheme is relatively straight foward to operate, but requires a little awkward handling of the inherent asymmetry between the upper and lower 64 bits within a 128-bit granule. A load of a full capability must verify that both out of band tag bits and t_{hi} are all asserted. A load of a short capability from the upper position must verify that T_{hi} is asserted and t_{hi} is clear. A load of a short capability from the lower position must verify that T_{low} is asserted, that t_{low} is clear, and that either T_{hi} or t_{hi} is clear. Data stores always clear the corresponding out-of-band bit; stores to the lower half of a capability granule must additionally access T_{hi} and, if T_{hi} is asserted, then access t_{hi} to determine whether T_{hi} should be cleared as well (to avoid the forbidden states marked with †). Fortunately, all of this state machine logic is localized within a cache line and its tag bits.

Similar considerations hold should we wish to mix all of 64-, 128-, and 256-bit capability forms. In such a system, there are 26 states for every 256-bit granule of memory: each 128-bit granule may be in each of the 5 states given above, or an adjacent pair may hold a 256-bit capability.

T_{hi}	T_{low}	t_{hi}	t_{low}	Meaning
0	0	X	X	Two data words
0	1	X	0	64 bits of data above a 64-bit capability
0	1	X	1	Forbidden
1	0	0	X	A 64-bit capability above 64 bits of data
1	0	1	X	Forbidden†
1	1	0	0	Two 64-bit capabilities
1	1	1	X	A 128-bit capability

Table D.1: A possible hybrid out-of-band and in-band tagging scheme for mixing 128-bit and 64-bit capabilities. t_{hi} and t_{low} are the intra-capability tag bits for the upper and lower 64-bit regions, respectively, while T_{hi} and T_{low} denote the corresponding two out-of-band tag bits. X indicates 'don't care' and stands for either bit value.

D.15.4 SoCs With Mixed-Size Capabilities

It is frequently the case that Systems on Chip (SoCs) contain 64-bit application cores and also 32-bit microcontrollers. One potential further use for this approach is to allow bridging between those two worlds: 64-bit cores with 128-bit capabilites that are able to load and store 64-bit capabilities used by 32-bit cores connected to the same memory fabric. Care would be required to ensure that capabilities originating on one core were derefenced only with a suitable address space on a second core able to access them.

D.16 Capabilities For Physical Addresses

D.16.1 Motivation

CHERI capabilities that authorize access to memory are typically interpreted in combination with an ambient virtual address translation configuration. That is, the addresses authorized by a CHERI memory capability are taken to be virtual addresses, which are then translated to physical addresses by the core's MMU. The MMU configuration defines a virtual address space; it is, ultimately, in all modern, mainstream architectures, described by *integers*.[22] The use of provenance-free integers to describe such configurations carries risks, just as with pointers. Necessarily, the ability to configure the MMU must be confined to privileged, and necessarily trusted, software; this software must enforce its intended policies concerning permitted access to the core's view of physical memory and it must do so with no architectural safeguards.

Moreover, a (software) system may, as part of timesharing the CPU core, reprogram the MMU to achieve isolation (and, possibly, controlled non-isolation) between different 'process contexts'. Further, these contexts may be dynamic, reshaping their associated MMU configurations across time. CHERI capabilities are not explicitly associated with a particular context and/or

[22] In architectures with hardware page table walkers, such as ARM and RISC-V, these integers are arranged in defined, tabular format. In architectures without, such as MIPS, the analogous structures are defined only in software, but the soft-loaded TLB is programmed using integer values written to architecturally specified registers.

time. As a result, software must ensure that capabilities are not transmissible improperly[23] from one context to another, nor retained improperly as context mappings evolve. Thus, the direct mechanisms available for capability passing within a single context (including between CHERI compartments therein) are likely not available for cross-context communication.

A similar story plays out in hardware: ‘physical’ addresses are meaningful only when paired with a *location*, as bus bridges may remap addresses in transit from one port to another. When devices or cores wish to communicate, they must model the action of the intermediate fabric and generate (integer) addresses that may not be meaningful locally but will be at the remote endpoint, across the bus fabric. Again, all the problems with integer addresses resurface and are exacerbated by the relatively minimal protection mechanisms available at the physical bus layer.

For this section, we focus on two cases: software on a CHERI core seeking to escalate its privilege, and peripheral devices wishing to attack the core (possibly in cooperation with software). In both cases, the intended victim of the attack(s) will be taken to be the CHERI core’s trusted computing base (e.g., a hypervisor). We restrict our attention to steady-state operation rather than attacks against the initial bootstrap; that is, we assume that any would-be attacker was not present during the load of said TCB and that the *core* itself is trusted to faithfully execute instructions.

D.16.2 Capability-Mediated CPU Physical Memory Protection

RISC-V has a notion of a Physical Memory Protection (PMP) unit that validates every (post-virtual-address-translation) memory request issued by a processor core. Roughly, for each request, an n-way associative lookup against a table of (region, permissions) pairs is performed, and the request is authorized only if the table contains a region containing the requested address and the request is of a type permitted by that region. For details, see the RISC-V Privileged Architecture specification [127, §3.6].

The control interface to the PMP is, as might be imagined, based on integers: coarsely speaking, machine-mode code is able to write arbitrary bits to the PMP table through the core’s CSR interface. Supervisor and user mode code are not permitted access to the table. Thus, any code in machine mode can alter restrictions imposed on supervisor or user memory access, and so a confused deputy attack on the machine mode could result in privilege escalation for the supervisor or user programs. We would prefer to have a more ‘least authority’-friendly option.

We propose a ‘capability-mediated PMP’ (CPMP). Its control interface will permit table entries to be populated only from valid (tagged) capabilities. We imagine using a pair of a CSR and a special capability register to provide row-by-row access to the augmented table.

Because machine-mode code on RISC-V has explicit control over whether address translation is enabled, a baseline capability-mediated PMP implementation could repurpose the existing CHERI capability mechanisms and rely on software to track the distinction between capabilities intended for use as physical addresses and those intended for use as virtual addresses. Such

[23] The simplest and most restrictive policy is to entirely prevent transmission of capabilities between contexts. However, if contexts have common identically interpreted regions of their address spaces, one could imagine utility in passing capabilities referencing only these spaces. Such passing would, in CHERI’s design, necessarily have to go via a software intermediate rather than more direct passing through the shared region itself.

an approach runs slightly against the grain of our design principles, and has limitations; for example, sealed forms must be used if these capabilities are to be given to supervisor (or user) code.

For these reasons, and to enable a wider series of uses, we envision creating a new capability provenance *root*. Capabilities derived from this root are distinct from existing CHERI capabilities (by, say, having a bit immutably set that the existing capabilities maintain cleared) and denote ranges of physical addresses, even in the presence of paging. Accesses via these capabilities bypass any paging mechanism and, dually, we can now make accesses via the existing CHERI capabilities that *always* go via address translation, even in machine mode.[24] These capabilities may have their born authority decreased as with any other CHERI capability, and may flow to non-machine-mode code to enable (for example) light-weight partitioning of physical resources between multiple supervisors.

D.16.3 Capability-Mediated DMA Physical Memory Protection

Whereas RISC-V considers PMPs only in the context of a CPU core, nearly identical hardware can be used to gate peripheral DMA requests. Here, the PMP's control interface is exposed to the CPU, most likely as a memory-mapped region, and the direction of requests is backwards, but the operation of the device is fundamentally the same. When presented with a memory request *by the peripheral*, such a gate performs an associative scan of the configured table and either permits the request to enter the bus or rejects the request. We tentatively call such a gate an IOPMP.

Whereas IOPMPs could be programmed using integers (as in the RISC-V PMPs), or using existing CHERI capabilities transported over the memory bus, the story is much more credible if they can require physical-address capabilities. So equipped, we reduce the risk of confusion or misbehavior of machine-mode code but, more excitingly, we gain the possibility of directly exposing peripheral IOPMPs to non-machine-mode code for efficient device pass-through.

This story is fairly satisfying for the control of the IOPMP itself; however, there remains a challenge of translating the authority carried by the CHERI CPU core into an address suitable for comprehension by the peripheral. That is, because the peripheral continues to speak in *integer* addresses in its control messages, software on the core could easily treat the peripheral as a confused deputy, causing it to DMA to regions authorized by, for example, other (software) compartments. It may be necessary to limit sharing of peripherals this way, or more directly involve the IOPMPs in device control. One could imagine, for example, that the IOPMP could 'back-translate' core-originated capabilities in control messages into integers for the peripheral's consumption, perhaps with a tag.

D.16.4 Capability-Based Page Tables

Traditionally, hypervisors must deny the supervisors they oversee the ability to directly control the memory translation tables. Towards the 'paravirtualization' end of the spectrum, the hypervisors require that the guests make hypercalls to manipulate the page tables. Towards

[24] This obviates the RISC-V `mstatus` MPRV mechanism for toggling address translation.

the ‘hardware-assisted’ end, the CPU’s MMU will use ‘nested translation’: the ‘guest physical’ addresses manipulated by the guest are subject to re-translation, through tables controlled by the hypervisor, before becoming ‘host physical’ addresses and exiting the CPU core. Both approaches have substantial costs.

A more radical approach would have us change the traditional memory management unit (MMU) page tables. Instead of mapping virtual addresses to *integer* physical addresses, the page tables would yield a *physical capability* for a virtual address. We envision repurposing the capability permission bits for the PTE permission bits, and extending the flags field of appendix D.1 to encompass non-authority flags of PTEs, notably including accessed, dirty, and global flags.

To simplify the system, we may require that physical capabilities installed in page tables have offset zero and length at least a full page (of the appropriate level of the tree). This allows us to skip a capability bounds check when translating a virtual address but retains proof of *provenance* of the authority to access a given region of physical addresses.

D.16.5 Capability-Based Page Tables in IOMMUs

As with the PMPs, this new facility also finds use in guarding peripherals. Rather than the associative table scans of the IOPMPs above, we could have capability-mediated IOMMUs whose page-table entires, again, contain physical-address capabilities. Of course, there is no reason that an IOPMP expose a 64-bit address space to the peripheral, nor that it use hierarchical pages. For many peripherals, a *single* page-sized aperature (or even smaller) may suffice. The concern of integer addresses in peripheral control messages continues to apply.

D.16.6 Exposing Capabilities Directly To Peripherals

Both IOPMPs and IOMMUs, mediated by capabilities or not, continue to expose an *integer* address space to the peripheral. While the peripheral may be using CHERI for its internal computations, its interface with the host remains capability-less. In some cases of mutually distrusting peers, this may suffice, and each side may have capability-mediating devices under its control to guard the interconnect.

However, in other cases the host may wish to extend the *tagged* memory bus all the way to the peripheral, and then grant capabilities directly to the device as though it were a software process. In such cases, we expect that an IOPMP- or IOMMU-like guarding device will still be useful, to prevent a malicious or errant device from synthesizing or retaining (and subsequently using) capabilities that the host does not intend. All capabilities transiting the guard would be checked to be a *subset* of a capability in the guard’s table. We note, in passing, that such guard devices are also useful for the case of direct peripheral-to-peripheral access, not merely the case of peripheral-to-memory as we have generally focused upon here. The details of the control interface to such a device, as well as its internal operation, are left to future work.

D.17 Distributed Capabilities For Peripherals And Accelerators

CHERI's design focuses on the 'main' CPU core(s), in which there is a single operating system, and capabilities are used within virtual address spaces, mediated via an MMU and a memory-coherency system.

Many systems are composed of distributed compute elements that share memory. In various contexts these are termed 'peripherals', 'DMA engines', 'accelerators' or 'remote DMA network cards'. These may be on a single system-on-chip using fabrics such as AXI, or across interconnect such as PCI Express, Thunderbolt or Infiniband.

When capabilities are used in such a system, there is a requirement to protect them from inappropriate modification by cores that might be outside the purview of the primary operating system. Additionally, it would be advantageous for such cores to use capabilities for their own code and data, without having to mediate them from centralized authority. Furthermore, such systems frequently use multiple levels of address translation – not just a virtual address space (as capabilities in this document primarily refer to), but a patchwork of multiple physical address spaces (including the guest physical address spaces used by hypervisors), as well as virtual address spaces used by accelerators and other cores.

There are two challenges: first, preventing a core from modifying a capability it does not own, and second, handling the case that capabilities can alias if they refer to an incorrect address space.

To achieve these goals, we propose several architectural features.

D.17.1 Scope and threat model

This feature assumes that peripherals are capability-aware, in that they are able to load, store and manipulate capabilities and their tags. A number of scenarios with trustworthy hardware and software, untrustworthy software on trustworthy hardware, or untrustworthy hardware and software may be envisaged. Hardware that is not capability-aware and uses integers as pointers is out of scope for this extension, although it may be constrained or otherwise capability-wrapped by some other structure.

D.17.2 Address-space coloring

We deconstruct systems into regions of address-space colors (ASCs). A region with a common color has addresses with a single unambiguous meaning. Generalizing, a color could apply to an application's virtual address space, the system's hardware physical address space, the guest physical address space of a virtual machine, or a piece of memory on a peripheral. A Processing Element (PE – processor, DMA engine or other core) is assigned a color based on its physical location in the system topography.

Colors also represent single regions of authority. Within a colored region, it is assumed that every device that can synthesize a capability has rights to do so. If a device is untrustworthy, it should be segmented into a different colored region.

An address space may be connected to a different address space via an Address Translation and Protection Unit (ATPU). Examples of ATPUs might be MMUs, IOMMUs, and hypervisor page translation, but also more limited cases such as PCI BAR mapping or driving upper address bits from a page register. An ATPU may provide no translation between mutually distrusting hardware that happens to share an address space, but still apply protection between them.

We generalize an ATPU as a bridge by which requests come in from one address space and are dispatched into another. The ATPU may itself make memory requests to determine the translation, such as when walking page tables. These would potentially occur in an address space of a third color.

D.17.3 Capability coloring

Capabilities refer to addresses in particular address spaces, hence capabilities are given a color that is stored within the capability. It is now possible to disambiguate the address within a capability with the address space to which it refers.

Representation

We describe architecturally the notion of address space color without specifying the specific representation. However microarchitecturally we expect that the **otype** field in a capability would be reused, based on a tagging scheme to distinguish them from a software-defined **otype**. Given the limited number of bits available in the **otype** for the otype-color, it may be impossible to represent all the colors within these bits. It is not necessary for the otype-color field to be unique, only that it is possible to disambiguate which address space region is referred by a capability. For example, the upper bits of the address may be used to distinguish two regions with the same otype-color field which are each smaller than 64-bit addressing. Architecturally such regions would be thought of as having different colors.

otype reuse Since 128-bit capabilities are constrained by size, we propose using the **otype** field to represent some or all of the ASC. To disambiguate from the softwaredefined **otype**, the data structure should be tagged.

To avoid reducing the bits for the **otype**, we propose a variable-length tag. This also allows embedding one instance of a variety of other metadata in the **otype** field. For example:

```
0x_xxxx_xxxx_xxxx_xxxx: Software-defined otype (17 bits)
10_xxxx_xxxx_xxxx_xxxx: Metadata type A (16 bits)
11_0xxx_xxxx_xxxx_xxxx: Metadata type B (15 bits)
...
11_1110_cccx_xxxx_xxxx: Metadata type E (8 alternatives of 9 bits each)
```

D.17.4 Operations on colored capabilities

Colors are used to enforce policy by processing elements and ATPUs. A processing element has, in its hardware, an awareness of the color of its local address space.

In this area exist a number of possibilities, subject to further research.
Most conservatively, a PE could deal only with capabilities of its own color. A capability with another color is treated as if the tag is cleared. This would allow PEs to use capabilities internally, without sharing between them. A privileged process (boot loader, management processor, hypervisor, operating system on an application core) is used to generate initial colored capabilities for each PE, from where they are used internally.
In this scenario, capability conversion between colors would be minimal or require a call to the privileged process. Conversion would require translation between address spaces, with a chance that a capability could not be directly represented in the target address space (if it represents disparate physical pages, for instance).
Other approaches are possible. For instance, colored capabilities could be treated as sealed. This would enable devices to be given 'handles' to memory in another address space that they cannot access, but can pass around to other data structures. For instance, networking data structures might contain linked list pointers in network stack address space – the NIC can build its own linked list, without the ability to access the data being pointed to. Much care would be required here to avoid confused deputy attacks.

D.17.5 Enforcement

Both PEs and system bridges are tasked with enforcing the capability model:

Bridges enforce operations on colored capabilities. For example, a bridge may disallow capabilities of other colors to pass through it. Bridges are viewed as more trustworthy than devices they connect, although a hierarchy exists – bridges closer to DRAM are able to disallow capabilities that are accepted by bridges further away. Bridges have an awareness of whether hardware might be untrustworthy (for instance, plugged in to a motherboard slot or external port) and apply external enforcement of properties where the hardware might be untrustworthy. ATPUs could also transform capabilities that pass through them according to their address space remapping – e.g., allowing a PE to store capabilities with its local address space, but remap them to physical addresses when storing to DRAM.

PEs are tasked with enforcing the capability model within their local software. Bridges enforce colors, but PEs enforce the remainder of the capability model (monotonicity, tagging, etc). An untrustworthy PE may corrupt its own capabilities, but since the coloring is enforced by the bridge it will only have detrimental effects on its own software.

D.18 Details of Proposed Instructions

The following instructions are described using the same syntax and approach as those in Chapter 7.

CAndAddr: Mask Address of Capability

Format

CAndAddr cd, cb, rs

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cb	rs	0x24

Description

cd is set to *cb* with *cb*.**a** logically ANDed with *rs*. If the new address causes the capability to become unrepresentable then an untagged capability with address set to the masked address is returned.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let rt_val = rGPR(rt);
if cb_val.tag & cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else
{
  let newAddr = cb_val.address & rt_val;
  let (representable, newCap) = setCapAddr(cb_val, newAddr);
  if representable then
      writeCapReg(cd, newCap)
  else
      writeCapReg(cd, int_to_cap(newAddr));
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is set and *cb* is sealed.

Notes

- This instruction may be appropriate for use cases in which C is masking pointers before use, for example when the lower bits are used to store additional metadata in a pointer with known alignment.

- This instructions also allows more efficient aligning down (and up) of capabilities. In C source code this will almost always be a compile time constant so we might discover that we need a version of this instruction that takes an immediate operand. Using a 5-bit immediate field we could encode align-down/align-up masks from 2 to 2^{32}.

- This instruction is also useful in order to create precisely representable bounds for heap and stack allocations.

CBuildCap: Import a Capability

Format

CBuildCap cd, cb, ct

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cb	ct	0x1d

Note: If the encoded value of *cb* is zero, this instruction will use **DDC** as the *cb* operand

Description

CBuildCap attempts to interpret the contents of *ct* as if it were a valid capability (even though *ct*.**tag** is not required to be set and so *ct* might contain any bit pattern) and extracts its **base**, **length**, **offset**, **perms** and **uperms** fields. If the bounds of *ct* cannot be extracted because the bit pattern in *ct* does not correspond to a permitted value of the capability type (e.g. **length** is negative), then an exception is raised.

If the extracted bounds of *ct* are within the bounds of *cb*, and the permissions of *ct* are within the permissions of *cb*, then *cd* is set equal to *cb* with the **base**, **length**, **offset**, **perms** and **uperms** of *ct*.

If *ct* is sealed, this instruction does not copy its **otype** into *cd*. With compressed capabilities, a different representation may be used for the bounds of sealed and unsealed capabilities. If *ct* is sealed, CBuildCap will change the representation of the bounds so that their values are preserved.

Because *ct*.**tag** is not required to be set, there is no guarantee that the bounds of *ct* will be in canonical form. CBuildCap may convert the bounds into canonical form rather than simply copying their bit representation.

CBuildCap does not copy the fields of *ct* that are reserved for future use.

CBuildCap can be used to set the tag bit on a capability (e.g., one whose non-tag contents has previously been swapped to disk and than reloaded into memory, or during dynamic linking as untagged capability values are relocated and tagged after being loaded from a file). This provides both improved efficiency relative to manual rederivation of the tagged capability via a series of instructions, and also provides improved architectural abstraction by avoiding embedding the rederivation sequence in code.

Semantics

```
checkCP2usable();
let cb_val = readCapRegDDC(cb);
let ct_val = readCapReg(ct);
let cb_base = getCapBase(cb_val);
let ct_base = getCapBase(ct_val);
let cb_top = getCapTop(cb_val);
let ct_top = getCapTop(ct_val);
let cb_perms = getCapPerms(cb_val);
let ct_perms = getCapPerms(ct_val);
let ct_offset = getCapOffset(ct_val);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if ct_base < cb_base then
  raise_c2_exception(CapEx_LengthViolation, cb)
else if ct_top > cb_top then
  raise_c2_exception(CapEx_LengthViolation, cb)
 else if ct_base > ct_top then /* check for length < 0 - possible because ct might
      be untagged */
  raise_c2_exception(CapEx_LengthViolation, ct)
 else if (ct_perms & cb_perms) != ct_perms then
  raise_c2_exception(CapEx_UserDefViolation, cb)
else
{
  let (exact, cd1) = setCapBounds(cb_val, to_bits(64, ct_base), to_bits(65,
      ct_top));
  let (representable, cd2) = setCapOffset(cd1, to_bits(64, ct_offset));
  let cd3 = setCapPerms(cd2, ct_perms);
  {
    assert(exact, "CBuildCap: setCapBounds was not exact"); /* base and top came
        from ct originally so will be exact */
    assert(representable, "CBuildCap: offset was not representable"); /* similarly
         offset should be representable XXX except for fastRepCheck */
    writeCapReg(cd, cd3);
  }
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cd*, *cb* or *ct* is a reserved register and **PCC.perms** does not grant *Permit_Access_System_Registers*.
- *cb*.**tag** is not set.
- *cb* is sealed.
- The bounds of *ct* are outside the bounds of *cb*.
- The values of **base** and **length** found in *ct* are not within the range permitted for a capability with its **tag** bit set.
- *ct*.**perms** grants a permission that is not granted by *cb*.**perms**.
- *ct*.**uperms** grants a permission that is not granted by *cb*.**uperms**.

Notes

- This instruction acts both as an optimization, and to provide architectural abstraction in the face of future change to the capability model. A similar effect, albeit with reduced abstraction, could be achieved by using CGetBase, CGetLen and CGetPerm to query *ct*, and then using CSetBounds and CAndPerm to set the bounds and **perms** of *cd*.

- Despite the description of its intended use above, CBuildCap does not actually require that *ct* have an unset tag.

- *ct* might be a sealed capability that has had its **tag** bit cleared. In this case (assuming an exception is not raised for another reason), *cd* will be unsealed and the bit representation of the **base** and **length** fields might be changed to take account of the differing compressed representations for sealed and unsealed capabilities.

- This instruction can not be used to break security properties of the capability mechanism (such as monotonicity) because *cb* must be a valid capability and the instruction cannot be used to create a capability that grants rights that were not granted by *cb*.

- As the tag bit on *ct* does not need to be set, there is no guarantee that the bit pattern in *ct* was created by clearing the tag bit on a valid capability. It might be an arbitrary bit pattern that was created by other means. As a result, there is no guarantee that the bit pattern in *ct* corresponds to the encoding of a valid value of the capability type, especially when capability compression is in use. Fields might have values outside of their defined range, and invariants such as $\textbf{base} \geq 0$, $\textbf{base} + \textbf{length} \leq 2^{64}$ or $\textbf{length} \geq 0$ might not be true. In addition, fields might not be in a canonical (normalized) form. CBuildCap checks that the **base** and **length** fields are within the permitted range for the type and satisfy the above invariants, raising a length exception if they are not. If the fields are not in normalized form, CBuildCap may renormalize them rather than simply copying the bit pattern from *ct* into *cd*.

- The type constraint $cd.\textbf{tag} \implies cd.\textbf{base} \geq 0$ is guaranteed to be satisfied because $cb.\textbf{base} \geq 0$ and an exception would be raised if $ct.\textbf{base} \leq cb.\textbf{base}$.

- The type constraint $cd.\textbf{tag} \implies cd.\textbf{base} + cd.\textbf{length} \leq 2^{64}$ is guaranteed to be satisfied because this constraint is true for *cb*, and an exception would be raised if $ct.\textbf{base} + ct.\textbf{length} > cb.\textbf{base} + cb.\textbf{length}$.

- Is the value of *cd* guaranteed to be representable? If *ct* was created by clearing the tag bit on a capability, then its bounds can be represented exactly and there will be no loss of precision. If *ct* is sealed, then there is a potential issue that the values of the bounds that are representable in a sealed capability are not the same as the range of bounds that are representable in an unsealed capability. We rely on a property of the existing capability formats that if a value of the bounds is representable in a sealed capability, then it is also representable in an unsealed capability.

- As CBuildCap is not able to restore the seal on a re-tagged capability, it is intended to be used alongside CCSeal, which will conditionally seal a capability based on a **otype** value extracted with `CCopyType`. These instructions will normally be used in sequence to *(i)* re-tag a capability with CBuildCap, *(ii)* extract a possible object type from the untagged value with `CCopyType`, and *(iii)* conditionally seal the resulting capability with `CCSeal`.

- The typical use of `CBuildCap` assumes that there is a single capability *cb* whose bounds include every capability value that is expected to be encountered in *ct* (with out of range values being an error). The following are two examples of situations where this is not the case, and the sequence of instructions to recreate a capability might need to decide which capability to use as *cb*: (a) The operating system has enforced a "write xor execute" policy, and the program attempting to recreate *ct* has a capability with *Permit_Write* permission and a capability with *Permit_Execute* permission, but does not have a capability with both permissions. (b) The capability in *ct* might be a capability that authorizes sealing with the *Permit_Seal* permission, and the program attempting to recreate it has a capability for a range of memory addresses and a capability for a range of **otype** values, but does not have a single capability that includes both ranges.

CCopyType: Import the otype field of a Capability

Format

CCopyType cd, cb, ct

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cb	ct	0x1e

Description

CCopyType attempts to interpret the contents of *ct* as if it were a valid capability (even though *ct*.**tag** is not required to be set, and so might contain any bit pattern), and extracts its **otype** field. If *ct* is sealed, *cd* is set to *cb* with its **offset** field set to *ct*.**otype** $-$ *cb*.**base**. If *ct* is not sealed, *cd* is set to the NULL capability with its **base** $+$ **offset** fields set to -1.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let ct_val = readCapReg(ct);
let cb_base = getCapBase(cb_val);
let cb_top  = getCapTop(cb_val);
let ct_otype = unsigned(ct_val.otype);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if ct_val.sealed then
{
  if ct_otype < cb_base then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if ct_otype >= cb_top then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else
  {
    let (success, cap) = setCapOffset(cb_val, to_bits(64, ct_otype - cb_base));
    assert(success, "CopyType: offset is in bounds so should be representable");
    writeCapReg(cd, cap);
  }
}
else
  writeCapReg(cd, int_to_cap(0xffffffffffffffff))
```

Exceptions

A coprocessor 2 exception is raised if:

- *cd*, *cb* or *ct* are reserved registers and **PCC** does not grant *Access_System_Registers* permission.
- *cb*.**tag** is not set.
- *cb* is sealed.
- *ct*.**otype** is outside the bounds of *cb*.

Notes

- The intended use case for this instruction is as part of a routine for resetting the tag bit on a capability that has had its tag bit cleared (e.g. by being swapped out to disk and then back into memory).

  It is a requirement of this specification that if a capability has its tag bit cleared (either with `CClearTag` or by copying it as data), and `CCopyType` is used to extract the **otype** field of the result, then *cd*.**base** + *cd*.**offset** will be equal to the **otype** of the original capability if it was sealed, and *cd*.**offset** will be -1 if the original capability was not sealed.

- Typical usage of this instruction will be to use `CBuildCap` to extract the bounds and permissions of a capability, `CCopyType` to extract the **otype**, and then use `CCSeal` to seal the result of the first step with the correct **otype**.
- This instruction is an optimization. A similar effect could be achieved by using `CGetType` to get *ct*.**otype** and then `CSetOffset` to set *cd*.**offset**.
- -1 is not a valid value for the **otype** field, so the result distinguishes between the case when *ct* was sealed and the case when it was not sealed.
- If *ct* is sealed and an exception is not raised, then the result is guaranteed to be representable, because the bounds checks ensure that *cd*'s cursor is within its bounds.
- If *ct*.**otype** is outside of the bounds of *ct*, this is an error condition (attempting to reconstruct a capability that *cb* does not give you permission to create). In order to catch this error condition near to where the problem occurred, we raise an exception. This also has the benefit of avoiding the case where changing *cb*'s **offset** results in a value that is not representable, as explained in the previous note.

CCSeal: Conditionally Seal a Capability

Format

CCSeal cd, cs, ct

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cs	ct	0x1f

Description

If *ct*.**tag** is false or *ct*.**base** + *ct*.**offset** = −1, *cs* is copied into *cd*. Otherwise, capability register *cs* is sealed with an **otype** of *ct*.**base** + *ct*.**offset** and the result is placed in *cd* as follows:

- *cd* is sealed with *cd*.**otype** set to *ct*.**base** + *ct*.**offset**;
- and the other fields of *cd* are copied from *cs*.

ct must grant *Permit_Seal* permission, and the new **otype** of *cd* must be between *ct*.**base** and *ct*.**base** + *ct*.**length** − 1.

Semantics

```
checkCP2usable();
let cs_val = readCapReg(cs);
let ct_val = readCapReg(ct);
let ct_cursor = getCapCursor(ct_val);
let ct_top    = getCapTop(ct_val);
let ct_base   = getCapBase(ct_val);
if not (cs_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cs)
else if not (ct_val.tag) | (getCapCursor(ct_val) == MAX_U64) then
  writeCapReg(cd, cs_val)
else if cs_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cs)
else if ct_val.sealed then
  raise_c2_exception(CapEx_SealViolation, ct)
else if not (ct_val.permit_seal) then
  raise_c2_exception(CapEx_PermitSealViolation, ct)
else if ct_cursor < ct_base then
  raise_c2_exception(CapEx_LengthViolation, ct)
else if ct_cursor >= ct_top then
  raise_c2_exception(CapEx_LengthViolation, ct)
else if ct_cursor > max_otype then
  raise_c2_exception(CapEx_LengthViolation, ct)
else
{
  let (success, newCap) = sealCap(cs_val, to_bits(24, ct_cursor));
```

```
    if not (success) then
        raise_c2_exception(CapEx_InexactBounds, cs)
    else
        writeCapReg(cd, newCap)
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cs*.**tag** is not set.
- *cs* is sealed.
- *ct*.**tag** is set and *ct* is sealed.
- *ct*.**perms**.*Permit_Seal* is not set.
- *ct*.**tag** and *ct*.**offset** $\geq$ *ct*.length
- *ct*.**tag** and *ct*.**base** $+$ *ct*.**offset** $>$ *max_otype*
- The bounds of *cb* cannot be represented exactly in a sealed capability.

Notes

- If capability compression is in use, the range of possible (**base**, **length**, **offset**) values might be smaller for sealed capabilities than for unsealed capabilities. This means that CCSeal can fail with an exception in the case where the bounds are no longer precisely representable.

- This instruction provides two means of indicating that the capability should not be sealed: either clearing the tag bit on *ct* or setting *ct*'s cursor to -1. A potential problem with just using a cursor of -1 (rather than clearing the tag bit) to disable sealing is that, depending on *ct*'s **base** and **offset**, setting *ct*'s cursor to -1 might have a result that is not representable. However, the NULL capability has **tag** clear and can always have its cursor set to -1. (We implement casts from `int` to `int_cap_t` by setting the cursor of NULL to the value of the integer, and so this can hold a value of -1.) Directly clearing *ct*'s tag to indicate that sealing should not be performed will work, because it is always possible to clear the tag bit. Setting *ct*'s cursor to -1 with CSetOffset to indicate that sealing should not be performed will also work, because this will either set the cursor to -1 or (if the result would not be representable) both clear the tag bit and set the cursor to -1. The latter method may be preferred in a code sequence that extracts the **otype** of a capability with CGetType, getting a value of -1 if the capability is not sealed, setting the cursor of *ct* to the result, and then using CCSeal to seal a new capability to the same **otype** as the original.

CGetAndAddr: Get Address of Capability with mask

Format

CGetAndAddr rd, cb, rs

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	rs	0x23

Description

rd is set to *cb*.**a** logically ANDed with *rs*.

Semantics

```
checkCP2usable();
let capVal = readCapReg(cb);
let rs_val = rGPR(rs);
wGPR(rd) = capVal.address & rs_val;
```

Notes

- This instruction may be useful when C is using the lower bits of a pointer with known alignment to store additional metadata.
- This instruction is also useful when using a capability type as a union of pointers, integers and floating pointer numbers using a NaN-boxing representation.
- In a merged register file this instruction would not be necessary since the integer bitwise-and could be used instead.

CGetFlags: Move Flags to an Integer Register

Format

CGetFlags rd, cb

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	0x12	0x3f

Description

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

Semantics

```
checkCP2usable();
let capVal = readCapReg(cb);
wGPR(rd) = zero_extend(getCapFlags(capVal));
```

CLoadTags: Read Multiple Tags to Integer Register

Format

CLoadTags rd, cb

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x00	rd	cb	0x1E	0x3F

Note: If the encoded value of *cb* is zero, this instruction will use **DDC** as the *cb* operand

Description

The *tags* from memory referenced by *cb* are loaded to *rd*, with bit significance increasing with memory address. The result of this instruction must be coherent with other processors, *as if* the corresponding data memory words had been loaded. The number of tags loaded is an implementation-defined constant but is constrained to be a power of two, at least 1, and no more than the width of *rd*.
Capability register *cb* must contain a capability that grants permission to load capabilities. The virtual address *cb*.**base** + *cb*.**offset** must be suitably aligned; the precise requirements are, again, implementation defined, but must equal the width of memory corresponding to the tags loaded. If any tag to be loaded corresponds to memory out of bounds of *cb*, a length violation is indicated.

Semantics

```
checkCP2usable();
let cb_val = readCapRegDDC(cb);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_load) then
  raise_c2_exception(CapEx_PermitLoadViolation, cb)
else if not (cb_val.permit_load_cap) then
  raise_c2_exception(CapEx_PermitLoadViolation, cb)
else
{
  let vAddr   = getCapCursor(cb_val);
  let vAddr64 = to_bits(64, getCapCursor(cb_val));
  if (vAddr + caps_per_cacheline ∗ cap_size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if vAddr < getCapBase(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if not(vAddr % (cap_size ∗ caps_per_cacheline) == 0) then
    SignalExceptionBadAddr(AdEL, vAddr64)
  else
    {
      let pAddr     = TLBTranslate(vAddr64, LoadData);
      x : bits(64)  = zeros();
      foreach(i from 0 to (caps_per_cacheline - 1)) {
        let (tag, _) = MEMr_tagged(pAddr + i*cap_size, cap_size);
        x[i] = tag;
      };
      wGPR(rd) = x;
    }
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb*.**tag** is clear.
- *cb* is sealed.
- *cb*.**perms**.*Permit_Load* is not set.
- *cb*.**base** $+$ *cb*.**offset** + $n * \mathit{capability\_size} >$ *cb*.**base** $+$ *cb*.**length**, where n is the number of capabilities to be fetched (`caps_per_cacheline` in the Sail code).
- *cb*.**base** $+$ *cb*.**offset** $<$ *cb*.**base**.

An address error during load (AdEL) exception is raised if:

- The virtual address *addr* is not $n * \mathit{capability\_size}$ aligned.

Notes

- In practice, the number of tags loaded is likely less arbitrary than it may appear. Usually, the implementation's cache fabric determines the minimum granularity of coherency and already tracks tag bits along with each cache line, and so this instruction fetches the tag bits from the cache line indicated by *cb*.**base** $+$ *cb*.**offset**. Of course, additional complexity in the cache and tag cache fabric could allow this instruction to retrieve more tags than in a cache line. Also, the number of tags this instruction loads should be a power-of-two to avoid alignment issues and to preserve page divisibility in systems with MMUs. In order to reduce DRAM traffic, it is desirable that this *tag fetch* not require loading the corresponding data and not necessarily evict other lines from caches. (However, a non-zero result is probably a reasonable hint that a capability load is likely to follow.)

CLShC: Load Short Capability via Capability

Format

CLShC cd, rt, offset(cb)
CLShCR cd, rt(cb)
CLShCI cd, offset(cb)

Note: If the encoded value of *cb* is zero, this instruction will use **DDC** as the *cb* operand

Description

Capability register *cd* is loaded with the decoded form of the short capability located at the memory location specified by $addr \triangleq cb.\mathbf{base} + cb.\mathbf{offset} + rt + 8 * offset$, provided this address is suitably aligned for short capabilities. Capability register *cb* must contain a capability that grants permission to load capabilities.
The tag bit associated with *cd* is set if the tag bits associated with *addr* indicate that this address contain a short capability and *cb* bears Permit_Load_Capability. See appendix D.15 for possible tag representations.

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb* is sealed.
- *cb*.**perms**.*Permit_Load* is not set.
- *addr* + *short_capability_size* $>$ *cb*.**base** $+$ *cb*.**length**.
- *addr* $<$ *cb*.**base**.

An address error during load (AdEL) exception is raised if:

- The virtual address *addr* is not *capability_size* aligned.

Notes

- *offset* is interpreted as a signed integer.
- The `CLShCI` mnemonic is equivalent to `CLShC` with *cb* being the zero register ($zero). The `CLShCR` mnemonic is equivalent to `CLShC` with *offset* set to zero.
- Although the *short_capability_size* may vary, the offset is always in multiples of 8 bytes (64 bits).
- See appendix D.15 for details of short capabilities. In particular, note that several permissions are not transported in short capabilities.

CSetFlags: Set Flags

Format

CSetFlags cd, cb, rt

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	cd	cs	rt	0xe

Description

Capability register *cd* is replaced with the contents of capability register *cb* with the **flags** field set to bits 0 .. *max_flags* of integer register *rd*.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let rt_val = rGPR(rt);
if cb_val.tag & cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else
{
  let newCap  = setCapFlags(cb_val, truncate(rt_val, num_flags));
  writeCapReg(cd, newCap);
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb* is sealed.

CSShC: Store Short Capability via Capability

Format

CSShC cs, rt, offset(cb)
CSShCR cs, rt(cb)
CSShCI cs, offset(cb)

Note: If the encoded value of *cb* is zero, this instruction will use **DDC** as the *cb* operand

Description

Capability register *cs* is encoded to a short capability form and stored at the memory location specified by $addr \triangleq cb.\mathbf{base} + cb.\mathbf{offset} + rt + 16 * \mathit{offset}$, provided *addr* is suitably aligned for short capabilities. Capability register *cb* must contain a capability that grants permission to store capabilities. The bits in the tag memory associated with *addr* are updated to indicate a short capability resides at this location if *cs*.**tag** is set and encoding was successful. See appendix D.15 for possible tag representations.

Encoding

As per appendix D.15, several properties must hold of the capabilities given to `CSShC` for a successful store to take place:

- *cs* may be out of bounds by strictly less than 4 GiB.
- The bounds of *cs* must be representable in short capabilities.
- If *cs* is sealed, its **otype** must be representable in short capabilities.

If any of the above tests fail, the store will take place but will update the tags corresponding to *addr* to indicate that the memory contains data.
Further, the following permission bits may not be stored in the short capability format, and may read back (via `CLShC`) as false:

- Permit_Seal
- Permit_Unseal
- Permit_SetCID
- Permit_Access_System_Registers
- Permit_Store_Local_Capability.

Software must, however, not depend on `CSShC` to clear these bits; an `CAndPerm` must be used to ensure that these rights are discarded.

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb* is sealed.
- *cb*.**perms**.*Permit_Store* is not set.
- *cb*.**perms**.*Permit_Store_Capability* is not set.
- *cb*.**perms**.*Permit_Store_Local* is not set and *cs.tag* is set and *cs*.**perms**.*Global* is not set.
- $addr + short\_capability\_size > cb.\mathbf{base} + cb.\mathbf{length}$.
- $addr < cb.\mathbf{base}$.
- *cs*.**base**, *cs*.**base** + *cs*.**length**, and the store's target memory location computed above differ in their top 32 bits.

A TLB Store exception is raised if:

- *cs*.**tag** is set and the *S* bit in the TLB entry for the page containing *addr* is not set.

An address error during store (AdES) exception is raised if:

- The virtual address *addr* is not *short_capability_size* aligned.

Notes

- If the address alignment check fails and one of the security checks fails, a coprocessor 2 exception (and not an address error exception) is raised. The priority of the exceptions is security-critical, because otherwise a malicious program could use the type of the exception that is raised to test the bottom bits of a register that it is not permitted to access.
- *offset* is interpreted as a signed integer.
- The `CSShCI` mnemonic is equivalent to `CSShC` with *cb* being the zero register ($zero). The `CSShCR` mnemonic is equivalent to `CSShC` with *offset* set to zero.
- Although the *short_capability_size* can vary, the offset is always in multiples of 8 bytes (64 bits).

CTestSubset: Test that Capability is a Subset of Another

Format

CTestSubset rd, cb, ct

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rd	cb	ct	0x20

Note: If the encoded value of *cb* is zero, this instruction will use **DDC** as the *cb* operand

Description

CTestSubset tests if the bounds of *ct* are within the bounds of *cb*, and the permissions of *ct* are within the permissions of *cb*, setting *rd* to *1* if so, and *0* if not.

Semantics

```
checkCP2usable();
let cb_val = readCapRegDDC(cb);
let ct_val = readCapReg(ct);
let ct_top   = getCapTop(ct_val);
let ct_base  = getCapBase(ct_val);
let ct_perms = getCapPerms(ct_val);
let cb_top   = getCapTop(cb_val);
let cb_base  = getCapBase(cb_val);
let cb_perms = getCapPerms(cb_val);
let result = if cb_val.tag != ct_val.tag then
              0b0
         else if ct_base < cb_base then
              0b0
         else if ct_top > cb_top then
              0b0
         else if (ct_perms & cb_perms) != ct_perms then
              0b0
         else
              0b1;
wGPR(rd) = zero_extend(result);
```

Exceptions

A coprocessor 2 exception is raised if:

- *cd*, *cb* or *ct* is a reserved register and **PCC.perms** does not grant *Permit_Access_System_Registers*.

Notes

- This instruction was originally motivated as an additional check for `CToPtr`. A conversion of a capability to a pointer with respect to a default capability would normally expect that the entire capability is accessible within the default capability with (at least) the original permissions. `CTestSubset` can perform this assertion, and a `CMove` instruction can replace the result of the `CToPtr` with NULL upon failure.

- Another use case for this instruction is in garbage collection. For this application, we want to be able to test if one capability is a subset of the other even if one is sealed and the other is not. (For the purposes of garbage collection, a sealed reference to a region of memory is still a reference to that region of memory). With compressed capabilities, the bounds are represented differently for sealed and unsealed capabilities, but `CTestSubset` is still able to perform the subset check.

CRepresentableAlignmentMask: Retrieve Mask to Align Capabilities to Precisely Representable Address

Format

CRepresentableAlignmentMask rt, rs

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rt	rs	0x11	0x3f

Description

rt is set to a mask that can be used to align down addresses to a value that is sufficiently aligned to set precise bounds for the nearest representable length of *rs* (as obtained by `CRoundRepresentableLength`).

Notes

- The result of `CRepresentableAlignmentMask` can be used as the mask argument for `CAndAddr` to create a capability with an address that is sufficiently aligned to perform `CSetBoundsExact`. This `CSetBoundsExact` is guarenteed to succeed if the size is rounded using `CRoundRepresentableLength`.
- The required alignment of an allocation of size *rs* can be computed by negating *rt* and adding one.
- Combined with `CRoundRepresentableLength` this instruction can be used in memory allocators to guarantee non-overlapping allocations.
- This instruction can be useful to adjust the stack pointer to an address that is suitably aligned for dynamic allocations.
- An alternative to this instruction is the use of count-leading-zeroes and count-trailing-zeros instructions followed by shifting and masking. However, this requires encoding knowledge of the underlying precision in the resulting binary and can therefore result in incompatibilities with future architectures that use a different compression algorithm.

CRoundRepresentableLength: Round to Next Precisely Representable Length

Format

CRoundRepresentableLength rt, rs

31 26	25 21	20 16	15 11	10 6	5 0
0x12	0x0	rt	rs	0x10	0x3f

Description

rt is set to the smallest value greater or equal to *rs* that can be used by `CSetBoundsExact` without trapping (assuming a suitably aligned base).

Exceptions

Notes

- This instruction is useful when implementing allocators to round up allocation sizes to a size that can be precisely bounded (and will therefore not overlap with any other allocations). It is also useful when using `mmap()`, since the requested size must be a precisely representable length.

- An alternative to this instruction is the use of count-leading-zeroes and count-trailing-zeros instructions followed by shifting and masking. However, this requires encoding knowledge of the underlying precision in the resulting binary and can therefore result in incompatibilities with future architectures that use a different compression algorithm.

Appendix E

CHERI-128 Compression (Deprecated)

On the path to developing our current capability compression scheme, CHERI Concentrate (see Section 3.4.5), we developed three earlier 128-bit formats based on floating-point compression of bounds relative to the virtual address in a capability. We present those techniques here to explore potential tradeoffs in their designs and potential alternative approaches.

E.1 CHERI-128 candidate 1

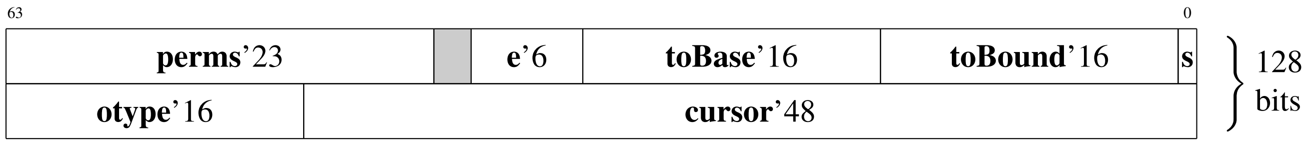

Figure E.1: CHERI-128 c1 memory representation of a capability

s The **s** flag is set if the capability is sealed and is clear otherwise. See the discussion of **otype** below.

e The 6-bit **e** field gives an exponent for the **toBase** and **toBound** fields. The exponent is the number of bits that **toBase** and **toBound** should be shifted before being added to **cursor** when performing bounds checking.

toBase This 16-bit field contains a signed integer that is to be shifted by **e** and added to **cursor** (with the lower bits set to 0) to give the **base** of the capability. This field must be adjusted upon any update to **cursor** to preserve the **base** of the capability.

$$\mathrm{mask} = -1 << \mathrm{e}$$

$$\mathrm{base} = (\mathrm{toBase} << \mathrm{e}) + \mathrm{cursor}\&\mathrm{mask}$$

perms The 23-bit **perms** field contains precisely the same 15-bits of permissions as the 256-bit version. The **perms** field has 8-bits of software-defined permissions at the top, down from 16-bits in the 256-bit version.

toBound This 16-bit field contains a signed integer that is to be shifted by **e** and added to **cursor** (with the lower bits set to 0) to give the bound of the capability. The **length** of the capability is reported by subtracting **base** from the resulting bound. This field must be adjusted upon any update to **cursor** to preserve the **length** of the capability.

$$\mathbf{base} + \mathbf{length} = (\mathbf{toBound} << \mathbf{e}) + \mathbf{cursor} \& \mathbf{mask}$$

otype The 16-bit **otype** field corresponds directly to the **otype** bit vector but is defined only when the capability is sealed. If **s** is cleared, the architectural **otype** is $2^{64} - 1$ but and the bits devoted to object type representation are instead an extension of **cursor**.

cursor The 64-bit **cursor** value holds a 48-bit absolute virtual address that is equal to the architectural **base + offset**. The address in **cursor** is the full 64-bit MIPS virtual address when the capability is unsealed, and it holds a compressed virtual address when the capability is sealed. The compression format places the 5 bits of the address segment in bits [47:42], replacing unused bits of the virtual address. When the capability is unsealed, the segment bits are placed at the top of a 64-bit address and the rest are “sign" extended.

$$\mathbf{cursor} = \mathbf{base} + \mathbf{offset}$$

Compression Notes When CSetBounds is not supplied with a length that can be expressed with byte precision, the resulting capability has an **e** that is non-zero and **toBase** and **toBound** describe units of size $2^{\mathbf{e}}$. **e** is selected such that the pointer can wander outside of the bounds by at least the entire size of the capability both below the base and above the bound without becoming unrepresentable. As a result, a 16-bit **toBase** and **toBound** require both a sign bit and a bit for additional range that cannot contribute to the size of representable objects. The greatest length that can be represented with byte granularity for a 16-bit **toBase** and **toBound** is $2^{14} = 16KiB$. The resulting alignment in bytes required for an allocation can be derived from the length by rounding to the nearest power of two and dividing by this number.

$$alignment\_bits = \lceil log_2(X) \rceil - 14$$

E.2 CHERI-128 candidate 2 (Low-fat pointer inspired)

baseBits This 16-bit field gives bits to be inserted into **cursor**[**e**+15:**e**], with the lower bits set to 0, to produce the base of the capability.

$$\mathbf{base} = \{\mathbf{cursor}[63 : \mathbf{e} + 16] + \mathbf{correction}, \mathbf{baseBits}\} \ll \mathbf{e}$$

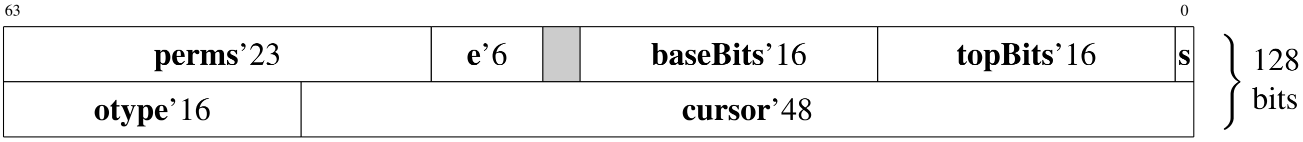

Figure E.2: CHERI-128 c2 memory representation of a capability

The bits above $(\mathbf{e}+16)$ in **cursor** may differ from **base** by at most 1, i.e.

$$\mathbf{correction} = f(\mathbf{baseBits}, \mathbf{topBits}, \mathbf{cursor}[\mathbf{e}+15:\mathbf{e}]) = (1, 0, or-1)$$

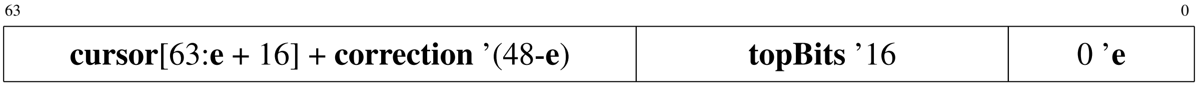

Figure E.3: CHERI-128 c2 base construction

topBits This 16-bit field gives bits to be inserted into the bits of **cursor** at **e** to produce the representable top of the capability equal to (top - 1024). To compute the top, a circuit must insert **topBits** at **e**, set the lower bits to 0, subtract 1024, and add a potential carry bit. The carry bit is implied if **topBits** is less than **baseBits**, as the top will never be less than the bottom of an object.

$$\mathbf{top} = \{\mathbf{cursor}[63:\mathbf{e}+16] + \mathbf{correction}, \mathbf{topBits}, 0\}$$

The bits above $(\mathbf{e}+16)$ in **cursor** may differ from **top** by at most 1:

$$\mathbf{correction} = f(\mathbf{baseBits}, \mathbf{topBits}, \mathbf{cursor}[\mathbf{e}+15:\mathbf{e}]) = (1, 0, or-1)$$

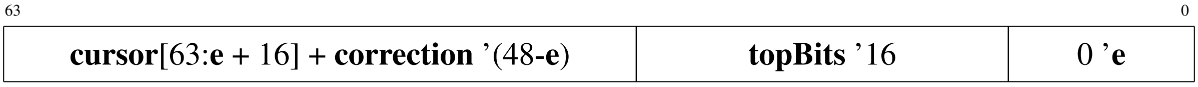

Figure E.4: CHERI-128 c2 top bound construction

Candidate 2 Notes Candidate 2 is inspired by “Low-fat pointers" [62], which insert selected bits into the pointer to produce the bounds. The Low-fat pointer representation does not allow a pointer to go out of bounds, but we observe that **cursor** could wander out of bounds without causing **base** and **top** to become ambiguous as long as these three remain within the same $2^{(\mathrm{e}+16)}$-sized region. Candidate 2 sets the edges of this range to a fixed 1024^{e} bytes beyond each bound, and encodes these in the top and bottom fields to allow high-speed access during pointer arithmetic.

E.3 CHERI-128 candidate 3

After substantial exploration, we developed a third compression model, CHERI-128, which is somewhat similar to candidate 2 with two improvements:

- Condense hardware and software permissions, making room for larger **baseBits** and **topBits** fields in the unsealed capability format.
- A new sealed capability format, which reduces the size of **baseBits** and **topBits** to make room for a larger **otype** and software-defined permissions. **otype** no longer aliases bits of **cursor** but rather the bounds metadata.

Subsequent refinement of CHERI-128 gave rise to our current compression scheme, CHERI Concentrate [152], detailed in Section 3.4.5.

Alternative exponents The CHERI-128 scheme treats the exponent (**e**) as a 2^{e} multiplier, though we note that in our current implementation the bottom two bits of **e** are forced to be zero, so the exponent is actually $16^{\mathrm{e}[5:2]}$. Clearly we could chose different precision for the exponent, trading precision for hardware cost and bits in the capability format.

Alternative precision for T and B Currently we use 20-bits to represent top and bottom bounds (**T** and **B**). This gives us a great deal of precision; however, reducing these bit widths may well be workable for a broad range of software. In particular, we may wish to reduce the size of these fields in the sealed capability format since sealed objects are a new concept and introducing strong alignment requirements does not appear to have significant penalty. Similarly, the bit widths could be increased for better precision.

Alternative otype size We may wish to adjust the field widths for the sealed capability format to allow a larger **otype**, thereby allowing more sandboxes without risk of **otype** reuse.

Alternative perms We may wish to adjust field widths to increase the number of permission bits.

E.3.1 Implementation

This section describes the compressed capability format known as CHERI-128 [55]. The compressed in-memory formats for CHERI-128 unsealed and sealed capabilities are depicted in Figures E.5 and E.6.

μperms Hardware permissions for this format are trimmed from those listed in Table 3.1 by consolidating system registers. The condensed format is listed in Table E.1

e Is an exponent for both the top (**T**) and bottom (**B**) bits — see calculations below. Currently the bottom two bits of **e** are zero.

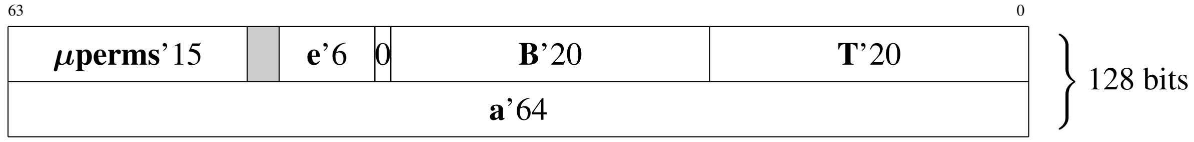

Figure E.5: Unsealed CHERI-128 memory representation of a capability

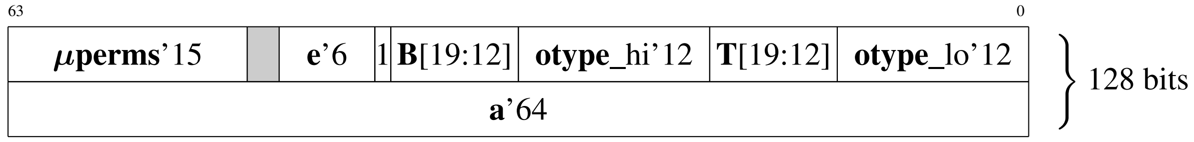

Figure E.6: Sealed CHERI-128 memory representation of a capability

s Indicates if a capability is sealed or not, listed simply as 0 or 1 in Figures E.5 and E.6 respectively due to each format being specific to the state of the sealed bit.

a A 64-bit value holding a virtual address equal to the architectural **base** + **offset**.

B A 20-bit value used to reconstruct the architectural **base**. When deriving a capability with a requested **base_req** and **rlength**, we have:

$$\mathrm{B} = \left\lfloor \frac{\mathbf{base\_req}}{2^{\mathrm{e}}} \right\rfloor \bmod 2^{20}$$

Which can be rewritten as a bit-manipulation:

$$\mathrm{B} = \mathbf{base\_req}[19 + \mathrm{e} : \mathrm{e}]$$

architectural bit#	*μ***perms** bit#	Name
perms[0]	0	Global
perms[1]	1	Permit_Execute
perms[2]	2	Permit_Load
perms[3]	3	Permit_Store
perms[4]	4	Permit_Load_Capability
perms[5]	5	Permit_Store_Capability
perms[6]	6	Permit_Store_Local_Capability
perms[7]	7	Permit_Seal
perms[8]	8	Permit_CCall
perms[9]	9	Permit_Unseal
perms[10]	10	Access_System_Registers
uperms[15–18]	11–14	Software-defined permissions

Table E.1: Permission bit mapping

For sealed capabilities, $\mathbf{B}[11:0] = 0$

T A 20-bit value used to reconstruct the architectural **top** $(\mathtt{base} + \mathtt{length})$. When deriving a capability with a requested **base_req** and **rlength**, we have:

$$\mathbf{T} = \left\lceil \frac{\textbf{base\_req} + \textbf{rlength}}{2^{\mathrm{e}}} \right\rceil \bmod 2^{20}$$

Rewritten as bit manipulations:

$$\mathbf{T} = \begin{cases} (\textbf{base\_req} + \textbf{rlength})[19 + \mathrm{e} : \mathrm{e}], & \text{if } (\textbf{base\_req} + \textbf{rlength})[\mathrm{e} - 1 : 0] = 0 \\ (\textbf{base\_req} + \textbf{rlength})[19 + \mathrm{e} : \mathrm{e}] + 1, & \text{otherwise} \end{cases}$$

otype The 24-bit **otype** field (concatenation of the two **otype** fields of Figure E.6) corresponds to the least-significant 24 bits of the architectural **otype** bit vector. These bits are not allocated in an unsealed capability, and the **otype** of an unsealed capability is $2^{64} - 1$; the encoded value $2^{24} - 1$ is reserved.

The hardware computes **e** according to the following formula:

$$\mathrm{e} = \left\lceil \mathbf{plog_2}\left(\frac{(\textbf{rlength}) \cdot (1 + 2^{-6})}{2^{20}}\right) \right\rceil \text{where} \mathbf{plog_2}(x) = \begin{cases} 0, & \text{if } x < 1 \\ \mathbf{log_2}(x), & \text{otherwise} \end{cases}$$

which is equivalent to the following bit manipulation:

$$\mathrm{e} = \mathrm{idxMSNZ}((\textbf{rlength} + (\textbf{rlength} \gg 6)) \gg 19)$$

where:

- $\mathrm{idxMSNZ}(x)$ returning the index of the most significant bit set in x
- $(\textbf{rlength} + (\textbf{rlength} \gg 6))$ being a 65-bit result

Note that:

- **e** is rounded up to the nearest representable value. In the current implementation the bottom two bits of **e** are zero. For example, the above **e** calculation returned the value 1, then it would be rounded up to 4.
- **rlength** is artificially inflated in the computation of **e** in such a way that:

  $$\textbf{rlength} + 8\mathrm{KiB} \leq 2^{\mathrm{e}+20}$$

  to ensure that there is a representable region which is at least one page above and below the base and bound. This allows pointers to stray up to a page beyond the base and bound without causing an exception, a feature which is necessary to run much legacy C-code.
- **e** is computed in such a way that loss of precision due to alignment requirements is minimized, i.e., **e** is the smallest natural n satisfying:

  $$\mathtt{maxLength}(n) \geq \textbf{rlength}\text{where}\mathtt{maxLength}(n) = \left\lfloor \frac{2^{n+20}}{1 + 2^{-6}} \right\rfloor$$

E.3.2 Representable Bounds Check

When **a** is incremented (or decremented) we need to ascertain whether the resulting capability is representable. We do not check to see if the capability is within bounds at this point, which is done only on dereference (load/store instructions).

We first ascertain if we are *inRange* and then if we are *inLimits*. The *inRange* test determines whether an inspection of only the lower bits of the pointer and increment can yield a definitive answer. The *inLimits* test assumes the success of the *inRange* test, and determines whether the update to $\mathbf{a}_{mid}$ could take it beyond the limits of the representable space.

The increment i is *inRange* if its absolute value is less than s, the size of the representable region:

$$inRange = -s < i < s$$

This reduces to a test that all the bits of I_{top} ($i[63 : \mathbf{e} + 20]$) are the same. For *inLimits*, we need only $\mathbf{a}_{mid}$ ($\mathbf{a}[19 + \mathbf{e} : \mathbf{e}]$), I_{mid} ($i[\mathbf{e} + 19 : \mathbf{e}]$), and the sign of i to ensure that we have not crossed either R ($\mathbf{B} - 2^{12}$), the limits of the representable region:

$$inLimits = \begin{cases} I_{mid} < (R - \mathbf{a}_{mid} - 1), & \text{if } i \geq 0 \\ I_{mid} \geq (R - \mathbf{a}_{mid}) \wedge R \neq \mathbf{a}_{mid}, & \text{if } i < 0 \end{cases}$$

When we are incrementing upwards, we must conservatively subtract one from the representable limit to account for any carry that may propagate up from the lower bits of the full pointer add. When the increment is negative, we must conservatively disallow any operation where $\mathbf{a}_{mid}$ begins at the representable limit as the standard test would spuriously allow any negative offset.

One final test is required that ensures that, if $\mathbf{e} \geq 44$, any increment is representable. This handles a number of corner cases related to T, B, and $\mathbf{a}_{mid}$ describing bits beyond the top of the pointer. Our final fast *representable* check composes these three tests:

$$representable = (inRange \wedge inLimits) \vee (\mathbf{e} \geq 44)$$

E.3.3 Decompressing Capabilities

When producing the architectural **base** of a capability, the value is computed by inserting **B** into **a**[19+**e**:**e**], inserting zeros in **a**[**e**-1:0], and adding a potential correction **c_b** to **a**[63:20+**e**] as defined in Table E.2:

$$\begin{aligned} \mathbf{base}[63 : 20 + \mathbf{e}] &= \mathbf{a}[63 : 20 + \mathbf{e}] + \mathbf{c_b} \\ \mathbf{base}[19 + \mathbf{e} : \mathbf{e}] &= \mathbf{B} \\ \mathbf{base}[\mathbf{e} - 1 : 0] &= 0 \end{aligned}$$

When producing the architectural **top** ($= \mathbf{base} + \mathbf{length}$) of a capability, the value is computed by inserting **T** into **a**[19+**e**:**e**], inserting zeros in **a**[**e**-1:0], and adding a potential correction **c_t** to **a**[63:20+**e**] as defined in Table E.2:

$$\begin{aligned} \mathbf{top}[64 : 20 + \mathbf{e}] &= \mathbf{a}[63 : 20 + \mathbf{e}] + \mathbf{c_t} \\ \mathbf{top}[19 + \mathbf{e} : \mathbf{e}] &= \mathbf{T} \\ \mathbf{top}[\mathbf{e} - 1 : 0] &= 0 \end{aligned}$$

Note that **top** is a 65-bit quantity to allow the upper bound to be larger than the address space. For example, this is used at reset to allow the default data capability to address all of the virtual address space, because **top** must be one byte more than the top address. In this special case, $\mathbf{e} \geq 45$.
For sealed capabilities, $\mathbf{B}[11:0] = 0$ and $\mathbf{T}[11:0] = 0$.

We define

$$\mathbf{a}_{mid} = \mathbf{a}[19 + \mathbf{e} : \mathbf{e}]$$

$$R = \mathbf{B} - 2^{12}$$

$\mathbf{a}_{mid} < R$	$\mathbf{B} < R$	$\mathbf{c_b}$
0	0	0
0	1	$+1$
1	0	-1
1	1	0

$\mathbf{a}_{mid} < R$	$\mathbf{T} < R$	$\mathbf{c_t}$
0	0	0
0	1	$+1$
1	0	-1
1	1	0

Table E.2: Calculating $\mathbf{c_b}$ and $\mathbf{c_t}$

E.3.4 Bounds Alignment Requirements

Unsealed capabilities: Compressed capabilities impose bounds alignment requirements on software if precise bounds are required. The calculation of **e** determines the alignment requirement (see Section E.3.1):

$$alignment = 2^{\mathbf{e}}$$

where **e** is determined by the requested length of the region (**rlength**). Note that in the current implementation the bottom two bits of **e** are zero, so the value is rounded up.
Since the calculation of **e** is a little complicated, it can be convenient to have a conservative approximation:

$$\mathbf{rlength} < 2^{\mathbf{e}} \cdot \tfrac{3}{4}\text{MiB}$$

So the conservative approximation of **e** can be computed as follows (or the precise version used from Section E.3.1), noting that **e** is also rounded up to ensure the bottom two bits are zero:

$$\mathbf{e} = \left\lceil plog_2\left(\frac{\mathbf{rlength}}{\frac{3}{4}\text{MiB}}\right)\right\rceil$$

i.e. for an object length less than $\frac{3}{4}$ MiB you get byte alignment (since **e**=0 so $alignment = 1$). You then go to 16-byte alignment for objects less than $2^4 \cdot \frac{3}{4}\text{MiB} = 12\text{MiB}$, etc. Page alignment (4 KiB pages) is required only when objects are between 1 GiB and 3 GiB.
Note that the actual length of the region covered will be rounded up to the nearest $alignment$ boundary.

Sealed capabilities have more restrictive alignment requirements due to fewer bits available to represent **T** and **B**. The hardware will raise an exception when sealing an unsealed capability where the bottom 12 bits of **T** and **B** are not zero. As a consequence, the alignment becomes:

$$alignment = 2^{\mathbf{e}+12}$$

The relationship between **rlength** and **e** remains the same, but the actual length of the region covered will be rounded up to the new $alignment$. Thus, for small regions alignment is on 4 KiB (page) boundaries and the length of the region protected is a multiple of pages up to $\frac{3}{4}$ MiB. Length of region up to $2^4 \cdot \frac{3}{4} = 12$ MiB are aligned on 64 KiB boundaries. Similarly, a region of length 1 GiB to 3 GiB will be 16 MiB aligned.

Glossary

abstract capability Abstract capabilities are a conceptual abstraction that overlays the concrete capabilities of the architecture to describe the intended maintenance of capability lifespan across operations that violate architectural capability provenance. For example, if an OS kernel swaps a page containing a capability to and from disk, it will have to have its capability tag restored through re-derivation, so there is no longer an architectural provenance relationship between the two, but for application-level reasoning it is sometimes useful to regard there to be one.

address An integer address suitable for dereference within an address space. In CHERI-MIPS, capabilities are always interpreted in terms of virtual addresses. In CHERI-RISC-V, capabilities may be interpreted as virtual addresses – or physical addresses when operating in Machine Mode.

capability A capability contains an address, capability bounds describing a range of bytes within which addresses may be dereferenced, capability permissions controlling the forms of dereference that may be permitted (e.g., load or store), a capability tag protecting capability validity (integrity and capability provenance), and a capability object type indicating whether it is a sealed capability (and, if so, under which capability object type they are sealed) or unsealed capability. The address embedded within a capability may be a virtual address or a physical addresses depending on the current addressing mode; when used to authorize (un)sealing, the address is instead a capability object type.

In CHERI, capabilities are used to implement pointers with additional protections in aid of fine-grained memory protection, control-flow robustness, and other higher-level protection models such as software compartmentalization. Unlike a fat pointer, capabilities are subject to capability provenance, ensuring that they are derived from a prior valid capability only via valid manipulations, and capability monotonicity, which ensures that manipulation can lead only to non-increasing rights. CHERI capabilities provide strong compatibility with C-language pointers and Memory Management Unit (MMU)-based system-software designs, by virtue of its hybrid capability model.

Architecturally, a capability can be viewed as an address equal to the sum of the capability base and capability offset, as well as associated metadata. Dereferencing a capability is done relative to that address. The size of an in-memory capability may be smaller than the sum of its architectural fields (such as base, offset, and permissions) if a compressed capability mechanism, such as CHERI Concentrate, is used.

In the ISA, capabilities may be used explicitly via capability-based instructions, an application of the principle of intentional use, but also implicitly using legacy load and store instructions via the default data capability (**DDC**), and instruction fetch via the program-counter capability (**PCC**). A capability is either sealed or unsealed, controlling whether it has software-defined or instruction-set-defined behavior, and whether or not its fields are immutable.

Capabilities may be held in a capability register in a dedicated capability register file, a merged register file, or a suitably aligned tagged memory.

capability base The lower of the two capability bounds, from which the address of a capability can be calculated by using the capability offset.

capability bounds Upper and lower bounds, associated with each capability, describing a range of addresss that may be dereferenced via the capability. Architecturally, bounds are with respect to the capability base, which provides the lower bound, and capability length, which provides the upper bound when added to the base. The bounds may be empty, connoting no right to dereference at any address. The address of a capability may float outside of the dereferenceable bounds; with a compressed capability, it may not be possible to represent all possible out-of-bounds addresses. Bounds may be manipulated subject to capability monotonicity using capability-based instructions.

capability length The distance between the lower and upper capability bounds.

capability monotonicity Capability monotonicity is a property of the instruction set that any requested manipulation of a capability, whether in a capability register or in memory, either leads to strictly non-increasing rights, clearing of the capability tag, or a hardware exception. Controlled violation of monotonicity can be achieved via the exception delivery mechanism, which grants rights to additional capability register, and also by the CCall instruction, which may deliver an exception or unseal (and jump to) suitably checked sealed capabilities. .

capability object type In addition to fat-pointer metadata such as capability bounds and capability permissions, a sealed capability also contains an integer object type. The object type is set during a sealing operation to the address of the sealing capability. Object types can be used to link a sealed code capability and a sealed data capability when used with CCall to implement a software object model.

capability offset The distance between capability base and the address accessed when the capability is used as a pointer.

capability permissions A bitmask, associated with each capability, describing a set of ISA- or software-defined operations that may be performed via the capability. ISA-defined permissions include load data, store data, instruction fetch, load capability, and store capability. Permissions may be manipulated subject to capability monotonicity using capability-based instructions.

capability provenance The property that a valid-for-use capability can only be constructed by deriving it from another valid capability using a valid capability operation. Provenance is implemented using a capability tag combined with capability monotonicity, irrespective of whether a capability is held in a capability register or tagged memory.

capability register A capability register is an architectural register able to hold a capability including its capability tag, address, other fat-pointer metadata such as its capability bounds and capability permissions, and optional capability object type. Capability registers may be held in a capability register file, a merged register file, or be a special capability register accessed by dedicated instructions. A capability register might be a dedicated register intended primarily for capability-related operations (e.g., the capability registers described in CHERI-MIPS), or a general-purpose integer register that has been extended with capability metadata (such as the program-counter capability (**PCC**), or the capability registers described in CHERI-RISC-V when using a merged register file). Capability registers must be used to retain tag bits on capabilities transiting through memory, as only capability-based instructions enforce capability provenance and capability monotonicity.

capability register file The capability register file is a register file dedicated to holding general-purpose capabilities, in contrast to a merged register file, in which general-purpose integer registers are extended to be able to hold tagged capabilities. Some general-purpose capability registers have well-known conventions for their use in software, including the return capability and the stack capability.

capability tag A capability tag is a 1-bit integrity tag associated with each capability register, and also with each capability-sized, capability-aligned location in memory. If the tag is set, the capability is valid and can be dereferenced via the ISA. If the tag is clear, then the capability is invalid and cannot be dereferenced via the ISA. Tags are preserved by ISA operations that conform to capability provenance and capability monotonicity rules – for example, that any attempted modification of capability bounds leads to non-increasing bounds, and that in-memory capabilities are written only via capability stores, not data stores – otherwise, tags are cleared.

capability validity A capability is valid if its capability tag is set, which permits use of the capability subject to its capability bounds, capability permissions, and so on. Attempts to dereference a capability without a tag set will lead to a hardware exception.

capability-based instructions These instructions accept capabilities as operands, allowing capabilities to be loaded from and stored memory, manipulated subject to capability provenance and capability monotonicity rules, and used for a variety of operations such as loading and storing data and capabilities, as branch targets, and to retrieve and manipulate capability fields – subject to capability permissions.

CCall The `CCall` instruction is a source of controlled non-monotonicity in the CHERI-MIPS and CHERI-RISC-V ISAs. It has two modes of operation determined by an opcode selector field: a trapping mode, similar to a system call, that allows a privileged software exception handler to perform a domain transition; and a jump-like mode in which sealed operands are unsealed to provide access to additional rights to allow userspace code to perform operations in a different domain.

The trapping mode, similar to a system call, is intended to support invoking objects expressed as a pair of sealed capabilities, representing a code capability and a data capability. The exception code generated depends on whether or not the two operand capabilities have valid capability tags, suitable capability permissions, are both sealed, have matching capability object types, and other requirements associated with joint invocation. The software exception handler is expected to implement software-defined aspects of the object model, including any necessary unsealing of the operand capabilities, storing of any return information (e.g., via a trusted stack), and handle any exceptions reporting failures of ISA-implemented checks. To facilitate optimized software implementations, a separate `CCall`/`CReturn` exception vector is used.

The jump-like mode can directly enter any userspace domain described by a pair of sealed capabilities with the *Permit_CCall* permission set. In particular, it can safely enter userspace domain-transition code described by the sealed code capability while also unsealing the sealed data capability. As with the trapping mode, the sealed operand capability registers are checked for suitable properties and correspondence, and the userspace domain-transition routine can store any return information, perform further error checking, and so on.

CHERI Concentrate CHERI Concentrate is a specific compressed capability format that represents a 64-bit address with full precision, and capability bounds relative to that address with reduced precision. Bounds have a floating-point representation, requiring that as the size of a bounded object increases, greater alignment of its capability base and capability length are required. CHERI Concentrate is the successor compression format to CHERI-128.

CHERI-128 CHERI-128 is a specific compressed capability format that represents a 64-bit address with full precision, and capability bounds relative to that address with reduced precision. Bounds have a floating-point representation, requiring that as the size of a bounded object increases, greater alignment of its capability base and capability length are required. CHERI-128 has been replaced with CHERI Concentrate.

CHERI-MIPS An application of the CHERI protection model to the 64-bit MIPS ISA.

CHERI-RISC-V An application of the CHERI protection model to the RISC-V ISA.

CHERI-x86-64 An application of the CHERI protection model to the x86-64 ISA.

code capability A capability whose capability permissions have been configured to permit instruction fetch (i.e., execute) rights; typically, write permission will not be granted via an executable capability, in contrast to a data capability. Code capabilities are used to implement control-flow robustness by constraining the available branch and jump targets.

compressed capability A capability whose capability bounds are compressed with respect to its address, allowing its in-memory footprint to be reduced – e.g., to 128 bits, rather than the architectural 256 bits visible to the instruction set when a capability is loaded into a register file. Certain architecturally valid out-of-bounds addresses may not be representable with capability compression; operations leading to unrepresentable capabilities will clear the capability tag or throw an exception in order to ensure continuing capability monotonicity. CHERI-128 and CHERI Concentrate are specific compressed capability models that select particular points in the tradeoff space around in-memory capability size, bounds alignment requirements, and representability.

control-flow robustness The use of code capabilities to constrain the set of available branch and jump targets for executing code, such that the potential for attacker manipulation of the program-counter capability (**PCC**) to simulate injection of arbitrary code is severely constrained; a form of vulnerability mitigation implemented via the principle of least privilege.

CReturn A trapping instruction, similar to a system call, intended to support returning from an object invoked via the trapping mode of the CCall instruction. Unlike `CCall`, in-ISA checks are not performed, leaving any required functionality to software – for example, popping an entry off of a trusted stack. To facilitate optimized software implementations, a separate `CCall`/`CReturn` exception vector is used.

data capability A capability whose capability permissions have been configured to permit data load and store, but not instruction fetch (i.e., execute) rights; in contrast to a code capability.

default data capability (DDC) A special capability register constraining legacy non-capability-based instructions that load and store data without awareness of the capability model. Any attempts to load and store will be relocated relative to the default data capability's capability base and capability offset, and controlled by its capability bounds and capability permissions. Use of the default data capability violates the principle of intentional use, but permits compatibility with legacy software. A suitably configured default data capability will prevent the use of non-capability-based load and store instructions.

dereference Dereferencing a address means that it is the target address for a load, store, or instruction fetch. A capability may be dereferenced only subject to it being valid – i.e., that its capability tag is present – and is also subject to appropriate checks of its capability bounds, capability permissions, and so on. Dereference may occur as a result of explicit use of a capability via capability-based instructions, or implicitly as a result of the program-counter capability (**PCC**) or default data capability (**DDC**).

exception program-counter capability (EPCC) A special capability register into which the running program-counter capability (**PCC**) will be moved into on an exception, and whose value will be moved back into the program-counter capability on exception return.

fat pointer A pointer (address) that has been extended with additional metadata such as capability bounds and capability permissions. In conventional fat-pointer designs, fat pointers to not have a notion of sealing (i.e., as in sealed capabilities and unsealed capabilities), nor rules implementing capability provenance and capability monotonicity.

fine-grained memory protection The granular description of available code and data in which capability bounds and capability permissions are made as small as possible, in order to limit the potential effects of software bugs and vulnerabilities. This approach applies both to code capabilities and data capabilities, offering effective vulnerability mitigation via techniques such as control-flow robustness, as well as supporting higher-level mitigation techniques such as software compartmentalization. Fine-grained memory protection will typically be driven by the goal of implementing the principle of least privilege.

hybrid capability model A capability model in which not all interfaces to use or manipulate capabilities conform to the principle of intentional use, such that legacy software is able to execute around, or within, capability-constrained environments, as well as other features required to improve compatibility with conventional software designs permitting easier incremental adoption of a capability-system model. In CHERI, composition of the capability-system model with the conventional Memory Management Unit (MMU), the support for legacy instructions via the program-counter capability (**PCC**) and default data capability (**DDC**), and strong compatibility with the C-language pointer model, all constitute hybrid aspects of its design, in comparison to a more pure capability-system model that might elide those behaviors at a cost to compatibility and adoptability.

invoked data capability (IDC) A capability register reserved by convention to hold the unsealed data capability on the callee side of CCall, and to be saved from the caller context on CCall, to be restored by CReturn. Typically, for the caller side, this will point at a frame on the caller stack sufficient to safely restore any caller state. On the callee side, the invoked data capability will be a data capability describing the objects internal state.

kernel code capability (KCC) A special capability register reserved to hold a privileged code capability for use by the kernel during exception handling. This value will be installed in the program-counter capability (**PCC**) on exception entry, with the previous value of the program-counter capability stored in the exception program-counter capability (**EPCC**).

kernel data capability (KDC) A special capability register reserved to hold a privileged data capability for use by the kernel during exception handling. Typically, this will refer either to the data segment for a microkernel intended to field exceptions, or for the full kernel. Kernels compiled to primarily use legacy instructions might install this in the default data capability (**DDC**) for the duration of kernel execution. Use of this register is controlled by capability permissions on the currently executing program-counter capability (**PCC**).

kernel reserved capabilities These capabilities, modeled on the MIPS kernel reserved registers, are set aside for use by a CHERI-MIPS operating-system kernel in exception handling – in particular, in allowing userspace registers to be saved so that the kernel context can be installed. As with the MIPS registers, the userspace ABI is not able to use capability registers set aside for kernel use; unlike the MIPS registers, the kernel reserved capabilities are available for use in the ISA only with a suitably authorized program-counter capability (**PCC**) installed. Due to a different exception-handling model in CHERI-RISC-V, that ISA does not have kernel reserved capabilities.

legacy instructions Legacy instructions are those that accept integer addresses, rather than capabilities, as their operands, requiring use of the default data capability (**DDC**) for loads and stores, or that explicitly set the program counter to a address, rather than doing setting the program-counter capability (**PCC**). These instructions allow legacy binaries (those compiled without CHERI awareness) to execute, but only without the benefits of fine-grained memory protection, granular control-flow robustness, or more efficient software compartmentalization. While still constrained, these instructions do not conform to the principle of intentional use.

merged register file A single general-purpose register file able to hold both integer and tagged capability values. In CHERI-MIPS, a dedicated capability register file is used, separate from the general-purpose integer register file. In CHERI-RISC-V, a merged register file is supported, reducing the amount of control logic required for a separate register file.

out of bounds When a capability's capability offset falls outside of its capability bounds, it is out of bounds, and cannot be dereferenced. Even if a capability's offset is in bounds, the width of a data access may cause a load, store, or instruction fetch to fall out of bounds, or the further offset introduced via a register index or immediate operand to an instruction. With 256-bit capabilities, all out-of-bounds pointers are representable capabilities. With compressed capabilities, if an instruction shifts the offset too far out of bounds, this may result in an unrepresentable capability, leading to the capability tag being cleared, or an exception being thrown.

physical address An address that is passed directly to the memory hierarchy without virtual-address translation. In CHERI-MIPS, capabilities contain only virtual addresses. In CHERI-RISC-V, capabilities addresses may be interpreted as physical addresses in Machine Mode.

pointer A pointer is a language-level reference to a memory object. In conventional ISAs, a pointer is typically represented as an address. In CHERI, pointers can be represented either as an address indirected via the default data capability (**DDC**) or program-counter capability (**PCC**), or as a capability. In the latter cases, its integrity and capability provenance are protected by the capability tag, and its use is limited by capability bounds and capability permissions. Capability-based instructions preserve the tag as required across both capability registers and tagged memory, and also enforce capability monotonicity: legitimate operations on the pointer cannot broaden the set of rights described by the capability.

principle of intentional use A design principle in capability systems in which rights are always explicitly, rather than implicitly exercised. This arises in the CHERI instruction set through explicit capability operands to capability-based instructions, which contributes to the effectiveness of fine-grained memory protection and control-flow robustness. When applied, the principle limits not just the rights available in the presence of a software vulnerability, but the extent to which software can be manipulated into using rights in an unintended (and exploitable) manner.

principle of least privilege A principle of software design in which the set of rights available to running code is minimized to only those required for it to function, often with the aim of vulnerability mitigation. In CHERI, this concept applies via fine-grained memory protection for both data and code, and also higher-level software compartmentalization.

program-counter capability (PCC) A special capability register that extends the existing program counter to include capability metadata such as a capability tag, capability bounds, and capability permissions. The program-counter capability ensures that instruction fetch occurs only subject to capability protections. When an exception fires, the value of the program-counter capability will be moved to the exception program-counter capability (**EPCC**), and the value of the kernel data capability (**KDC**) moved into the program-counter capability. On exception return, the value of the exception program-counter capability will be moved into the program-counter capability.

representable capability A compressed capability whose capability offset is representable with respect to its capability bounds; this does not imply that the offset is "within bounds", but does require that it be within some broader window around the bounds.

return capability A capability designated as the destination for the return address when using a capability jump-and-link instruction. A degree of control-flow robustness is provided due to capability bounds, capability permissions, and the capability tag on the resulting capability, which limits sites that may be jumped back to using the return capability.

sealed capability A sealed capability is one whose capability object type is set (i.e., is not the reserved value $2^{64} - 1$). A sealed capability's address, capability bounds, capability permissions, and other fields are immutable – i.e., cannot be modified using capability-based instructions. Sealed capabilities also have a capability object type derived from their sealing capabilities's address. CHERI's sealing feature allows capabilities to be used to describe software-defined objects, permitting implementation of encapsulation. A sealed capability cannot be directly dereferenced using the instruction set. Unsealing can be performed using a jump-based CCall instruction, or using the `CUnseal` instruction combined with a suitable sealing capability. Sealed capabilities provide the necessary architectural encapsulation support to implement fine-grained compartmentalization via both object-oriented and non-object-centric models.

sealing capability A sealing capability is one with the *Permit_Seal* permission, allowing it to be used to create sealed capabilities using a capability object type set to the sealing capability's address, and subject to its bounds.

software compartmentalization The configuration of code capabilities and data capabilities available via the capability register file or merged register file, accessible special capability registers, and tagged memory such that software components can be isolated from one another, enabling vulnerability mitigation via the application of the principle of least privilege at the application layer. One approach to implementing software compartmentalization on CHERI is to use sealed capabilities to represent security domains, which can be safely invoked using a suitably crafted CCall exception handler, providing mutual distrust. Another uses the jump-based CCall semantics to jump into sealed code and data capabilities describing a trusted intermediary and destination protection domain.

special capability register Special capability registers have special architectural meanings, and include the program-counter capability (**PCC**), the default data capability (**DDC**), the exception program-counter capability (**EPCC**), the kernel code capability (**KCC**), and the kernel data capability (**KDC**). Not all registers are accessible at all times; for example, some may be available only in certain rings, or when **PCC** has the Access_System_Registers permission set.

stack capability A capability referring to the current stack, whose capability bounds are suitably configured to allow access only to the remaining stack available to allocate at a given point in execution.

tagged memory Tagged memory associates a 1-bit capability tag with each capability-aligned, capability-sized word in memory. Capability-based instructions that load and store capabilities maintain the tag as the capability transits between memory and the capability register file, tracking capability provenance. When data stores (i.e., stores of non-capabilities), the tag on the memory location will be atomically cleared, ensuring the integrity of in-memory capabilities.

Trusted Computing Base (TCB) The subset of hardware and software that is critical to the security of a system; in secure system designs, there is often a goal to minimize the size of the TCB in order to minimize the opportunity for exploitable software vulnerabilities.

trusted stack Some software-defined object-capability models offer strong call-return semantics – i.e., that if a return is issued by an invoked object, or an uncaught exception is generated, then the appropriate caller will be returned to – exactly once. This can be implemented via a trusted stack, maintained by the software Trusted Computing Base (TCB) via CCall and CReturn exception handlers. A trusted stack for an object-oriented model will likely maintain at least the caller's program-counter capability (**PCC**) and invoked data capability (**IDC**) to be restored on return.

unrepresentable capability A compressed capability whose capability offset is sufficiently outside of its capability bounds that the combined pointer value and bounds cannot be represented in the compressed format; constructing an unrepresentable capability will lead to the tag being cleared (and information loss) or an exception, rather than a violation of capability provenance or capability monotonicity.

unsealed capability An unsealed capability is one whose capability object type is unset (i.e., is the reserved value $2^{64}-1$). Its remaining capability fields are mutable, subject to capability provenance and capability monotonicity rules. These capabilities have hardware-defined behaviors – i.e., subject to capability bounds, capability permissions, and so on, can be dereferenced.

virtual address An integer address translated by the Memory Management Unit (MMU) into a physical address for the purposes of load, store, and instruction fetch. Capabilities embed an address, represented in the instruction set as the sum of the capability base and capability offset, as well as capability bounds relative to the address. The integer addresses passed to legacy load and store instructions that would previously have been interpreted as virtual addresses are, with CHERI, transformed (and checked) using the default data capability (**DDC**). Similarly, the integer addresses passed to legacy branch and jump instructions are transformed (and checked) using the program-counter capability (**PCC**). This in effect introduces a further relocation of legacy addresses prior to virtual address translation.

vulnerability mitigation A set of techniques limiting the effectiveness of the attacker to exploit a software vulnerability, typically achieved through use of the principle of least privilege to constrain injection of arbitrary code, control of the program-counter capability (**PCC**) via control-flow robustness using code capabilities, minimization of data rights granted via available data capabilities, and higher-level software compartmentalization.

Bibliography

[1] *ARM® A64 Instruction Set Architecture ARMv8, for ARMv8-A architecture profile*.

[2] Introduction to SPARC M7 and Application Data Integrity (ADI). https://swisdev.oracle.com/_files/What-Is-ADI.html.

[3] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. Technical report, Computer Science Department, Carnegie Mellon University, August 1986.

[4] W. B. Ackerman and W. W. Plummer. An implementation of a multiprocessing computer system. In *SOSP '67: Proceedings of the First ACM Symposium on Operating System Principles*, pages 5.1–5.10, New York, NY, USA, 1967. ACM.

[5] J. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, October 1972. (Two volumes).

[6] G. R. Andrews. Partitions and principles for secure operating systems. Technical report, Cornell University, Ithaca, NY, USA, 1975.

[7] Apple Inc. Mac OS X Snow Leopard. http://www.apple.com/macosx/, 2010.

[8] 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. Proc. ACM Program. Lang. 3, POPL, Article 71.

[9] R. S. Barton, H. Berce, G. A. Collins, B. A. Creech, D. M. Dahm, B. A. Dent, V. J. Ford, B. A. Galler, J. E. S. Hale, E. A. Hauck, J. T. Hootman, P. D. King, N. L. Kreuder, W. R. Lonergan, D. MacDonald, F. B. MacKenzie, C. Oliphint, R. Pearson, R. F. Rosin, L. D. Turner, and R. Waychoff. Discussion: The burroughs b 5000 in retrospect. *Annals of the History of Computing*, 9(1):37–92, Jan 1987.

[10] D. Bell and L. L. Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, The Mitre Corporation, Bedford, Massachusetts, March 1976.

[11] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–283, New York, NY, USA, 1995. ACM.

[12] R. Bisbey II and D. Hollingworth. Protection Analysis: Project final report. Technical report, USC Information Sciences Institute (ISI), Marina Del Rey, California, 1978.

[13] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 309–322. USENIX Association, 2008.

[14] Bluespec, Inc., Waltham, MA. *Bluespec SystemVerilog Version 3.8 Reference Guide*, November 2004.

[15] M. Branstad and J. Landauer. Assurance for the Trusted Mach operating system. In *Proceedings of the Fourth Annual Conference on Computer Assurance COMPASS '89*, pages 9–13. IEEE, June 1989.

[16] B. Campbell and I. Stark. Randomised testing of a microprocessor model using SMT-solver state generation. *Sci. Comput. Program.*, 118:60–76, 2016.

[17] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *ATEC '04: Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2004. USENIX Association.

[18] N. P. Carter, S. W. Keckler, and W. J. Dally. Hardware support for fast capability-based addressing. *SIGPLAN Not.*, 29(11):319–327, Nov. 1994.

[19] S. Chiricescu, A. DeHon, D. Demange, S. Iyer, A. Kliger, G. Morrisett, B. C. Pierce, H. Reubenstein, J. M. Smith, G. T. Sullivan, et al. Safe: A clean-slate architecture for secure systems. In *IEEE International Conference on Technologies for Homeland Security (HST)*, pages 570–576, 2013.

[20] D. Chisnall, B. Davis, K. Gudka, D. Brazdil, A. Joannou, J. Woodruff, A. T. Markettos, J. E. Maste, R. Norton, S. Son, M. Roe, S. W. Moore, P. G. Neumann, B. Laurie, , and R. N. M. Watson. CHERI-JNI: Sinking the Java security model into the C. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2017)*, April 2017.

[21] D. Chisnall, C. Rothwell, B. Davis, R. Watson, J. Woodruff, S. Moore, P. G. Neumann, and M. Roe. Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XX, New York, NY, USA, 2014. ACM.

[22] E. Cohen and D. Jefferson. Protection of the Hydra operating system. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, pages 141–160, 1975.

[23] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley. An experimental time-sharing system. In *AIEE-IRE '62 (Spring): Proceedings of the May 1–3, 1962, Spring Joint Computer Conference*, pages 335–344, New York, NY, USA, 1962. ACM.

[24] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the Multics system. In *AFIPS '65 (Fall, part I): Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, part I*, pages 185–196, New York, NY, USA, 1965. ACM.

[25] B. Corporation. B 6700 information processing system reference manual. 1972.

[26] B. A. Creech. Architecture of the b-6500. In J. T. Tou, editor, *Software Engineering: Proceedings of the Third Symposium on Computer and Information Sciences*, volume 1, pages 29–43, December 1970.

[27] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. In *AFIPS Conference Proceedings, Fall Joint Computer Conference*, pages 213–229. Spartan Books, November 1965.

[28] B. Davis, R. N. M. Watson, A. Richardson, P. Neumann, S. Moore, J. Baldwin, D. Chisnall, J. Clarke, K. Gudka, A. Joannou, B. Laurie, A. T. Markettos, E. Maste, E. T. Napierala, R. 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 24nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019)*, April 2019.

[29] P. J. Denning. Fault tolerant operating systems. *ACM Computing Surveys*, 8(4):359–389, 1976.

[30] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[31] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. *SIGPLAN Not.*, 43(3):103–114, Mar. 2008.

[32] M. S. Doerrie. *Confidence in Confinement: An Axiom-free, Mechanized Verification of Confinement in Capability-based Systems*. PhD thesis, 2015.

[33] W. E. Boebert. On the inability of an unmodified capability machine to enforce the *-property. 2019.

[34] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. *SIGOPS Oper. Syst. Rev.*, 39:17–30, October 2005.

[35] R. S. Fabry. The case for capability based computers (extended abstract). In *SOSP '73: Proceedings of the Fourth ACM Symposium on Operating System Principles*, page 120, New York, NY, USA, 1973. ACM.

[36] R. J. Feiertag and P. G. Neumann. The foundations of a Provably Secure Operating System (PSOS). In *Proceedings of the National Computer Conference*, pages 329–334. AFIPS Press, 1979. http://www.csl.sri.com/neumann/psos.pdf.

[37] A. Fox. *Improved Tool Support for Machine-Code Decompilation in HOL4*, pages 187–202. Springer International Publishing, Cham, 2015.

[38] P. George, W. Susan, W. Jean, and T. Glen. MIPSpro^{TM} N32 ABI Handbook. 2002.

[39] P. G.Neumann, R. Boyer, R. Feiertag, K. Levitt, and L. Robinson. A Provably Secure Operating System: The system, its applications, and proofs. Technical Report CSL-116, Second edition, Computer Science Laboratory, SRI International, Menlo Park, California, May 1980.

[40] L. Gong. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, Reading, Massachusetts, 1999.

[41] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[42] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[43] S. Govindavajhala and A. Appel. Using memory errors to attack a virtual machine. In *Proceedings of the 2003 Symposium on Security and Privacy*, pages 154–165, Oakland, California, May 2003. IEEE Computer Society.

[44] R. Graham. Protection in an information processing utility. *Communications of the ACM*, 11(5), May 1968.

[45] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, S. J. Murdoch, P. G. Neumann, and A. Richardson. Clean application compartmentalization with SOAAP (extended version). Technical Report UCAM-CL-TR-873, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, Dec. 2015.

[46] K. Gudka, R. N. M. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015)*, October 2015.

[47] R. H. Gumpertz. *Error Detection with Memory Tags*. PhD thesis, December 1981.

[48] N. Hardy. KeyKOS architecture. *SIGOPS Operating Systems Review*, 19(4):8–25, 1985. Also available at http://cap-lore.com/CapTheory/upenn/OSRpaper.html.

[49] N. Hardy. The confused deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4), October 1988.

[50] J. Heinrich. *MIPS R4000 User's Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[51] International Standards Organization. *Common Criteria for Information Technology Security Evaluation – Part 3: Security assurance components*, September 2012. Version 3.1, revision 4.

[52] International Standards Organization. *Common Criteria for Information Technology Security Evaluation – Part 1: Introduction and General Model*, April 2017. Version 3.1, revision 5.

[53] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *ATEC '02: Proceedings of the USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[54] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. M. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. Son, and A. T. Markettos. Efficient tagged memory. In *Proceedings of the 2017 IEEE 35th International Conference on Computer Design (ICCD)*, November 2017.

[55] A. J. P. Joannou. High-performance memory safety: optimizing the CHERI capability machine. Technical Report UCAM-CL-TR-936, University of Cambridge, Computer Laboratory, May 2019.

[56] A. Jones and W. Wulf. Towards the design of secure systems. In *Protection in Operating Systems, Proceedings of the International Workshop on Protection in Operating Systems*, pages 121–135, Rocquencourt, Le Chesnay, France, 13–14 August 1974. Institut de Recherche d'Informatique.

[57] P. Karger. Limiting the damage potential of discretionary Trojan horses. In *Proceedings of the 1987 Symposium on Security and Privacy*, pages 32–37, Oakland, California, April 1987. IEEE Computer Society.

[58] P. Karger and R. Schell. Multics security evaluation: Vulnerability analysis. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC), Classic Papers section*, Las Vegas, Nevada, December 2002. Originally available as U.S. Air Force report ESD-TR-74-193, Vol. II, Hanscomb Air Force Base, Massachusetts.

[59] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the Tenth ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[60] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an operating-system kernel. *Communications of the ACM*, 53:107–115, June 2009.

[61] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[62] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *20th ACM Conference on Computer and Communications Security*, November 2013.

[63] B. Lampson. Redundancy and robustness in memory protection. In *Information Processing 74 (Proceedings of the IFIP Congress 1974)*, volume Hardware II, pages 128–132. North-Holland, Amsterdam, 1974.

[64] B. W. Lampson. Dynamic protection structures. In *AFIPS '69 (Fall): Proceedings of the November 18-20, 1969, Fall Joint Computer Conference*, pages 27–38, New York, NY, USA, 1969. ACM.

[65] B. W. Lampson. Protection. *SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[66] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the international symposium on code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[67] N. G. Leveson and W. Young. An integrated approach to safety and security based on system theory. *Communications of the ACM*, 57(2):31–35, February 2014. `http://www.csl.sri.com/neumann/insiderisks.html`.

[68] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[69] J. Liedtke. On microkernel construction. In *SOSP'95: Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, Dec. 1995.

[70] S. B. Lipner, W. A. Wulf, R. R. Schell, G. J. Popek, P. G. Neumann, C. Weissman, and T. A. Linden. Security kernels. In *AFIPS '74: Proceedings of the May 6-10, 1974, National Computer Conference and Exposition*, pages 973–980, New York, NY, USA, 1974. ACM.

[71] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[72] A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh. Cryptographically enforced control flow integrity. *arXiv preprint arXiv:1408.1451*, 2014.

[73] A. J. W. Mayer. The architecture of the burroughs b5000: 20 years later and still ahead of the times? *SIGARCH Comput. Archit. News*, 10(4):3–10, June 1982.

[74] A. Mazzinghi, R. Sohan, and R. N. M. Watson. Pointer Provenance in a Capability Architecture. In *Proceedings of the 10th USENIX Workshop on the Theory and Practice of Provenance (TaPP'18)*, 2018.

[75] S. McCanne and V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *USENIX'93: Proceedings of the USENIX Winter 1993 Conference*, Berkeley, CA, USA, 1993. USENIX Association.

[76] E. McCauley and P. Drongowski. KSOS: The design of a secure operating system. In *National Computer Conference*, pages 345–353. AFIPS Conference Proceedings, 1979. Vol. 48.

[77] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson. *The Design and Implementation of the FreeBSD Operating System, Second Edition*. Pearson Education, 2014.

[78] K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell. Exploring C semantics and pointer provenance. In ***POPL 2019****: Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, Jan. 2019. Proc. ACM Program. Lang. 3, POPL, Article 67.

[79] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. Watson, and P. Sewell. Into the depths of C: elaborating the de facto standards. In *Proceedings of PLDI 2016*, June 2016.

[80] A. Mettler and D. Wagner. Class properties for security review in an object-capability subset of Java. In *PLAS '10: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 1–7, New York, NY, USA, 2010. ACM.

[81] M. S. Miller. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006.

[82] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript, May 2008. http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf.

[83] M. S. Miller and J. S. Shapiro. Paradigm Regained: Abstraction Mechanisms for Access Control. pages 224–242, 2003.

[84] M. S. Miller, K.-P. Yee, and J. Shapiro. Capability Myths Demolished. page 15.

[85] J. H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.

[86] A. C. Myers and B. Liskov. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31:129–142, October 1997.

[87] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[88] NCSC. *Department of Defense Trusted Computer System Evaluation Criteria (TCSEC)*. National Computer Security Center, December 1985. DOD-5200.28-STD, Orange Book.

[89] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI '96: Proceedings of the Second USENIX symposium on Operating Systems Design and Implementation*, pages 229–243, New York, NY, USA, 1996. ACM.

[90] G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 128–139, New York, NY, USA, 2002. ACM.

[91] P. G. Neumann. Holistic systems. *ACM Software Engineering Notes*, 31(6):4–5, November 2006.

[92] P. G. Neumann. Fundamental Trustworthiness Principles in CHERI. In H. Shrobe, D. L. Shrier, and A. Pentland, editors, *New Solutions for Cybersecurity*, chapter 6. MIT Press/Connection Science, 2018.

[93] P. G. Neumann, R. Boyer, R. Feiertag, K. Levitt, and L. Robinson. A Provably Secure Operating System: The system, its applications, and proofs. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, May 1980. 2nd edition, Report CSL-116.

[94] P. G. Neumann and R. J. Feiertag. PSOS revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), Classic Papers section*, pages 208–216, Las Vegas, Nevada, December 2003. IEEE Computer Society. http://www.acsac.org/ and http://www.csl.sri.com/neumann/psos03.pdf.

[95] P. G. Neumann and R. N. M. Watson. Capabilities revisited: A holistic approach to bottom-to-top assurance of trustworthy systems. In *Fourth Layered Assurance Workshop*, Austin, Texas, December 2010. U.S. Air Force Cryptographic Modernization Office and AFRL. http://www.csl.sri.com/neumann/law10.pdf.

[96] R. M. Norton. Hardware support for compartmentalisation. Technical Report UCAM-CL-TR-887, University of Cambridge, Computer Laboratory, May 2016.

[97] E. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, Massachusetts, 1972.

[98] E. I. Organick. Computer system organization. May 1973.

[99] D. A. Patterson and C. H. Sequin. RISC I: A Reduced Instruction Set VLSI Computer. In *ISCA '81: Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 443–457, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[100] G. J. Popek, M. Kampe, M. Urban, A. Stoughton, E. J. Walton, and C. S. Kline. Ucla secure unix. In *International Workshop on Managing Requirements Knowledge (AFIPS)*, page 355, 6 1979.

[101] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, 2003.

[102] R. Rashid and G. Robertson. Accent: A communications oriented network operating system kernel. In *Proceedings of the Eighth ACM Symposium on Operating System Principles*, pages 64–75, Asilomar, California, December 1981. (ACM Operating Systems Review, Vol. 15, No. 5).

[103] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems*, pages 219–232, New York, NY, USA, 2009. ACM.

[104] REMS Project. Sail CHERI-MIPS implementation. https://github.com/CTSRD-CHERI/sail-cheri-mips/, 2018.

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

[106] D. Rémy and J. Vouillon. Objective ML: a simple object-oriented extension of ML. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 40–53, New York, NY, USA, 1997. ACM.

[107] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974.

[108] Ruby Users Group. Ruby Programming Language. http://www.ruby-lang.org/, October 2010.

[109] J. Rushby. The design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating System Principles*, pages 12–21, Asilomar, California, December 1981. (ACM Operating Systems Review, 15(5)).

[110] J. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[111] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[112] W. Schiller. The design and specification of a security kernel for the PDP-11/45. Technical Report MTR-2934, Mitre Corporation, Bedford, Massachusetts, March 1975.

[113] M. D. Schroeder. Engineering a security kernel for Multics. In *SOSP '75: Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, pages 25–32, New York, NY, USA, 1975. ACM.

[114] E. J. Sebes. Overview of the architecture of Distributed Trusted Mach. In *Proceedings of the USENIX Mach Symposium*, pages 20–22. USENIX Association, November 1991.

[115] J. Shapiro, M. S. Doerrie, E. Northup, and M. Miller. Towards a verified, general-purpose operating system kernel. In *Proceedings of the NICTA Invitational Workshop on Operating System Verification*, pages 1–19, 2004.

[116] J. Shapiro, J. Smith, and D. Farber. EROS: a fast capability system. In *SOSP '99: Proceedings of the seventeenth ACM Symposium on Operating Systems Principles*, Dec 1999.

[117] J. S. Shapiro and J. W. Adams. Coyotos microkernel specification.

[118] L. Skorstengaard, D. Devriese, and L. Birkedal. Stktokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proc. ACM Program. Lang.*, 3(POPL), Jan. 2019.

[119] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proceedings of the 8th USENIX Security Symposium*, pages 123–139, Washington, D.C., USA, Aug. 1999. USENIX Association.

[120] J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. *USENIX Security*, 15-17 August 2018. http://foreshadowattack.eu/foreshadowattack.pdf.

[121] R. Wahbe, S. Lucco, T. E. Anderson, and S. u. L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, New York, NY, USA, 1993. ACM.

[122] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.

[123] S. T. Walker. The advent of trusted computer operating systems. In *AFIPS '80: Proceedings of the May 19-22, 1980, national computer conference*, pages 655–665, New York, NY, USA, 1980. ACM.

[124] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the 18th USENIX Security Symposium*, pages 417–432, Berkeley, CA, USA, 2009. USENIX Association.

[125] A. Waterman. Design of the RISC-V Instruction Set Architecture, 2016.

[126] A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2.* May 2017.

[127] A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10.* May 2017.

[128] R. N. M. Watson. New Approaches to Operating System Security Extensibility. Technical report, Ph.D. Thesis, University of Cambridge, Cambridge, UK, October 2010.

[129] R. N. M. Watson. New approaches to operating system security extensibility. Technical Report UCAM-CL-TR-818, University of Cambridge, Computer Laboratory, Apr. 2012.

[130] R. N. M. Watson. A decade of OS access-control extensibility. *Commun. ACM*, 56(2), Feb. 2013.

[131] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for Unix. In *Proceedings of the 19th USENIX Security Symposium*. USENIX, August 2010.

[132] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, P. G. Neumann, and J. Woodruff. Bluespec Extensible RISC Implementation (BERI): Software Reference. Technical Report UCAM-CL-TR-853, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, June 2014.

[133] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, P. G. Neumann, and J. Woodruff. Capability Hardware Enhanced RISC Instructions (CHERI): User's guide. Technical Report UCAM-CL-TR-851, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, June 2014.

[134] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, P. G. Neumann, and J. Woodruff. Bluespec Extensible RISC Implementation: BERI Software reference. Technical Report UCAM-CL-TR-869, University of Cambridge, Computer Laboratory, Apr. 2015.

[135] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, P. G. Neumann, and J. Woodruff. Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide. Technical Report UCAM-CL-TR-877, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, Nov. 2015.

[136] R. N. M. Watson, P. G. Neumann, and S. W. Moore. Balancing Disruption and Deployability in the CHERI Instruction-Set Architecture (ISA). In H. Shrobe, D. L. Shrier, and A. Pentland, editors, *New Solutions for Cybersecurity*, chapter 5. MIT Press/Connection Science, 2018.

[137] R. N. M. Watson, P. G. Neumann, J. Woodruff, J. Anderson, D. Chisnall, B. Davis, B. Laurie, S. W. Moore, S. J. Murdoch, and M. Roe. Capability Hardware Enhanced RISC Instructions (CHERI): Instruction-Set Architecture. Technical Report UCAM-CL-TR-850, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, June 2014.

[138] R. N. M. Watson, P. G. Neumann, J. Woodruff, J. Anderson, D. Chisnall, B. Davis, B. Laurie, S. W. Moore, S. J. Murdoch, and M. Roe. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture. Technical Report UCAM-CL-TR-864, University of Cambridge, Computer Laboratory, Dec. 2014.

[139] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, D. Chisnall, B. Davis, N. W. Filardo, A. Joannou, B. Laurie, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Sewell, S. Son, and H. Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7). Technical Report UCAM-CL-TR-927, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, Oct. 2018.

[140] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, J. Baldwin, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S. W. Moore, S. J. Murdoch, R. Norton, S. Son, and H. Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6). Technical Report UCAM-CL-TR-907, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, Apr. 2017.

[141] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S. W. Moore, S. J. Murdoch, R. Norton, and S. Son. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture. Technical Report UCAM-CL-TR-876, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, Nov. 2015.

[142] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S. W. Moore, S. J. Murdoch, R. Norton, S. Son, and H. Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 5). Technical Report UCAM-CL-TR-891, University of Cambridge, Computer Laboratory, June 2016.

[143] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera. Fast Protection-Domain Crossing in the CHERI Capability-System Architecture. *IEEE Micro*, 36(5):38–49, Sept 2016.

[144] R. N. M. Watson, J. Woodruff, D. Chisnall, B. Davis, W. Koszek, A. T. Markettos, S. W. Moore, S. J. Murdoch, P. G. Neumann, R. Norton, and M. Roe. Bluespec Extensible RISC Implementation (BERI): Hardware Reference. Technical Report UCAM-

CL-TR-852, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, June 2014.

[145] R. N. M. Watson, J. Woodruff, D. Chisnall, B. Davis, W. Koszek, A. T. Markettos, S. W. Moore, S. J. Murdoch, P. G. Neumann, R. Norton, and M. Roe. Bluespec Extensible RISC Implementation: BERI Hardware reference. Technical Report UCAM-CL-TR-868, University of Cambridge, Computer Laboratory, Apr. 2015.

[146] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. s Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, May 2015.

[147] R. N. M. Watson, J. Woodruff, M. Roe, S. W. Moore, and P. G. Neumann. Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks. Technical Report UCAM-CL-TR-916, University of Cambridge, Computer Laboratory, Feb. 2018.

[148] R. N. M. Watson, P. G. N. J. Woodruff, J. Anderson, R. Anderson, N. Dave, B. Laurie, S. W. Moore, S. J. Murdoch, P. Paeps, M. Roe, and H. Saidi. CHERI: a research platform deconflating hardware virtualization and protection. In *Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012)*, March 2012.

[149] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. Foreshadow: Breaking the virtual memory abstraction with transient out-of-order execution. 14 August 2018. http://foreshadowattack.eu/foreshadow-NG.pdf.

[150] M. Wilkes and R. Needham. *The Cambridge CAP Computer and Its Operating System*. Elsevier North Holland, New York, 1979.

[151] A. Wilkinson et al. A penetration study of a Burroughs large system. *ACM Operating Systems Review*, 15(1):14–25, January 1981.

[152] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. Norton, T. Bauereiss, 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. *IEEE Transactions on Computers*, 2019.

[153] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA 2014)*, June 2014.

[154] J. D. Woodruff. CHERI: A RISC capability machine for practical memory safety. Technical Report UCAM-CL-TR-858, University of Cambridge, Computer Laboratory, July 2014.

[155] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, 1974.

[156] W. Wulf, R. Levin, and S. Harbison. *Hydra/C.mmp: An Experimental Computer System*. McGraw-Hill, New York, 1981.

[157] 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 *Proceedings of the 2018 IEEE 36th International Conference on Computer Design (ICCD)*, October 2018.

[158] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society.

[159] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society.

[160] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.