Technical Report

UCAM-CL-TR-987
ISSN 1476-2986

Number 987

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)

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

September 2023

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 contract FA8750-10-C-0237 ("CTSRD"), with additional support from FA8750-11-C-0249 ("MRC2"), HR0011-18-C-0016 ("ECATS"), FA8650-18-C-7809 ("CIFV"), HR0011-22-C-0110 ("ETC"), and HR0011-23-C-0031 ("MTSS") 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.

This work was supported in part by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 105694.

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 EPSRC CHaOS grant (EP/V000292/1), the EPSRC SIPP grant (EP/S030867/1), 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
DOI *https://doi.org/10.48456/tr-987*

The CHERI-RISC-V pseudocode is derived from the Sail CHERI-RISC-V model:

```
This CHERI Sail RISC-V architecture model here, comprising all files and
directories except for the snapshots of the Lem and Sail libraries in the
prover_snapshots directory (which include copies of their licenses), is subject
to the BSD two-clause licence below.

Copyright (c) 2017-2021 Alasdair Armstrong, Thomas Bauereiss, Brian Campbell,
  Jessica Clarke, Nathaniel Wesley Filardo (contributions prior to July 2020,
  thereafter Microsoft), Alexandre Joannou, Microsoft, Prashanth Mundkur,
  Robert Norton-Wright (contributions prior to March 2020, thereafter
  Microsoft), Alexander Richardson, Peter Rugg, Peter Sewell

All rights reserved.

This software was developed by SRI International and the University of
Cambridge Computer Laboratory (Department of Computer Science and
Technology) under DARPA/AFRL contract FA8650-18-C-7809 ("CIFV"), and
under DARPA contract HR0011-18-C-0016 ("ECATS") as part of the DARPA
SSITH research programme.

This software was developed within the Rigorous Engineering of
Mainstream Systems (REMS) project, partly funded by EPSRC grant
EP/K008528/1, at the Universities of Cambridge and Edinburgh.

This project has received funding from the European Research Council
(ERC) under the European Union’s Horizon 2020 research and innovation
programme (grant agreement 789108, ELVER).

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in
   the documentation and/or other materials provided with the
   distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ‘‘AS IS’’
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

The RISC-V pseudocode is derived from the Sail RISC-V model:

```
This Sail RISC-V architecture model, comprising all files and
directories except for the snapshots of the Lem and Sail libraries
in the prover_snapshots directory (which include copies of their
licences), is subject to the BSD two-clause licence below.

Copyright (c) 2017-2021 Prashanth Mundkur, Rishiyur S. Nikhil and
  Bluespec, Inc., Jon French, Brian Campbell, Robert Norton-Wright,
  Alasdair Armstrong, Thomas Bauereiss, Shaked Flur, Christopher Pulte,
  Peter Sewell, Alexander Richardson, Hesham Almatary,
  Jessica Clarke, Microsoft, for contributions by Robert Norton-Wright and
  Nathaniel Wesley Filardo, Peter Rugg,
  Aril Computer Corp., for contributions by Scott Johnson

All rights reserved.

This software was developed by the above within the Rigorous
Engineering of Mainstream Systems (REMS) project, partly funded by
EPSRC grant EP/K008528/1, at the Universities of Cambridge and
Edinburgh.

This software was developed by SRI International and the University of
Cambridge Computer Laboratory (Department of Computer Science and
Technology) under DARPA/AFRL contract FA8650-18-C-7809 ("CIFV"), and
under DARPA contract HR0011-18-C-0016 ("ECATS") as part of the DARPA
SSITH research programme.

This project has received funding from the European Research Council
(ERC) under the European Union’s Horizon 2020 research and innovation
programme (grant agreement 789108, ELVER).


Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in
   the documentation and/or other materials provided with the
   distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ‘‘AS IS’’
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

Abstract

This technical report describes CHERI ISAv9, the ninth version of the CHERI architecture being developed by SRI International and the University of Cambridge. This design captures thirteen years of research, development, experimentation, refinement, formal analysis, and validation through hardware and software implementation.

CHERI introduces an architecture-neutral capability-based protection model, which has been instantiated in various commodity base architectures to give CHERI-RISC-V, Arm's prototype Morello architecture, and (sketched) CHERI-x86-64. It enables software to efficiently implement fine-grained memory protection and scalable software compartmentalization, by providing strong, deterministic, efficient mechanisms to support the principles of least privilege and intentional use in the execution of software at multiple levels of abstraction, preventing and mitigating vulnerabilities. Design goals include incremental adoptability from current ISAs and software stacks, low performance overhead for memory protection, significant performance improvements for software compartmentalization, formal grounding, and programmer-friendly underpinnings.

CHERI blends traditional paged virtual memory with an in-address-space capability model that includes capability values in registers, capability instructions, and tagged memory to enforce capability integrity. This hybrid approach addresses the performance and robustness issues that arise when trying to express more secure, privilege minimising programming models, above conventional architectures that provide only MMU-based protection. 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 improving control-flow robustness. Strong capability integrity and monotonicity properties allow CHERI to express a variety of protection idioms, 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 approach allows incremental adoption of capability-oriented design: critical components can be ported and recompiled to use capabilities throughout, providing fine-grain memory protection, or be largely unmodified but encapsulated in ways that permit only controlled interaction. Potential early deployment scenarios include low-level software Trusted Computing Bases (TCBs) such as separation kernels, hypervisors, and operating-system kernels, userspace TCBs such as language runtimes and web browsers, and 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).

CHERI ISAv9 is a substantial enhancement to prior ISA versions. CHERI-RISC-V has replaced CHERI-MIPS as the primary reference platform, and CHERI-MIPS has been removed from the specification. CHERI architectures now always use merged register files where existing general-purpose registers are extended to support capabilities. CHERI architectures have adopted two design decisions from Arm Morello: 1) CHERI architectures now clear tags rather than raising exceptions if an instruction attempts a non-monotonic modification of a capability; and 2) **DDC** and **PCC** no longer relocate legacy memory accesses by default. CHERI-RISC-V has received numerous updates to serve as a better baseline for an upstream standard proposal

including a more mature definition of compressed instructions in capability mode. CHERI-x86-64 now includes details of extensions to existing x86 instructions and proposed new instructions in a separate ISA reference chapter along with various other updates.

Acknowledgments

The authors of this report thank members of the CTSRD, MRC2, ECATS, CIFV, REMS, ETC, MTSS, and CHaOS 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
Ruben Ayrapetyan
Rosie Baish
Hadrien Barral
Thomas Bauereiss
Stuart Biles
Andrew Bivin
Peter Blandford-Baker
Matthias Boettcher
David Brazdil
Reuben Broadfoot
Kevin Brodsky
Ruslan Bukin
Brian Campbell
Gregory Chadwick
Jianyi Cheng
Serban Constantinescu
Kian Cross
Chris Dalton
Nirav Dave
Dominique Devriese
Mike Dodson
Lawrence Esswood
Jonas Fiala
Wedson Filho
Anthony Fox
Paul J. Fox
Ivan Gomes Ribeiro
Dapeng Gao
Paul Gotch
Tom Grocutt
Khilan Gudka
Brett Gutstein
Jong Hun Han
Andy Hopper
Alex Horsman
Graeme Jenkinson
Timothy Jones
Asif Khan
Myron King
Joe Kiniry
Chris Kitching
Wojciech Koszek
Robert Kovacsics
Karthik Muthusamy
Brian Li
Nix Li
Patrick Lincoln
Marno van der Maas
Anil Madhavapeddy
Jessica Man
Ilias Marinos
Tim Marsland
Ed Maste
Alfredo Mazzinghi
Kayvan Memarian
Paul Metzger
Dejan Milojicic
Andrew W. Moore
Will Morland
Alan Mujumdar
Prashanth Mundkur
Edward Napierala
George V. Neville-Neil
Philip Paeps
Lucian Paul-Trifu
Alex Porter
Austin Roach
Colin Rothwell
John Rushby
Hassen Saidi
Hans Petter Selasky
Andrew Scull
Muhammad Shahbaz
Bradley Smith
Lee Smith
Ian Stark
Samuel Stark
Ramy Tadros
Andrew Turner
Richard Uhler
Munraj Vadera
Jacques Vidrine
Hugo Vincent
Philip Withnall
Vadim Zaliva
Bjoern A. Zeeb

The CTSRD team also thanks past and current members of its external oversight group:

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), Keith Rebello (DARPA SSITH program manager), Lok Yan (DARPA SSITH program manager), Brad Martin (DARPA program manager for ETC seedling), Sergey Bratus (DARPA program manager for MTSS seedling), Daniel Adams and Laurisa Goergen (DARPA I2O SETAs supporting the CRASH and MRC programs), Marnie Dunsmore and John Marsh (DARPA MTO SETAs supporting the SSITH program), Garrett Ramela (DARPA I2O SETA supporting the ETC seedling), Thomas Maddern (DARPA I2O SETA supporting the MTSS seedling), John Goodacre (UKRI), and Georgios Papadakis (UKRI).

Contents

Chapter 1

Introduction

CHERI (Capability Hardware Enhanced RISC Instructions) extends 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 enables *fine-grain protection* and *intentional use* by introducing 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- and C++-language types, memory allocators, and run-time linking.

This application of least privilege and intentional use provides strong, non-probabilistic 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* – for example, implementing *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. There is also the potential to direct compartmentalization through language- or linker-level structures and annotations, such as class or module definitions.

This application of least privilege and intentional use mitigates 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- and 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 13), 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 implemented by processor MMUs (Memory Management Units).

A C/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-RISC-V and Arm's Morello), an introduction to the CHERI-RISC-V 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 12. Historical context and related work for CHERI may be found in Chapter 13. 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. When C and C++ pointers are implemented using capabilities, capability protections constrain the 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.

Capabilities can be used to implement a variety of language-level pointer types including heap, stack, global, thread-local, and function pointers. These protect against application-level mismanipulation and misuse of source-visible pointers, including out-of-range accesses to explicit or implied memory allocations, and corruption of pointers within those allocations. Capabilities can also be used to implement sub-language pointers created and maintained by the compiler, runtime, and operating system, such as the stack pointer, control-flow pointers such as return addresses, and the pointers to internal linkage structures such as Procedure Linkage Tables (PLTs) and the Global Offset Table (GOT). Collectively, strong pointer integrity, pointer provenance validity, bounds, permissions, monotonicity, and encapsulation prevent corrupted pointers from allowing a range of vulnerable behaviors (e.g., buffer overflows) and also directly impede common exploit techniques (e.g., pointer injection).

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. While CHERI has not been specifically designed with other languages in mind, there appear to be many potential applications of CHERI within the language runtimes for higher-level and managed languages.

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. With CHERI, 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.

As with an MMU, CHERI enables a variety of *software operational models* for compartmentalization. Taking advantage of CHERI's strong in-address-space protection, we have explored both intra-process sandboxing models, providing robust encapsulation for dynamic libraries, and also acceleration of the traditional process model (*co-processes*) by colocating multiple separated processes within the same virtual address space.

Formal modeling and verification 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. To this end, we have produced formal models of our instruction-set extensions for multiple architectures, and used both pragmatic SMT-based validation and formal proof to validate that the models satisfy essential security properties such as provenance validity and monotonicity.

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, energy use, and compatibility in order to compete with less secure alternatives.

CHERI is therefore necessarily disruptive, but its architectural, microarchitectural, and software interventions are bounded in scope, and constrained in their implications. Introducing architectural capabilities and tagged memory has significant implications for instruction-set design and maintainability, as well as microarchitectural implications for the processor design and memory subsystem. On the other hand, the general computational, operating-system, and software compilation models are largely retained, and most existing architectural, microarchitectural, and software implementation choices are preserved.

For example, although tagged capabilities are used instead of unadorned integers in the instruction-level descriptions of loads and stores, once checks are performed capabilities are largely reduced to integer addresses for processing by the MMU and memory subsystem. Compilers must use capability-constrained loads and stores for capability-aware code, but the structure (and often format) of those instructions remain the same, and although code is generated to use capabilities rather than integers for pointers, the vast majority of source code remains the same. Running native capability-unaware binary code is specifically designed for and supported, allowing conventional OS compatibility ABI techniques (such as used in the 32-bit to 64-bit transition) to be used.

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 is an *architecture-neutral protection model* in that, like virtual memory, it can be deployed within multiple ISAs. Our initial mapping into the 64-bit MIPS ISA allowed us to develop the CHERI approach; using it, we explored 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 to derive an architecture-neutral CHERI protection model. This architecture-neutral model is discussed in detail in Chapter 3. We have since added CHERI protection to the RISC-V ISA (Chapter 4), developed a lightweight architectural sketch for the x86-64 ISA (Chapter 5), and collaborated with Arm as they have developed the experimental Morello architecture, an application of CHERI to the ARMv8-A architecture [7].

Over the course of this evolution, 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 supports 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. In this report, 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 checked 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.

1.2.2 An Architecture-Specific Mapping into RISC-V

The CHERI-RISC-V ISA (see Chapter 4) is an instantiation of the CHERI protection model to the 32-bit and 64-bit variants of the open-source RISC-V architecture. CHERI adds the following features 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 extend the existing general-purpose integer registers as well as various CSRs to hold capabilities.

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.

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 (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) produce a capability with a cleared tag that can no longer be dereferenced, 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 RISC-V 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.

CHERI-RISC-V applies CHERI protections in different ways compared to our initial CHERI-MIPS architecture. Some of these differences arise from differences in the base ISA; we anticipate that adaptations of CHERI to ISAs will adopt conventions such as instruction-encoding in keeping with their specific flavor and design.

Other design decisions reflect maturity of the CHERI model and lessons learned from CHERI-MIPS. In our initial work on CHERI, we utilized an uncompressed capability format in which each capability was 256 bits in size. This gave us significant flexibility to experiment with capability contents and semantics for bounds checking and capability behaviors. As our protection and software models matured, we turned our attention to the performance implications of large capability sizes, including potentially substantial data-cache overhead for pointer-intensive applications. We adapted an approach explored in fat-pointer research – bounds compression exploiting redundancy between a pointer value and its bounds – to implement 128-bit compressed capabilities. As of CHERI ISAv8, capabilities are assumed to use compressed bounds whether based on a 32-bit or 64-bit base address size.

CHERI-MIPS used a separate *capability register file* to hold capability registers rather than extending general-purpose registers. CHERI-RISC-V instead extends general-purpose registers to hold capabilities.

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 CHERI-x86-64 and Arm Morello

The abstract CHERI memory protection and security models described above have been applied to two other architectures:

CHERI-x86-64 is a sketch application of CHERI to the Intel x86-64 architecture (Chapter 5). It is not fully elaborated, and has not been implemented, but serves to demonstrate how CHERI could be applied to a CISC architecture.

Arm Morello is an experimental application of CHERI to the ARMv8-A architecture [7]. Developed by Arm, Morello is a complete integration into a rich commercially used load-store architecture that includes features such as vector instructions and virtualization support. We have ported our complete CHERI software stack to Morello.

There is a high degree of source-level compatibility between software across all CHERI architectures. Compilers and low-level operating-system components necessarily have modest amounts of architecture-specific code generation, assembly code, and status or control registers. However, high-level CHERI-aware systems C/C++ code is entirely portable across the multiple architectures. This is comparable to the portability of architecture-neutral virtual-memory abstractions and APIs despite architecture-specific interfaces to MMUs and TLB management.

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 [52], and now for CHERI-MIPS and CHERI-RISC-V in Sail [9]. 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-RISC-V 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 [21].

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 [114, 115].

The CHERI design process has also benefitted from an interplay with our work on rigorous semantics for C [97, 98].

1.5 CHERI ISA Version History

This is the nineth version of the CHERI ISA specification document. A high-level summary of CHERI ISA versions and their corresponding contributions can be found in Table 1.5. A much more detailed version summary and complete change log can be found in Appendix A. A more narrative exploration of the research and development cycle leading to our current specification can be found in Chapter 12.

1.5.1 Changes in CHERI ISA 9.0

This version of the *CHERI Instruction-Set Architecture* is a full release of the Version 9 specification:

- We have shifted to CHERI-RISC-V as our primary reference platform instead of CHERI-MIPS. This included several changes to Chapter 2 and Chapter 3 to replace MIPS-specific details with more architectural-neutral concepts. Section 3.8 was also moved to Chapter 3.

- The privileged architecture portions of CHERI-RISC-V are now defined as an extension to version 1.11 of the RISC-V privileged architecture specification.

- CHERI-RISC-V reports capability exception details in *x*`tval` rather than *x*`ccsr`.

- The RISC-V `JAL` and `JALR` instructions are now mode-dependent meaning that they use capability register operands in capability mode rather than always using integer registers. The capability mode version of these instructions are named `CJAL` and `CJALR`. The previous `CJALR` instruction has been renamed to `JALR.CAP`. In addition, `JALR.PCC` has been added to permit integer jump and links in capability mode.

- Section 4.3.8 has been rewritten to reflect an initial implementation of CHERI-RISC-V compressed instructions in capability encoding mode.

- Opcode encodings have been reserved for CHERI-RISC-V memory versioning instructions as well as `CRelocate`.

- CHERI-RISC-V always uses a merged register file and the architecture-neutral chapters now assume a merged register file on all CHERI architectures. This included removing the dirty bit from *x*`ccsr` as well as the `CGetAddr`, `Clear`, and `CSub` instructions.

- CHERI-RISC-V clears tags rather than raising exceptions for non-monotonic modifications to capabilities.

- Added `CGetHigh` and `CSetHigh` to retrieve and modify the upper half of a capability.

- Added `CGetTop` to retrieve the upper limit of a capability.

- **DDC** and **PCC** no longer relocate legacy memory accesses. These registers still constrain legacy memory accesses. This included deprecating `CFromPtr` and `CToPtr`.

- Removed CHERI-MIPS from the specification as it is deprecated and no longer actively developed.

- Added a new section in Chapter 2 describing potential uses of capabilities to protect physical addresses.

- CHERI-RISC-V now enables/disables CHERI extensions via a bit in the `menvcfg` and `senvcfg` CSRs rather than *x*`ccsr`.

- CHERI-RISC-V ***x*ScratchC** capability registers now extend the existing *x*`scratch` registers.

- We have expanded the CHERI-x86-64 sketch in Chapter 5 to include details on extensions to existing instructions to support operations on capabilities as well as details for new instructions in a new ISA reference in Chapter 8.
- Added a description of the 64-bit CHERI Concentrate capability format.

1.6 Experimental Features

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

- Efficient tag rederivation for use with swapping, memory compression, memory encryption, and virtual-machine migration
- A number of architectural features to accelerate temporal memory safety and capability revocation: fast capability subset testing, non-temporal tag loading, and non-temporal capability loading.
- A recursive mutable load permission that limits the store rights via future capability loads
- More efficient capability permission representations
- Memory versioning for use with capabilities
- Linear capabilities
- Indirect capabilities
- Indirect sealed entry capabilities
- 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

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 the specific CHERI-RISC-V ISA.

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 32-bit and 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 5 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 CHERI-RISC-V instruction.

Chapter 8 provides a detailed description of each CHERI-x86-64 instruction.

Chapter 9 discusses the design rationale for many aspects of the CHERI ISA, as well as our thoughts on future refinements based on lessons learned to date.

Chapter 10 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 11 provides a high-level summary of how CHERI can be integrated into RISC pipelines and the memory subsystem.

Chapter 12 describes the motivations and hardware-software co-design research approach taken in developing CHERI, including major phases of the research.

Chapter 13 describes the historical context for this work, including past systems that have influenced our approach.

Chapter 14 discusses our short- and long-term plans for the CHERI protection model and CHERI ISAs, 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 ISAs.

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

Appendix C specifies a number of CHERI 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 instruction-set architectures, their 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* [179]. 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* [27], 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* [172], 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* [59], 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* [98], 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* [149], 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* [26]. 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 [160]. *Fundamental Trustworthiness Principles in CHERI* discusses how CHERI fulfills a number of critical trustworthiness principles [111].

- In the International Conference on Computer Design (ICCD 2017), we published *Efficient Tagged Memory* [70]. 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 2018), we published *CheriRTOS: A Capability Model for Embedded Devices* [185]. 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 [9].

- 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 [97].

- 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* [40].

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* [178]. 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.

- In the IEEE/ACM International Symposium on Microarchitecture (IEEE MICRO 2019), we published *CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety* [184]. This paper performs a simulation study of a potential approach to temporal memory safety using CHERI.

- In the IEEE Symposium on Security and Privacy (Oakland 2020), we published *Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process* [114]. This paper describes formal modeling and proof of isolation properties for the CHERI-MIPS ISA.

- In the IEEE Symposium on Security and Privacy (Oakland 2020), we published *Cornucopia: Temporal Safety for CHERI Heaps* [51]. This paper describes a full hardware-software implementation of temporal memory safety for CHERI, including architectural accelerations.

- In the proceedings of Hardware and Architectural Support for Security and Privacy (HASP 2020), we published *Position Paper: Defending Direct Memory Access with CHERI Capabilities* [89]. This paper proposes new solutions that can efficiently address the problem of malicious memory access from pluggable computer peripherals and microcontrollers embedded within a system-on-chip.

- In Workshop on Computer Architecture Research with RISC-V (CARRV 2021), we published *Developing a Test Suite for Transient-Execution Attacks on RISC-V and CHERI-RISC-V* [53]. This paper presents a flexible and extensible bare-metal test suite containing replications of all major transient-execution attacks in RISC-V.

- In the proceedings of the 31st European Symposium on Programming (ESOP 2022) we published *Verified Security for the Morello Capability-enhanced Prototype Arm Architecture* [11]. In this paper we define the fundamental security property that Morello aims to provide, reachable capability monotonicity, and prove that the architecture definition satisfies it. We also published an extended version as a technical report [12].

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 (Version 9)* [161, 162, 163, 164, 165, 166, 167, 168], describes the CHERI ISA, both as a high-level, software-facing model and the specific mapping into multiple instruction sets. 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* [158] 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* [157].

- The *Bluespec Extensible RISC Implementation (BERI): Hardware Reference* [170, 171] describes hardware aspects of our prototyping platform, including physical platform and practical user concerns.

- The *Bluespec Extensible RISC Implementation (BERI): Software Reference* [155, 156] 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)* [60], 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* [173], 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 technical report, *Rigorous engineering for hardware security: formal modelling and proof in the CHERI design and implementation process* [115], is a preprint version of our similarly named paper published at Oakland 2020 [114].

- *An Introduction to CHERI* [159] provides a high-level introduction to CHERI: the abstract protection model, architectural instantiations, formal modeling, microarchitectural implementation, and software stack. It was released alongside the announcement of the Arm Morello board [7].

- The *CHERI C/C++ Programming Guide* [169] is an introduction to pure-capability C/C++, variants of the C and C++ programming languages targeting implementation of all pointers using CHERI architectural capabilities.

- The *Arm Morello Programme: Architectural security goals and known limitations* [154] lays out the specific architectural security objectives of the Arm Morello prototype, as well as areas that fell out of scope for the project.

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 [151, 152].

- Jonathan Woodruff's PhD dissertation, *CHERI: A RISC capability machine for practical memory safety*, describes our CHERI1 prototype implementation [180].

- Robert Norton's PhD dissertation, *Hardware support for compartmentalisation*, describes how hardware support is provided for optimized domain transition using the CHERI2 prototype implementation [116].

- 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 [71].

- Alexander Richardson's PhD dissertation, *Complete spatial safety for C and C++ using CHERI capabilities*, describes the implementation of C/C++ compilation and linkage using CHERI capabilities for spatial memory safety [127].

- Hongyan Xia's PhD dissertation, *Capability memory protection for embedded systems*, describes 64-bit CHERI capabilities along with a real-time OS to evaluate their effectiveness [183].

- Lawrence Esswood's PhD dissertation, *CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor*, describes CheriOS, a clean-slate CHERI-specific operating system and hypervisor [48].

- Michael Dodson's PhD dissertation, *Capability-based access control for cyber physical systems*, demostrates the composition of hardware-enforced architectural capabilities and cryptographic network tokens to implement object capabilities in a distributed cyber physical system [45].

- Brett Gutstein's PhD dissertation, *Memory safety with CHERI capabilities: security analysis, language interpreters, and heap temporal safety*, describes a memory-operations framework for reasoning about memory-safety mitigations, presents a CHERI-aware implemention of Apple's JavaScriptCore, and describes the Cornucopia temporal memory safety implementation [62].

- Hesham Almatary's PhD dissertation, *CHERI compartmentalisation for embedded systems*, describes CompartOS, a new lightweight hardware-software compartmentalisation model building on CHERI to secure mainstream and complex embedded software systems [3].

- Kayvan Memarian's PhD dissertation, *The Cerberus C semantics*, describes Cerberus, an executable model for a substantial fragment of C11 expressed as a translation to a purpose-built language called Core [96].

- Peter Rugg's PhD dissertation, *Efficient spatial and temporal safety for microcontrollers and application-class processors*, describes the implementation of CHERI on three RISC-V microarchitectures as well as the refinement of support for temporal memory safety [130].

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) [113], 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) [174].

- 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* [113].

- In the USENIX Workshop on the Theory and Practice of Provenance (TaPP), we published *Pointer Provenance in a Capability Architecture* [92]. 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.

Table 1.1: 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 [162]	Fat pointers + capabilities, compiler Instructions to optimize hybrid code Sealed capabilities, `CCall`/`CReturn`
2015	ISAv4 [167]	MMU-CHERI integration (TLB permissions) ISA support for compressed capabilities Hardware-accelerated domain switching Multicore instructions: `LL`/`SC` variants
2016	ISAv5 [168]	CHERI-128 compressed capability model Improved generated code efficiency Initial in-kernel privilege limitations
2017	ISAv6 [166]	Mature kernel privilege limitations Further generated code efficiency CHERI-x86 and CHERI-RISC-V sketches Jump-based protection-domain transition
2019	ISAv7 [165]	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
2020	ISAv8 [164]	Compressed capabilities in abstract model 32- and 64-bit address sizes Deployed sentry capabilities Fully elaborated CHERI-RISC-V MMU-assisted load-side-barrier revocation Richer microarchitectural exploration Synchronized with Arm Morello architecture [7]
2023	ISAv9 [163]	CHERI-RISC-V as primary reference platform CHERI-MIPS removed Capabilities stored in general-purpose registers Clear tags for non-montonic modifications DCC and PCC relocation disabled by default CHERI-x86-64 instruction descriptions

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 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 support common (and portable) programming models and approaches.

We describe the cross-architecture aspects of CHERI in Chapter 3. Collectively, our instantiations of the CHERI protection model in the 32- and 64-bit RISC-V (Chapter 4) and 64-bit ARMv8-A (Morello [7]) ISAs demonstrate the portability of the model despite diverse underlying architectural implementations.

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 [64]. 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.

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* [111].

2.2 CHERI Capabilities: Strong Protection for Pointers

A key purpose of the CHERI protection model is to provide architectural primitives to support strong protection for C and C++-language pointers. Typically, language-level pointer types are implemented using architectural integers in registers and in memory. CHERI provides a new architectural data type, the *capability*, which software (such as a compiler) can use use instead when implementing pointers. CHERI protections complement existing architectural protection mechanisms such as virtual memory implemented by a Memory Management Unit (MMU) or non-virtualized protection implemented by a Memory Protection Unit (MPU). CHERI 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 scalable 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 only by low-level operating-system components such as kernels, hypervisors, and system libraries, CHERI capabilities are also targeted more broadly at compiler and language-runtime use, allowing program structure and dynamic memory allocation to direct their use.

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 address use – while providing additional metadata to the architecture allowing the implementation of stronger memory safety. Code in which all pointers (and implied 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*

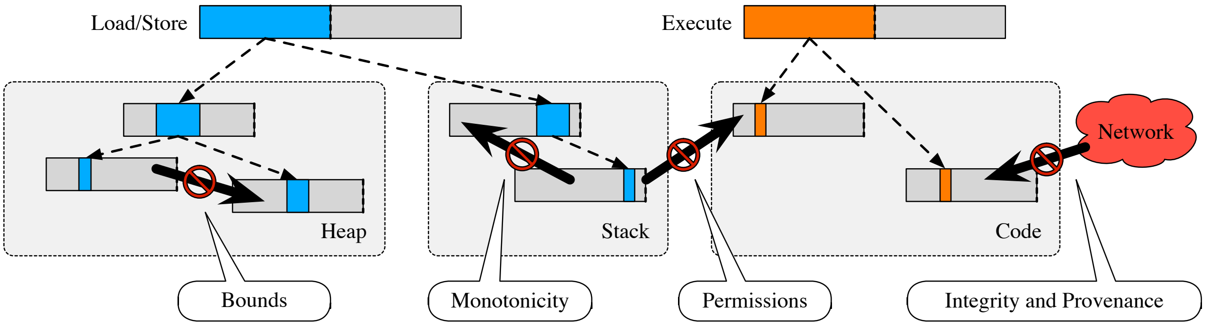

Figure 2.1: CHERI enforces strict *integrity*, *provenance validity*, *monotonicity*, *bounds*, *permissions*, and *encapsulation* on pointers, mitigating common vulnerabilities and exploit techniques.

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, software typically implements pointers as integer values stored in two architectural forms: in integer registers, and in memory. *Architectural capabilities* are a new architectural data type likewise stored in register and memory, and containing an integer value that will most frequently be interpreted as an address. Capabilities also contain a number of other fields that contain additional metadata associated with the address, such as bounds and permissions, as well as a tag protecting their integrity. Compilers, toolchains, and runtimes implementing pointers in terms of architectural capabilities can imbue pointers with those protections, enforced in hardware, subject to appropriately managing that metadata. Capabilities are $2\times$ (plus 1 bit) the size of the architecture's natural address size, with the metadata compressed to fit in the additional space. On 32-bit architectures, capabilities are 64 bits (+1 tag bit), and on 64-bit architectures, capabilities are 128 bits (+1 tag bit). In many senses, architectural capabilities used to implement pointers act like the integers they replace, being loaded or stored, loaded or stored via, jumped to, and so on. The operating system, compiler, linker, and language runtimes are able to use them to implement fine-grained memory safety for C and C++, as well as for other purposes such as in-address-space software compartmentalization.

The majority of the capability is stored in a register or in addressable memory, as is the case for current integer pointers, with additional metadata stored adjacent to the integer address it protects. 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 architectural 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. While implementation 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 64-bit or 128-bit granularity. The capability's 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 a 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 address space, which can help support techniques such as garbage collection.

Our CHERI prototypes implement tagged memory using partitioned memory, with tags and associated capability-sized units linked close to the memory controller, and propagated by the cache hierarchy in order to provide strong atomicity with the data it protects. However, it is also possible to imagine implementations in which DRAM – e.g., alongside ECC metadata – or non-volatile memory is extended to store tags with capability-sized units as well. 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 capabilities in descriptor rings).

2.3.2 Bounds on Capabilities

Capabilities contain lower and upper bounds for the memory they authorize access to. While a capability's address may move out of bounds (and perhaps back in again), attempts to dereference (e.g., via a load, store, or instruction fetch) an out-of-bounds capability 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 addresses to sometimes be out-of-bounds with respect to their bounds – without faulting – is important for de-facto C-language compatibility. In an ideal world, addresses could be arbitrarily out of bounds. However, our bounds-compression scheme places restrictions on this property, as bounds compression depends on redundancy between the address and bounds, which is reduced when addresses are substantially outside of their bounds (see Section 3.5.3 for details).

Bounds originate in allocation and linking events. The operating system is able to place bounds on pointers to initial address-space allocations during process startup (e.g., via the initial register file, and ELF auxiliary arguments in memory), and on an ongoing basis as new address-space mappings are made available (e.g., via `mmap` system calls). In practice, 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 Capabilities

Capabilities additionally extend addresses with a permissions mask controlling how the capability may be used. For example, the run-time linker or compiler may set a capability's permissions so that pointers to data cannot be reused for instruction fetch, 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 can 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 protection model 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.

Stripping the tag in register write-back Some instructions are able to represent non-monotonic operations, but attempts to use them non-monotonically will write back a capability 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.

---

[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.

Exceptions on monotonicity violation As an alternative to stripping the tag, attempts to use instructions non-monotonically could lead to an exception being delivered. We generally avoid this approach in favour of stripping the tag for the reasons discussed in Section 9.27.

Stripping the tag in memory store Tagged memory ensures that direct modification of capabilities stored in memory using data store instructions (whether non-monotonically or otherwise) will clear the tag on affected in-memory capabilities. This causes 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-A). 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 strict 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 2.4.5).

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. That is, these flags are intended to affect the *semantics* of access, rather than impose access control. 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, such as to hint as to cache interactions for shared-memory rings, or to control the behavior of operations such as capability equality testing.

2.3.6 Sealed Capabilities

Capability *sealing* allows capabilities to be marked as *immutable* and *non-dereferenceable*, causing the tag to be cleared if attempts are made to modify them, and 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. There are two forms of capability sealing: pairs of capabilities sealed using a common *object type*, and stand-alone *sealed entry capabilities* (sentry capabilities).

Sealed pairs are primarily designed to support the linking of a pair of code and data capabilities for use together during domain transition. A jump-like instruction, `CInvoke`, allows the two sealed capabilities to be atomically unsealed as control flow transfers to the code pointed to by the code capability, if their object types match. This can be used to implement controlled privilege escalation for the purposes of domain transition.

Sealed entry capabilities simply seal a single code capability, which likewise can be jumped to leading to an atomic unsealing and control-flow transfer. This can also be used to implement domain transition with privilege escalation, but to date has primarily been used to strengthen control-flow robustness within a single protection domain by preventing the undesired manipulation and use of code pointers. Jump-and-link instructions acting on sealed entry capabilities also generate a sealed return capability.

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, 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.

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.

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). The `CInvoke` 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 by virtue of unsealing of the operand registers.

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. `CInvoke` behaves much like a conventional jump to register, permitting an in-address-space domain switch without changing rings.

The newly executing code has the ability to further manipulate execution state, and impose semantics such as call-return secure function invocation or secure asynchronous message passing, which will likely be followed by a privilege de-escalation as a target domain is entered (see Section 2.4.5).

Object-Capability Policies in CHERI

Consider an execution environment having access to several capabilities sealed with the same `otype`. The tests required by the `CInvoke` 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 `CInvoke` tests pass.

Non-Cartesian and/or stateful policies can 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 PTE bits extend the existing load and store permissions on 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 an 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.

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 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 PTE 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

Architecturally, capability fields are exposed via a set of accessor instructions that get or set field values, such as the address, upper bound, lower bound, and permissions. The in-register and in-memory formats for capability contents may differ substantially, permitting a more efficient representation using compressed capability bounds. CHERI utilizes a floating-point-like *fat-pointer compression technique* that relies on redundancy between the address, lower bound, and upper bound. The compressed representation exchanges stronger alignment requirements (proportional to object size) for a more compact representation.

The CHERI Concentrate compression model (see Section 3.5.3) maintains monotonicity: 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, for example, a 256-bit representation as found in earlier CHERI versions.

2.3.12 Hybridization with Integer Pointers

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

Default Data Capability (DDC) DDC constrains legacy instructions that load and store relative to integer addresses rather than capabilities.

Program Counter Capability (PCC) PCC extends the conventional program counter with capability metadata, constraining instruction fetches.

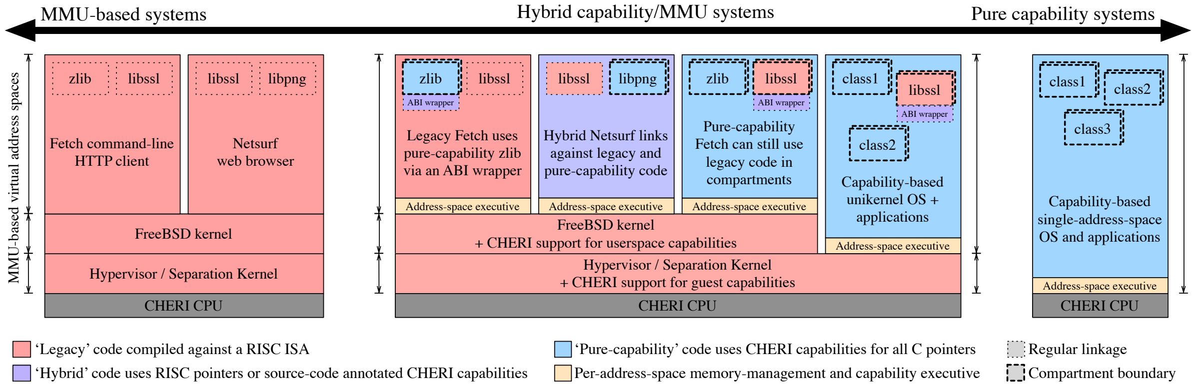

Figure 2.2: CHERI supports a wide range of operational software models including: unmodified MMU-based operating systems; hybrid operating systems utilizing the MMU to support a process model and/or virtualization while using CHERI within virtual address spaces; and pure single-address-space CHERI-based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

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. 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). We have selected to deliver failures as *hardware exceptions*; for example, on attempts to perform disallowed load and store operations. 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.

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. 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 address space, or temporarily limiting access to portions of the 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 handler, there is the opportunity for that handler to implement revocation semantics explicitly. For example, the object invocation handler of a trusted userspace supervisor entered by `CInvoke` could interpret the address of a sealed capability as pointing to a table entry within its domain, rather than directly encapsulating a pointer to the target object's data. 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 invocation handler 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. 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 [40], 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 (and not just the operating system) is intended to play a significant 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 user-space 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 Protecting Physical Addresses

A new capability type (distinguished using permissions, for example) could be used to express and protect physical addresses. The operating system would use these capabilities internally to control access to physical addresses to IO memory, for example. As most physical memory would be addressed using virtual memory mappings, it would be natural to use physical capabilities as leaf node page table entries (PTEs), reusing capability permissions as PTE permissions, though some additional permissions would be necessary (e.g. dirty).

Synergy with linear capabilities and enclaves Linear physical capabilities could support low-overhead enclaves. If a physical capability could be linear, such that each instance is guaranteed to be the unique reference to that physical memory, an enclave could be certain that its virtual memory mapping is the only route to access that physical page. Additionally, it would be necessary to remove low-level page-table management from the general operating system to ensure that the physical capability is not otherwise dereferenced.

Synergy with sparse, contiguous page tables ASAP [88] has proposed speculatively assuming that page tables are contiguous, if sparse, such that an address can directly index each level of the page table to produce the probable address of its PTEs. This allows all levels to be queried in parallel, and will be likely to be correct if the operating system succeeds in using the expected pages for the page table. If the PTEs were capabilities of a unique type, it should be possible to accurately distinguish a PTE from data. With appropriate guarantees in page table layout (i.e. no overlapping contiguous sparse page tables), the page walker can directly load the expected leaf entry and know if it is a valid translation. For practical contiguous page table sizes, it may be appropriate to use this optimisation for the level before the leaf such that we have a fixed 2-level page walk, at least in the common case. We should note that this optimisation could be used apart from CHERI if physical pages could be tagged to distinguish PTE pages from data pages.

2.4.5 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 [59, 73, 121, 153]. 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 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 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.

2.4.6 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.

2.4.7 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.

2.4.8 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. 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 [77]. 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* [173].

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.

3.1 Architectural Instantiations of CHERI Protection

Our current instantiations within concrete ISAs are:

CHERI-RISC-V is our mature reference instantiation. It is an instantiation of the CHERI protection model against 32-bit and 64-bit RISC-V (Chapter 4).

CHERI-RISC-V has been validated with a complete end-to-end hardware-software stack including a formal ISA model, ISA-level simulations, three FPGA implementations, adaptations of our CheriBSD and CheriFreeRTOS operating systems, Clang/LLVM/LLD toolchain, GDB debugger, and application suite.

We aim to propose 64-bit CHERI-RISC-V as a RISC-V extension with minimal adjustments. We consider 32-bit CHERI-RISC-V less mature and expect future disruptive modifications as it transitions to a more mature status.

Arm Morello is an experimental instantiation created by Arm in collaboration with the CHERI team [7]. It is an instantiation of the CHERI protection model against the 64-bit ARMv8-A ISA.

Morello is the target of an in-progress CPU, SoC, and board design based on Arm's Neoverse N1 system architecture, and has been validated for much of the end-to-end-hardware-stack including a formal ISA model, ISA-level simulations, an adaptation of our CheriBSD operating system, Clang/LLVM/LLD toolchain, GDB debugger, and application suite.

CHERI-x86-64 is a sketch instantiation intended to describe a potential approach to applying the CHERI protection model to the x86-64 ISA – the dominant non-RISC architecture (Chapter 5).

3.2 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 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 processors, 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. Protection primitive use is common-case, not exceptional, and occurs in performance-centric code paths such as stack and heap allocation, on pointer arithmetic, and on

pointer-relative 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 or the use of architectural or microarchitectural secrets subject to leaking or side-channel attacks. For example, we avoid the use of cryptographic hashes, random address-space bits, and version numbers that must be truncated to small numbers of bits within a pointer or capability, instead making use of tagging. This offers resistance to attacks at stastical scale (e.g., millions of devices), and also protects software structures that might otherwise reuse secrets allowing multiple attempts (e.g., forked daemon or zygote processes). Tags allow strong 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, deterministically true, 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.

3.3 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.

3.4 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.

3.4.1 Address Size and Capability Size

Architectural capabilities are sized with respect to the address size of the architecture. As we define CHERI capability variants for both 32-bit architectures and 64-bit architectures, we parameterize the definitions in this chapter as follows:

XLEN is the architectural address size in bits. For 32-bit architectures, `XLEN` is 32. For 64-bit architectures, `XLEN` is 64.

CLEN is the architectural capability size in bits, which is 2× the architectural address size (and does not include the tag bit). For 32-bit architectures, `CLEN` is 64. For 64-bit architectures, `CLEN` is 128.

3.4.2 Capability Contents

Capabilities contain a number of software-accessible architectural fields, which may differ in content and size from the microarchitectural implementation or that is apparent from its 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**”, 4 bits for 64-bit capabilities or 18 bits for 128-bit capabilities)
- Offset (“**offset**”, `XLEN`)
- Base virtual address (“**base**”, `XLEN`)
- Length in bytes (“**length**”, `XLEN`)

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 capability is invalid, and cannot be dereferenced. Section 3.5.2 describes the behavior of tagged memory.

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_INVOKE	✓	Sealed	-
9	PERMIT_UNSEAL	✓	Unsealed	Object Type
10	PERMIT_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.

Permission Bits

The **perms** bit vector governs the architecturally defined permissions of the capability including read, write, and execute permissions.[1] 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.

---

[1] Although these values are used in CHERI-RISC-V, the specific integer constants – and in some cases the named permissions – differ in Arm's Morello.

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_INVOKE Allow this sealed capability to be used with `CInvoke`.

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) 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.

otype value	Interpretation
$2^{\mathsf{XLEN}} - 1$	Unsealed capability
$2^{\mathsf{XLEN}} - 2$	Sealed entry (“sentry”) capabilities; see Section 3.9
$2^{\mathsf{XLEN}} - 3$	Reserved (experimental “memory type tokens”; see Appendix C.11)
$2^{\mathsf{XLEN}} - 4$	Reserved (experimental “indirect enter capabilities”; see Appendix C.8)
$2^{\mathsf{XLEN}} - 5$ through $2^{\mathsf{XLEN}} - 16$	Reserved
other	Capability sealed by `CSeal`

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

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. We define 0 software-defined permission bits for 64-bit capabilities, and 4 software-defined permission bits for 128-bit capabilities.

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. Capabilities with all hardware-defined permission bits cleared have only software-defined interpretations, making them suitable for potential use as unforgeable tokens of authority 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 **otype** field is 4 bits for 64-bit capabilities, and 18 bits for 128-bit capabilities. The 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. 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. 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 `CInvoke` instruction, as in Section 3.8, or `CJALR` instructions, as in Section 3.9). 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.

Base

The `XLEN`-bit **base** field is the base 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 `XLEN`-bit values of **base** will be representable (see Section 3.5.3).

Offset

The `XLEN`-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 3.5.3).

Address

The 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 single combined entity (e.g., via `CSetAddr`) depending on the software use case.

Length

The `XLEN`-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 `XLEN`-bit values of **length** will be representable (see Section 3.5.3).

3.4.3 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_INVOKE, may also be present. The following architectural values will normally be used:

- The **tag** is set.
- The capability is unsealed (has **otype** of $2^{\mathsf{XLEN}} - 1$).
- **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 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. Due to bounds compression, the memory allocation may require stronger than word alignment or padding so as to ensure non-overlapping bounds with other allocations. 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 $2^{\mathsf{XLEN}} - 1$).
- **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^{XLEN}).

3.4.4 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 $2^{\mathsf{XLEN}} - 1$).
- **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^{XLEN}).

Note 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.

3.4.5 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 3.7) 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.

CHERI architectures extend the existing general-purpose integer register file to allow it to hold `XLEN`-sized integers and also capabilities, with instructions selecting the desired semantics when utilizing a register. 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.

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 checked using an appropriate implied capability such as the Program-Counter Capability (**PCC**) or Default Data Capability (**DDC**).

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-A), 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. See Section 3.5.3 for a discussion of capability compression, used to avoid storing a minimum of 3× `XLEN` bits in each capability.

3.4.6 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 provide similar functionality to architecture-specific special registers such as RISC-V *Control and Status Registers*. In many cases, SCRs extend an existing special register. SCRs are accessed via new variants of architecture-specific instructions used to access special registers, and serve specific architectural functions. Access to special capability registers is controlled on a case-by-case basis and may be restricted based on PERMIT_ACCESS_SYSTEM_REGISTERS, execution ring, or exception-handling state. The specific registers vary by underlying architecture, but will include the following:

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) constrains legacy non-capability loads and stores, controlling data accesses to memory.

Although these capability special registers may be viewed as extensions to existing special registers (e.g., **PC**), 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.

Where existing special registers, such as the Program Counter (**PC**), 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, so that capability-unaware code can jump within its code segment without experiencing a tag violation.

3.4.7 Values Extended to Capabilities

Several other existing values also require extending to hold capabilities. These values may be stored in a general-purpose capability register, a special capability register, or some other architecture-specific location. When possible, the capability variant should be stored as an extension of the equivalent value from the base architecture:

Exception Program Counter Capability Just as conventional architectures save the **PC** following an exception and restore the **PC** on exception return, CHERI architectures must save and restore the full **PCC** when handling exceptions.

Exception Code Capability When an exception is taken, **PCC** must be replaced with a code capability containing a suitable execution and security context for the exception handler.

Exception Data Capability When an exception is taken, the exception handler must have a way to access a suitable data capability for use by the exception handler. This capability should permit access to a stack pointer as well as a value for **DDC**.

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

3.5 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.

3.5.1 In-Memory Representation

As implemented in 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 65th or 129th 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:

NULL has an all zeroes in-memory representation, with 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.

The bottom `XLEN` bits of a capability hold its address value. 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.

The CHERI Concentrate compression format used for both 64-bit and 128-bit capabilities is described in Section 3.5.3. 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 heterogeneous 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 `XLEN` 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.

3.5.2 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 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.

3.5.3 Compressed Capabilities

In the abstract, full precision capabilities (i.e., those containing all of the architectural capability fields at full width in their in-memory representation) offer higher levels of software compatibility, but at a cost: quadrupling the memory size of pointers implemented using capabilities. 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 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 64-bit architectural view of their fields. 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 3.5.4 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.

CHERI ISAv8 specifies a single compression scheme for capabilities, CHERI Concentrate.[2]

3.5.4 CHERI Concentrate Compression

In this section, we describe how CHERI Concentrate compresses the bounds used in 128-bit capabilities with 64-bit architectural addresses.[3]

CHERI Concentrate is a compressed capability encoding that uses a floating point representation to encode the bounds relative to the capability's address [178]. 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 9.23.

Figure 3.1 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), a flag bit, 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.

---

[2] CHERI-128 (Appendix E), our previous compression format, is now deprecated.

[3] A variant of CHERI Concentrate is used in Arm Morello, but with different precision constants and a slightly different encoding format.

63								0
p'16	f	▨	otype'18	I_E	$T[11:3]$	T_E'3	$B[13:3]$	B_E'3
a'64								

f: flag p: permissions otype: object type a: pointer address

If $I_E = 0$:

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

If $I_E = 1$:

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

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
false	false	0
false	true	+1
true	false	−1
true	true	0

$A_3 < R$	$B_3 < R$	c_b
false	false	0
false	true	+1
true	false	−1
true	true	0

Figure 3.1: CHERI Concentrate 128-bit capability format and decoding

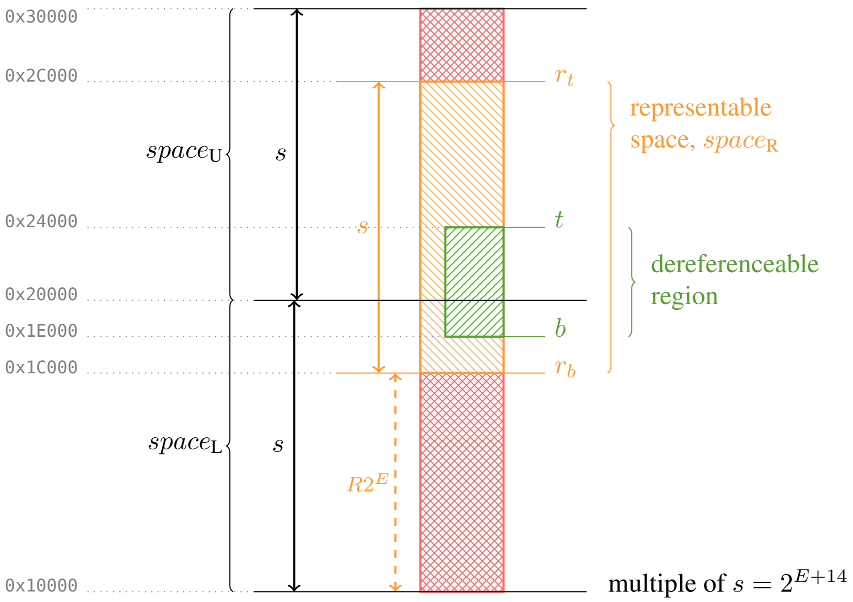

Figure 3.2: Graphical representation of memory regions encoded by CHERI Concentrate. The example addresses on the left are for a `0x6000`-byte object located at `0x1E000`; the representable region extends `0x2000` below the object's base and `0x8000` above the object's limit.

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 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 an aligned $s = 2^{E+14}$ space of values over which a can range without changing the decoded bounds. Requiring this space to be aligned would be an unacceptable restriction for software, so we make use of 'spare' encodings where T is less than B to allow an arbitrary space boundary, R, that is relative to the base, calculated 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 3.2. This allows us to compute the corrections to a_{top}, c_b and c_t, shown in the tables in Figure 3.1. 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:

$$\textbf{if } ((E < 51)\,\&((t[64:63] - b[63]) > 1)) \textbf{ then } t[64] = !t[64]$$

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 $E = 52$, $T = 2^{12}$ 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:

$$B = b[13:0] T = t[11:0]$$

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

$$B = b[E+13:E+3] T = t[E+11:E+3]$$

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`. Rounding up t to a 2^{E+3} aligned value may increase the length, and therefore might cause L_{msb} to increase by one, therefore mandating that the E of the resulting capability also increase so that L_{msb} lands at exactly $E+12$ to ensure correct decoding. Selecting a new E forces a fresh selection of T and B, but is certain not to overflow again.

CHERI Concentrate Alignment Requirements

For a requested base and length 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 length must be aligned to 2^{E+3} byte boundaries (i.e., the $E + 3$ least significant bits are zero). E is determined from the requested length l and is subject to rounding such that an E_{initial} is calculated for l, which is then aligned up to l_{aligned} which is used to derive the final E. Specifically, $E = E_{\text{initial}} + C$, where $E_{initial} = 52 - \text{CountLeadingZeros}(l[64 : 13])$ and C is an additional carry bit from rounding up the truncated (to $E_{\text{initial}} + 3 + MW - 4$ bits) length to a multiple of 2^{E+3} (that is, $C = 1$ if and only if any of the $E_{\text{initial}} + 3$ least significant bits of l are non-zero and the next $MW - 4$ least significant bits of l are all 1).
- 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 Fast Representable Limit Checking

Pointer arithmetic is typically performed using addition, and does not raise an exception. If we wish to preserve these semantics for capabilities, capability pointer addition must fit comfortably within the delay of simple arithmetic in the pipeline, and should not introduce the possibility of an exception. For CC, as with Low-fat, typical pointer addition requires adding only an offset to the pointer address, leaving the rest of the capability fields unchanged. However, it is possible that the address could pass either the upper or the lower limits of the representable space, beyond which the original bounds can no longer be reconstituted. In this case, CHERI Concentrate clears the tag of the resulting capability to maintain memory safety, preventing an illegal reference to memory from being forged. This check against the representable limit, R, has been designed to be much faster than a precise bounds check, thereby eliminating the costly measures the Low-fat design required to achieve reasonable performance.

To ensure that the critical path is not unduly lengthened, CHERI Concentrate verifies that an increment i will not compromise the encoding by inspecting only i and the original address field. We first ascertain if i is *inRange*, and then if it is *inLimit*. The *inRange* test determines whether the magnitude of i is greater than that of the size of the representable space, s, which would certainly take the address out of representable limits:

$$inRange = -s < i < s$$

The *inLimit* test assumes the success of the *inRange* test, and determines whether the update to A_{mid} could take it beyond the representable limit, outside the representable space:

$$inLimit = \begin{cases} I_{\text{mid}} < (R - A_{\text{mid}} - 1), & \text{if } i \geqslant 0 \\ I_{\text{mid}} \geqslant (R - A_{\text{mid}}) \text{ and } R \neq A_{\text{mid}}, & \text{if } i < 0 \end{cases}$$

The *inRange* test reduces to a test that all the bits of I_{top} ($i[63 : E + 14]$) are the same. The *inLimit* test needs only 14-bit fields ($I_{\text{mid}} = i[E + 13, E]$) and the sign of i.

The I_{mid} and A_{mid} used in the *inLimit* test do not include the lower bits of i and a, potentially ignoring a carry in from the lower bits, presenting an *imprecision hazard*. We solve this by conservatively subtracting one from the representable limit when we are incrementing upwards, and by not allowing any subtraction when A_{mid} is equal to R.

One final test is required to ensure that if $E \geqslant 50$, any increment is representable. (If $E = 50$, the representable space, s, encompases the entire address space.) This handles a number of corner cases related to T, B, and A_{mid} describing bits beyond the top of a virtual address. Our final fast *representability* check composes these three tests:

$$representable = (inRange \text{ and } inLimit) \text{ or } (E \geqslant 50)$$

To summarize, the representability check depends only on four 14-bit fields, T, B, A_{mid}, and I_{mid}, and the sign of i. Only I_{mid} must be extracted during execute, as A_{mid} is cached in our register file. This fast representability check allows us to perform pointer arithmetic on compressed capabilities directly, avoiding decompressing capabilities in the register file that introduces both a dramatically enlarged register file and substantial load-to-use delay.

CHERI Concentrate 64-bit format for 32-bit address spaces

In this section, we describe how CHERI Concentrate compresses the bounds used in 64-bit capabilities with 32-bit architectural addresses.

Figure 3.3 shows the capability format and decoding method for 64-bit CHERI concentrate. The format contains a 32-bit address, a, 12 hardware defined permission bits, a flag bit, a 4-bit object type and 15 bits that encode the bounds relative to the address.

3.5.5 Capability Address and Length Rounding Instructions

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 parametrization would be beneficial. We propose two new instructions that allow the architecture to provide information to memory allocators regarding precision effects:

CRepresentableAlignmentMask (CRAM) `CRAM` accepts a proposed bounds length, and returns a mask suitable for use in aligning down the address of an allocation.

CRoundRepresentableLength (CRRL) `CRRL` 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

31							0
p'12	f	otype'4	I_E	$T[5:3]$	T_E'3	$B[7:3]$	B_E'3
a'32							

f: flag p: permissions otype: object type a: pointer address

If $I_E = 0$:

$$\begin{aligned} E &= 0 \\ T[2:0] &= T_\mathrm{E} \\ B[2:0] &= B_\mathrm{E} \\ L_\text{carry\_out} &= \begin{cases} 1, & \text{if } T[5:0] < B[5:0] \\ 0, & \text{otherwise} \end{cases} \\ L_\text{msb} &= 0 \end{aligned}$$

If $I_E = 1$:

$$\begin{aligned} E &= \{T_\mathrm{E}, B_\mathrm{E}\} \\ T[2:0] &= 0 \\ B[2:0] &= 0 \\ L_\text{carry\_out} &= \begin{cases} 1, & \text{if } T[5:3] < B[5:3] \\ 0, & \text{otherwise} \end{cases} \\ L_\text{msb} &= 1 \end{aligned}$$

Reconstituting the top two bits of T:

$$T[7:6] = B[7:6] + L_\text{carry\_out} + L_\text{msb}$$

Decoding the bounds:

address, $a =$	$a_\text{top} = a[31 : E+8]$	$a_\text{mid} = a[E+7 : E]$	$a_\text{low} = a[E-1 : 0]$
top, $t =$	$a_\text{top} + c_\text{t}$	$T[7:0]$	0'E
base, $b =$	$a_\text{top} + c_\text{b}$	$B[7:0]$	0'E

To calculate corrections c_t and c_b:

$$\begin{aligned} A_3 &= a[E+7 : E+5] \\ B_3 &= B[7:5] \\ T_3 &= T[7:5] \\ R &= B_3 - 1 \end{aligned}$$

$A_3 < R$	$T_3 < R$	c_t
false	false	0
false	true	$+1$
true	false	-1
true	true	0

$A_3 < R$	$B_3 < R$	c_b
false	false	0
false	true	$+1$
true	false	-1
true	true	0

Figure 3.3: CHERI Concentrate 64-bit capability format and decoding

to be well suited for use with dynamic stack allocation – e.g., using `alloca`, but also other types of allocation.

3.5.6 32-bit Modes on 64-bit Architectures

We currently consider 32-bit execution modes on 64-bit processors to be legacy compatibility modes, and hence do not define capability instruction-set extensions for those modes. The essential design goal is therefore to maintain CHERI's security properties while enabling the execution of 32-bit capability-unaware code.

Our recommendation is that capability-aware instructions be inaccessible in 32-bit modes, and that it be impossible for executing code to introduce capability values that violate provenance validity and monotonicity properties. Any writes of non-capability values into capability-extended general-purpose registers should be treated in a similar manner to integer writes into those registers in the capability-aware execution environment: they should clear the remainder of the register including the tag bit. Writes of integer values into capability-extended special-purpose registers will need similar handling to 64-bit writes: in some cases they should clear the tag, and in other cases they should modify the offset being accessed, in a manner similar to changes to **PC**, and so on.

While this appears to be a coherent design direction, we have not validated this approach in architecture.

3.6 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 C.4.

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.

3.6.1 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^{XLEN}.
- **otype** = $2^{\mathsf{XLEN}} - 1$ (truncated as required by the implementation's encoding).
- All available permission bits are set; other bits will be returned as zero architecturally.
- Concrete architectures specify the reset value of **flags** for root capabilities.
- All unused bits are cleared.

Capability registers not intended to act as roots will 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^{XLEN}.
- **otype** = $2^{\mathsf{XLEN}} - 1$ (truncated as required by the implementation's encoding).
- All available permission bits are unset.

- **flags** = 0x0.
- All unused bits are cleared.

3.6.2 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. This property is often already enforced by real-world hardware and systems – whether due to Error-Correcting Codes (ECC),[4] 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.

3.7 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.

Where possible, the structure and semantics of capability instructions have been aligned with similar core 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:

---

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

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

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

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: `CAndPerm`, `CClearTag`, `CIncOffset`, `CIncOffsetImm`, `CSetAddr`, `CSetBounds`, `CSetBoundsExact`, `CSetBoundsImm`, `CSetFlags`, `CSetHigh`, and `CSetOffset`.

Capability pointer comparison These instructions provide pointer comparison: `CSetEqualExact` and `CTestSubset`.

Load or store via a capability These instructions access memory 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. CHERI-RISC-V adds `CLC` and `CSC` to load and store capabilities as well as a new instruction decoding mode in which existing memory access instructions use capability registers as the base address instead of integer registers. CHERI-RISC-V also adds new instructions which explicitly use a capability register as the base address regardless of decoding mode including `L[BHWD][U].CAP`, `LC.CAP`, `S[BHWD][U].CAP`, and `SC.CAP`.

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.

Additionally, the `CLoadTags` instruction provides direct, *read-only* access to capability tags; see Section 9.26.

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). CHERI-RISC-V extends the base `AUIPC` instruction with `AUIPCC` that adds an offset to **PCC**.

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: `JALR.CAP` and `CJALR`.

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. The `CSealEntry` instruction constructs *hardware-interpreted* sealed entry ('sentry') capabilities; see Section 3.9.

Protection-domain switching The `CInvoke` instruction is a primitive upon which protection-domain switching can be implemented. `CInvoke` has a jump-based semantic that unseals its sealed code and data capability-register operands. This allows software-controlled non-monotonicity by granting access to additional state via unsealing.

Fast register clear The `CClear` and `FPClear` instructions clear a range of capability or floating-point registers to support fast protection-domain transition.

Special capability registers Special capability registers are read and written via `CSpecialRW`.

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 sequences are 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`.

Capability Address and Length Rounding Instructions Capability compression requires stronger alignment as allocation sizes increase as described in Section 3.5.5. `CRAM` and `CRRL` can be used by allocators to enforce non-overlapping bounds for distinct allocations.

3.8 Protection-Domain Transition with CInvoke

Cross-domain procedure calls are implemented using the `CInvoke` 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. `CInvoke` 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.

`CInvoke` unseals the sealed code and data capabilities and places them in **PCC** and **IDC** (an architecture-specific capability register), 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.

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, if the OS the clears non-argument registers on call, and non-return registers on return, caller and callee can avoid clearing those registers allowing that clearing to occur exactly once. Efficient register clearing instructions (e.g., `CClear`) 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. 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.

3.9 Sealed Entry Capabilities

CHERI borrows from earlier capability architectures a notion of immutable capabilities that are usable solely as jump targets, most notably the M-machine [24], where these are called "*enter* capabilities." These reside somewhere between CHERI's unsealed and sealed PERMIT_EXECUTE-bearing capabilities. Because they act in tandem with CHERI's sealing mechanism and describe function entry points, we use the name 'sealed entry' capability or just 'sentry,' for short. Similar to sealed capabilities, sentry capabilities are immutable by their bearer and do not authorize memory loads or stores. Like unsealed capabilities, the bearer may directly jump to the sentry to begin executing the instructions it references. The jump instruction atomically unseals the sentry and installs it to the program counter capability register. In our implementations, we use the same instruction (e.g., `CJR` or `CJALR`) to vector control through either unsealed or sentry capabilities, so that code can be oblivious to whether it is jumping through an ordinary code capability or a sentry. One could, of course, imagine instructions that enforced the type of their operand.

Since userspace function pointers are often passed to kernels for use in callbacks, such as signal handlers, performing an exception return (`ERET` on ARMv8-A, *x*`RET` on RISC-V and `IRET` on x86) also atomically unseals the implicit jump target when installing it to **PCC** just like a normal capability jump instruction, rather than forcing the kernel to re-derive an unsealed capability for the same function. However, due to the need for kernels to perform actions such as emulating unaligned accesses or unimplemented instructions, and thus manually increment the application's **PCC**, exception handlers may need to return via the original unsealed **PCC** rather than creating a sentry and similarly forcing the kernel to re-derive the unsealed capability. In addition to eliminating unnecessary work, reducing the need for kernels to unseal or re-derive sentry capabilities in software provides a security benefit by reducing the authority present in userspace-facing code paths.

Creating sentry capabilities is taken to be an ambient monotonic action, requiring no additional permission than to have a capability bearing PERMIT_EXECUTE. The `CSealEntry` instruction derives a sentry capability from any PERMIT_EXECUTE-bearing capability, otherwise preserving permissions, bounds, and cursor. Sentry capabilities have **otype** of $2^{\mathsf{XLEN}} - 2$ (truncated as required by the implementation; recall Table 3.2) but are not intended to be unsealable within general system software[5] except by entry of control flow.[6]

CHERI-RISC-V creates sentry capabilities whenever it stores the **PCC** to a link register, as in `CJALR`. This behavior furthers our adherence to the principle of least privilege and reduces the number of "gadgets" available to adversarial code.

Because the full, unsealed sentry is installed as the program counter, PCC-relative addressing permits the invoked instructions to use authority beyond PERMIT_EXECUTE. We exhibit some examples of such usage below.

3.9.1 Per-Library Globals Pointers

Sentry capabilities are useful for multiply instantiated objects (e.g., shared libraries), as schematically shown in Figure 3.4. In this scenario, 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`.

---

[5] While it would be ideal if the permission to unseal **otype** $2^{\mathsf{XLEN}} - 2$ (and $2^{\mathsf{XLEN}} - 1$) were excluded from the primordial capability set, instead, early boot code can enforce this when it partitions its boot capabilities into the provenance roots it uses in the steady state.

[6] Of course, one could create a 'self-unsealing enter capability' that transferred PCC to the return value (capability) 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.

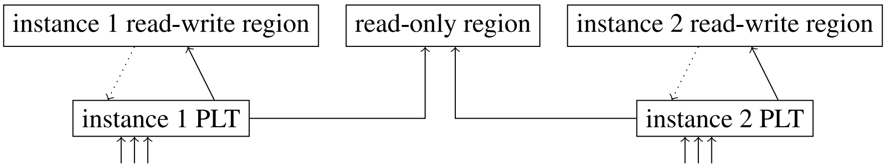

Figure 3.4: PLT-style multiple instantiation showing capability reachability. The RO region is referenced with a subset of execute and load (data and capability) permissions by the PLT. The PLT references its corresponding RW region with any desired set of permissions. The PLT is referenced using sentry 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.

In order to achieve the desired 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 sentry capabilities is ideal: the PLT may contain the capability to the state, and the sentry can authorize its (PCC-relative) load once it has been entered, yet the user can neither fetch nor manipulate capabilities through the sentry 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 sentry 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 distrusting 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 beyond sentry capabilities, namely, just PCC-relative loads of 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 sentry-guarded library instance may engage in non-stack-discipline control flow and skip its return.

3.9.2 Environment Calls via Sentry Capabilities

Sentry capabilities are also useful for sandboxing. While sandboxed code can be made to look like a library to the caller, a more interesting observation is that the reverse is also possible and that sentry capabilities are also viable for calls *from* the sandbox back to a single-threaded supervisor environment. On sandbox construction, the supervisor allocates space for its state closure (a `longjmp` buffer and 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 a sentry 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 sentry capability from one thread within another thread's execution. The trampoline code for invoking or returning to the supervisor environment will, ultimately, involve asking the *supervisor environment's supervisor* 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 sentry capability itself must bear PERMIT_ACCESS_SYSTEM_REGISTERS or have access to another capability that does).

3.10 Handling Failures

Instruction-set architectures have various recourses 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. Throwing an exception prior to writing the new value to **PCC**, on the other hand, provides more complete debugging information: the errant jump **PCC** 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-A), 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.

However, in all current architectures, we 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 invalid 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-A); 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. In addition, stripping the tag avoids encouraging implementations that are vulnerable to speculative side channel attacks.

When using tag stripping in ISAs with status registers (e.g., ARMv8-A), 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.

3.11 Composing Architectural Capabilities with 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.

As CHERI is applied to an ISA, various low-level design choices must be made including the storage of capabilities in registers, opcode encoding, and MMU changes. These choices are generally specific to each ISA – 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 two ISAs: 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 both, 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.

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.

3.11.1 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, two 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 [73], 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.

3.11.2 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.

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.

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 in architecture-specific state (for example, in a special register). 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 an architecture-specific manner. 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 must save a copy of **PCC** and not just **PC**. 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-A, the stack pointer register is banked, allowing exception handlers to use their own stack pointer to save remaining registers. In x86-64, the stack pointer register is potentially replaced and the original stack pointer is saved on the exception stack. 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-A stack pointer and x86-64 exception stack pointers would need to be widened to full capabilities.

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 extensions are required to provide a suitable **PCC** and exception data capability to provide additional rights to the exception handler authorizing its execution. CHERI-RISC-V provides this by extending the *x*`tvec` and *x*`scratch` special registers and adding the *x*`tdc` special capability registers.

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**, 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 **PCC**.

Capability Exception Causes

In each of the target ISAs (RISC-V and x86-64), we introduce a new exception to report capability violations. Since this exception covers a variety of error cases, each CHERI ISA must provide an architecture-specific capability exception code which indicates the specific violation.

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.

3.11.3 Virtual Memory

Where virtual memory is present and enabled, CHERI capabilities are interpreted with respect to the current virtual address space. In architectures, such as CHERI-RISC-V, where virtual-address translation can be enabled or disabled dynamically, the embedded address in a capability 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 themselves, 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

Modern Memory Management Units (MMUs) support architectural page tables. A series of instructions or control registers configures parameters such as the page-table format being used and 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 or access 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. Exceptions may fire if operations are rejected as a result of page permission checks (e.g., an 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, and flushing the TLB. The ACCESS_SYSTEM_REGISTERS permission must be present on **PCC** to perform these operations.

- 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.

MMU Capability Permissions

Virtual-address translation is itself unmodified, but permission checking is extended with new page permissions in the MMU mappings (i.e., PTEs):

MMU Load Capability Permission If this permission is present, as well as the existing page-table read permission, then loading tagged capabilities is permitted. If this permission is not present, architectures may either trap or clear the tag bit of the loaded capability.

If the PERMIT_LOAD_CAPABILITY permission is not present on the authorizing capability for the memory read, then the tag is cleared from the loaded capability, and this page permission is ignored.

If an exception is raised, the exception should resemble other MMU exceptions for the architecture. In particular, the virtual address of the attempted memory access should be provided by the exception in a similar manner to other MMU exceptions.

As both trapping and tag-clearing semantics might be useful in different circumstances, architectures may use a separate bit in the MMU mapping to indicate which behavior is requested. It is possible to emulate the tag-clearing semantics given only the trapping semantics, at the cost of efficiency; if trapping semantics are desired but the architecture only permits a single semantic (e.g. due to limited MMU mapping bits), we suggest providing only the trapping behavior.

While traps would ideally only be raised if a *set* tag were returned from memory, this would be a *data-dependent trap*, a potentially uncomfortable proposition for high-performance microarchitectures. Instead, we believe it permissible for a (micro)architecture to always trap on capability load instructions fetching through MMU mappings without this permission. System software could minimize spurious traps for capability-oblivious code such as memory copies by using capabilities without the PERMIT_LOAD_CAPABILITY permission for mappings lacking this permission.

MMU 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.

If an exception is raised, the exception should resemble other MMU exceptions for the architecture. In particular, the virtual address of the attempted memory access should be provided by the exception in a similar manner to other MMU exceptions.

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.

Per-Page Capability Load Barriers

Garbage collectors and capability revocation, in addition to lazily tracking capability flow through stores, would like to be able to catch (attempted) capability transfers on loads from memory. Such software could avail itself of the trapping variant of the page-table load capability permission, in which loading a (tagged) capability *trapped*.

Across an application's lifetime, its address space may need to be repeatedly scanned. That is, its capability-bearing pages may need to transition in tandem between permitting capability loads and trapping thereupon to cue the collector or revoker's inspection of the source page before restoring that page to permitting capability loads. While it is certainly possible to update all MMU mappings to toggle between the two, this operation would take linear time and touch linear memory, likely with all application threads paused. We instead envision a parallel constant-time operation for the bulk update, achieved by equipping each *CPU core* and all MMU mappings with one-bit "generation counters" delimiting per-address-space "epochs." Prior to the start of an epoch, all generation counters within the address space's page tables and actively associated cores are equal. The beginning of an epoch is signaled by all cores incrementing their generation (synchronously, from the perspective of the application), and the epoch comes to an end when all MMU mappings' generation counters once again equal this incremented value.

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`. `CLoadTags` allows efficient gathering of tag data from full cache lines, and has 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.

3.11.4 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 corrupting 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.

3.11.5 Caching and Explicit Prefetch

Some architectures have explicit prefetch instructions that give a hint to the CPU that data at a particular virtual address might be used in the near future. For performance reasons, these prefetch instructions do not raise MMU exceptions. This allows a highly parallel CPU implementation to start executing the next instruction in the program without waiting for the TLB check to complete. (Imagine, for example, that a prefetch is followed by a store to a different address). For similar performance reasons, with CHERI a **DDC**-relative prefetch instruction may fail without raising an exception if the address is outside the range of **DDC**. On the other hand, there is a potential covert timing channel if programs are allowed to prefetch memory addresses to which they do not have access. If a prefetch to the out of bounds address changed the contents of the memory caches, then another subprogram (one that did have a capability granting access to the memory address) could test whether or not the prefetch had happened by doing a load and timing how long it took. The compromise between performance and security is that prefetch of an out of bounds address does not raise an exception, but also does not change the memory caches or (in multicore CPUs) affect the behavior of other cores; it acts like a no-op.

Prefetch will typically fetch an entire cache line, not just the address that has been explicitly prefetched. The question then arises as to what happens if **DDC** grants access to the address that is explicitly prefetched, but not all of the cache line. As prefetch does not raise an exception on failure, it can silently fail in this case without affecting program correctness (though there will be a performance penalty). If covert channels via memory caches are a concern, subsystems that are intended to be isolated should not share cache lines: ordinary loads, and load-linked/store conditional also provide opportunities for covert channels via caches. Rounding up a protected subsystem's memory region to a cache line boundary will mean that the reduced performance case where part of the cache line is outside the range of **DDC** will not be encountered.

To allow explicit prefetch in pure capability mode, a prefetch via capability instruction may also be added. The security and performance trade-offs are the same as for prefetch relative to **DDC**: an out of bounds prefetch can fail silently without raising an exception, as long as it does not perturb the memory caches.

3.12 Implications for Software Models and Code Generation

3.12.1 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 Native 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.

3.12.2 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 `CInvoke` instruction.

In traditional capability designs, invocation of an object capability triggered microcode responsible for state management. Initially, we implemented a pair of `CCall` and `CReturn` instructions via software exception handlers in the kernel, but have since refined this model to a single `CInvoke` which performs a jump-like operation to minimize 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.[7] 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 fragments 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.

3.13 Deep Versus Surface Design Choices

In adapting an ISA to implement the CHERI protection model, we find it useful to contrast between two types of changes:

Deep design choices include the decision to expose capability use and management for explicit use by the compiler, employing tagged memory to protect capability values, enforcing monotonicity using limitations on the instruction set, preventing capability use if its valid provenance has been violated, and introducing (or extending) registers (including control registers such as the Program Counter) to hold capability values.

Surface design choices reflect to the specific possible integrations with the target ISA, including the specific blend of instructions and their encodings, whether the address embedded

---

[7] 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.

in a capability is physical or virtual, how to extend existing registers to hold capability values, and the specific number (or mix) of capability registers.

Further, applications to an ISA are necessarily sensitive to existing choices in the ISA – for example, 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 virtual memory, or with virtual memory disabled).
- Tags on registers or in memory determine whether they are valid capabilities for loading, fetching, or jumping to;
- Tagged registers can contain both data and capabilities, allowing (for example) capability-oblivious memory copies;
- Tags on capability-sized, capability-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 (rather than a valid capability) into memory that has one or more valid tags will atomically clear the tags on any affected memory;
- 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;
- Capabilities contain bounds and permissions; a capability's address is able to float freely within (and to varying extents, beyond) the bounds;
- Permissions control both data and control-flow operations;
- Guarded manipulation in the architecture (and, implicitly, microarchitecture) implements monotonicity: rights can be reduced but not increased through valid manipulations of capabilities;
- Invalid manipulations of capabilities 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 – for jumps, this could be on the jump instruction, or on instruction fetch at the target;
- 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) via extensions to the page-table entries for hardware-managed TLBs, contains additional permissions controlling the loading and storing of capabilities;
- That MMU-enforced permissions may clear tags or throw exceptions if violated (possibly as configurable option);
- That operations violating guarded manipulation clear the tag and yield a later exception on use, rather than triggering an immediate exception;
- 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 in special registers allow a software supervisor to operate with greater rights than non-supervisor code, recovering those rights on exception delivery;
- A capability flow-control model to allow the propagation of capabilities to be constrained, preventing capabilities marked as local from being stored via capabilities marked to prevent that;
- Sealed capabilities allow a non-monotonic escalation of privilege associated with a constrained control-flow transition to a defined address. Subject to the use of suitable instructions, and appropriate permissions, a pair of sealed capabilities with identical object types allow access to unsealed versions of the capabilities, with code beginning execution at one of them. This enables software-enabled behaviors such as software compartmentalization.
- Sealed entry capabilities likewise allow non-monotonic escalation of privilege associated with a constrained control-flow transition to a defined address. Subject to use of suitable instructions, and appropriate permissions, a single sealed entry (sentry) capability allows code to begin execution via an unsealed version of the same capability.
- 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, addresses can stray well out-of-bounds without becoming unrepresentable;
- For compressed capabilities, alignment requirements do not restrict common object sizes and do not overly 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 integrations of the CHERI model into the 32/64-bit RISC-V and x86-64 ISAs, and might be revisited in various forms in integrating CHERI support into these or other ISAs:

- 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;
- 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
- The specific parameter choices in capability values, including the number of dereferenceable bits in the address, the investment of bits in bounds-related fields (such as the exponent size), the size of the object-type field, the number of software-defined permissions, and also the specific in-memory layout.
- 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.

3.14 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`, and `CGetLen`. However, there is also a desire that `CSetOffset` should preserve the values of T and B for debugging purposes, where possible.

- 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 C.10 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 `CInvoke` 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.

- 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**.

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

- 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, "non-manipulable") 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 C.6 describes an experimental proposal for how this might be implemented.

- Today, a uniform set of capability roots are provided: **PCC**, **DDC**, 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., MMU configuration) 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-RISC-V 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 architecture: 32-bit and 64-bit RISC-V. Wherever possible, CHERI-RISC-V implements the architecture-neutral concepts described in Chapter 3. Detailed descriptions of specific capability-aware instructions can be found in Chapter 7.

4.1 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.

4.2 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).

- 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 reflected pragmatic design choices made prior to the development of full compiler and operating-system stacks, and were 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.

4.2.1 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 supports 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 RVG, 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 the supervisor-level and machine-level ISAs defined in the privileged portion of the ISA.

We view 64-bit CHERI-RISC-V as a mature specification suitable as a starting point for an official RISC-V extension. However, we feel that 32-bit CHERI-RISC-V is less mature. In particular, the current encoding for 64-bit capabilities provides insufficient precision. Further research is needed to determine if an alternate encoding, perhaps using an alternate scheme for permissions, can provide better precision.

4.2.2 CHERI-RISC-V Strategy

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.

4.2.3 Common Architectural Features

CHERI-RISC-V shares the following features with other CHERI architectures:

- Tagged memory with capability-width tag granularity and alignment.
- Registers able to hold capabilities are tagged.
- **PCC** constrains program-counter-relative fetches.
- **DDC** constrains legacy RISC-V load-store instructions.
- Floating point is fully supported, including capability-relative floating-point load and store instructions.
- General-purpose registers are extended to hold capabilities.
- Capability-related violations (such as loads/stores/fetches via untagged capabilities, out-of-bound accesses, and so on) trigger immediate precise exceptions.
- Requests for non-monotonic capability transformations result in the tag of the written back value being stripped.
- 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 reinforces intentionality.
- PERMIT_ACCESS_SYSTEM_REGISTERS limits privileged ISA operations when within privileged rings. While RISC-V's specific privileged operations differ, the intent remains the same: to allow code compartmentalization within the privileged ring.

4.2.4 Unique Architectural Features

The following changes are specific to CHERI-RISC-V:

- RISC-V exception handling – including register banking, scratch registers, and cause mechanism – is used.
- A new exception code, 0x1A, is reported in the RISC-V *x*`cause` CSRs when a load attempts to fetch a capability through a valid page table entry granting read permission but forbidding loads of capabilities. This fault otherwise behaves like a RISC-V load page fault.
- A new exception code, 0x1B, is reported in the RISC-V *x*`cause` CSRs when a store attempts to write a capability through a valid page table entry granting write permission but forbidding stores of capabilities. This fault otherwise behaves like a RISC-V store/AMO page fault.

- A new exception code, 0x1C, is reported in the RISC-V *x*`cause` CSRs when other capability-related exceptions (such as tag violations) occur. Additional capability-specific exception cause information, such as more specific cause information and the identity of the faulting register is reported in the existing *x*`tval` CSRs (see Section 4.3.11).
- A new bit is defined in `menvcfg` and `senvcfg` to enable CHERI support.
- New per-mode capability CSRs are added as *x*`ccsr` (see Section 4.3.5).
- CHERI-related page permissions are added to RISC-V architectural page-table formats.
- The interpretation of addresses in memory capabilities depends on whether virtual addressing is enabled via the RISC-V `satp` CSR[1]. 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 are supported (albeit not dynamically). In the future, it may be desirable to also support XLEN=128.
- A rich set of atomic instructions is extended with capability support.
- 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`, and jumps to be interpreted as expecting capability rather than integer operands (reducing opcode footprint while maintaining intentionality).
- In the compressed RISC-V encoding, the capability encoding mode allows existing load, store, jump, and stack addressing opcodes to be interpreted as expecting capability rather than integer operands.

4.3 CHERI-RISC-V Specification

In this section, we describe in greater detail the integration of CHERI into the RISC-V instruction set. Instruction opcode encodings can be found in Appendix B.

4.3.1 CHERI as a non-standard RISC-V extension

CHERI is integrated into the RISC-V ISA as a non-standard extension named Xcheri, and follows the idioms for RISC-V extensions to the extent possible. In the extension terminology of the RISC-V specification, CHERI is mostly a *greenfield* extension since it adds most new

---

[1] This is not a substantially different design choice than in other architectures: 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.

instructions by populating a new instruction encoding space. The prefix used for the encoding is currently “1011011”, placing it in the *custom-2/rv128* opcode space that the specification allows for use for custom instruction set extensions on RV64; this makes it a standard-compatible global encoding. However, we also propose a few new instructions in existing encoding ranges. The new instructions to load and store capabilities are *brownfield* extensions to the LOAD and STORE opcodes in the base integer ISAs. In addition, CHERI adds new atomic operation instructions which are *brownfield* extensions to the AMO opcode.

A CHERI-RISC-V processor has the X bit of the `misa` register hardwired to 1 on boot to indicate the presence of a non-standard extension. Information tying this set X bit to the Xcheri extension would be communicated to system software in a platform-specific manner.

CHERI-RISC-V is currently defined as a non-standard extension to version 2.2 of the RISC-V userspace ISA [147] and version 1.11 of the RISC-V privileged ISA [148].

4.3.2 Tagged Capabilities and Memory

CHERI-RISC-V allows 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, supports the implementation of capability-oblivious memory copy operations, and maintains strong C-language pointer compatibility for capabilities. This implies the use of tagged memory consisting of 1-bit tags protecting capability-aligned, capability-sized words of memory implemented with suitable protection and atomicity properties.

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.

4.3.3 Capability Register File

In CHERI-RISC-V, general-purpose integer registers are extended to optionally hold full capabilities, along with a tag.

Extending general-purpose integer registers raises the question of whether and how capability-unaware 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.

The bottom `XLEN` bits of the register contain the integer interpretation (which, for a capability, will be its address), and the top `XLEN` bits (plus additional tag bit) 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, the new integer value is stored in its bottom `XLEN` bits, and the top `XLEN` bits and tag bit are 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` is 64 bits, and for 64-bit CHERI-RISC-V, `CLEN` is 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 reuse these opcode reservations and encodings to load 64-bit `CLEN` words as well as their tag bit. Similarly, when `XLEN` is 64, we use the anticipated 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.

4.3.4 Capability-Aware Instructions

In CHERI-RISC-V, two general categories of instructions are added: those that query or manipulate capability fields within registers, and those that utilize capability registers for the purposes of load, store, or jump operations.

Register-to-register instructions querying and manipulating fields allow integer values to be moved in and out of portions of an in-register capability, subject to guarded manipulation. They are simply new instructions defined in CHERI-RISC-V and added to the opcode space.

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.

4.3.5 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 4.1. New Special Capability Registers (SCRs), accessed via new CSR-like instructions, are described in Section 4.3.6.

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 4.1: Control and Status Registers (CSRs)

Controlling Access to CSRs

Accessing some 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). We adopt a whitelist approach: reading or writing any CSR requires the permission, with the exceptions listed in Table 4.2.

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

Table 4.2: CSR Whitelist. The accesses shown are the only CSR accesses that are permitted when the installed PCC does not have Permit_Access_System_Registers.

CHERI Extension Control

A new bit in the `menvcfg` and `senvcfg` CSRs enables CHERI for lower privilege levels. When CHERI is disabled, attempting to execute CHERI-specific instructions raises an illegal instruction fault, including loads and stores which use a capability register (excluding the implicit **DDC** operand for legacy loads/stores) as the memory operand.

Other CHERI extensions are always enabled regardless of the state of this bit. Specifically, bounds and permissions on **PCC** and **DDC** are always honored. Exceptions always copy **PCC** to *x***EPCC** on exception entry and restore the full **PCC** on exception return. Capability mode is always honored if enabled in **PCC**. Software which disables CHERI in lower modes must take care to ensure that **PCC** and **DDC** are set to suitable values while lower modes execute.

Bit 28 (`0x1C`) in the `menvcfg` and `senvcfg` CSRs is defined as the CHERI enable bit. Its allocation within these CSRs may change until CHERI is ratified as a RISC-V extension.

Capability Control and Status Registers (CCSRs)

New per HART *x*`ccsr` `XLEN`-bit RISC-V CSRs are defined as per Figure 4.1 (shown for XLEN=32):

Figure 4.1: *x*`ccsr` register format; WPRI bits are Write Preserve Read Ignore.

nr The `nr` "no relocation" read-only bit indicates if integer memory addresses are relocated by the base address of **DDC** and **PCC**. If this bit is 1, then integer addresses are not relocated.

tc The `tc` "tag-clearing" read-only bit indicates if attempts to update a capability non-monotonically clear the resulting capability's tag (1) rather than raising a 0x1C exception (0).

An implementation compliant with the current version of this specification sets the `nr` and `tc` bits to 1 and all other bits to 0 at reset.

4.3.6 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 is the zero register, 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 is the zero register, 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 4.3 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 3.6. Whether a

	Register	**Modes**	**Access**	**Reset**	**Extends**
0	Program counter capability (**PCC**)	U, S, M	RO	∞	**PC**
1	Default data capability (**DDC**)	U, S, M	-	∞	-
4	User trap code capability (**UTCC**)	U, S, M	ASR	∞	`utvec`
5	User trap data capability (**UTDC**)	U, S, M	ASR	$\emptyset$	-
6	User scratch capability (**UScratchC**)	U, S, M	ASR	$\emptyset$	`uscratch`
7	User exception PC capability (**UEPCC**)	U, S, M	ASR	∞	`uepc`
12	Supervisor trap code capability (**STCC**)	S, M	ASR	∞	`stvec`
13	Supervisor trap data capability (**STDC**)	S, M	ASR	$\emptyset$	-
14	Supervisor scratch capability (**SScratchC**)	S, M	ASR	$\emptyset$	`sscratch`
15	Supervisor exception PC capability (**SEPCC**)	S, M	ASR	∞	`sepc`
28	Machine trap code capability (**MTCC**)	M	ASR	∞	`mtvec`
29	Machine trap data capability (**MTDC**)	M	ASR	$\emptyset$	-
30	Machine scratch capability (**MScratchC**)	M	ASR	$\emptyset$	`mscratch`
31	Machine exception PC capability (**MEPCC**)	M	ASR	∞	`mepc`

Table 4.3: Special Capability Registers (SCRs). SCRs 4-7 are available only with the N extension, and 12-15 only with supervisor mode. **Modes** shows which RISC-V privilege modes are allowed to access the registers. **Access** indicates additional restrictions on accessing the registers: **PCC** is read-only via `CSpecialRW`, but is set by `CJALR` and during exceptions; *ASR* indicates **PCC**.**perms** must grant Permit_Access_System_Registers to permit access (in addition to being in a permitted mode). **Reset** indicates whether the register is initialised to the default root capability (∞) or NULL capability ($\emptyset$) on reset. Some special capabilities registers are extensions of existing RISC-V registers, with the capability address being equal to the original register.

register is initialized to NULL or the omnipotent capability, its flags field is initialized to zero (specifying integer encoding mode).

Where an SCR extends a RISC-V CSR, e.g. **MTCC** extending `mtvec`, any read to the CSR shall return the address of the corresponding SCR. Similarly, any write to the CSR shall set the address of the SCR to the value written. This shall be equivalent to a `CSetAddr` instruction. This allows sealed capabilities to be held in SCRs without allowing them to be modified in a tag-preserving way. Some RISC-V CSRs have write ignore bits, or otherwise implicitly modify the written value to restrict the CSR to legal values. These modifications must be applied to the SCR's new address when writing a CSR extended by an SCR, or to the address of the newly written capability when using `CSpecialRW`. `CSpecialRW` of a sealed capability to an SCR which extends a CSR with any non-preserved bits clears the tag on the capability, even if the address would not be changed. As per the rest of the RISC-V specification, should the SCR become unrepresentable as a result of the address being set, the tag is cleared but the resulting address and the encoded capability metadata are preserved.

4.3.7 Capability Encoding Mode

RISC-V instructions that interpret arguments or results as addresses (e.g. loads, stores, jumps, `AUIPC`) can either act on integer pointers or on explicit capabilities. For example, capability-relative load and store instructions accept (and expect) capability operands that 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.

To avoid occupying large chunks of remaining encoding space by supplementing each address-manipulating instruction with a corresponding capability-relative version, we introduce a new *capability encoding mode* in which some existing RISC-V opcodes are reused for capability-relative accesses. The encoding mode is 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, store, jump, and `AUIPC` opcodes contain *integer addresses*. The upper `XLEN` bits and tag bit of the operand register are ignored. For loads and stores, 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, store, jump, and `AUIPCC` opcodes contain *capabilities*. For loads and stores, 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.

To maintain intentionality, this approach is never ambiguous in either mode as to whether memory accesses are relative to an integer or capability operand: address operands of existing RISC-V opcodes are always integer relative in integer encoding mode, and always capability relative in capability encoding mode.

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 *x***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.

In addition, a small set of both capability-relative and integer-relative loads, stores, and jumps are added, tuned to limit opcode space utilization – e.g., by having small or no immediates – at the cost of increased code footprint. These instructions are available in both encoding modes to permit alternate memory accesses.

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.

Non-Compressed Instructions Affected by Capability Encoding Mode

The following non-compressed RISC-V instructions are affected by the capability encoding-mode bit (see the following section for further details on compressed instructions):

Integer load	LB	LH	LW	LD	LQ
Integer load (unsigned)	LBU	LHU	LWU	LDU	
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	
Control flow	JAL	JALR			
Address calculation	AUIPC[2]				

4.3.8 Compressed Instructions

While the compressed instruction extension is not mandatory for RV32G and RV64G, it is widely used in existing RISC-V software to improve code density.

Given the tight encoding space for compressed instructions, it is not practical to support both integer and capability variations of common instructions. Instead, the encoding mode must be used to alter the interpretation of existing instructions to support optimal code density. Two problems arise in adding compressed instruction support for capabilities: determining which existing instructions should use capability semantics and the need to add new instructions to load and store capabilities.

For some instructions, the choice to use capability semantics in capability encoding mode is straightforward. Compressed loads and stores should use capability-relative addresses just as for non-compressed instructions. Similarly, compressed jump instructions should use capability registers for jump target registers and link registers. Finally, instructions which use the stack pointer as an implicit operand should use **CSP** as the stack pointer. This includes loads and stores as well as the stack addressing instructions `C.ADD16SP` and `C.ADDI4SPN`.

For other instructions, the decision is less clear. Pure capability code will use both `MV` and `CMove` instructions as well as both `ADDI` and `CIncOffsetImm`. Our current approach for these types of instructions has been to not alter their semantics in capability encoding mode until more research can be done to determine the most common semantics in pure capability code.

Adding compressed instructions to load and store capabilities requires repurposing some existing opcodes. For this case we follow the pattern used by RV64C and RV128C of repurposing compressed floating point loads and stores to load and store capabilities. Just as RV64C reuses the floating-point single-precision loads and stores (e.g. `C.FLW` and `C.FSW`) for 64-bit integer loads and stores (`C.LD` and `C.SD`), compressed CHERI-RV32 reuses these opcodes in capability-encoding mode for capability loads and stores (`C.CLC` and `C.CSC`). Similarly, CHERI-RV64 reuses the opcodes for floating-point double loads and stores (`C.FLD` and `C.FSD`) in capability-encoding mode for capability loads and stores. The same rules apply to stack-relative memory access instructions.

---

[2] See Section 4.3.14.

When a compressed instruction in capability encoding mode encodes a capability register operand using a 3-bit field rather than a 4-bit field, the selected capability register is one of `c8`–`c15` using the same mapping used for `x8`–`x15` in integer encoding mode.

Compressed Instructions Affected by Capability Encoding Mode

The following compressed instructions are affected by capability encoding mode:

Stack addressing	C.ADDI4SPN	C.ADDI16SP		
Control flow	C.JAL	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

4.3.9 Floating Point

The vast majority of floating-point instructions are not impacted by the presence of CHERI-RISC-V. Existing RISC-V floating-point load and store instructions use capability-relative addresses in capability encoding mode, and integer-relative addresses constrained by **DDC** in integer encoding mode.

The floating point control registers (`fcsr`, `frm`, and `fflags`) are whitelisted in Table 4.2 so they can be accessed without needing PERMIT_ACCESS_SYSTEM_REGISTERS.

4.3.10 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, when a capability exception is raised, *x*`tval` provides details about the exception as described in Section 4.3.11.

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 (extends `mscratch`)

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 (extends `sscratch`)

Exceptions to User Mode

When present, we extend the "N" extension (for "User-Level Interrupts") 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 `utvec`)
- **UScratchC** - User Mode Scratch Capability (extends `uscratch`)

This extension could be leveraged for user-space-only implementations of `CInvoke`, as well as routing specific interrupts from suitable devices to user-level compartments for handling by sandboxed device drivers.

Explicit vector and data capabilities give each ring its own code and data capabilities to utilize during exception handling. We extend the existing RISC-V *x*`scratch` registers as 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.

When exception behavior, e.g. a trapping instruction, `ecall`, or *x*`RET`, causes **PCC** to take a value stored in an SCR, it is possible that the SCR contains a capability that would not be a valid **PCC** (untagged, sealed, not executable, or improperly aligned). In these cases, the value is still installed in **PCC**, and a check on the next instruction fetch triggers a further exception.

4.3.11 Capability Exception Reporting

CHERI-RISC-V extends the definition of the Trap Value CSRs, *x*`tval`, to report capability exception details as described in Figure 4.2 (shown for XLEN=32):

cause The `cause` field reports the capability exception code from Table 4.4.

Value	Description
0x00	None
0x01	Length Violation
0x02	Tag Violation
0x03	Seal Violation
0x04	Type Violation
0x05-0x07	*reserved*
0x08	Software-defined Permission Violation
0x09-0x0f	*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	*reserved*
0x18	PERMIT_ACCESS_SYSTEM_REGISTERS Violation
0x19	PERMIT_INVOKE Violation
0x1a-0x1b	*reserved*
0x1c	PERMIT_SET_CID Violation
0x1d-0x1f	*reserved*

Table 4.4: CHERI-RISC-V Capability Exception Codes

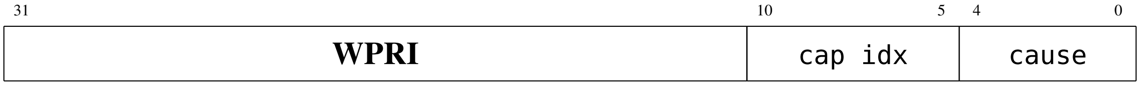

Figure 4.2: `xtval` register format for Capability Exception

Priority	Description
1	PERMIT_ACCESS_SYSTEM_REGISTERS Violation
2	Tag Violation
3	Seal Violation
4	Type Violation
5	PERMIT_INVOKE Violation
	PERMIT_SET_CID 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	Software-defined Permission Violation

Table 4.5: CHERI-RISC-V Capability Exception Priority

cap idx The `cap idx` field reports the index of the capability register that caused the last 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 4.3, otherwise, they index the general-purpose capability register file.

If an instruction could potentially throw more than one capability exception, the capability exception code is set to the highest priority exception (numerically lowest priority value) as shown in Table 4.5.

4.3.12 Virtual Memory and Page Tables

In CHERI-RISC-V, capability addresses are interpreted with respect to the privilege level of the processor in line with RISC-V's handling of integer addresses. In Machine Mode, capability addresses are generally interpreted as physical addresses; if the `mstatus` `MPRV` flag is asserted, then data accesses (but not instruction accesses) will be interpreted as if performed by lower-privileged modes. In Supervisor and User Modes, capability addresses are interpreted as dictated by the current `satp` configuration: addresses are virtual if paging is enabled and physical if not.

In CHERI-RISC-V, we require PERMIT_ACCESS_SYSTEM_REGISTERS to change the page-table root (`satp`) 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; see Appendix C.13.4.)

It is desirable to extend the Memory Management Unit to constrain the loading and storing of valid capabilities via specific page mappings by adding new 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, we could 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, some of which we allocate for use by CHERI-RISC-V; see Figure 4.3.

Capability Stores

Capability stores are mediated with two bits per PTE, called CW and CD. Their effect on capability flow parallels the existing W and D bits and is described by the following table:

CW	**CD**	**Behavior**
0	X	Trap on capability stores (exception code 0x1B)
1	0	Capability stores atomically raise CD or fault (as above)
1	1	Capability stores permitted

Currently, implementations must apply these behaviors to all instructions which would store an asserted capability tag; that is, they are dependent on the tag bit. This may be relaxed in future versions of this specification to all instructions which *could* store an asserted capability tag, removing the dependence on the tag bit. Instructions which are able to move only data (and so necessarily clear tags) will not interact with these PTE flags. CHERI-aware Sv32 implementations, lacking room in their PTEs, act as though CW and CD are *set*.

As with the existing D bit, there are two permitted approaches for hardware to take in response to an attempted store with an asserted CHERI tag and through a PTE with clear CD: ① raise a store capability page fault (exception code 0x1B), or ② atomically update the PTE to set CD. In this case, the existing rules regarding atomicity continue to apply: the PTW must check, atomically, that the PTE is valid and has W and CW both set, and the PTE update must become visible no later than the causal store. Capability-store instructions are still stores and so are expected to check the W permission, in addition to CW, and to set the existing D bit (or fault if it is clear, using the existing RISC-V *x*`cause` code) in addition to the CD bit (or fault, using the new capability store/AMO page fault *x*`cause` code). The ordering of checks of, and updates to, the PTE follows the scheme of RISC-V but interdigitates capability mediation: V, U, and W must be checked first, followed by CW, before any of D and/or CD (and/or A) are atomically asserted or are used as grounds for faulting. In the latter case, D and/or A take precedence over CD.

The PTE bits CW and CD have no necessary relationship to any of the CHERI tag bits on the corresponding physical page. In particular, CD does not reflect the presence of capabilities on the page, much as D does not reflect anything about the particular values of data on a

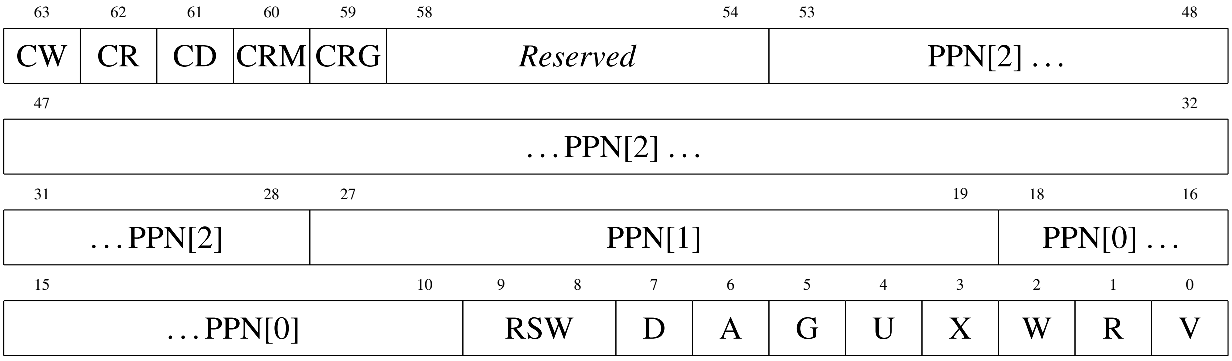

Figure 4.3: A Sv39 PTE showing CHERI extensions in bits 59 through 63.

page. Software-enforced temporal safety mechanisms, for example, are anticipated to regularly clear CD (and even, occasionally, CW) on PTEs referencing pages that nevertheless contain capabilities.

Capability Loads

Aside from experimental behavior, there are three behaviors we would like to elicit upon attempted load of a capability: succeed, strip loaded tags, or raise a fault. To leave sufficient room for experimentation, we reserve three bits for mediation of capability loads, CR, CRM, and CRG, but we reserve most configurations (which will raise page faults by virtue of being invalid settings in PTEs). These bits interact with capability flow as follows:

CR	**CRM**	**CRG**	**Behavior**
0	0	0	Capability loads strip tags on loaded result
0	1	0	Capability loads fault (exception code 0x1A)
0	X	1	*Reserved for future use*
1	0	0	Capability loads are unaltered
1	0	1	*Reserved for future use*
1	1	X	*Reserved for generational load barriers*

As with CW/CD, implementations are required to take a data-dependent disposition when a PTE is configured to fault on a capability load, raising faults only when the resulting tag is set. Future versions of this specification may, similarly, relax this to any instruction which *could* load a capability with a set tag. CHERI-aware Sv32 implementations act as though CR is *set* and both CRM and CRG are clear.

4.3.13 The RV128 LQ, SQ, and Atomic Instructions

The putative 128-bit RISC-V ISA (RV128) 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 RV64, we reuse these hypothesised opcode encodings for our 129-bit capability load and store instructions, `LC` and `SC`, to avoid additional opcode commitment. We also introduce corresponding atomics on capabilities reusing the quad-word atomic opcodes.

Should the future RV128 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 RV128 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. 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 RV128 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.

4.3.14 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, `AUICGP`, which returns a capability derived from a global capability table register.

4.4 Design Rationale

The CHERI-RISC-V specification has evolved over time. This section describes some of the alternatives considered and other changes made.

4.4.1 Capability Exception Reporting

Initially CHERI-RISC-V reported additional information for capability faults via `cause` and `cap idx` fields in a new per-mode CSR *x*`ccsr`. During CHERI-RISC-V's development, the RISC-V privileged spec renamed the per-mode *x*`badaddr` CSR to *x*`tval` and reused it to store non-address information for some faults. Following this change, CHERI-RISC-V moved the `cause` and `cap idx` fields to *x*`tval`.

4.4.2 MMU Capability Exceptions

Initially CHERI-RISC-V reported MMU tag permission violations via the 0x1C exception code. Unique values were assigned for these violations to the `cause` field of *x*`tval`. However, storing the faulting address in *x*`tval` similar to other MMU faults was found to be more useful. As a result, additional top-level exception codes (0x1A and 0x1B) were added to report MMU tag permission violations instead.

4.4.3 Separate Capability Register File

CHERI-RISC-V was originally specified with two different approaches for adding general-purpose capability registers. One approach described earlier extended existing integer general-purpose registers to hold capabilities (a "merged" register file). The second approach left the integer register file as-is and added a separate capability register file to hold capabilities (a "split" register file). This second approach mirrored the approach used on CHERI-MIPS.

Most of the ISA is agnostic to this choice due to the principle of intentional-use such as the explicit use of capability-relative addresses versus integer pointers constrained by **DDC** or **PCC**. However, the "split" register file approach did require a few additional changes:

- A separate `Clear` instruction similar to `CClear` and `FPClear` which cleared a set of general-purpose integer registers.
- A separate `CGetAddr` instruction to fetch the address of a capability register. In a "merged" register file this can be obtained by reading the aliased general-purpose integer register.
- A separate `CSub` instruction to compute the difference between addresses in two capability registers. In a "merged" register file this can be computed via the existing `SUB` instruction on the aliased general-purpose integer registers.
- Dirty bits were defined in the per-mode CSR *x*`ccsr`. This bit was set anytime a general-purpose capability register was changed. It was intended to be used similar to `FS` and `XS` fields in `mstatus` to optimize context switching.

In practice, only variants of CHERI-RISC-V using a "merged" register file were implemented in emulators, soft cores, toolchains, and operating systems.

4.4.4 DDC and PCC Relocation

CHERI-RISC-V originally specified that legacy memory accesses using integer pointers were relocated by **DDC** and **PCC** in addition to being constrained. In this model, integer pointers were treated as offsets relative to the address of **DDC** or base of **PCC**[3] rather than addresses.

The current version of CHERI-RISC-V does not relocate integer pointers. However, it may be desirable to provide optional relocation in the future, in particular to permit compartmentalization of hybrid code in user mode. Such compartments would run with **DDC** and **PCC** set to narrow bounds and integer pointers relocated to the base address of **DDC**.

---

[3] While offsetting with respect to **DDC**'s address avoids unnecessary alignment requirements, the base of **PCC** must be used since its address must change as the program runs, so cannot be used for offsetting.

Chapter 5

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.

5.1 CHERI-x86-64 Approach

In applying CHERI to the 64-bit x86 architecture, we aim to provide a model similar to CHERI-RISC-V and Morello. This model should have the following properties:

- A new capability hardware type that is usable for C language pointers.
- Capability values should be intentionally used. Instructions should explicitly specify whether a register operand should be used as a capability or an integer scalar. Specifically, the presence (or lack) of a tag should not determine if a value is treated as a capability rather than an integer.
- While new instructions will be required to manipulate capabilities, common code patterns for pure-capability C such as function prologues and epilogues should use similar opcode density to 64-bit x86.

5.1.1 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.

5.1.2 Common Architectural Features

CHERI-x86-64 shares the following features with other CHERI architectures:

- Tagged memory with capability-width tag granularity and alignment.

- Registers able to hold capabilities are tagged.
- **CIP** controls program-counter-relative fetches.
- **DDC** controls memory operands using integer addresses.
- Floating point is fully supported, including capability-relative floating-point load and store instructions.
- General-purpose registers are extended to hold capabilities.
- It is never left ambiguous as to whether a register operand used as the base address of a memory operand or branch target is a capability and therefore must have a tag set.
- PERMIT_ACCESS_SYSTEM_REGISTERS limits privileged ISA operations when within privileged rings.

5.1.3 Unique Architectural Features

The following changes are specific to CHERI-x86-64:

- CHERI-x86-64 makes use of opcode prefixes to permit altering the addressing mode and operand size of individual instructions, both in 64-bit mode and capability mode.
- **RIP** is the full integer value (virtual address) of **CIP** and not **CIP**.offset.
- Integer addresses are treated as absolute virtual addresses bounded by **DDC**, and are not treated as offsets to **DDC**.base.
- x86 exception handling is extended to support capabilities including a new architectural stack frame for exception entry and return.
- A new exception code is used to report CHERI-related exceptions.
- New PTE bits and page-fault exception code bits are defined for loading and storing capabilities in memory.
- The **IA32_FS_BASE** and **IA32_GS_BASE** registers are extended as capabilities.
- As with CHERI-RISC-V, the **flags** field contains a single bit used to enable capability mode in code capabilities installed into **CIP**.
- Operations on capabilities can set bits in the **RFLAGS** register.

5.2 CHERI-x86-64 Specification

5.2.1 Tagged Capabilities and Memory

As with CHERI-RISC-V, we recommend that both memory and registers contain tagged 128-bit capabilities. Since capabilities require 16-byte alignment in memory, attempts to load or store capabilities at misaligned addresses should raise a General Protection Fault with an error code of zero, similar to misaligned loads and stores of SSE registers.

5.2.2 General-Purpose Capability 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 the bottom 64 bits of capability registers contain the integer value (virtual address) and the upper 64 bits contain capability metadata. Reads of capability registers as integers return the integer value. Integer writes to capability registers should clear the tag and upper 64 bits of capability metadata, storing the desired integer value in the bottom 64 bits.

The **RIP** register (which contains the address of the current instruction in the existing x86 architecture) would also be extended into a **CIP** capability. This would function as the equivalent of **PCC**.

5.2.3 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 control registers will be required to manage user to kernel transitions as described in Section 5.2.12.

These additional registers will be stored as a separate bank of capability registers. As with other x86 register banks such as control registers and debug registers, additional capability registers cannot be used as operands (with a limited exception for **CFS** and **CGS** described below) in existing instructions.

5.2.4 Capability Mode

As with other CHERI architectures, CHERI-x86-64 should support running existing x86-64 code, capability-aware code, and hybrid code. This requires the architecture to support an additional addressing mode using capabilities as well as a new operand size for instructions that use capabilities as operands. The x86 architecture has supported similar extensions in the past when it was extended to support 32-bit operation.

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 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.

For CHERI support, we propose a similar scheme of using a default execution mode along with prefixes to toggle the individual addressing mode and operand size of individual instructions. We define a new **capability mode**. As with CHERI-RISC-V, this mode is enabled by setting the low bit of the **flags** field in **CIP**. This mode is valid only in 64-bit mode. A far call or jump that uses a 32-bit code segment along with a target code capability with this flag set will raise a General Protection Fault with the error code set to the target segment selector.

In capability mode, instructions will use capability-aware addressing (Section 5.2.5) by default. Some existing opcodes will also assume a capability sized operand in this mode. Finally, instructions which work with the stack would use **CSP** as the implicit stack pointer.

Removed Instructions in Capability Mode

In capability mode, the following 64-bit mode instructions would no longer be valid:

- `PUSH FS`
- `POP FS`
- `PUSH GS`
- `POP GS`
- `LFS`
- `LGS`
- `LSS`
- `LAR`

- `LSL`
- Direct memory-offset `MOV`
- Far branches (`CALL`, `JMP`, and `RET`)
- Segment Prefixes for **CS**, **DS**, **ES**, and **SS**

5.2.5 Using Capabilities with Memory Address Operands

We propose a new capability-aware addressing mode that can be toggled via a new 0x07 opcode prefix. (In 32-bit x86, the 0x07 opcode is the `POP ES` instruction, which is invalid in 64-bit mode.) In capability mode, instructions will use the capability-aware addressing mode by default. Individual instructions can toggle between capability-aware and "plain" 64-bit addressing via the 0x07 opcode prefix. Addresses using the "plain" 32-bit or 64-bit addressing will be constrained by **DDC** (for example, bounds and permissions). Instructions using capability-aware addressing would always use 64-bit virtual addresses.

The 0x07 prefix would be a Group 4 prefix meaning that a single instruction would not be permitted to use both 0x67 and 0x07 prefixes. In addition, the use of the 0x67 prefix in capability mode would not be permitted.

Capability-Aware Addressing

For instructions with register-based memory operands, capability-aware addressing would use the capability version of the register rather than the integer register as a virtual address constrained by **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 the address **RBP**+8 constraining the memory access to the bounds and permissions of the **DDC** capability. Both instructions would use the same opcode aside from the addition of an 0x07 opcode prefix. In capability mode, the second instruction would require the prefix. In plain 64-bit mode, the first instruction would require the prefix.

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.

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 constrained by **DDC** when using "plain" 64-bit addressing.

When capability-aware addressing is used, **RIP**-relative addresses would instead be treated as **CIP**-relative addresses constrained by the bounds and permissions of **CIP**.

Absolute Addresses

Memory operands can be encoded without a base register, either as an absolute address, or an absolute address added to a scaled index register. If these addresses are not used as offsets relative to **CFS** or **CGS** as described below in Section 5.2.5, they are always constrained by **DDC**, including in capability mode.

Direct Memory-Offset MOVs

The direct memory-offset `MOV` instructions store the absolute address of a memory operand as an immediate operand. Extending these instructions to support capability immediates would require padding nops to align the capability immediate as well as text relocations (even for position-dependent code). However, we do not anticipate wide use of these instructions so instead choose to remove them in capability mode and restrict them to using integer operands and integer addressing in 64-bit mode. Attempting to use these instructions with capability-aware addressing would be reserved and raise a UD# exception.

Addresses Relative to CFS and CGS

Capability-aware addressing must also permit addresses defined as offsets relative to **CFS** and **CGS** to support TLS with capability-aware addresses. When an instruction uses the **FS** or **GS** segment prefix with capability-aware addressing, the memory operand (registers and displacement) is interpreted as an integer offset relative to the **CFS** or **CGS** capability register, respectively.

Other segment prefixes are not permitted in capability-aware addressing. Attempting to use a segment prefix other than **FS** or **GS** with a capability-aware address should raise an illegal instruction exception.

Instructions with Implicit Memory Operands

Some x86 instructions have implicit memory operands addressed by a register. These instructions should support addressing memory with capabilities.

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 capability addressing mode is enabled, these string instructions should use **CSI** instead of **RSI** and **CDI** instead of **RDI**.

`XLAT` should use **CBX** as the implicit table address when using capability-aware addressing.

Stack Address Size

Instructions that work with the stack such as `PUSH` or `CALL` use the stack pointer as an implicit operand. In 32-bit x86, the 'B' flag of the stack segment selector determines if the 16-bit or 32-bit stack pointer register is used. In 64-bit long mode, **RSP** is always used as the stack pointer. In capability mode, **CSP** would always be used as the stack pointer.

Code that needs to use the alternate stack pointer interpretation would simulate these instructions using `MOV` instructions and adjusting the desired stack pointer using instructions such as `ADD` or `SUB`. Emulation of `CALL` or `RET` would use `JMP` to adjust the instruction pointer.

5.2.6 Capability-Aware Instructions

CHERI-x86-64 will require new instructions to examine and modify capabilities. Many of these new instructions can be implemented as new variants of existing instructions that use an opcode that specifies a capability operation rather than an integer operation. Existing x86 toolchains already use instruction suffixes such as `b`, `w`, `l`, and `q` to explicitly state the operand size. We recommend that the `c` suffix be used to explicitly state a capability operand size.

Capability Operands for Existing Opcodes

Previous extensions to the x86 architecture have relied on opcode prefixes combined with the 'D' and 'L' flags of the current code segment to determine the operand size. We propose a similar scheme for supporting capability-sized operands with existing opcodes.

First, we propose reusing a single-byte opcode declared invalid in 64-bit mode such as 0x06 (`PUSH ES`) as an opcode prefix (**capability operand prefix**). This prefix would be classified as a Group 3 prefix meaning that a single instruction would not be permitted to use both 0x66 and 0x06 prefixes.

When not executing in capability mode, existing instructions will follow the existing rules for 64-bit long mode as defined by the 0x66 prefix and `REX.W` flag to set the operand size. If an instruction supports capability-sized operands, the capability operand prefix can be used to use a capability-sized operand instead. This prefix would have higher precedence than `REX.W`.

In capability mode, most instructions that can operate on either integer or capability-sized values would follow the same logic in the previous paragraph to determine the operand size. However, two groups of existing instructions would default to using a capability-sized operand when executed in capability mode:

- Near branches.
- Instructions that implicitly reference the stack pointer (**CSP**).

This matches the approach used to select a default operand size of 64 bits in 64-bit long mode. For some of these instructions, the capability operand prefix could be used to revert to a smaller operand size. The effective operand size would then determined by `REX.W`.

Extending Existing Instructions to Support Capability Operands

Several existing instructions should be extended to support capability operands:

- `MOVC` would handle loads and stores of capabilities similar to `CLC` and `CSC` as well as copying capabilities between registers similar to `CMove`.

  To permit moving the contents of an additional capability register to a general-purpose register or vice versa, two new `MOV` opcodes would be used. These opcodes would permit access to **CFS**, **CGS**, and **DDC** in all privilege levels. Access to other additional capability registers would be permitted only in privilege level 0.

- `MOVNTIC` would store a single capability to memory using a non-temporal hint.

- The string instructions `LODS`, `MOVS`, and `STOS` would be extended to support capability operands.

  We do not currently foresee a need to extend `CMPS` and `SCAS` with support for capability operands. If that did prove necessary, they could be extended.

- `CMOVC` would handle conditional loads and stores of capabilities.

- `ADDC` and `SUBC` would be used to adjust the **address** field of a capability similar to `CIncOffset`. Note that for these instructions, the source operand would either be a sign-extended immediate or a 64-bit integer register whose value is either added to or subtracted from the **address** field of the capability-sized destination operand.

  For example:

  ```
  add %csp,$16
  ```

  would move the capability stack pointer up by 16 bytes.

  We do not anticipate a need for capability-sized variants of `ADC` or `SBB`.

- `INCC` and `DECC` would permit simple increments and decrements of the **address** field of capabilites.

- `ANDC`, `ORC`, and `XORC` would permit bit manipulation of the **address** field of a capability. As with `ADDC`, the second operand would always be an integer operand.

- `CMPC` would permit comparison of capability values including the functionality of both `CSetEqualExact` (via `ZF`) and `CTestSubset` (via `SF`). This is somewhat different from the existing variants of `CMP` that perform the equivalent `SUB` instruction and then discard the result as in this case the flags set would not be identical to the flags set as a result of `SUBC`.

  We do not anticipate a need for a capability-sized variant of `TEST`.

- `CMPXCHGC` 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.)

- `CMPXCHG2C` will be required to support atomic operations on pairs of capabilities.

- `XCHGC` will also be required to support atomic operations on capabilities.

- It may also be desirable to support `XADDC`. For this instruction, only the integer portion of the second (source) operand would be added to the first (destination) operand to determine the value stored to the destination. Any tag or capability metadata in the second operand would be ignored and would be overwritten with the original value of the first operand.

- `PUSHC` and `POPC` would be used to save and restore capability registers on the stack.

- `LEAC` would store the resulting address in a destination capability register.

  `LEA` would not support the 0x07 opcode prefix. The address size would always match the operand size. Storing an integer address in a capability register would have the same effect as the equivalent version of `LEA` storing the integer address to the integer alias register. Using a capability-aware address with an integer `LEA` would also be identical in effect to using "plain" addressing.

- `ENTER` and `LEAVE` could be extended to support implicit capability operands, or they could be deprecated and remain as integer-only instructions.

  If these instructions were extended to support capability operands, the capability-sized versions would operate on **CSP** and **CBP** rather than **RSP** and **RBP**. These instructions would also default to capability operands in capability mode if extended.

  If these instructions were deprecated then they would would be removed in capability mode.

5.2.7 Control-Flow Instructions

Absolute near branches would be extended to support capability operands. In 64-bit long mode, a capability operand prefix would select a capability operand size. In capability mode, absolute near branches would support only capability operands. Absolute near branches that use an integer operand would set the **address** field of the **CIP** capability while absolute near branches using a capability operand would load a new capability into **CIP**. Relative near branches would always modify the **address** field of the **CIP** capability and would not support the capability operand prefix.

The size of return addresses pushed to and popped from the stack for near branches would be determined by the operand size. Capability-sized branches would save and restore a full capability on the stack while integer-sized branches would save and restore an integer address.

Far calls, jumps, and returns would not support capability operands and would be invalid in capability mode. Far branches would set the **address** field of **CIP**.

If the resulting value of **CIP** after any branch is invalid, a capability violation fault would be raised on the branch instruction (see Section 5.2.10).

`IRETC` should pop a capability exception frame (see Section 5.2.12) from the stack loading capabilities into **CIP** and **CSP**. This instruction would require the capability operand prefix. An attempt to restore a 32-bit code segment paired with a **CIP** that uses capability mode should raise a General Protection fault with the error code set to the destination code segment.

Note that attempting to push or pop a misaligned capability will raise an exception. The stack pointer must be suitably aligned before the use of `CALLC`, `IRETC`, and `RETC`.

5.2.8 New CHERI Instructions

For other capability operations we propose adding new CHERI-specific instructions. Existing general-purpose x86 instructions support two operands rather than three operands. To avoid requiring a **VEX** prefix for all new CHERI instructions, most instructions are defined with two operands rather than three. New instructions that require three operands must be encoded using a **VEX** prefix.

Note that all of these instructions would only be valid in 64-bit mode and capability mode.

Capability-Inspection Instructions

These instructions fetch a single field from a capability.

- `GCPERM` – Get Capability Permissions
- `GCTYPE` – Get Capability Object Type
- `GCBASE` – Get Capability Base
- `GCLEN` – Get Capability Length
- `GCTAG` – Get Capability Tag
- `GCOFF` – Get Capability Offset

- `GCHI` – Get Capability High Half
- `GCLIM` – Get Capability Limit
- `GCFLAGS` – Get Capability Flags

Capability-Modification Instructions

If these instructions fail, they should clear the tag in the resulting capability.

- `SEAL` – Seal Capability
- `UNSEAL` – Unseal Capability
- `ANDCPERM` – Mask Capability Permissions
- `SCOFF` – Set Capability Offset
- `SCADDR` – Set Capability Address
- `SCBND` – Set Capability Bounds
- `SCBNDE` – Set Exact Capability Bounds
- `SCHI` – Set Capability High Half
- `SCFLAGS` – Set Capability Flags
- `CLCTAG` – Clear Capability Tag
- `BUILDCAP` – Construct Capability
- `CPYTYPE` – Construct Sealing Capability
- `CSEAL` – Conditional Capability Seal
- `SENTRY` – Seal Capability as a Sentry

Control-Flow Instructions

- `CINVOKE` – Invoke sealed capability pair

Adjusting to Compressed Capability Precision Instructions

- `CRRL` – Round Representable Length
- `CRAM` – Representable Alignment Mask

Tag-Memory Access Instructions

These instructions permit bulk access to a set of in-memory tags. Each instruction accesses the tags in a "stride" of capabilities. The size of a stride is implementation dependent. It must be a power of two, and it is suggested that a stride contain the number of tags in a single cache line. The stride size should either be reported in a new `CPUID` leaf or be defined as equal to the value returned by an existing `CPUID` leaf.

- `LCTAGS` – Load Capability Tags
- `CLCTAGS` – Clear Capability Tags

5.2.9 Interactions with Vector Extensions

CHERI should have minimal impact on existing vector extensions to the x86 architecture including MMX, SSE, AVX, and AVX-512.

Vector Registers and Memory Tags

We propose that vector registers should not contain tags. Loads of vector registers should ignore tags in memory, and stores of vector registers to memory should always clear tags. Existing vector instructions that manipulate vector register contents do not make sense for tagged capability values. However, vector extensions are also used to perform certain classes of memory loads and stores, which may require additional care.

Memory Copies

Vector loads and stores are often used to implement `memcpy()`. In CHERI C, `memcpy()` must preserve tags. A `memcpy()` implementation that uses `MOVC` will operate at the same width as existing memory copies implemented using SSE, which may mitigate some of the cost. Another option may be to support an optimized `REP MOVSC` similar to the existing optimization for `REP MOVSB` where the former instruction would preserve tags during a copy unlike the latter.

Non-Temporal Stores

Blocks of data stored to memory mapped with write-combining (WC) are often written via non-temporal vector register stores. However, such data is generally consumed by an I/O device via DMA and rarely contains pointers. We believe that permitting a non-temporal store of a single capability via `MOVNTIC` is sufficient for cases requiring non-temporal stores of tagged capabilities.

Memory Addressing

Vector instructions with memory operands would support capability-aware addressing in the same manner as general-purpose register instructions. For scatter/gather instructions using VSIB, the base address register would use a capability register instead of an integer address when using capability-aware addressing.

Value	Description
0x0	Tag Violation
0x1	Length Violation
0x2	Seal Violation
0x3	Type Violation
0x4	Software-defined Permission Violation
0x5	GLOBAL Violation
0x6	PERMIT_EXECUTE Violation
0x7	PERMIT_LOAD Violation
0x8	PERMIT_STORE Violation
0x9	PERMIT_LOAD_CAPABILITY Violation
0xa	PERMIT_STORE_CAPABILITY Violation
0xb	PERMIT_STORE_LOCAL_CAPABILITY Violation
0xc	PERMIT_ACCESS_SYSTEM_REGISTERS Violation
0xd	PERMIT_INVOKE Violation
0xe	PERMIT_SET_CID Violation

Table 5.1: CHERI-x86-64 Capability Exception Error Codes

5.2.10 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. This error code would contain the capability exception code as described in Table 5.1 to indicate the specific violation.

If an instruction could potentially throw more than one capability exception, the capability exception error code is set to the highest priority exception (numerically lowest priority value) as shown in Table 5.2.

CHERI-RISC-V includes the name of the register, which triggers a capability violation. It is not feasible to provide a direct analog of this on x86. Indirect jumps and calls may raise an exception while loading a capability from memory that is not present in any register at the start of the instruction. However, unlike page faults, capability violation faults are not generally restartable and the register name's primary use is for debugging convenience rather than correctness. There are a few possible options for providing similar information:

1. Provide a copy of the faulting capability via a new capability control register similar to the PF# virtual address stored in **CR2**. This faulting capability would include the result of any offset adjustments from immediates or scaled indices. If the result of offset adjustments made the capability unrepresentable, the faulting capability would have its tag cleared.

2. Similar to the above, but ignore offset adjustments and provide only the base capability value.

Priority	Description
1	PERMIT_ACCESS_SYSTEM_REGISTERS Violation
2	Tag Violation
3	Seal Violation
4	Type Violation
5	PERMIT_INVOKE Violation PERMIT_SET_CID 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	Software-defined Permission Violation

Table 5.2: CHERI-x86-64 Capability Exception Priority

3. Provide the virtual address from the faulting capability in **CR2** similar to PF#. A debugger could examine the faulting instruction's operands to determine which capability triggered the fault.

4. Do nothing as the prior approaches may be too expensive to implement.

Like Morello and CHERI-RISC-V, CHERI-x86-64 would raise capability violation faults when a invalid memory access is performed such as an out-of-bounds access or access via an untagged capability. Instructions which modify capabilities should not raise capability violation faults (for example, when a capability becomes unrepresentable) but should instead clear the tag of the resulting capability. This permits compilers to speculatively reorder these instructions without raising spurious faults during execution.

5.2.11 Call Gates

We do not recommend extending call gates to support capabilities. Supporting capabilities with call gates would likely require the following changes:

- Extending the global and local descriptor table format to support a new capability call gate that stores a full capability rather than a 64-bit address. This will be more invasive than the 64-bit call gate that depends on the ability to force a number of reserved bits in the fourth double word to zero as a sentinel type for the second half of a 64-bit call gate.

- As with 64-bit call gates, capability call gates would not support parameter copying.

- Calls to a capability call gate would need to push a modified call frame containing both a code segment and code capability that would be returned from via `RETFC`.

5.2.12 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. As with exceptions in long mode, the stack pointer would be 16-byte aligned prior to pushing the exception frame to ensure capabilities are aligned. The **RIP** and **RSP** fields in the exception frame would be replaced with the full **CIP** and **CSP** capabilities. Other fields in this frame would be padded to 16 bytes. To minimize padding, it may be desirable to pack multiple smaller registers into a single 16-byte slot; for example, **SS**, **CS**, and **RFLAGS** could be stored in a single slot. However, this would result in a frame layout inconsistent with far calls. `IRETC` would be used in interrupt service routines to unwind this frame.

Capability Control Registers

Interrupt and exception handlers require new capabilities for the program counter (**CIP**) and stack pointer (**CSP**) registers. These values must be derived from valid, privileged capabilities. To support this, we propose the addition of a new class of capability registers: capability control registers.

Capability control registers are capability-sized control registers. As with other control registers such as **CR4**, direct access to capability control registers would be restricted to supervisor mode as well as requiring PERMIT_ACCESS_SYSTEM_REGISTERS in **CIP**. Unlike other control registers, however, capability control registers would not be accessed via the `0F 20` and `0F 22` opcodes of `MOV`. Instead, capability control registers would be named as additional capability registers as described in Section 5.2.3.

We consider two possible approaches for deriving **CIP** and **CSP** at the start of an interrupt or exception.

Kernel Code and Stack Capabilities

The first approach would add two new capability control registers: the Kernel Code Capability (**KCC**) and Kernel Stack Capability (**KSC**). Transitions into supervisor mode would load new addresses from existing data structures and tables to derive 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 address to derive a new **CIP** from **KCC**, and the address stored in the Interrupt Stack Table (**IST**) entry in the current Task-State Segment (**TSS**) would be used as an address to derive a new **CSP** from **KSC**. Transitions via the `SYSCALL` instruction would use the address 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.

Capabilities in Entry Points

The second approach would 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.

Setting the desired kernel stack pointers in **CSP** would require a new **TSS** layout that expanded the existing **RSP** and **IST** entries to capabilities.

For `SYSCALL`, a new capability control register **CSTAR** would be added to hold the target instruction pointer.

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.

`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. We recommend defining a new capability control register **KGS**. `SWAPGS` in capability mode would swap the **CGS** and **KGS** registers.

5.2.13 FS and GS Aliases

The **FS** and **GS** segment descriptors have grown several related aliases over time such as the **IA32_FS_BASE** and **IA32_GS_BASE** MSRs and `RDFSBASE` family of instructions. These aliases should be implemented as the addresses of the appropriate capability register. Reads of the **IA32_FS_BASE**, **IA32_GS_BASE**, and **IA32_KERNEL_GS_BASE** MSRs should return the **address** field of the **CFS**, **CGS**, and **KGS** capabilities, respectively. Writes to these MSRs should set the **address** field of the respective capability equivalent to `SCADDR`. Similarly, the `RDFSBASE` and `RDGSBASE` instructions should return the **address** field of the **CFS** and **CGS** capabilities, respectively. The `WRFSBASE` and `WRGSBASE` instructions should set the **address** field of the respective capability equivalent to `SCADDR`. If a new address is set that makes the capability unrepresentable, the capability's tag should be cleared.

5.2.14 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 (`MAXPHYADDR`) to bit 62. The Protection Keys extension uses bits 59-62 from that range. To avoid conflicting with Protection Keys, CHERI-x86-64 could use bits starting at bit 58 as described in Table 5.3. Higher bits are preferred, to permit maximal room for growth of the physical address field that currently ends at bit 51.

If an instruction performs a memory access that violates a CHERI page permission (such as a store of a tagged capability to a page where the `CW` bit is clear), a page-fault (PF#) exception should be raised. Bit 8 (currently reserved) should be set in the page-fault error code provided

Bit	Name	Description
58	CW	Permits writes of tagged capabilities
57	CR	Permits reads of tagged capabilities
56	CD	Set when a tagged capability is written to this page

Table 5.3: CHERI-x86-64 Page Table Bits

by the processor indicating that the fault was caused by a capability permission violation. Other bits in the page fault error code such as `P`, `W/R`, `U/S`, and `I/D` should be set to indicate the type of memory access. In addition, the virtual address of the memory access should be provided in the **CR2** register similar to other page-fault exceptions.

Note that the `CR` and `CW` bits fault only if the capability being read or written is tagged. Untagged capability values can be read from or written to memory regardless of the `CR` and `CW` permissions. In addition, if the authorizing capability for a capability read does not hold PERMIT_LOAD_CAPABILITY, then reading a tagged capability will always return a capability with the tag cleared instead of faulting.

Instruction fetches always ignore tags and will never raise a capability page-fault exception.

5.2.15 Controlling Access to System Registers

In CHERI-x86-64, PERMIT_ACCESS_SYSTEM_REGISTERS would be required to directly access the following registers:

- Control registers including **KCC**, **KSC**, **CSTAR**, and **KGS**
- Debug registers
- Model-specific registers

5.3 Design Rationale

We have considered several alternatives to various aspects of the CHERI-x86-64 design. This section describes some of those alternatives.

5.3.1 Capability Mode

Currently capability mode is enabled via a single-bit flags field in **CIP**. We did consider more closely matching older extensions to the x86 architecture by repurposing the ‘D’ flag of the current code segment descriptor to enable capability mode. Similarly, we considered using the ‘B’ flag of the current stack segment descriptor to select the implicit stack pointer of **CSP**. While this approach would match traditional x86, it would not protect instruction decoding by sealing. For example, a sentry capability could be used in either plain 64-bit mode or capability mode. By storing the mode in capability metadata protected by sealing, sealed code capabilities can be used only in the intended mode. Also, while it may be less flexible to permit the stack alignment to be chosen orthogonally to the default instruction encoding mode, it does not seem useful in practice. Instead, capability mode is designed as a single knob to optimize pure capability code.

5.3.2 Additional Capability Registers as Operands

We considered various options for using additional capability registers such as **CGS** as explicit operands in instructions rather than as a separate bank of registers accessed only via `MOV`. All of these approaches add complexity to instruction decoding, but we do not anticipate frequent direct access to additional capability registers beyond the use of the existing **FS** and **GS** segment prefixes.

Using Segment Prefixes

One approach to expand register selector fields would be to make use of existing segment prefixes to indicate a set 5th bit for a specific field. For example, the **GS** prefix could be used in capability-aware addressing mode to indicate that the base capability register used in a memory operand would be an additional capability register with an index of 16 or higher. The lower four bits of the register selector would be determined by the existing register selector fields. Note that this approach would void the earlier use of the **FS** and **GS** segment prefixes. Instead, **CFS** and **CGS** would be used as base address registers in memory operands via the expanded register selector field.

Additional prefixes could be used to extend other register selector fields at the cost of potentially using multiple segment prefixes in a single instruction. For example, the **FS** prefix could be used to extend the “r” register selector field.

`VEX.mmmmm` Field

A second approach can be used with instructions that can be encoded with a **VEX** prefix. The upper three bits of the `VEX.mmmmm` field could be reused as the 5th bit of register selector fields similar to `EVEX.R'`, `EVEX.X`, and `EVEX.V'` fields in **EVEX** prefixes.

EVEX Prefixes

A third approach would be to require **EVEX** prefixes for instructions using an additional capability register.

5.3.3 Access to Additional Capability Registers

The **CFS**, **CGS**, and **DDC** capability registers must be accessible in all privilege levels. However, other additional capability registers such as **KGS** are suitable only for privilege level 0. Currently the **0F 24** and **0F 25** instructions permit access to a subset of registers in privilege levels other than 0.

A few other approaches are enumerated below:

- For **CFS** and **CGS** the `RDFSBASE` family of instructions could be expanded to support a capability operand size. This would not provide a solution for access to **DDC** but would otherwise permit restricting **0F 24** and **0F 25** to privilege level 0.
- Capability control registers such as **KGS** could be allocated unused indices in the existing control register bank. This would require the **0F 20** and **0F 22** opcodes to vary the operand size based on the control register index.
- The additional capability registers could be split into two separate banks. One for **CFS**, **CGS**, and **DDC** accessible via **0F 24** and **0F 25** accessible at all privilege levels, and a second bank of capability control registers accessible via a a second set of opcodes such as **0F 26** and **0F 27** that were restricted to privilege level 0.

5.3.4 Additional Capability Arithmetic Opcodes

Two operand arithmetic instructions such as `ADD` overwrite one of the source operands with the arithmetic result. For operations that are commutative (such as adding two integers), a compiler can choose which of the source operands to overwrite. For example, if a series of instructions computes integer pointers to fields of an object by adding offsets to a base integer pointer, the compiler can use the register holding the offset as the destination operand of `ADD` to preserve the operand holding the base object integer pointer. Capability arithmetic instructions such as `ADDC` are not commutative since only one source operand holds a capability.

To mitigate this, it may be desirable to add alternate opcodes for `ADD`, `SUB`, `AND`, `OR`, and `XOR`, which treat the destination operand as an integer input to the arithmetic operation applied to the second capability operand. This could be implemented by extending a subset of the 8-bit opcodes of these instructions to perform a three operand operation when used with a capability operand prefix. However, two of these operands would be encoded by a single ModRM field similar to the encoding of `XADDC`.

For example, the `00` opcode would be extended to support three operand `ADDC` by adding a 64-bit offset read from ModRM:r/m to the **address** field of the capability read from ModRM:reg. The result would then be stored to the capability identified by ModRM:r/m.

The instruction

```
addc %cax, %cbx, %rax
```

would add **RAX** to the address field of **CBX** and store the result in **CAX**. It would be encoded identically to the instruction

```
addc %cax, %rbx
```

except for using the opcode `00` instead of `01`.

Note that this approach would permit encoding variants of `ANDC` and `SUBC` that preserve the base capability pointer input operand – which is not possible in the existing ISA for integer pointers.

Instructions that use an immediate source operand would not be extended in this manner.

5.3.5 Vector Registers and Tags

It may be desirable to support loading and storing tags in vector registers. In particular, if a tag preserving extension of `REP MOVSB` is not added, then loads and stores of multiple packed capabilities via new instructions may be desirable to support optimized implementations of `memcpy()`. For example, new tag-preserving variants of `MOVDQA` and `MOVDQU` could be added via two new two-byte opcodes.

This would require extending the vector registers to contain one or more tags (1 tag for XMM registers, 2 tags for YMM, 4 tags for ZMM). Instructions that modify vector registers should not permit non-monotonic operations on tagged capabilities embedded in vector registers. The simplest approach would be to clear all tags for any instruction other than simple move operations. However, it may be desirable to preserve tags for operations that are safe. For example, tags belonging to capabilities in the unshuffled half of a YMM or ZMM register used with `VPSHUFHW` could be safely preserved.

5.3.6 Far Branches and Capabilities

Supporting far branches with capability operands would add additional complexity. For example, far branches need to ensure that code capability pointers that enable capability mode are used only with 64-bit code segments. In-memory far capability pointers would also have odd alignment requirements due to the 16-bit code selector being adjacent to an aligned capability. Far branches are also little used in existing 64-bit x86 programs. Significantly, 64-bit x86 still defaults to 32-bit operands for far branches (unlike near branches that are commonly used and default to 64-bit operands).

5.3.7 Direct Memory-Offset MOVs

These four `MOV` instructions store the address of their memory operand inline as an immediate. Currently, we propose removing these instructions in capability code and not extending them to support capability operands in 64-bit mode. These instructions could instead be extended to support capabilities both as immediates for the memory offset and as operands. In that case, the opcodes would be retained in capability mode rather than removed.

5.3.8 XCHG [ER]AX Opcodes

If the `XCHG` instructions `91` – `97` are not commonly used, they could be removed in capability mode.

Chapter 6

Sail Overview

The instruction descriptions contained in the following chapters are accompanied by code in the Sail languge [9, 125]. 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 instructions can be implemented with finite arithmetic.
- Bit vector operators: `&` (bitwise AND), `<_s` (signed less than), `@` (bit vector concatenation)
- Equality: `==` (equal), `!=` (not equal)
- Vector slice:

  ```
  v[a..b]
  ```

  Creates a sub-range of a vector from index a down to b inclusive.
- Local variables:

  ```
  mutable_var = exp;
  let immutable_var = exp;
  ```

  Mutable variables are introduced by simply assigning to them. An explicit type may be given following a colon, but types can usually be inferred. Sail supports mutable or immutable variables where immutable ones are introduced by `let` and assigned only once when created.
- Functional if:

  ```
  if cond then exp1 else exp2
  ```

  May return a value similar to C ternary operator.
- Foreach loop:

  ```
  foreach(i from start_exp to end_exp) {
    body
  };
  ```
- Function invocation:

  ```
  func_id (arg1, arg2)
  ```
- Field selection from struct:

  ```
  struct_val.field
  ```

  Returns the value of the given field from structure.

- Functional update of structure:

  `{struct_val with field=exp}`

  A copy of the structure with the named field replaced with another value.

Chapter 7

The CHERI-RISC-V Instruction-Set Reference

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

7.1 Sail language used in instruction descriptions

The instruction descriptions contained in this chapter are accompanied by code in the Sail language taken from the Sail CHERI-RISC-V implementation [23]. 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. A brief description of the Sail language features used in this chapter can be found in Chapter 6. For a full description see the Sail language documentation [125].

7.2 Constant Definitions

The following constants are used in various type and function definitions throughout the specification. The concrete values listed here apply to the CHERI-RISC-V ISA with 128-bit capabilities that extends the base 64-bit RISC-V ISA. The constants for with 64-bit CHERI capabilities (extending the 32-bit RISC-V ISA) can be found in the CHERI-RISC-V Sail model [23].

```
type xlen         : Int = 64
type cap_addr_width : Int = xlen
type cap_len_width : Int = cap_addr_width + 1
type cap_size : Int = 16
type cap_mantissa_width : Int = 14
type cap_hperms_width : Int = 12
type cap_uperms_width : Int = 4
type cap_uperms_shift : Int = 15
```

```
type cap_flags_width : Int = 1
type cap_otype_width : Int = 18
let reserved_otypes = 4
let cap_max_otype = MAX(cap_otype_width) - reserved_otypes
type caps_per_cache_line : Int = 4
```

7.3 Function Definitions

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

Functions for integer and bit vector manipulation

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

`unsigned : forall 'n. bits('n) -> range(0, 2 ^ 'n - 1)`

converts a bit vector of length n to an integer in the range 0 to $2^n - 1$.

`signed : forall 'n, 'n > 0. bits('n) -> range(- (2 ^ ('n - 1)), 2 ^ ('n - 1) - 1)`

converts a bit vector of length n to an integer in the range -2^{n-1} to $2^{n-1} - 1$ using twos-complement.

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

`bool_to_bit : bool -> bit`

`bool_to_bits : bool -> bits(1)`

`truncate : forall 'm 'n, ('m >= 0 & 'm <= 'n). (bits('n), int('m)) -> bits('m)`

`truncate(v, n)` truncates `v`, keeping only the *least* significant `n` bits.

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

`MAX : forall 'n, 'n >= 0. int('n) -> int(2 ^ 'n - 1)`

`EXTZ`

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

`EXTS`

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

`zeros`

Produces a bit vector of all zeros

`ones`

Produces a bit vector of all ones

Types used in function definitions

```
type CapBits = bits(8 * cap_size)
type CapAddrBits  = bits(cap_addr_width)
```

```
type CapLenBits   = bits(cap_len_width)
type CapPermsBits = bits(cap_perms_width)
```

Many functions also use `struct Capability`, a structure holding a partially-decompressed representation of CHERI capabilities. The following functions can be used to convert between the structure representation and the raw capability bits:

`capBitsToCapability : (bool, CapBits) -> Capability`

`capToBits : Capability -> CapBits`

Functions for reading and writing registers and memory

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

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

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

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

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

The overloaded function `F(n)` is used to read or write floating-point register `n`.

`memBitsToCapability : (bool, CapBits) -> Capability`

`capToMemBits : Capability -> CapBits`

`int_to_cap : CapAddrBits -> Capability`

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

`ddc_and_resulting_addr : xlenbits -> (Capability, xlenbits)`

Returns the current value of DDC (for subsequent access checks) as well as the resulting address for a non-CHERI integer-based memory access.

`get_cheri_mode_cap_addr : (regidx, xlenbits) -> (Capability, xlenbits, capreg_idx)`

For given base register and offset returns, depending on current capability mode flag, a bounding capability, effective address, and capreg_idx (for use in cap cause).

`handle_load_cap_via_cap : (regidx, capreg_idx, Capability, xlenbits) -> Retired`

```
handle_load_data_via_cap : (regidx, capreg_idx, Capability, xlenbits, bool,
    word_width) -> Retired
```

`handle_store_cap_via_cap : (regidx, capreg_idx, Capability, xlenbits) -> Retired`

```
handle_store_data_via_cap : (regidx, capreg_idx, Capability, xlenbits, word_width)
    -> Retired
```

Functions for ISA exception behavior

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

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

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

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

Causes the processor to raise a capability exception by writing the given capability exception cause and register number to the xtval register then signalling an exception.

`handle_cheri_pcc_exception : CapEx -> unit`

Is as `handle_cheri_cap_exception` except that the capability register number uses the special value 0x10 indicating the PCC register.

`pcc_access_system_regs : unit -> bool`

`privLevel_to_bits : Privilege -> priv_level`

`min_instruction_bytes : unit -> CapAddrInt`

`legalize_epcc : Capability -> Capability`

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

Functions for manipulating capabilities

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

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

Note that for 128-bit capabilities top can be up to 2^{64}, meaning the entire 64-bit address space can be addressed.

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

`getCapBaseBits : Capability -> CapAddrBits`

`getCapTop : Capability -> CapLen`

`getCapLength : Capability -> CapLen`

`inCapBounds : (Capability, CapAddrBits, CapLen) -> bool`

The capability's address (also known as cursor) and offset (relative to base) are related by:

$$\mathit{base} + \mathit{offset} \ \mathbf{mod}\ 2^{64} = \mathit{cursor}$$

The following functions return the cursor and offset of a capability respectively:

`getCapCursor : Capability -> CapAddrInt`

`getCapOffsetBits : Capability -> CapAddrBits`

The following functions adjust the bounds and offset of 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, CapAddrBits, CapLenBits) -> (bool, Capability)`

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

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

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

`clearTag : Capability -> Capability`

`clearTagIf : (Capability, bool) -> Capability`

`clearTagIfSealed : Capability -> Capability`

`getRepresentableAlignmentMask : xlenbits -> xlenbits`

`getRepresentableLength : xlenbits -> xlenbits`

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

`unsealCap : Capability -> Capability`

`isCapSealed : Capability -> bool`

`hasReservedOType : Capability -> bool`

Tests whether the capability has a reserved otype (larger than `cap_max_otype`). Note that this includes both sealed (e.g. sentry) and unsealed (all ones) otypes.

Capability permissions and flags are accessed using the following functions:

`getCapPerms : Capability -> CapPermsBits`

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

`getCapFlags : Capability -> CapFlagsBits`

Gets the architecture specific capability flags for given capability.

`setCapFlags : (Capability, CapFlagsBits) -> Capability`

`setCapFlags(cap, flags)` sets the architecture specific capability flags on `cap` to `flags` and returns the result as new capability.

Checking for availability of ISA features

`haveRVC : unit -> bool`

`haveFExt : unit -> bool`

`haveNExt : unit -> bool`

`haveSupMode : unit -> bool`

`have_pcc_relocation : unit -> bool`

7.4 CHERI-RISC-V Instructions

AUIPCC

Format

AUIPCC cd, imm *(capability mode)*

31 … 12	11 … 7	6 … 0
imm[31:12]	cd	0x17

Description

Capability register *cd* is replaced with the contents of **PCC**, with the **address** replaced with **PCC**.**address** $+$ *imm* $\times$ 4096.

Semantics

```
let off : xlenbits = EXTS(imm @ 0x000);
let (representable, newCap) = setCapAddr(PCC, PC + off);
C(cd) = clearTagIf(newCap, not(representable));
RETIRE_SUCCESS
```

CAndPerm

Format

CAndPerm cd, cs1, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0xd	rs2	cs1	0x0	cd	0x5b

Description

Capability register *cd* is replaced with the contents of capability register *cs1* with the **perms** field set to the bitwise and of its previous value and bits 0 to `cap_hperms_width`-1 of integer register *rs2* and the **uperms** field set to the bitwise and of its previous value and bits `cap_uperms_shift` to `cap_uperms_shift`+`cap_uperms_width`-1 of *rs2*. If *cs1* was sealed then *cd*.**tag** is cleared.

Semantics

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

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

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

C(cd) = newCap;
RETIRE_SUCCESS
```

CBuildCap

Format

CBuildCap cd, cs1, cs2

31–25	24–20	19–15	14–12	11–7	6–0
0x1d	cs2	cs1	0x0	cd	0x5b

Description

Capability register *cd* is set equal to capability register *cs1* with its **base**, **length**, **address**, **perms**, **uperms** and **flags** replaced with the corresponding fields in capability register *cs2*. If *cs2* is a sentry then *cd* is also sealed as a sentry. If the resulting capability is not a subset of *cs1* in bounds or permissions, or is not a legally derivable capability, or if *cs1* did not have its **tag** field set, or if *cs1* was sealed, *cd* is replaced with *cs2* with its **tag** field cleared.

Semantics

```
let cs1_val = if unsigned(cs1) == 0 then DDC else C(cs1);
let cs2_val = C(cs2);

let authorityCap = cs1_val;
let requestedCap' = {C(cs2) with tag=true};
let requestedSentry = signed(requestedCap'.otype) == otype_sentry;
let requestedCap = if requestedSentry then requestedCap' else unsealCap(requestedCap
    ');

let (authorityBase, authorityTop) = getCapBounds(authorityCap);
let (requestedBase, requestedTop) = getCapBounds(requestedCap);
let authorityPerms = getCapPerms(authorityCap);
let requestedPerms = getCapPerms(requestedCap);
let requestedFlags = getCapFlags(requestedCap);

let subset = (requestedBase >= authorityBase)
           & (requestedTop <= authorityTop)
           & (requestedBase <= requestedTop) /* check for length < 0 - possible
               because requested might be untagged */
           & ((requestedPerms & authorityPerms) == requestedPerms);

let inCap = clearTagIfSealed(authorityCap);
let (exact, cd1) = setCapBounds(inCap, to_bits(cap_addr_width, requestedBase), to_-
    bits(cap_len_width, requestedTop));
let (_, cd2) = setCapOffset(cd1, getCapOffsetBits(requestedCap)); /* Ignore
    representability check, since Fast Rep Check not relevant */
let cd3 = setCapPerms(cd2, requestedPerms);
let cd4 = setCapFlags(cd3, requestedFlags);
```

```
let cd5 = if requestedSentry then sealCap(cd4, to_bits(cap_otype_width, otype_sentry
    )) else cd4;
let derivable = cd5 == requestedCap; /* True iff requestedCap has bounds not
    exceeding address space and no reserved bits set, and authority was tagged and
    unsealed */
assert(not(derivable) | exact, "CBuildCap: setCapBounds was not exact"); /* If
    requestedCap was a derivable encoding then setBounds should be exact */
let cd6 = if subset & derivable then cd5 else clearTag(requestedCap);

C(cd) = cd6;
RETIRE_SUCCESS
```

Notes

- Implementations may instead choose to set *cd* to *cs2* with its **tag** set after performing all checks, but the specification derives the result from *cs1* in order to convey the provenance associated with this operation.

CClear

Format

CClear q(uarter), m(ask)

31 25	24 20	19 18	17 15	14 12	11 7	6 0
0x7f	0xe	q	$\mathrm{m}_{[7:5]}$	0x0	$\mathrm{m}_{[4:0]}$	0x5b

Description

Capability registers $8 \times q + i$ are each set to **NULL** if the ith bit of m is set, with the exception that the 0th bit of m refers to **DDC** when q is 0, rather than **C0**.

Semantics

```
let q_u = unsigned(q);
foreach (i from 0 to 7)
  if m[i] == bitone then
    if q_u == 0 & i == 0 then
      DDC = null_cap
    else
      C(8 * q_u + i) = null_cap;
RETIRE_SUCCESS
```

Notes

- This instruction is designed to accelerate the register clearing that is required for secure domain transitions. It is expected that it can be implemented efficiently in hardware using a single ‘valid’ bit per register that is cleared by this instruction and set on any subsequent write to the register.

CClearTag

Format

CClearTag cd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0xb	cs1	0x0	cd	0x5b

Description

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

Semantics

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

CCopyType

Format

CCopyType cd, cs1, cs2

31 25	24 20	19 15	14 12	11 7	6 0
0x1e	cs2	cs1	0x0	cd	0x5b

Description

Capability register *cd* is replaced with the contents of capability register *cs1* with the **address** set to *cs2*.**otype** and the **tag** field cleared if *cs2* has a reserved **otype** or if *cs1* was sealed.

Semantics

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

let reserved = hasReservedOType(cs2_val);
let otype : xlenbits = if reserved then EXTS(cs2_val.otype)
                                   else EXTZ(cs2_val.otype);

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

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

Notes

- Reserved otypes always result in untagged capabilities, as, at the moment, all reserved otypes are constructed using ambiently-available actions. `CCSeal` knows how to work with these.

CCSeal

Format

CCSeal cd, cs1, cs2

31 25	24 20	19 15	14 12	11 7	6 0
0x1f	cs2	cs1	0x0	cd	0x5b

Description

Capability register *cd* is replaced with capability register *cs1*, and is conditionally sealed with **otype** equal to the **address** field of capability register *cs2*. The conditions under which the input is passed through unaltered are intended to permit a fast branchless rederivation sequence with multiple sealing authorities with a single `CBuildCap` and a set of `CCopyType` and `CCSeal` pairs when swapping capabilities in from disk. Other than these conditions, if *cs2* is unable to authorize the sealing, the **tag** field of *cd* is cleared.

Semantics

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

let cs2_cursor = getCapCursor(cs2_val);
let (cs2_base, cs2_top) = getCapBounds(cs2_val);
let passthrough = not(cs2_val.tag)
                | isCapSealed(cs1_val)
                | (cs2_cursor < cs2_base)
                | (cs2_cursor >= cs2_top)
                | (signed(cs2_val.address) == otype_unsealed);
if passthrough then {
  C(cd) = cs1_val;
  RETIRE_SUCCESS
} else {
  let permitted = not(isCapSealed(cs2_val))
                & cs2_val.permit_seal
                & (cs2_cursor <= cap_max_otype);
  let newCap = sealCap(cs1_val, to_bits(cap_otype_width, cs2_cursor));
  C(cd) = clearTagIf(newCap, not(permitted));
  RETIRE_SUCCESS
}
```

Notes

- The intent is that this is used for rederiving swapped-out capabilities, so the expectation is that this whole sequence is guarded by a check on whether the **tag** field of the capability was valid.

- If the input to be conditionally sealed is already sealed it is passed through before any futher checks are made. This allows multiple `CCSeal`s in a chain, any of which can be the one to seal the initial input. The intent is that all of these `CCSeal`s' authorities will have been produced by `CCopyType`s of the same input (i.e., they will all attempt to seal to the same type), but that's not, strictly, required. Sealed capabilities with a reserved **otype** are also constructed directly by `CBuildCap`.

- To avoid the need to branch on whether the original capability was sealed, attempts to seal with the reserved unsealed **otype** will leave the capability unmodified rather than trap.

- To avoid the need to check which is the correct authority, any sealing request where the **address** of capability register *cs2* is out of bounds will leave the capability unmodified rather than trap, as will attempts to seal with an invalid capability since it may have become unrepresentable but be within its reinterpreted bounds.

CFromPtr

Format

CFromPtr cd, cs1, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0x13	rs2	cs1	0x0	cd	0x5b

Description

If the value of integer register *rs2* is 0 then capability register *cd* is set to **NULL**. Otherwise capability register *cd* is set to capability register *cs1* with its **offset** replaced with *rs2*. If the resulting capability cannot be represented exactly, or if *cs1* was sealed, then *cd*.**tag** is cleared. The remaining capability fields are set to what the in-memory representation of *cs1* with the address set to *cd*.**address** decodes to.

Semantics

```
let cs1_val = if unsigned(cs1) == 0 then DDC else C(cs1);
let rs2_val = X(rs2);

if rs2_val == zeros() then {
  C(cd) = null_cap;
  RETIRE_SUCCESS
} else {
  let inCap = clearTagIfSealed(cs1_val);
  let (success, newCap) = setCapOffset(inCap, rs2_val);
  C(cd) = clearTagIf(newCap, not(success));
  RETIRE_SUCCESS
}
```

Deprecated

This instruction is deprecated and may be removed in a future version.

CGetBase

Format

CGetBase rd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x2	cs1	0x0	rd	0x5b

Description

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

Semantics

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

CGetFlags

Format

CGetFlags rd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x7	cs1	0x0	rd	0x5b

Description

Integer register *rd* is set equal to the zero-extended **flags** field of capability register *cs1*.

Semantics

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

CGetHigh

Format

CGetHigh rd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x17	cs1	0x0	rd	0x5b

Description

Integer register *rd* is set equal to the **high half** of capability register *cs1*.

The bits returned here are of the **in-memory** form of the capability, which may differ from microarchitectural forms in use within implementations. (Notably, in the sail implementation, see the distinction between `capToBits` and `capToMemBits`.) That is, applying `CGetHigh` to a capability loaded from address *m* will yield the same result as loading the high half of the capability-sized granule at *m* (that is, bits above **XLEN** when a capability is interpreted as a twice-**XLEN**-bit integer).

Semantics

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

CGetLen

Format

CGetLen rd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x3	cs1	0x0	rd	0x5b

Description

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

Semantics

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

Notes

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

CGetOffset

Format

CGetOffset rd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x6	cs1	0x0	rd	0x5b

Description

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

Semantics

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

CGetPerm

Format

CGetPerm rd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x0	cs1	0x0	rd	0x5b

Description

The least significant `cap_hperms_width` bits of integer register *rd* are set equal to the **perms** field of capability register *cs1*; bits `cap_uperms_shift` to `cap_uperms_shift`+`cap_uperms_width`-1 of *rd* are set equal to the **uperms** field of *cs1*. The other bits of *rd* are set to zero.

Semantics

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

CGetSealed

Format

CGetSealed rd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x5	cs1	0x0	rd	0x5b

Description

The low bit of integer register *rd* is set to 0 if *cs1* is unsealed and to 1 otherwise. All other bits of *rd* are cleared.

Semantics

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

CGetTag

Format

CGetTag rd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x4	cs1	0x0	rd	0x5b

Description

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

Semantics

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

CGetTop

Format

CGetTop rd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x18	cs1	0x0	rd	0x5b

Description

Integer register *rd* is set equal to the **top** field (i.e. one past the last addressable byte) of capability register *cs1*.

Semantics

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

Notes

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

CGetType

Format

CGetType rd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x1	cs1	0x0	rd	0x5b

Description

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

Semantics

```
let capVal = C(cs1);
X(rd) = if   hasReservedOType(capVal)
        then EXTS(capVal.otype)
        else EXTZ(capVal.otype);
RETIRE_SUCCESS
```

CIncOffset

Format

CIncOffset cd, cs1, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0x11	rs2	cs1	0x0	cd	0x5b

Description

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

Semantics

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

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

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

CIncOffsetImm

Format

CIncOffsetImm cd, cs1, imm

31 20	19 15	14 12	11 7	6 0
imm[11:0]	cs1	0x1	cd	0x5b

Description

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

Semantics

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

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

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

CInvoke

Format

CInvoke cs1, cs2

31 25	24 20	19 15	14 12	11 7	6 0
0x7e	cs2	cs1	0x0	0x1	0x5b

Description

PCC is set equal to capability register *cs1* and unsealed with the 0th bit of its **address** set to 0, whilst **C31** is set equal to capability register *cs2* and unsealed. This provides a constrained form of non-monotonicity, allowing for fast jumps between protection domains, with *cs1* providing the target domain's code and *cs2* providing the target domain's data. The capabilities must have a matching **otype** to ensure the right data is provided for the given jump target.

Semantics

```
let cs1_val = C(cs1);
let cs2_val = C(cs2);
let newPC = [cs1_val.address with 0 = bitzero]; /* clear bit zero as for RISCV JALR
    */
let newPCCBase = getCapBaseBits(cs1_val);
if not(cs1_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cs1);
  RETIRE_FAIL
} else if not(cs2_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cs2);
  RETIRE_FAIL
} else if hasReservedOType(cs1_val) then {
  handle_cheri_reg_exception(CapEx_SealViolation, cs1);
  RETIRE_FAIL
} else if hasReservedOType(cs2_val) then {
  handle_cheri_reg_exception(CapEx_SealViolation, cs2);
  RETIRE_FAIL
} else if cs1_val.otype != cs2_val.otype then {
  handle_cheri_reg_exception(CapEx_TypeViolation, cs1);
  RETIRE_FAIL
} else if not(cs1_val.permit_cinvoke) then {
  handle_cheri_reg_exception(CapEx_PermitCInvokeViolation, cs1);
  RETIRE_FAIL
} else if not(cs2_val.permit_cinvoke) then {
  handle_cheri_reg_exception(CapEx_PermitCInvokeViolation, cs2);
  RETIRE_FAIL
} else if not(cs1_val.permit_execute) then {
  handle_cheri_reg_exception(CapEx_PermitExecuteViolation, cs1);
```

```
  RETIRE_FAIL
} else if cs2_val.permit_execute then {
  handle_cheri_reg_exception(CapEx_PermitExecuteViolation, cs2);
  RETIRE_FAIL
} else if have_pcc_relocation() & (newPCCBase[0] == bitone | (newPCCBase[1] ==
    bitone & ~(haveRVC()))) then {
  handle_cheri_reg_exception(CapEx_UnalignedBase, cs1);
  RETIRE_FAIL
} else if newPC[1] == bitone & ~(haveRVC()) then {
  handle_mem_exception(newPC,  E_Fetch_Addr_Align());
  RETIRE_FAIL
} else if not(inCapBounds(cs1_val, newPC, min_instruction_bytes())) then {
  handle_cheri_reg_exception(CapEx_LengthViolation, cs1);
  RETIRE_FAIL
} else {
  C(31) = unsealCap(cs2_val);
  nextPC = newPC;
  nextPCC = unsealCap(cs1_val);
  RETIRE_SUCCESS
}
```

Exceptions

An exception is raised if:

- *cs1*.**tag** is not set.
- *cs2*.**tag** is not set.
- *cs1*.**otype** is reserved.
- *cs2*.**otype** is reserved.
- *cs1*.**otype** $\neq$ *cs2*.**otype**.
- *cs1*.**perms** does not grant PERMIT_INVOKE.
- *cs2*.**perms** does not grant PERMIT_INVOKE.
- *cs1*.**perms** does not grant PERMIT_EXECUTE.
- *cs2*.**perms** grants PERMIT_EXECUTE.
- *cs1*.**address** $<$ *cs1*.**base**.
- *cs1*.**address** $+$ min_instruction_bytes $>$ *cs1*.**top**.
- *cs1*.**base** is unaligned (only possible if PCC relocation is enabled).
- *cs1*.**address** is unaligned, ignoring bit 0.

Notes

- From the point of view of security, this 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.

CJAL

Format

CJAL cd, imm *(capability mode)*

31	30 21	20	19 12	11 7	6 0
i[20]	imm$_{[10:1]}$	i[11]	imm$_{[19:12]}$	cd	0x6f

Description

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

Semantics

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

Exceptions

An exception is raised if:

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

CJALR

Format

CJALR cd, cs1, imm *(capability mode)*

31 20	19 15	14 12	11 7	6 0
imm[11:0]	cs1	0	cd	0x67

Description

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

Semantics

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

```
  C(cd) = sealCap(linkCap, to_bits(cap_otype_width, otype_sentry));
  nextPC = newPC;
  nextPCC = unsealCap(cs1_val);
  RETIRE_SUCCESS
}
```

Exceptions

An exception is raised if:

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

[C]LC

Format

LC cd, rs1, imm *(RV32, integer mode)*
CLC cd, cs1, imm *(RV32, capability mode)*

31 20	19 15	14 12	11 7	6 0
imm	rs1/cs1	0x3	cd	0x3

LC cd, rs1, imm *(RV64, integer mode)*
CLC cd, cs1, imm *(RV64, capability mode)*

31 20	19 15	14 12	11 7	6 0
imm	rs1/cs1	0x2	cd	0xf

Description

In integer mode, capability register *cd* is replaced with the capability located in memory at **DDC**.**address** + *rs1* + *imm*, and if **DDC**.**perms** does not grant Permit_Load_Capability then *cd*.**tag** is cleared. In capability mode, capability register *cd* is replaced with the capability located in memory at *cs1*.**address** + *imm*, and if *cs1*.**perms** does not grant Permit_Load_Capability then *cd*.**tag** is cleared.

Semantics

```
let offset : xlenbits = EXTS(imm);
let (auth_val, vaddr, auth_idx) = get_cheri_mode_cap_addr(rs1_cs1, offset);
handle_load_cap_via_cap(cd, auth_idx, auth_val, vaddr)
```

Exceptions

In integer mode, an exception is raised if:

- **DDC**.**tag** is not set.
- **DDC** is sealed.
- **DDC**.**perms** does not grant Permit_Load.
- **DDC**.**address** + *rs1* + *imm* < **DDC**.**base**.
- **DDC**.**address** + *rs1* + *imm* + **CLEN** / 8 > **DDC**.**top**.
- **DDC**.**address** + *rs1* + *imm* is unaligned, regardless of whether the implementation supports unaligned data accesses.

In capability mode, an exception is raised if:

- *cs1*.**tag** is not set.
- *cs1* is sealed.
- *cs1*.**perms** does not grant PERMIT_LOAD.
- *cs1*.**address** + *imm* < *cs1*.**base**.
- *cs1*.**address** + *imm* + **CLEN** / 8 > *cs1*.**top**.
- *cs1*.**address** + *imm* is unaligned, regardless of whether the implementation supports unaligned data accesses.

CLoadTags

Format

CLoadTags rd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x12	cs1	0x0	rd	0x5b

Description

Integer register *rd* is replaced with the tags of the capabilities located in memory at and above *cs1*.**address**. The 0th bit corresponds to the first capability in memory. The result is coherent with other processors, as if the corresponding data words had also been loaded. The number of tags loaded is implementation-defined; typical implementations are expected to return the tags held in an L1 cache line, and so we use the constant `caps_per_cache_line`. The number of tags loaded must be a power of two, at least 1, and no more than **XLEN**.

Semantics

```
let cs1_val = C(cs1);
let vaddr = cs1_val.address;
let aq : bool = false;
let rl : bool = false;
if not(cs1_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cs1);
  RETIRE_FAIL
} else if isCapSealed(cs1_val) then {
  handle_cheri_reg_exception(CapEx_SealViolation, cs1);
  RETIRE_FAIL
} else if not(cs1_val.permit_load) then {
  handle_cheri_reg_exception(CapEx_PermitLoadViolation, cs1);
  RETIRE_FAIL
} else if not(cs1_val.permit_load_cap) then {
  handle_cheri_reg_exception(CapEx_PermitLoadCapViolation, cs1);
  RETIRE_FAIL
} else if not(inCapBounds(cs1_val, vaddr, caps_per_cache_line * cap_size)) then {
  handle_cheri_reg_exception(CapEx_LengthViolation, cs1);
  RETIRE_FAIL
} else if not(unsigned(vaddr) % (caps_per_cache_line * cap_size) == 0) then {
  handle_mem_exception(vaddr, E_Load_Addr_Align());
  RETIRE_FAIL
} else match translateAddr(vaddr, Read(Cap)) {
  TR_Failure(E_Extension(_), _) => { internal_error(__FILE__, __LINE__, "unexpected
       cheri exception for tags load") },
  TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
  TR_Address(addr, ptw_info) => {
```

```
    if ptw_info.ptw_lc != PTW_LC_OK then {
      handle_mem_exception(vaddr, E_Extension(EXC_LOAD_CAP_PAGE_FAULT));
      RETIRE_FAIL
    } else {
      mtags : MemoryOpResult(bits(caps_per_cache_line)) = MemValue(zeros());

      foreach (i from 0 to (caps_per_cache_line - 1)) {
        match mtags {
          MemException(_) => (),
          MemValue(tags) => {
            match mem_read_cap(addr + i * cap_size, aq, aq & rl, false) {
              MemException(e) => mtags = MemException(e),
              MemValue(v) =>
                mtags = MemValue([tags with i = bool_to_bit(v.tag)])
            }
          }
        }
      };

      match mtags {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(v) => { X(rd) = EXTZ(v); RETIRE_SUCCESS }
      }
    }
  }
}
```

Exceptions

An exception is raised if:

- *cs1*.**tag** is not set.
- *cs1* is sealed.
- *cs1*.**perms** does not grant both PERMIT_LOAD and PERMIT_LOAD_CAPABILITY.
- *cs1*.**address** $<$ *cs1*.**base**.
- *cs1*.**address** $+$ `caps_per_cache_line` $\times$ **CLEN** $/ 8 >$ *cs1*.**top**.
- *cs1*.**address** is unaligned.
- The page table entry for *cs1*.**address** would cause the tag to be cleared.

Notes

- In order to reduce DRAM traffic, implementations may choose to load only the tags and not the corresponding data, and may wish to not evict other cache lines by treating it as a non-temporal/streaming load.
- Software can easily discover the number of tags loaded by an implementation by storing a series of **XLEN** capabilities to an aligned array and performing a `CLoadTags` operation. This need only be done once.
- For heterogeneous multi-core or multi-processor systems, all cores must return the same number of tags, which will often be based on the smallest cache line size in the system.
- Unlike CLC, this instruction traps if tags will always be unset due to lacking PERMIT_LOAD_CAPABILITY or page table entry permissions, since that is likely indicative of a software bug that could lead to temporal safety vulnerabilities if capabilities are erroneously missed.

CMove

Format

CMove cd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0xa	cs1	0x0	cd	0x5b

Description

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

Semantics

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

CRepresentableAlignmentMask

Format

CRepresentableAlignmentMask rd, rs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x9	rs1	0x0	rd	0x5b

Description

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

Semantics

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

CRoundRepresentableLength

Format

CRoundRepresentableLength rd, rs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x8	rs1	0x0	rd	0x5b

Description

Integer register *rd* is set to the smallest value greater or equal to *rs1* that can be used as a length to set exact bounds on a capability that has a suitably aligned base (as obtained with the help of `CRAM`).

Semantics

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

[C]SC

Format

SC cs2, rs1, imm *(RV32, integer mode)*
CSC cs2, cs1, imm *(RV32, capability mode)*

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	cs2	rs1/cs1	0x3	imm[4:0]	0x23

SC cs2, rs1, imm *(RV64, integer mode)*
CSC cs2, cs1, imm *(RV64, capability mode)*

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	cs2	rs1/cs1	0x4	imm[4:0]	0x23

Description

In integer mode, the capability located in memory at **DDC**.**address** + *rs1* + *imm* is replaced with capability register *cs2*. In capability mode, the capability located in memory at *cs1*.**address** + *imm* is replaced with capability register *cs2*.

Semantics

```
let offset : xlenbits = EXTS(imm);
let (auth_val, vaddr, auth_idx) = get_cheri_mode_cap_addr(rs1_cs1, offset);
handle_store_cap_via_cap(cs2, auth_idx, auth_val, vaddr)
```

Exceptions

In integer mode, an exception is raised if:

- **DDC**.**tag** is not set.
- **DDC** is sealed.
- **DDC**.**perms** does not grant PERMIT_STORE.
- **DDC**.**perms** does not grant PERMIT_STORE_CAPABILITY and *cs2*.**tag** is set.
- **DDC**.**perms** does not grant PERMIT_STORE_LOCAL_CAPABILITY, *cs2*.**tag** is set and *cs2*.**perms** does not grant GLOBAL.
- **DDC**.**address** + *rs1* + *imm* < **DDC**.**base**.
- **DDC**.**address** + *rs1* + *imm* + **CLEN** / 8 > **DDC**.**top**.

In capability mode, an exception is raised if:

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

CSeal

Format

CSeal cd, cs1, cs2

31 25	24 20	19 15	14 12	11 7	6 0
0xb	cs2	cs1	0x0	cd	0x5b

Description

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

Semantics

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

let cs2_cursor = getCapCursor(cs2_val);
let (cs2_base, cs2_top) = getCapBounds(cs2_val);

let permitted = cs2_val.tag
              & not(isCapSealed(cs2_val))
              & cs2_val.permit_seal
              & (cs2_cursor >= cs2_base)
              & (cs2_cursor < cs2_top)
              & (cs2_cursor <= cap_max_otype);

let inCap = clearTagIfSealed(cs1_val);
let newCap = sealCap(inCap, to_bits(cap_otype_width, cs2_cursor));

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

CSealEntry

Format

CSealEntry cd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x11	cs1	0x0	cd	0x5b

Description

Capability register *cd* is replaced with capability register *cs1* and sealed as a sentry.

Semantics

```
let cs1_val = C(cs1);
let inCap = clearTagIfSealed(cs1_val);
C(cd) = sealCap(inCap, to_bits(cap_otype_width, otype_sentry));
RETIRE_SUCCESS
```

CSetAddr

Format

CSetAddr cd, cs1, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0x10	rs2	cs1	0x0	cd	0x5b

Description

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

Semantics

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

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

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

CSetBounds

Format

CSetBounds cd, cs1, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0x8	rs2	cs1	0x0	cd	0x5b

Description

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

Semantics

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

let newBase = cs1_val.address;
let newTop : CapLenBits = EXTZ(newBase) + EXTZ(rs2_val);
let inBounds = inCapBounds(cs1_val, newBase, unsigned(rs2_val));

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

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

CSetBoundsExact

Format

CSetBoundsExact cd, cs1, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0x9	rs2	cs1	0x0	cd	0x5b

Description

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

Semantics

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

let newBase = cs1_val.address;
let newTop : CapLenBits = EXTZ(newBase) + EXTZ(rs2_val);
let inBounds = inCapBounds(cs1_val, newBase, unsigned(rs2_val));

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

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

CSetBoundsImm

Format

CSetBoundsImm cd, cs1, uimm

31 20	19 15	14 12	11 7	6 0
uimm[11:0]	cs1	0x2	cd	0x5b

Description

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

Semantics

```
let cs1_val = C(cs1);

let newBase = cs1_val.address;
let newTop : CapLenBits = EXTZ(newBase) + EXTZ(uimm);
let inBounds = inCapBounds(cs1_val, newBase, unsigned(uimm));

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

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

CSetEqualExact

Format

CSetEqualExact rd, cs1, cs2

31 25	24 20	19 15	14 12	11 7	6 0
0x21	cs2	cs1	0x0	rd	0x5b

Description

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

Semantics

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

CSetFlags

Format

CSetFlags cd, cs1, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0xe	rs2	cs1	0x0	cd	0x5b

Description

Capability register *cd* is replaced with the contents of capability register *cs1* with the **flags** field set to bits 0 to `cap_flags_width`-1 of integer register *rs2*. If *cs1* was sealed then *cd*.**tag** is cleared.

Semantics

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

let inCap = clearTagIfSealed(cs1_val);
let newCap = setCapFlags(inCap, truncate(rs2_val, cap_flags_width));

C(cd) = newCap;
RETIRE_SUCCESS
```

CSetHigh

Format

CSetHigh cd, cs1, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0x16	rs2	cs1	0x0	cd	0x5b

Description

Capability register *cd* comes to hold the capability from *cs1* with its high bits replaced with the value in the integer register *rs2*. The tag of *cd* is cleared.

rs2 holds the **in-memory** form of capability bits. That is, this instruction yields the same result as writing *cs1* out to memory, overwriting the high word with *rs2*, and loading that capability-sized granule into *cd*, although without the memory mutation side-effects.

Semantics

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

CSetOffset

Format

CSetOffset cd, cs1, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0xf	rs2	cs1	0x0	cd	0x5b

Description

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

Semantics

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

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

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

CSpecialRW

Format

CSpecialRW cd, scr, cs1

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

Description

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

Semantics

```
let (specialExists, ro, priv, needASR) : (bool, bool, Privilege, bool) = match
    unsigned(scr) {
  0  => (true, true,  User, false),
  1  => (true, false, User, false),
  4 if haveNExt() => (true, false, User, true),
  5 if haveNExt() => (true, false, User, true),
  6 if haveNExt() => (true, false, User, true),
  7 if haveNExt() => (true, false, User, true),
  12 if haveSupMode() => (true, false, Supervisor, true),
  13 if haveSupMode() => (true, false, Supervisor, true),
  14 if haveSupMode() => (true, false, Supervisor, true),
  15 if haveSupMode() => (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) |
    ro & cs1 != zeros() |
    (privLevel_to_bits(cur_privilege) <_u privLevel_to_bits(priv))) then {
  handle_illegal();
  RETIRE_FAIL
} else if (needASR & not(pcc_access_system_regs())) then {
  handle_cheri_cap_exception(CapEx_AccessSystemRegsViolation, 0b1 @ scr);
  RETIRE_FAIL
} else {
  let cs1_val = C(cs1);
  C(cd) = match unsigned(scr) {
    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  => legalize_epcc(UEPCC),
    12 => STCC,
    13 => STDC,
    14 => SScratchC,
    15 => legalize_epcc(SEPCC),
    28 => MTCC,
    29 => MTDC,
    30 => MScratchC,
    31 => legalize_epcc(MEPCC),
    _  => {assert(false, "unreachable"); undefined}
  };
  if (cs1 != zeros()) then {
    match unsigned(scr) {
      1  => DDC = cs1_val,
      4  => UTCC = legalize_tcc(UTCC, cs1_val),
      5  => UTDC = cs1_val,
      6  => UScratchC = cs1_val,
      7  => UEPCC = cs1_val,
      12 => STCC = legalize_tcc(STCC, cs1_val),
      13 => STDC = cs1_val,
      14 => SScratchC = cs1_val,
      15 => SEPCC = cs1_val,
      28 => MTCC = legalize_tcc(MTCC, cs1_val),
      29 => MTDC = cs1_val,
      30 => MScratchC = cs1_val,
      31 => MEPCC = cs1_val,
      _  => assert(false, "unreachable")
    };
    if get_config_print_reg() then
      print_reg(scr_name_map(scr) ^ " <- " ^ RegStr(cs1_val));
  };
  RETIRE_SUCCESS
}
```

Exceptions

An exception is raised if:

- *scr* does not exist.
- *scr* is read-only and *cs1* is not **C0**.

- *scr* is only accessible to a higher privilege mode.
- *scr* requires PERMIT_ACCESS_SYSTEM_REGISTERS and that is not granted by **PCC**.**perms**.

Notes

- Writing **NULL** to a special capability register cannot be done with **C0** as that only performs a read. An alternative implementation would allocate a separate two-operand CSpecialR instruction and interpret *cs1* being **C0** as a write of **NULL** if the need to use a temporary capability register proves to be overly problematic for software. For U-mode transitions to domains without PERMIT_ACCESS_SYSTEM_REGISTERS only **DDC** should need clearing, which can be done with `CClear`.

CTestSubset

Format

CTestSubset rd, cs1, cs2

31 25	24 20	19 15	14 12	11 7	6 0
0x20	cs2	cs1	0x0	rd	0x5b

Description

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

Semantics

```
let cs1_val = if unsigned(cs1) == 0 then DDC else C(cs1);
let cs2_val = C(cs2);

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

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

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

Notes

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

CToPtr

Format

CToPtr rd, cs1, cs2

31 25	24 20	19 15	14 12	11 7	6 0
0x12	cs2	cs1	0x0	rd	0x5b

Description

If the **tag** field of capability register *cs1* is not set, integer register *rd* is set to 0, otherwise integer register *rd* is set to *cs1*.**address** − *cs2*.**base**.

Semantics

```
let cs2_val = if unsigned(cs2) == 0 then DDC else C(cs2);
let cs1_val = C(cs1);

/* Note: returning zero for untagged values breaks magic constants such as SIG_IGN
    */
X(rd) = if not(cs1_val.tag) then
          zeros()
        else
          cs1_val.address - getCapBaseBits(cs2_val);
RETIRE_SUCCESS
```

Notes

- *cs2* being sealed will not set *rd* to 0. This is for further study.

Deprecated

This instruction is deprecated and may be removed in a future version.

CUnseal

Format

CUnseal cd, cs1, cs2

31 25	24 20	19 15	14 12	11 7	6 0
0xc	cs2	cs1	0x0	cd	0x5b

Description

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

Semantics

```
let cs1_val = C(cs1);
let cs2_val = C(cs2);
let cs2_cursor = getCapCursor(cs2_val);
let (cs2_base, cs2_top) = getCapBounds(cs2_val);
let permitted = cs2_val.tag
              & isCapSealed(cs1_val)
              & not(isCapSealed(cs2_val))
              & not(hasReservedOType(cs1_val))
              & (cs2_cursor == unsigned(cs1_val.otype))
              & cs2_val.permit_unseal
              & (cs2_cursor >= cs2_base)
              & (cs2_cursor < cs2_top);
let new_global = cs1_val.global & cs2_val.global;
let newCap = {unsealCap(cs1_val) with global=new_global};
C(cd) = clearTagIf(newCap, not(permitted));
RETIRE_SUCCESS
```

FPClear

Format

FPClear q(uarter), m(ask)

31 25	24 20	19 18	17 15	14 12	11 7	6 0
0x7f	0x10	q	$m_{[7:5]}$	0x0	$m_{[4:0]}$	0x5b

Description

Floating-point registers $8 \times q + i$ are each set to 0 if the ith bit of m is set.

Semantics

```
if haveFExt() then {
  foreach (i from 0 to 7)
    if m[i] == bitone then
      F(8 * unsigned(q) + i) = zeros();
  RETIRE_SUCCESS
} else {
  handle_illegal();
  RETIRE_FAIL
}
```

Notes

- This instruction is designed to accelerate the register clearing that is required for secure domain transitions. It is expected that it can be implemented efficiently in hardware using a single ‘valid’ bit per register that is cleared by this instruction and set on any subsequent write to the register.
- The 0 value written is FLEN bits wide, the largest supported by the implementation, such that the in-memory representation of the register is 0, rather than a NaN-boxed narrower value.

JALR.CAP

Format

JALR.CAP cd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0xc	cs1	0x0	cd	0x5b

Description

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

Semantics

```
execute(CJALR(zeros(), cs1, cd))
```

Exceptions

An exception is raised if:

- *cs1*.**tag** is not set.
- *cs1* is sealed and is not a sentry.
- *cs1*.**perms** does not grant PERMIT_EXECUTE.
- *cs1*.**address** < *cs1*.**base**.
- *cs1*.**address** + min_instruction_bytes > *cs1*.**top**.
- *cs1*.**base** is unaligned (only possible if PCC relocation is enabled).
- *cs1*.**address** is unaligned, ignoring bit 0.

JALR.PCC

Format

JALR.PCC rd, rs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x14	rs1	0x0	rd	0x5b

Description

Integer register *rd* is replaced with the next instruction's **PCC.offset**. **PCC.offset** is replaced with the value of register *rs1* with the 0th bit set to 0.

Semantics

```
execute(RISCV_JALR(zeros(), rs1, rd))
```

Exceptions

An exception is raised if:

- **PCC.address** + *rs1* < **PCC.base**.
- **PCC.address** + *rs1* + min_instruction_bytes > **PCC.top**.
- **PCC.address** + *rs1* is unaligned, ignoring bit 0.

L[BHWD][U].CAP

Format

LB.CAP rd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x08	cs1	0x0	rd	0x5b

LH.CAP rd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x09	cs1	0x0	rd	0x5b

LW.CAP rd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x0a	cs1	0x0	rd	0x5b

LD.CAP rd, cs1 *(RV64/128)*

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x0b	cs1	0x0	rd	0x5b

LBU.CAP rd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x0c	cs1	0x0	rd	0x5b

LHU.CAP rd, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x0d	cs1	0x0	rd	0x5b

LWU.CAP rd, cs1 *(RV64/128)*

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x0e	cs1	0x0	rd	0x5b

Description

Integer register *rd* is replaced with the signed or unsigned byte, halfword, word or doubleword located in memory at *cs1*.**address**.

Semantics

```
let cs1_val = C(cs1);
let vaddr = cs1_val.address;
handle_load_data_via_cap(rd, 0b0 @ cs1, cs1_val, vaddr, is_unsigned, width)
```

Exceptions

An exception is raised if:

- *cs1*.**tag** is not set.
- *cs1* is sealed.
- *cs1*.**perms** does not grant PERMIT_LOAD.
- *cs1*.**address** $<$ *cs1*.**base**.
- *cs1*.**address** $+$ *size* $>$ *cs1*.**top**.

L[BHWD][U].DDC

Format

LB.DDC rd, rs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x00	rs1	0x0	rd	0x5b

LH.DDC rd, rs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x01	rs1	0x0	rd	0x5b

LW.DDC rd, rs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x02	rs1	0x0	rd	0x5b

LD.DDC rd, rs1 *(RV64/128)*

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x03	rs1	0x0	rd	0x5b

LBU.DDC rd, rs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x04	rs1	0x0	rd	0x5b

LHU.DDC rd, rs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x05	rs1	0x0	rd	0x5b

LWU.DDC rd, rs1 *(RV64/128)*

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x06	rs1	0x0	rd	0x5b

Description

Integer register *rd* is replaced with the signed or unsigned byte, halfword, word or doubleword located in memory at **DDC.address** $+$ *rs1*.

Semantics

```
let (ddc_val, vaddr) = ddc_and_resulting_addr(X(rs1));
handle_load_data_via_cap(rd, DDC_IDX, ddc_val, vaddr, is_unsigned, width)
```

Exceptions

An exception is raised if:

- **DDC**.**tag** is not set.
- **DDC** is sealed.
- **DDC**.**perms** does not grant PERMIT_LOAD.
- **DDC**.**address** $+$ *rs1* $<$ **DDC**.**base**.
- **DDC**.**address** $+$ *rs1* $+$ *size* $>$ **DDC**.**top**.

LC.CAP

Format

LC.CAP cd, cs1 *(RV32)*

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x0b	cs1	0x0	cd	0x5b

LC.CAP cd, cs1 *(RV64)*

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x1f	cs1	0x0	cd	0x5b

Description

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

Semantics

```
let cs1_val = C(cs1);
let vaddr = cs1_val.address;
handle_load_cap_via_cap(cd, 0b0 @ cs1, cs1_val, vaddr)
```

Exceptions

An exception is raised if:

- *cs1*.**tag** is not set.
- *cs1* is sealed.
- *cs1*.**perms** does not grant PERMIT_LOAD.
- *cs1*.**address** $<$ *cs1*.**base**.
- *cs1*.**address** $+$ **CLEN** $/$ 8 $>$ *cs1*.**top**.
- *cs1*.**address** is unaligned, regardless of whether the implementation supports unaligned data accesses.

LC.DDC

Format

LC.DDC cd, rs1 *(RV32)*

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x03	rs1	0x0	cd	0x5b

LC.DDC cd, rs1 *(RV64)*

31 25	24 20	19 15	14 12	11 7	6 0
0x7d	0x17	rs1	0x0	cd	0x5b

Description

Capability register *cd* is replaced with the capability located in memory at **DDC**.**address** + *rs1*, and if **DDC**.**perms** does not grant PERMIT_LOAD_CAPABILITY then *cd*.**tag** is cleared.

Semantics

```
let (ddc_val, vaddr) = ddc_and_resulting_addr(X(rs1));
handle_load_cap_via_cap(cd, DDC_IDX, ddc_val, vaddr)
```

Exceptions

An exception is raised if:

- **DDC**.**tag** is not set.
- **DDC** is sealed.
- **DDC**.**perms** does not grant PERMIT_LOAD.
- **DDC**.**address** $+$ *rs1* $<$ **DDC**.**base**.
- **DDC**.**address** $+$ *rs1* $+$ **CLEN** $/ 8 >$ **DDC**.**top**.
- **DDC**.**address** $+$ *rs1* is unaligned, regardless of whether the implementation supports unaligned data accesses.

S[BHWD].CAP

Format

SB.CAP rs2, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7c	rs2	cs1	0x0	0x08	0x5b

SH.CAP rs2, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7c	rs2	cs1	0x0	0x09	0x5b

SW.CAP rs2, cs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7c	rs2	cs1	0x0	0x0a	0x5b

SD.CAP rs2, cs1 *(RV64/128)*

31 25	24 20	19 15	14 12	11 7	6 0
0x7c	rs2	cs1	0x0	0x0b	0x5b

Description

The byte, halfword, word or doubleword located in memory at *cs1*.**address** is replaced with integer register *rs2*.

Semantics

```
let cs1_val = C(cs1);
let vaddr = cs1_val.address;
handle_store_data_via_cap(rs2, 0b0 @ cs1, cs1_val, vaddr, width)
```

Exceptions

An exception is raised if:

- *cs1*.**tag** is not set.
- *cs1* is sealed.
- *cs1*.**perms** does not grant PERMIT_STORE.
- *cs1*.**address** $<$ *cs1*.**base**.
- *cs1*.**address** $+$ *size* $>$ *cs1*.**top**.

S[BHWD].DDC

Format

SB.DDC rs2, rs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7c	rs2	rs1	0x0	0x00	0x5b

SH.DDC rs2, rs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7c	rs2	rs1	0x0	0x01	0x5b

SW.DDC rs2, rs1

31 25	24 20	19 15	14 12	11 7	6 0
0x7c	rs2	rs1	0x0	0x02	0x5b

SD.DDC rs2, rs1 *(RV64/128)*

31 25	24 20	19 15	14 12	11 7	6 0
0x7c	rs2	rs1	0x0	0x03	0x5b

Description

The byte, halfword, word or doubleword located in memory at **DDC**.**address** $+$ *rs1* is replaced with integer register *rs2*.

Semantics

```
let (ddc_val, vaddr) = ddc_and_resulting_addr(X(rs1));
handle_store_data_via_cap(rs2, DDC_IDX, ddc_val, vaddr, width)
```

Exceptions

An exception is raised if:

- **DDC**.**tag** is not set.
- **DDC** is sealed.
- **DDC**.**perms** does not grant PERMIT_STORE.
- **DDC**.**address** $+$ *rs1* $<$ **DDC**.**base**.
- **DDC**.**address** $+$ *rs1* $+$ *size* $>$ **DDC**.**top**.

SC.CAP

Format

SC.CAP cs2, cs1 *(RV32)*

31 25	24 20	19 15	14 12	11 7	6 0
0x7c	cs2	cs1	0x0	0x0b	0x5b

SC.CAP cs2, cs1 *(RV64)*

31 25	24 20	19 15	14 12	11 7	6 0
0x7c	cs2	cs1	0x0	0x0c	0x5b

Description

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

Semantics

```
let cs1_val = C(cs1);
let vaddr = cs1_val.address;
handle_store_cap_via_cap(cs2, 0b0 @ cs1, cs1_val, vaddr)
```

Exceptions

An exception is raised if:

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

SC.DDC

Format

SC.DDC cs2, rs1 *(RV32)*

31 25	24 20	19 15	14 12	11 7	6 0
0x7c	cs2	rs1	0x0	0x03	0x5b

SC.DDC cs2, rs1 *(RV64)*

31 25	24 20	19 15	14 12	11 7	6 0
0x7c	cs2	rs1	0x0	0x04	0x5b

Description

The capability located in memory at **DDC.address** + *rs1* is replaced with capability register *cs2*.

Semantics

```
let (ddc_val, vaddr) = ddc_and_resulting_addr(X(rs1));
handle_store_cap_via_cap(cs2, DDC_IDX, ddc_val, vaddr)
```

Exceptions

An exception is raised if:

- **DDC.tag** is not set.
- **DDC** is sealed.
- **DDC.perms** does not grant PERMIT_STORE.
- **DDC.perms** does not grant PERMIT_STORE_CAPABILITY and *cs2*.**tag** is set.
- **DDC.perms** does not grant PERMIT_STORE_LOCAL_CAPABILITY, *cs2*.**tag** is set and *cs2*.**perms** does not grant GLOBAL.
- **DDC.address** + *rs1* < **DDC.base**.
- **DDC.address** + *rs1* + **CLEN** / 8 > **DDC.top**.

Chapter 8

The CHERI-x86-64 Instruction-Set Reference

In this chapter, we specify new CHERI instructions as well as extensions to existing instructions to support capability-sized operands. Instructions are described using similar syntax to Volume 2 of Intel's Software Developer's Manual [65] with a few extensions.

An additional symbol is defined to represent object code in the "Opcode" column:

- **CAP** — Indicates the use of the capability operand prefix.
- **+rc** — Indicates the lower 3 bits of the opcode byte is used to encode the register operand without a modR/M byte.

Additional symbols are defined to represent operands in the "Instruction" column:

- **rc** — One of the general-purpose capability registers: **CAX**, **CBX**, **CCX**, **CDX**, **CDI**, **CSI**, **CBP**, **CSP**, **C8**-**C15**.
- **r/mc** — A capability operand that is either the contents of one of the capability registers for **rc** or a capability in memory.
- **m2c** — A pair of adjacent capabilities in memory.
- **mcs** — An aligned stride of capabilities in memory.

In addition, all of these instructions are either invalid or not encodable in Compatibility/Legacy mode, so that column is omitted from opcode tables. However, a new column is added to describe capability mode support using one of the following annotations:

- **V** — Supported.
- **I** — Not supported.
- **N. E.** — Not encodable in capability mode.

8.1 Extensions to x86-64 Instructions

This section contains extensions to existing instructions to support capability operands. For each of these instructions, the instruction description should be treated as an extension to the description of the existing instruction in Volume 2 of Intel's Software Developer's Manual. Many of the instruction descriptions in this section reuse language from Intel's manual to highlight the similarity in semantics between the base instructions and their CHERI extensions.

ADD – Add

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + 05 *id*	ADD CAX, *imm32*	I	Valid	Valid	Add *imm32 sign-extended to 64-bits* to the address field of CAX.
CAP + 81 /0 *id*	ADD *r/mc, imm32*	MI	Valid	Valid	Add *imm32 sign-extended to 64-bits* to the address field of *r/mc*.
CAP + 83 /0 *ib*	ADD *r/mc, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to the address field of *r/mc*.
CAP + 01 */r*	ADD *r/mc, r64*	MR	Valid	Valid	Add *r64* to the address field of *r/mc*.
CAP + 03 */r*	ADD *rc, r/m64*	RM	Valid	Valid	Add *r/m64* to the address field of *rc*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:Reg (r,w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r,w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r,w)	imm8/32	NA	NA
I	CAX	imm32	NA	NA

Description

Adds the source operand to the **address** field of the destination operand and then stores the result in the destination operand. The destination operand can be a register or a memory location. The source operand can be an immediate, a register, or a memory location.

If the new value of the **address** field makes the resulting capability unrepresentable, the **tag** field in the resulting capability is cleared.

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the value of the resulting **address** field.

AND – Logical AND

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + 25 *id*	AND CAX, *imm32*	I	Valid	Valid	Bitwise AND of *imm32 sign-extended to 64-bits* with the address field of CAX.
CAP + 81 /4 *id*	AND *r/mc, imm32*	MI	Valid	Valid	Bitwise AND of *imm32 sign-extended to 64-bits* with the address field of *r/mc*.
CAP + 83 /4 *ib*	AND *r/mc, imm8*	MI	Valid	Valid	Bitwise AND of *sign*-extended *imm8* with the address field of *r/mc*.
CAP + 21 */r*	AND *r/mc, r64*	MR	Valid	Valid	Bitwise AND of *r64* with the address field of *r/mc*.
CAP + 23 */r*	AND *rc, r/m64*	RM	Valid	Valid	Bitwise AND of *r/m64* with the address field of *rc*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:Reg (r,w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r,w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r,w)	imm8/32	NA	NA
I	CAX	imm32	NA	NA

Description

Derives a new capability from the destination operand whose **address** field is set to the bitwise AND of the source operand and the **address** field of the destination operand and then stores the result in the destination operand. The destination operand can be a register or a memory location. The source operand can be an immediate, a register, or a memory location.

If the new value of the **address** field makes the resulting capability unrepresentable, the **tag** field in the resulting capability is cleared.

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the value of the resulting **address** field.

CALL – Call Procedure

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
E8 *cw*	CALL *rel16*	D	Invalid	N.S.	Call near, relative displacement.
E8 *cd*	CALL *rel32*	D	Valid	Valid	Call near, relative displacement, 32-bit displacement sign-extended to 64-bits.
FF /2	CALL *r/m16*	M	N.E.	N.E.	Call near, absolute indirect, address given in *r/m16*.
FF /2	CALL *r/m32*	M	N.E.	N.E.	Call near, absolute indirect, address given in *r/m32*.
FF /2	CALL *r/m64*	M	Invalid	Valid	Call near, absolute indirect, address given in *r/m64*.
FF /2	CALL *r/mc*	M	Valid	Valid	Call near, absolute indirect, address given in *r/mc*.
9A *cd*	CALL *ptr16:16*	D	N.E.	N.E.	Call far, absolute, address in operand.
9A *cp*	CALL *ptr16:32*	D	N.E.	N.E.	Call far, absolute, address in operand.
FF /3	CALL *m16:16*	M	Invalid	Valid	Call far, absolute indirect, address given in *m16:16*.
FF /3	CALL *m16:32*	M	Invalid	Valid	Call far, absolute indirect, address given in *m16:32*.
REX.W FF /3	CALL *m16:64*	M	Invalid	Valid	Call far, absolute indirect, address given in *m16:64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

Description

Saves return address on the stack and branches to the location specified by the first operand. In 64-bit mode the `CAP` prefix can be used with opcode `FF /2` to select the capability operand size instead of 64-bit. In capability mode, near calls always use the capability operand size.

Relative calls always apply the relative displacement to the address of the next instruction to compute a new value of the **address** field of **CIP**.

Near calls which use a capability operand size always push **CIP** onto the current stack before executing the first instruction at the target rather than pushing the value of **RIP**.

Flags Affected

None

CMOV*cc* – Conditional Move

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + 0F 40 */r*	CMOVO *rc, r/mc*	RM	Valid	Valid	Move if overflow (OF=1).
CAP + 0F 41 */r*	CMOVNO *rc, r/mc*	RM	Valid	Valid	Move if not overflow (OF=0).
CAP + 0F 42 */r*	CMOVC *rc, r/mc*	RM	Valid	Valid	Move if carry (CF=1).
CAP + 0F 43 */r*	CMOVNC *rc, r/mc*	RM	Valid	Valid	Move if not carry (CF=0).
CAP + 0F 44 */r*	CMOVZ *rc, r/mc*	RM	Valid	Valid	Move if zero (ZF=1).
CAP + 0F 45 */r*	CMOVNZ *rc, r/mc*	RM	Valid	Valid	Move if not zero (ZF=0).
CAP + 0F 46 */r*	CMOVBE *rc, r/mc*	RM	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
CAP + 0F 47 */r*	CMOVA *rc, r/mc*	RM	Valid	Valid	Move if above (CF=0 and ZF=0).
CAP + 0F 48 */r*	CMOVS *rc, r/mc*	RM	Valid	Valid	Move if sign (SF=1).
CAP + 0F 49 */r*	CMOVNS *rc, r/mc*	RM	Valid	Valid	Move if not sign (SF=0).
CAP + 0F 4A */r*	CMOVP *rc, r/mc*	RM	Valid	Valid	Move if parity (PF=1).
CAP + 0F 4B */r*	CMOVNP *rc, r/mc*	RM	Valid	Valid	Move if no parity (PF=0).
CAP + 0F 4C */r*	CMOVL *rc, r/mc*	RM	Valid	Valid	Move if less (SF$\neq$ OF).
CAP + 0F 4D */r*	CMOVGE *rc, r/mc*	RM	Valid	Valid	Move if greater or equal (SF=OF).
CAP + 0F 4E */r*	CMOVLE *rc, r/mc*	RM	Valid	Valid	Move if less or equal (ZF=1 or SF$\neq$ OF).
CAP + 0F 4F */r*	CMOVG *rc, r/mc*	RM	Valid	Valid	Move if greater (ZF=0 and SF=OF).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:Reg (w)	ModRM:r/m (r)	NA	NA

Description

Copies the source operand to the destination operand if one or more status flags in the **RFLAGS** register are in the required state. The destination operand is a register. The source operand can be a register or memory location.

Flags Affected

None

CMP – Compare Two Operands

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + 39 */r*	CMP *r/mc, rc*	MR	Valid	Valid	Compare *rc* with *r/mc*.
CAP + 3B */r*	CMP *rc, r/mc*	RM	Valid	Valid	Compare *r/mc* with *rc*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:Reg (r)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r)	ModRM:reg (r)	NA	NA

Description

Compares the first source operand with the scond source operand and sets status flags in **RFLAGS**. Sets ZF to 1 if the two operands are equal and 0 otherwise. Sets SF to 1 if the **tag** fields of the two operands are equal and the bounds and permissions of the second operand are a subset of the first operand.

Flags Affected

The ZF and SF flags are set as described above. The OF, SF, AF, CF, and PF flags are undefined.

CMPXCHG – Compare and Exchange

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + 0F B1 */r*	CMPXCHG *r/mc, rc*	MR	Valid	Valid	Compare CAX with *r/mc*. If equal, store *rc* to *r/mc*, else load *r/mc* into CAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r,w)	ModRM:reg (r)	NA	NA

Description

Compares **CAX** with the destination operand. If the values are equal, stores the source operand in the destination operand. If the values are not equal, loads the destination operand into **CAX**.

The instruction can be used with a `LOCK` prefix to execute atomically.

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the value of the result of comparing **CAX** with the destination operand.

CMPXCHG2C – Compare and Exchange Pair

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + 0F C7 /1	CMPXCHG2C *m2c*	MR	Valid	Valid	Compare CDX:CAX with *m2c*. If equal, set ZF and CCX:CBX to *m2c*, else load *m2c* into CDX:CAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r,w)	ModRM:reg (r)	NA	NA

Description

Compares the pair of capabilities in **CDX**:**CAX** with the destination operand. **CAX** is compared with the first capability in memory, and **CDX** is compared with the second capability in memory. If the capabilities are equal, stores **CCX**:**CBX** in the destination operand. If the capabilities are not equal, loads the destination operand into **CDX**:**CAX**.

The instruction can be used with a `LOCK` prefix to execute atomically.

Flags Affected

The ZF flag is set to the result of the comparison (1 if equal). The OF, SF, AF, CF, and PF flags are unaffected.

DEC – Decrement by 1

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + FF /1	DEC *r/mc*	M	Valid	Valid	Decrement address field of *r/mc* by 1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r,w)	NA	NA	NA

Description

Subtracts 1 from the **address** field of the destination operand and then stores the result in the destination operand. The destination operand can be a register or a memory location.

If the new value of the **address** field makes the resulting capability unrepresentable, the **tag** field in the resulting capability is cleared.

Flags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the value of the resulting **address** field.

INC – Increment by 1

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + FF /0	INC *r/mc*	M	Valid	Valid	Increment address field of *r/mc* by 1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r,w)	NA	NA	NA

Description

Adds 1 to the **address** field of the destination operand and then stores the result in the destination operand. The destination operand can be a register or a memory location.

If the new value of the **address** field makes the resulting capability unrepresentable, the **tag** field in the resulting capability is cleared.

Flags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the value of the resulting **address** field.

JMP – Jump

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
E9 *cw*	JMP *rel16*	D	Invalid	N.S.	Jump near, relative displacement.
E9 *cd*	JMP *rel32*	D	Valid	Valid	Jump near, relative displacement, 32-bit displacement sign-extended to 64-bits.
FF /4	JMP *r/m16*	M	N.E.	N.S.	Jump near, absolute indirect, address given in *r/m16*.
FF /4	JMP *r/m32*	M	N.E.	N.S.	Jump near, absolute indirect, address given in *r/m32*.
FF /4	JMP *r/m64*	M	Invalid	Valid	Jump near, absolute indirect, address given in *r/m64*.
FF /4	JMP *r/mc*	M	Valid	Valid	Jump near, absolute indirect, address given in *r/mc*.
EA *cd*	JMP *ptr16:16*	D	N.E.	Invalid	Jump far, absolute, address in operand.
EA *cp*	JMP *ptr16:32*	D	N.E.	Invalid	Jump far, absolute, address in operand.
FF /5	JMP *m16:16*	M	Invalid	Valid	Jump far, absolute indirect, address given in *m16:16*.
FF /5	JMP *m16:32*	M	Invalid	Valid	Jump far, absolute indirect, address given in *m16:32*.
REX.W FF /5	JMP *m16:64*	M	Invalid	Valid	Jump far, absolute indirect, address given in *m16:64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

Description

Branches to the location specified by the first operand. In 64-bit mode the `CAP` prefix can be used with opcode `FF /4` to select the capability operand size instead of 64-bit. In capability mode, `FF /4` supports only the capability operand size.

Relative jumps always apply the relative displacement to the address of the next instruction to compute a new value of the **address** field of **CIP**.

Flags Affected

None

LEA – Load Effective Address

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + 8D */r*	LEA *rc, m*	RM	Valid	Valid	Store effective capability address of *m* in *rc*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Computes the effective capability of the second operand and stores the result in the first operand. When used with the `CAP` prefix, the effective address is always computed using capability-aware addressing.

Flags Affected

None

LODS/LODSC – Load String

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + AD	LODS *mc*	ZO	Valid	Valid	Load capability at address (C|R)SI into CAX.
CAP + AD	LODSC	ZO	Valid	Valid	Load capability at address (C|R)SI into CAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Loads a capablity from the source operand into the **CAX** register. The source operand is a memory location identified by the **RSI** or **CSI** register (depending on the addressing mode). After the capability is loaded from memory, the address register is incremented or decremented by the size of a capability according to the setting of `DF` in **RFLAGS**.

Flags Affected

None

MOV – Move

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + 89 */r*	MOV *r/mc, rc*	MR	Valid	Valid	Move *rc* to *r/mc*.
CAP + 8B */r*	MOV *rc, r/mc*	RM	Valid	Valid	Move *r/mc* to *rc*.
CAP + C7 */0 id*	MOV *r/mc, imm32*	MI	Valid	Valid	Move *imm32 sign-extended to 64-bits* to *r/mc*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:Reg (w)	ModRM:r/m (r)	NA	NA
MI	ModRM:r/m (w)	imm32	NA	NA

Description

Copies the source operand to the destination operand. The destination operand can be a register or a memory location. The source operand can be an immediate, a register, or a memory location. If the source operand is an immediate, the value is sign-extended to 64-bits and used as the address of a NULL-derived capability.

Note that some `MOV` opcodes such as `B8+ rw` and `C6 /0` are not extended to support the `CAP` prefix as the behavior would be identical. The `C7 /0` opcode is extended primarily to support storing constants such as NULL to capabilities in memory without requiring an intermediate register.

The `A1` and `A3` opcodes are not extended to support capabilities.

Flags Affected

None

MOVNTI – Store Using Non-Temporal Hint

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
NP CAP + 0F C3 */r*	MOV *mc, rc*	MR	Valid	Valid	Move *rc* to *mc* using non-temporal hint.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the capability in the source operand to the destination operand using a non-temporal hint.

Flags Affected

None

MOVS/MOVSC – Move Data from String to String

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + A5	MOVS *mc, mc*	ZO	Valid	Valid	Move capability from address (C|R)SI to (C|R)DI.
CAP + A5	MOVSC	ZO	Valid	Valid	Move capability from address (C|R)SI to (C|R)DI.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Moves a capablity from the source operand to the destination operand. The source operand is a memory location identified by the **RSI** or **CSI** register (depending on the addressing mode). The destination operand is a memory location identified by the **RDI** or **CDI** register (depending on the addressing mode). After the capability is copied, the address registers are incremented or decremented by the size of a capability according to the setting of `DF` in **RFLAGS**.

Flags Affected

None

OR – Logical Inclusive OR

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + 0D *id*	OR CAX, *imm32*	I	Valid	Valid	Bitwise inclusive OR of *imm32 sign-extended to 64-bits* with the address field of CAX.
CAP + 81 /1 *id*	OR *r/mc, imm32*	MI	Valid	Valid	Bitwise inclusive OR of *imm32 sign-extended to 64-bits* with the address field of *r/mc*.
CAP + 83 /1 *ib*	OR *r/mc, imm8*	MI	Valid	Valid	Bitwise inclusive OR of *sign*-extended *imm8* with the address field of *r/mc*.
CAP + 09 */r*	OR *r/mc, r64*	MR	Valid	Valid	Bitwise inclusive OR of *r64* with the address field of *r/mc*.
CAP + 0B */r*	OR *rc, r/m64*	RM	Valid	Valid	Bitwise inclusive OR of *r/m64* with the address field of *rc*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:Reg (r,w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r,w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r,w)	imm8/32	NA	NA
I	CAX	imm32	NA	NA

Description

Derives a new capability from the destination operand whose **address** field is set to the bitwise inclusive OR of the source operand and the **address** field of the destination operand and then stores the result in the destination operand. The destination operand can be a register or a memory location. The source operand can be an immediate, a register, or a memory location.

If the new value of the **address** field makes the resulting capability unrepresentable, the **tag** field in the resulting capability is cleared.

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the value of the resulting **address** field.

POP – Pop Value from the Stack

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
8F /0	POP *r/m16*	M	N.E.	Valid	Pop *r/m16*.
8F /0	POP *r/m32*	M	N.E.	N.E.	Pop *r/m32*.
8F /0	POP *r/m64*	M	Valid	Valid	Pop *r/m64*.
8F /0	POP *r/mc*	M	Valid	Valid	Pop *r/mc*.
58+*rw*	POP *r16*	O	N.E.	Valid	Pop *r16*.
58+*rd*	POP *r32*	O	N.E.	N.E.	Pop *r32*.
58+*ro*	POP *r64*	O	Valid	Valid	Pop *r64*.
58+*rc*	POP *rc*	O	Valid	Valid	Pop *rc*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA
O	opcode + rd (w)	NA	NA	NA

Description

Stores the value at the top of the stack in the destination operand and increments the stack pointer. The following extensions apply to these instructions in 64-bit mode:

- Address size: The 0x07 prefix selects a capability-aware address.
- Operand size: The `CAP` prefix selects a capability operand size.

In capability mode, the various sizes are:

- Address size: The default addressing mode uses capability-aware addressing. 64-bit addresses can be used by specifying the 0x07 prefix.
- Operand size: The default operand size is a capability. If the `CAP` prefix is specified, the operand size is 64-bits. 16-bit and 32-bit operands cannot be used in capability mode.
- Stack-address size: In capability mode the stack pointer is always **CSP**.

Flags Affected

None

PUSH – Push Value Onto the Stack

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
FF /6	PUSH *r/m16*	M	N.E.	Valid	Push *r/m16*.
FF /6	PUSH *r/m32*	M	N.E.	N.E.	Push *r/m32*.
FF /6	PUSH *r/m64*	M	Valid	Valid	Push *r/m64*.
FF /6	PUSH *r/mc*	M	Valid	Valid	Push *r/mc*.
50+*rw*	PUSH *r16*	O	N.E.	Valid	Push *r16*.
50+*rd*	PUSH *r32*	O	N.E.	N.E.	Push *r32*.
50+*ro*	PUSH *r64*	O	Valid	Valid	Push *r64*.
50+*rc*	PUSH *rc*	O	Valid	Valid	Push *rc*.
6A *ib*	PUSH *imm8*	I	Valid	Valid	Push *imm8*.
68 *iw*	PUSH *imm16*	I	Valid	Valid	Push *imm16*.
68 *id*	PUSH *imm32*	I	Valid	Valid	Push *imm32*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA
O	opcode + rd (r)	NA	NA	NA
I	imm8/16/32	NA	NA	NA

Description

Decrements the stack pointer and stores the source operand on the stack. The following extensions apply to these instructions in 64-bit mode:

- Address size: The 0x07 prefix selects a capability-aware address.
- Operand size: The `CAP` prefix selects a capability operand size. When a capability operand size is used, immediate operands are sign-extended to 64-bits and the result used as the address of a null-derived capability.

In capability mode, the various sizes are:

- Address size: The default addressing mode uses capability-aware addressing. 64-bit addresses can be used by specifying the 0x07 prefix.
- Operand size: The default operand size is a capability. If the `CAP` prefix is specified, the operand size is 64-bits. Immediate operands are sign-extended to 64-bits. When a capability operand size is used, sign-extended immediate operands are used as the address of a null-derived capability.
- Stack-address size: In capability mode the stack pointer is always **CSP**.

Flags Affected

None

RET – Return from Procedure

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
C3	RET	ZO	Valid	Valid	Near return.
CB	RET	ZO	Invalid	Valid	Far return.
C2 *iw*	RET *imm16*	I	Valid	Valid	Near return and pop *imm16* bytes from stack.
CA *iw*	RET *imm16*	I	Invalid	Valid	Far return and pop *imm16* bytes from stack.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA
I	imm16	NA	NA	NA

Description

Pops return address from the stack and transfers control to the popped address. In 64-bit mode the `CAP` prefix can be used with near returns to select the capability operand size instead of 64-bit. In capability mode, near returns always use the capability operand size.

Near returns which use a capability operand size always pop a capability off of the stack to load into **CIP** rather than popping off the new value of **RIP**.

Flags Affected

None

STOS/STOSC – Store String

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + AB	STOS *mc*	ZO	Valid	Valid	Store CAX at address (C|R)DI.
CAP + AB	STOSC	ZO	Valid	Valid	Store CAX at address (C|R)DI.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Stores capability in the **CAX** register into the destination operand. The destination operand is a memory location identified by the **RDI** or **CDI** register (depending on the addressing mode). After the capability is stored to memory, the address register is incremented or decremented by the size of a capability according to the setting of `DF` in **RFLAGS**.

Flags Affected

None

SUB – Subtract

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + 2D *id*	SUB CAX*, imm32*	I	Valid	Valid	Subtract *imm32 sign-extended to 64-bits* from the address field of CAX.
CAP + 81 /5 *id*	SUB *r/mc, imm32*	MI	Valid	Valid	Subract *imm32 sign-extended to 64-bits* from the address field of *r/mc*.
CAP + 83 /5 *ib*	SUB *r/mc, imm8*	MI	Valid	Valid	Subtract *sign*-extended *imm8* from the address field of *r/mc*.
CAP + 29 */r*	SUB *r/mc, r64*	MR	Valid	Valid	Subtract *r64* from the address field of *r/mc*.
CAP + 2B */r*	SUB *rc, r/m64*	RM	Valid	Valid	Subtract *r/m64* from the address field of *rc*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:Reg (r,w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r,w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r,w)	imm8/32	NA	NA
I	CAX	imm32	NA	NA

Description

Subtracts the source operand from the **address** field of the destination operand and then stores the result in the destination operand. The destination operand can be a register or a memory location. The source operand can be an immediate, a register, or a memory location.

If the new value of the **address** field makes the resulting capability unrepresentable, the **tag** field in the resulting capability is cleared.

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the value of the resulting **address** field.

XADD – Exchange and Add

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + 0F C1 */r*	XADD *r/mc, rc, r64*	MRR	Valid	Valid	Load original value of *r/mc* into *rc*. Add *r64* to the address field of *r/mc*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRR	ModRM:r/m (r,w)	ModRM:reg (w)	ModRM:reg (r)	NA

Description

Stores the original value of the destination (first) operand in the second operand. Derives a new capability value by adding the third operand to the **address** field of the destination operand and stores the result in the destination operand. Note that the third operand must be the 64-bit register which aliases the low 64-bits of the second operand.

If the new value of the **address** field makes the resulting capability unrepresentable, the **tag** field in the resulting capability is cleared.

The instruction can be used with a `LOCK` prefix to execute atomically.

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the value of the resulting **address** field.

XCHG – Exchange

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + 90+*rc*	XCHG CAX, *rc*	O	Valid	Valid	Exchange *rc* with CAX.
CAP + 90+*rc*	XCHG *rc*, CAX	O	Valid	Valid	Exchange CAX with *rc*.
CAP + 87 */r*	XCHG *r/mc, rc*	MR	Valid	Valid	Exchange *rc* with *r/mc*.
CAP + 87 */r*	XCHG *rc, r/mc*	RM	Valid	Valid	Exchange *r/mc* with *rc*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
O	CAX (r,w)	opcode + rc (r, w)	NA	NA
O	opcode + rc (r, w)	CAX (r,w)	NA	NA
MR	ModRM:r/m (r,w)	ModRM:reg (r, w)	NA	NA
RM	ModRM:reg (r,w)	ModRM:r/m (r, w)	NA	NA

Description

Exchanges the contents of the source and destination operands. The operands can be two general-purpose registers or a general-purpose register and a memory location.

The specific opcode `90` would remain a single byte `NOP` that would not alter the value of **CAX**.

Flags Affected

None

XOR – Logical Exclusive OR

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
CAP + 35 *id*	XOR CAX, *imm32*	I	Valid	Valid	Bitwise exclusive OR of *imm32 sign-extended to 64-bits* with the address field of CAX.
CAP + 81 /6 *id*	XOR *r/mc, imm32*	MI	Valid	Valid	Bitwise exclusive OR of *imm32 sign-extended to 64-bits* with the address field of *r/mc*.
CAP + 83 /6 *ib*	XOR *r/mc, imm8*	MI	Valid	Valid	Bitwise exclusive OR of *sign*-extended *imm8* with the address field of *r/mc*.
CAP + 31 */r*	XOR *r/mc, r64*	MR	Valid	Valid	Bitwise exclusive OR of *r64* with the address field of *r/mc*.
CAP + 33 */r*	XOR *rc, r/m64*	RM	Valid	Valid	Bitwise exclusive OR of *r/m64* with the address field of *rc*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:Reg (r,w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r,w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r,w)	imm8/32	NA	NA
I	CAX	imm32	NA	NA

Description

Derives a new capability from the destination operand whose **address** field is set to the bitwise exclusive OR of the source operand and the **address** field of the destination operand and then stores the result in the destination operand. The destination operand can be a register or a memory location. The source operand can be an immediate, a register, or a memory location.

If the new value of the **address** field makes the resulting capability unrepresentable, the **tag** field in the resulting capability is cleared.

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the value of the resulting **address** field.

8.2 CHERI-x86-64 Instructions

This section contains new instructions added to support operations on capabilities. The opcode assignments in this section are tentative and subject to change. Single byte opcodes have been used for instructions which we believe may either be used frequently or in frequently-accessed code paths.

ANDCPERM – Mask Capability Permissions

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
NP 0F 0C */r*	ANDCPERM *r/mc, r64*	MR	Valid	Valid	Mask permissions of *r/mc* by *r64*.
37 /2 *id*	ANDCPERM *r/mc, imm32*	MI	Valid	Valid	Mask permissions of *r/mc* by sign-extended *imm32*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r,w)	imm32	NA	NA

Description

Derives a new capability from the destination operand with the **perms** and **uperms** field bitwise ANDed with the source operand and stores the result in the destination operand. The destination operand can be a register or memory location; the source operand can be a register or immediate. If the destination operand is sealed and tagged, set the destination operand to its original value with the **tag** field cleared.

Flags Affected

None

BUILDCAP – Construct Capability

Opcode	Instruction	Op/ En	Cap Mode	64-bit Mode	Description
VEX.LZ.0F.W0 0E */r*	BUILDCAP *rca, r/mc, rcb*	RMV	Valid	Valid	Construct capability from *r/mc* and *rcb* and store in *rca*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (w)	ModRM:r/m (r)	VEX.vvvv (r)	NA

Description

Constructs a new capability equal to the second operand with the **base**, **length**, **address**, **perms**, and **uperms** fields replaced with the corresponding fields from the third operand and stores the result in the first (destination) operand. If the third operand is a sentry then the result is also sealed as a sentry. If the resulting capability is not a subset of the second operand in bounds or permissions, or is not a legally-derivable capability, or if the second operand did not have its **tag** field set, or if the second operand was sealed as a non-sentry, the resulting capability is set to the third operand with the **tag** field cleared.

Flags Affected

ZF is set to the **tag** field of the resulting capability. The CF, PF, AF, and OF flags are undefined.

CINVOKE – Invoke Sealed Capability Pair

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
EA */r*	CINVOKE *rc, r/mc*	RM	Valid	Valid	Set CAX to *r/mc* and jump to *rc*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

Jumps to a pair of sealed capabilities. The first source operand can be a register; the second source operand can be a register or memory location.

If both operands are sealed with the same **otype**, sets **CIP** to the unsealed first operand and sets **CAX** to the unsealed second operand. Note that this control transfer is a jump and does not push any values onto the stack. If this instruction fails, it raises a Capability Violation Fault (see Section 5.2.10).

Flags Affected

None

CLCTAG – Clear Capability Tag

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
0E /1	CLCTAG *r/mc*	M	Valid	Valid	Clear tag of *r/mc*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA

Description

Clears the **tag** field of the destination operand. The destination operand can be a register or memory location.

Flags Affected

None

CLCTAGS – Clear Capability Tags

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
0E /2	CLCTAGS *mcs*	M	Valid	Valid	Clear capability tags of *mcs*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Clears the capability tags for a stride of capabilities in memory starting at the destination operand. The authorizing capability for the destination operand must include PERMIT_STORE and authorize access to the entire stride of capabilities. The address of the destination operand must also be aligned to a stride of capabilities.

If this instruction fails, it raises either a Capability Violation Fault (see Section 5.2.10) or a Page Fault.

Flags Affected

None

CPYTYPE – Construct Sealing Capability

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
VEX.LZ.66.0F.W0 0E */r*	CPYTYPE *rca, r/mc, rcb*	RMV	Valid	Valid	Construct a sealing capability from *r/mc* and *rcb* and store in *rca*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (w)	ModRM:r/m (r)	VEX.vvvv (r)	NA

Description

Constructs a new capability equal to the second operand with the **address** field set to the **otype** field of the third operand and stores the result in the first (destination) operand. If the third operand's **otype** field is reserved or if the second operand was sealed, the **tag** in the resulting capability is cleared.

Flags Affected

ZF is set to the **tag** field of the resulting capability. The CF, PF, AF, and OF flags are undefined.

CRAM – Representable Alignment Mask

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
1F */r*	CRAM *r64, r/m64*	RM	Valid	Valid	Set *r64* to mask sufficient for aligned bounds spanning the length in *r/m64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Sets the destination operand to a mask that can be used to round addresses down to a value that is sufficiently aligned to set exact bounds for the nearest representable length of the source operand. The source operand can be a register or memory location.

Flags Affected

None

CRRL – Round Representable Length

Opcode	Instruction	Op/ En	Cap Mode	64-bit Mode	Description
1E */r*	CRRL *r64, r/m64*	RM	Valid	Valid	Set *r64* to minimum representable length greater or equal to *r/m64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Sets the destination operand to the smallest value greater or equal to the source operand that can be used as a length to set exact bounds on a capability with a suitably aligned base. The source operand can be a register or memory location.

Flags Affected

None

CSEAL – Conditional Capability Seal

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
VEX.LZ.F2.0F.W0 0E */r*	CSEAL *rca, r/mc, rcb*	RMV	Valid	Valid	Conditionally seal *r/mc* with type from address field of *rcb* and store in *rca*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (w)	ModRM:r/m (r)	VEX.vvvv (r)	NA

Description

Seals the second operand with the **otype** equal to the **address** field of the third operand and stores the result in the first (destination) operand. If the sealing operation fails, the second operand is copied to the first operand.

Flags Affected

The ZF flag is set to 1 if the sealing operation succeeds; otherwise 0. The CF, PF, AF, SF, and OF flags are undefined.

GCBASE – Get Capability Base

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
F2 0F 7A */r*	GCBASE *r64, r/mc*	RM	Valid	Valid	Store the base field of *r/mc* in *r64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Sets the destination operand to the **base** field of the source operand. The source operand can be a register or memory location.

Flags Affected

None

GCFLAGS – Get Capability Flags

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
NP 0F 7B */r*	GCFLAGS *r64, r/mc*	RM	Valid	Valid	Store the flags field of *r/mc* in *r64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Sets the destination operand to the **flags** field of the source operand. The source operand can be a register or memory location.

Flags Affected

None

GCHI – Get Capability High Word

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
F2 0F 7B */r*	GCHI *r64, r/mc*	RM	Valid	Valid	Store the upper half of *r/mc* in *r64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Sets the destination operand to the **high half** of the source operand. The source operand can be a register or memory location.

Flags Affected

None

GCLEN – Get Capability Length

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
F3 0F 7A */r*	GCLEN *r64, r/mc*	RM	Valid	Valid	Store the bounds length of *r/mc* in *r64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Sets the destination operand to the **length** field of the source operand. The source operand can be a register or memory location.

Flags Affected

None

GCLIM – Get Capability Limit

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
F3 0F 7B */r*	GCLIM *r64, r/mc*	RM	Valid	Valid	Store the limit of *r/mc* in *r64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Sets the destination operand to the limit (i.e. one past the last addressable byte) of the source operand. The source operand can be a register or memory location. If the source operand permits accessing the last byte of the address space, the destination operand is set to the value $2^{64} - 1$.

Flags Affected

None

GCOFF – Get Capability Offset

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
66 0F 7B */r*	GCOFF *r64, r/mc*	RM	Valid	Valid	Store the offset of *r/mc* in *r64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Sets the destination operand to the **offset** field of the source operand. The source operand can be a register or memory location.

Flags Affected

None

GCPERM – Get Capability Permissions

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
NP 0F 7A */r*	GCPERM *r64, r/mc*	RM	Valid	Valid	Store the permissions mask of *r/mc* in *r64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Sets the destination operand to the combined **perms** and **uperms** fields of the source operand. The source operand can be a register or memory location.

Flags Affected

None

GCTAG – Get Capability Tag

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
0E /3	GCTAG *r/mc*	M	Valid	Valid	Store the tag of *r/mc* in ZF.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Sets ZF in **RFLAGS** to the **tag** field of the source operand. The source operand can be a register or memory location.

Flags Affected

ZF is set to 1 if the result is zero. The CF, PF, AF, SF, and OF flags are undefined.

GCTYPE – Get Capability Object Type

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
66 0F 7A */r*	GCTYPE *r64, r/mc*	RM	Valid	Valid	Store the object type of *r/mc* in *r64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Sets the destination operand to the **otype** field of the source operand. The source operand can be a register or memory location.

Flags Affected

ZF is set to 1 if the result is zero. SF is set to 1 if the result is less than zero. The CF, PF, AF, and OF flags are undefined.

LCTAGS – Load Capability Tags

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
2F */r*	LCTAGS *r64, mcs*	RM	Valid	Valid	Load bitmask of capability tags of *mcs* into *r64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:Reg (w)	ModRM:r/m (r)	NA	NA

Description

Loads a packed bitmask of capability tags for a stride of capabilities in memory starting at the source operand and stores the bitmask in the destination operand. Bit 0 corresponds to tag for the capability at the address of the source operand. The authorizing capability for the source operand must include PERMIT_LOAD and PERMIT_LOAD_CAPABILITY and authorize access to the entire stride of capabilities. The source operand address must also be aligned to a stride of capabilities.

If this instruction fails, it raises either a Capability Violation Fault (see Section 5.2.10) or a Page Fault.

Flags Affected

ZF is set to 1 if the result is zero. The CF, PF, AF, SF, and OF flags are undefined.

MOV – Move to/from Additional Capability Registers

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
0F 24 */r*	MOV *rc,* CFS/CGS/DDC	MR	Valid	Valid	Move additional capability register to *rc*.
0F 25 */r*	MOV CFS/CGS/DDC, *rc*	RM	Valid	Valid	Move *rc* to additional capability register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Moves the contents of an additional capability register to a general-purpose capability register or vice versa.

Similar to the `MOV` opcodes for control and debug registers, the **reg** field of the ModRM byte always identifies the additional capability register to read or write. The **mod** field of ModRM is ignored, and the **r/m** field identifies the general-purpose capability register. Attempts to reference invalid additional capability registers will raise a UD# exception.

Attempts to access additional capability registers other than **CFS**, **CGS**, or **DDC** from a privilege level other than 0 will raise a GP#(0) exception.

Flags Affected

None

SCADDR – Set Capability Address

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
16 */r*	SCADDR *r/mc, r64*	MR	Valid	Valid	Set the address field of *r/mc* to *r64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA

Description

Sets the **address** field of the destination operand to the source operand and stores the result in the destination operand. The destination operand can be a register or memory location; the source operand can be a register. If the destination operand is sealed and tagged, the destination operand is set to its original value with the **tag** field cleared.

If the new value of the **address** field makes the resulting capability unrepresentable, the **tag** field in the resulting capability is cleared.

Flags Affected

None

SCBND – Set Capability Bounds

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
17 */r*	SCBND *r/mc, r64*	MR	Valid	Valid	Set bounds of *r/mc* to *r64*.
37 /0 *id*	SCBND *r/mc, imm32*	MI	Valid	Valid	Set bounds of *r/mc* to zero-extended *imm32*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm32	NA	NA

Description

Derives a new capability from the destination operand and source operand and then store the result in the destination operand. The destination operand can be a register or memory location; the source operand can be an immediate or a register. When an immediate value is used as an operand, it is zero-extended.

The new capability's **base** field is set to the current **address** field of the destination operand, and the **length** field is set to the source operand. If the resulting capability cannot be represented exactly, the **base** will be rounded down and the **length** will be rounded up by the smallest amount needed to form a representable capability covering the requested bounds. If the original capability is sealed or the bounds of the new capability exceed the original capability, the **tag** field in the resulting capability is cleared.

Flags Affected

The ZF flag is set to the value of the **tag** field in the resulting capability. The CF, PF, AF, SF, and OF flags are undefined.

SCBNDE – Set Exact Capability Bounds

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
27 */r*	SCBNDE *r/mc, r64*	MR	Valid	Valid	Set bounds of *r/mc* to *r64*.
37 /1 *id*	SCBNDE *r/mc, imm32*	MI	Valid	Valid	Set bounds of *r/mc* to zero-extended *imm32*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm32	NA	NA

Description

Derives a new capability from the destination operand and source operand and then store the result in the destination operand. The destination operand can be a register or memory location; the source operand can be an immediate or a register. When an immediate value is used as an operand, it is zero-extended.

The new capability's **base** field is set to the current **address** field of the destination operand, and the **length** field is set to the source operand. If the resulting capability cannot be represented exactly, the **tag** field will be cleared, the **base** will be rounded down, and the **length** will be rounded up by the smallest amount needed to form a representable capability covering the requested bounds. If the original capability is sealed or the bounds of the new capability exceed the original capability, the **tag** field in the resulting capability is cleared.

Flags Affected

The ZF flag is set to the value of the **tag** field in the resulting capability. The CF, PF, AF, SF, and OF flags are undefined.

SCFLAGS – Set Capability Flags

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
66 0F 0A */r*	SCFLAGS *r/mc, r64*	MR	Valid	Valid	Set the flags field of *r/mc* to *r64*.
37 /2 *ib*	SCFLAGS *r/mc, imm8*	MI	Valid	Valid	Set the flags field of *r/mc* to *imm8*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r,w)	imm8	NA	NA

Description

Sets the **flags** field of the destination operand to the source operand and stores the result in the destination operand. The destination operand can be a register or memory location; the source operand can be a register. If the destination operand is sealed and tagged, the **tag** field of the destination operand is cleared.

Flags Affected

None

SCHI – Set Capability High Half

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
NP 0F 0A */r*	SCHI *r/mc, r64*	MR	Valid	Valid	Set high half of *r/mc* to *r64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA

Description

Sets the **high half** of the destination operand to the source operand and stores the result in the destination operand. The destination operand can be a register or memory location; the source operand can be a register. The **tag** field in the destination operand is cleared.

Flags Affected

None

SCOFF – Set Capability Offset

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
66 0F 0C */r*	SCOFF *r/mc, r64*	MR	Valid	Valid	Set offset of *r/mc* to *r64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA

Description

Sets the **offset** field of the destination operand to the source operand and stores the result in the destination operand. The destination operand can be a register or memory location; the source operand can be a register. If the destination operand is sealed and tagged, the destination operand is set to its original value with the **tag** field cleared.

If the new value of the **offset** field makes the resulting capability unrepresentable, the **tag** field in the resulting capability is cleared.

Flags Affected

None

SEAL – Seal Capability

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
F2 0F 0C */r*	SEAL *r/mc, rc*	MR	Valid	Valid	Seal *r/mc* with type from the address field of *rc*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA

Description

Seals the destination operand with the **otype** equal to the **address** field of the source operand and stores the result in the destination operand. If the sealing operation fails, the destination operand is set to the original value of the destination operand with the **tag** field cleared.

Flags Affected

The ZF flag is set to 1 if the sealing operation succeeds; otherwise 0. The CF, PF, AF, SF, and OF flags are undefined.

SENTRY – Seal Capability as a Sentry

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
0E /0	SENTRY *r/mc*	M	Valid	Valid	Seal *r/mc* as a sentry.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA

Description

Seals the destination operand as a sentry capability. If the sealing operation fails, leave the destination operand unchanged.

Flags Affected

The ZF flag is set to 1 if the sealing operation succeeds; otherwise 0. The CF, PF, AF, SF, and OF flags are undefined.

UNSEAL – Unseal Capability

Opcode	**Instruction**	**Op/ En**	**Cap Mode**	**64-bit Mode**	**Description**
F3 0F 0C */r*	UNSEAL *r/mc, rc*	MR	Valid	Valid	Unseal *r/mc* using *rc* as the authority.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA

Description

Unseals the destination operand using the source operand as the unsealing authority and stores the result in the destination operand. If the unsealing operation fails, the destination operand is set to the original value of the destination operand with the **tag** field cleared.

Flags Affected

The ZF flag is set to 1 if the unsealing operation succeeds; otherwise 0. The CF, PF, AF, SF, and OF flags are undefined.

8.3 Summary of New Opcodes

The following new opcodes are added in 64-bit mode and are also available in capability mode.

Opcode	**Instruction**
06	Capability Operand (Prefix)
07	Capability Address (Prefix)
0E /0	`SENTRY` *r/mc*
0E /1	`CLCTAG` *r/mc*
0E /2	`CLCTAGS` *mcs*
0E /3	`GCTAG` *r/mc*
16 */r*	`SCADDR` *r/mc, r64*
17 */r*	`SCBND` *r/mc, r64*
1E */r*	`CRRL` *r64, r/m64*
1F */r*	`CRAM` *r64, r/m64*
27 */r*	`SCBNDE` *r/mc, r64*
2F */r*	`LCTAGS` *r64, mcs*
37 /0 *id*	`SCBND` *r/mc, imm32*
37 /1 *id*	`SCBNDE` *r/mc, imm32*
37 /2 *id*	`ANDCPERM` *r/mc, imm32*
37 /3 *id*	`SCFLAGS` *r/mc, imm32*
EA */r*	`CINVOKE` *rc, r/mc*
NP 0F 0A */r*	`SCHI` *r/mc, r64*
66 0F 0A */r*	`SCFLAGS` *r/mc, r64*
NP 0F 0C */r*	`ANDCPERM` *r/mc, r64*
66 0F 0C */r*	`SCOFF` *r/mc, r64*
F2 0F 0C */r*	`SEAL` *r/mc, rc*
F3 0F 0C */r*	`UNSEAL` *r/mc, rc*
VEX.LZ.0F.W0 0E */r*	`BUILDCAP` *rca, r/mc, rcb*
VEX.LZ.66.0F.W0 0E */r*	`CPYTYPE` *rca, r/mc, rcb*
VEX.LZ.F2.0F.W0 0E */r*	`CSEAL` *rca, r/mc, rcb*
0F 24 */r*	`MOV` *rc*, CFS/CGS/DDC
0F 25 */r*	`MOV` CFS/CGS/DDC, *rc*
NP 0F 7A */r*	`GCPERM` *r64, r/mc*
66 0F 7A */r*	`GCTYPE` *r64, r/mc*
F2 0F 7A */r*	`GCBASE` *r64, r/mc*
F3 0F 7A */r*	`GCLEN` *r64, r/mc*
NP 0F 7B */r*	`GCFLAGS` *r64, r/mc*
66 0F 7B */r*	`GCOFF` *r64, r/mc*
F2 0F 7B */r*	`GCHI` *r64, r/mc*
F3 0F 7B */r*	`GCLIM` *r64, r/mc*

8.4 Instructions Deprecated in Capability Mode

The following instructions are valid in 64-bit mode but invalid in capability mode. For conciseness, the placeholder `XX` in the table below indicates any valid size for the instruction (i.e. 8, 16, 32, or 64).

Opcode	**Instruction**	**Description**
26	SEG=ES	ES segment prefix.
2E	SEG=CS	CS segment prefix.
36	SEG=SS	SS segment prefix.
3E	SEG=DS	DS segment prefix.
A0	MOV AL,*moffs8*	Move byte at (*offset*) to AL.
A1	MOV [ER]AX,*moffsxx*	Move value at (*offset*) to [ER]AX.
A2	MOV *moffs8*,AL	Move AL to (*offset*).
A3	MOV *moffs8*,[ER]AX	Move [ER]AX to (*offset*).
CA	RET *imm16*	Far return and pop *imm16* bytes from stack.
CB	RET	Far return.
FF /3	CALL *m16:xx*	Call far, absolute indirect, address given in *m16:xx*.
FF /5	JMP *m16:xx*	Jump far, absolute indirect, address given in *m16:xx*.
0F 02 */r*	LAR *rxx,rxx/m16*	Load access rights.
0F 03 */r*	LSL *rxx,rxx/m16*	Load segment limit.
0F A0	PUSH FS	Push FS.
0F A1	POP FS	Pop FS.
0F A8	PUSH GS	Push GS.
0F A9	POP GS	Pop GS.
0F B2 */r*	LSS *rxx,m16:xx*	Load SS:*rxx* with far pointer.
0F B4 */r*	LFS *rxx,m16:xx*	Load FS:*rxx* with far pointer.
0F B5 */r*	LGS *rxx,m16:xx*	Load GS:*rxx* with far pointer.

Chapter 9

Detailed Design Rationale

During the design of CHERI that began in 2010, we considered many different capability architectures and design approaches. This chapter describes various design choices; it briefly outlines some possible alternatives, and provides rationales for the selected choices.

9.1 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 various 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, page table layout, and so on.

9.2 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 [70, 71], 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) [90] 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.

9.3 Capability Register File

CHERI extends existing general-purpose integer registers to hold capabilities. This design is similar to the manner in which the 32-bit x86 ISA was extended to support 64-bit registers. However, this is not the only way to add CHERI capability registers to an architecture.

We initially used a separate register file for capability registers on CHERI-MIPS for a few pragmatic reasons:

- Coprocessor interfaces on MIPS assume additional register files (a la floating-point registers).
- The initial 256-bit capability registers were quite large, and by giving the capability coprocessor its own pipeline for manipulations, we could avoid enforcing a 256-bit-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.

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.

With the separate register file on CHERI-MIPS, we also added instructions for copying non-capability data from a capability register into a general-purpose integer register. A use case for this was when a function was called with a parameter whose type is the union of a pointer and a non-pointer type, such as an int. This parameter had to be passed in a capability register, because the tag needed to be preserved when it held a capability. If the body of the function accessed the non-capability branch of the union, it needed to get the non-capability bits out of the capability register and into a general purpose register. This was originally done by spilling the capability register to the stack and then reading it back into a general-purpose integer register, but the register to register copy of `CGetAddr` proved faster.

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.

With register tags and 128-bit compressed capabilities, extending existing general-purpose registers to support capabilities became a feasible approach, as register size doubled rather than quadrupled. This approach resulted in improved efficiency in implementations as well as greater software compatibility. For example, in the case described above for a function parameter with a union, the integer branch of the union can be accessed by using the integer portion of the relevant general-purpose register without requiring a separate instruction. As a result, all of the current CHERI architectures extend existing general-purpose registers to hold capabilities.

9.4 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 10 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 alternative approach taken in capability machines such as CAP, in which the compiler was not privileged.

9.5 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, make 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 (with the CHERI-LLVM back-end), 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.

9.6 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.

9.7 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.

9.8 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. (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`; however, this requires a complex series of instructions. `CBuildCap` can add a tag to an untagged value in a capability-register operand authorized by a second operand holding a suitably authorized capability. This avoids software awareness of the in-memory layout and accelerates 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.

9.9 Reading Capabilities as Bytes

In CHERI, if a non-capability data load instruction such as `LD` 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.

9.10 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 invoking `CInvoke` 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 `CInvoke` on to an object of **otype** O (e.g. after having performed some additional access control checks or auditing first).

9.11 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 `memcpy()`. In CHERI, `memcpy()` copies the tag bit as well as the data so that it can be used to copy structures containing capabilities. As capability registers are dynamically tagged, `memcpy()` 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. `memcpy()` can be regarded as a function whose parameter (technically a `void *`) is dynamically typed.

9.12 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.)

9.13 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.

---

[1] CHERI Concentrate (Section 3.5.4) 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.

9.14 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 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.

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.

9.15 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.

9.16 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 is unsealed (a **otype** of $2^{64} - 1$) and has 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 has 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 C.4 for a less-orthogonal representation.)

9.17 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 constrained to a limited number of 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.)

9.18 Unseal is an Explicit Operation

In CHERI, it requires 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 `CInvoke` 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 [64]: 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.

9.19 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`.

9.20 Instruction-Set Randomization

CHERI does not include features for instruction set randomization [75]; 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.

9.21 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., MMU special register 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, 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 CHERI-specific privileges (i.e., exception-handling capabilities); in the current version, the bit controls all privileges available only in kernel mode including MMU registers and exception return instructions. 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 ring-based authorization are imaginable. For example, the permission bit could grant privileged ISA use in userspace regardless of ring. 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 added new capability-based privilege instructions; instead, we chose to limit use of existing instructions (such as those used in MMU 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 PERMIT_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 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, with a simple and deterministic composition of features.

9.22 CInvoke: Jump-Based Domain Transition

Earlier versions of the CHERI-MIPS ISA included an exception-based mechanism for domain transition via a pair of `CCall` and `CReturn` instructions. The use of exceptions introduced both runtime overhead and implementation complexity in the kernel. We replaced this mechanism with `CInvoke`, which provides jump-like semantics. Non-monotonicity is accomplished by virtue of unsealing the sealed operand capabilities to `CInvoke`.

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 `AUIPCC`.

- A variation that seals caller **PCC** and **IDC** to construct a return-capability pair. We choose not to implement this to avoid multiple register writes in one instruction, and 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 `CInvoke` will be used to invoke trusted software routines. For situations involving mutual distrust, `CInvoke` can be used to invoke a trusted supervisor responsible for mediating messages and requests between distrusting parties. The supervisor would be responsible for clearing non-argument capability and general-purpose integer registers and performing any additional checks. The `CInvoke` 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.

9.23 Compressed Capabilities

In prior CHERI ISA versions, we specified a 256-bit capability representation able to fully represent byte-granularity protection. This allowed 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. We now specify only compressed capabilities, whether 64-bit capabilities for 32-bit architectural addresses, or 128-bit capabilities for 64-bit architectural addresses. Prior versions of our compression approaches, the CHERI-128 candidates, are described in Appendix E.

9.23.1 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.

9.23.2 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 those of 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 legacy 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.

9.23.3 The Value of Architectural Minimum Precision

The `CRepresentableAlignmentMask` and `CRoundRepresentableLength` instructions avoid encoding the details of the capability compression scheme into programs. This is in turn allows different microarchitectures to choose different tradeoffs in the precision of compressed capabilities. With no constraints on implementations, this may lead to unnecessary work when performing operations like loading programs into memory. For example, if linkers allow the generation of static binary load addresses that are insufficiently aligned to be representable, then program loaders must potentially pad the beginning of the mapped region, and enter a loop that adjusts the base and length of the region until they match. This could be avoided if an architectural minimum precision as specified so that an OS ABI could forbid under-aligned load addresses and thus image activators could simply refuse to load such programs rather than having to handle every edge case. Microarchitectures could still implement more precision if they choose and that should be handled by `CRepresentableAlignmentMask` and `CRoundRepresentableLength`.

9.24 Capability Encoding Mode

CHERI-MIPS duplicated 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**-constrained 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.

CHERI-RISC-V instead uses 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**-constrained 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 *x*`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 `CInvoke`), 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 9.25 for further considerations.

Potential encoding mode-switch mechanisms

We are considering the following mechanisms:

New jump instruction sets mode flag in *x*`ccsr` An architecture mode bit, held in *x*`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 *x*`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 *x*`ccsr` Because of a 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. *x*`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 *x*`ccsr`, the lowest bit would persist in **PCC** and be ignored

---

[2] The `JAL` instruction shifts its immediate operand, and could not be used to change mode.

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 *x*`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 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.

9.25 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 `CInvoke`.

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.

9.26 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 have introduced the `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 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.

Full details of the `CLoadTags` instruction may be found on Page 206.

9.27 Attempted Montonicity Violations Clear Tags

To ensure pointer provenance, attempts to violate non-monotonicity, for example to broaden (rather than narrow) bounds, must be forbidden. This can be achieved in several ways. The instruction could throw a hardware exception, or generate a non-deferenceable pointer as its output, in effect deferring the exception until the time of an attempted load, store, or instruction fetch. Both of these implementations ensure monotonicity by preventing derived pointers from improperly allowing increased access following guarded manipulation, and are consistent with the CHERI model.

Initially, in our prototyping, we selected to deliver exceptions as early as possible when such events occur. However, all current CHERI ISA instantiations defer exceptions to the use of a capability's authority, instead clearing the tag on operations that would otherwise violate monotonicity.

The early exception approach offers slightly improved debuggability by exposing the error earlier. Clearing the capability tag may make debugging more expensive (if additional checks are introduced) or more tricky (if loss of the tag is discovered only substantially later).

However, early exceptions limit compiler optimization as instructions that may throw exceptions are restricted in how they can safely be reordered. For example, this prevents a bounds restriction performed within a loop from being hoisted outside the loop, unless that instruction is always executed. If the loop is not always entered, this could turn a conditional execution of a trapping instruction into an unconditional one.[3] In addition, code that manipulates untrusted capabilities is forced to branch when the operation would be illegal, or risk being vulnerable to denial-of-service attacks. This may require it to recreate the hardware-performed checks in software.

With a deferred-exception approach, as well as avoiding these issues, microarchitecture is simplified by reducing the set of instructions that can throw exceptions. While it is initially tempting to delay performing the required checks, forwarding the common-case value and later flushing the pipeline if a check fails, this leads to exploitable speculative side channel attacks. As such, in either approach, microarchitecture must perform the checks before forwarding the result.

Early exceptions can still be achieved if desired for debugging by instrumenting potentially tag-clearing instructions with assertions about the tag, either manually or in a compiler sanitization pass. The CHERI ISA instantiation can ensure these checks are cheap, for example by

---

[3] This is not just a theoretical possibility – we observed this happening in the FreeBSD kernel and had to modify the compiler to avoid hoisting any potentially-trapping CHERI instructions.

providing an instruction to throw an exception based on the tag.

9.28 DDC and PCC Offsetting

Originally, CHERI always treated integer pointers used for legacy memory accesses as offsets. For example, loads and stores using an integer pointer treated integer address as an offset relative to the base of **DDC**[4]. Similarly, branch instructions that targeted an absolute integer pointer set the offset of **PCC** to the value of the integer pointer.

Offsetting also impacted CHERI C in multiple ways. Casts of a capability to an integer value returned the offset of the capability rather than its address. Similarly, casts between capability pointers and integer pointers used special instructions (`CFromPtr` and `CToPtr`), which took the offset of **DDC** into account. Specifically, the compiler would use `CFromPtr` to generate integer pointers that were not an absolute virtual address of an object, but the offset of an object's address relative to the base of **DDC**. Similarly, capability pointers created via casts were derived from **DDC** assuming that the integer pointer was an offset.

To provide consistent semantics with pointer casts, arithmetic operations performed on integer values of capabilities (uintcap_t) used the offset of the capability as the scalar value. For example, to mask off the low bits of a pointer, the compiler fetched the offset, applied the requested mask, and saved the result as the new offset.

Finally, offsetting affected sub-language integer pointers. Integer function pointers, such as those stored in GOT entries, had to store offsets relative to the base of **PCC** rather than absolute addresses. Similarly, pointers to data objects were stored as offsets relative to **DDC**.

As CHERI matured and developers gained more experience, several caveats of this approach arose:

- Using offset interpretation for arithmetic operations on uintcap_t broke several common idioms in pure-capability CHERI C (where uintptr_t is the same as uintcap_t). Aligning pointers did not work reliably since the offset of a misaligned capability was still zero. Code using the integer value of pointers as a key for hash tables would see far more collisions. Due to these types of issues CHERI LLVM switched the default interpretation for arithmetic operations to work with addresses.

  However, this did result in inconsistent semantics compared to pointer casts. In particular, converting between capabilities and integers can have different results if intermediate uintcap_t values are used compared to direct casts.

- Adding the base of **DDC** to the effective address of legacy loads and stores can have a prohibitive cost in microarchitecture.

- Tight bounds for **DDC** and **PCC** for hybrid code required that the hybrid code be relocatable and position independent. For simple support of legacy 64-bit processes for which **DDC** and **PCC** bounds covered the entire user portion of the address space with a base address of 0 this did not matter. However, this was a hurdle for hybrid operating system kernels, which tended to run in a higher range of virtual addresses and were not always

---

[4] Some CHERI instantiations performed offsetting with respect to the address of DDC, rather than the base.

relocatable. In practice hybrid kernels ran with **DDC** and **PCC** whose bounds spanned the entire address space.

- The Morello architecture shipped with knobs to toggle the offsetting behavior of **DDC** and **PCC**. When offsetting was disabled, **DDC** and **PCC** still constrained legacy memory accesses via bounds and permissions, but legacy integer pointers were interpreted as addresses rather than offsets.

CHERI no longer mandates **DDC** and **PCC** offsetting by default. CHERI architectures may provide it as an optional feature, which can be enabled at runtime or may omit it entirely. CHERI compilers always treat integer pointers as addresses using `CSetAddr` to handle conversions between capabilities and integers. The `CFromPtr` and `CToPtr` instructions may be provided on architectures supporting offsetting.

Chapter 10

CHERI in High-Assurance Systems

This chapter considers the roles of formal methods relating to the assurance of CHERI-MIPS 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 and were published in our IEEE Symposium on Security and Privacy in 2020 [114].

10.1 Unpredictable Behavior

In the semantics for the CHERI instructions 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.

10.2 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.

10.3 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, with compressed 128-bit capabilities, there are bit patterns corresponding to $\textbf{base} + \textbf{length} > 2^{64}$. 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.

10.4 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).

10.5 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") [43] 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 [66] 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 [57] 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* $\Rightarrow$ *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* $\Rightarrow$ *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 $\Rightarrow$ **F** *TCBEntryState*

It then follows that *SecureStateTimer* $\Rightarrow$ *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 [67] 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 [87] and Spectre [77] 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 [20, 175].

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.

Chapter 11

Microarchitectural Techniques for CHERI

The CHERI architecture has been designed to fit into modern RISC pipelines without disturbing existing control-flow or data paths. As a result, microarchitectural concerns are simpler than they could otherwise be, but there are nevertheless several innovations that have been developed for prototype implementations that should be considered in any microarchitecture that supports the CHERI model.

The following repositories hold open-source CHERI implementations or libraries, and are referenced in the sections below.

CHERI-MIPS [34] Original reference implementation of the CHERI-MIPS instruction set

cheri-cap-lib [33] A library of reference capability algorithms developed for CHERI-MIPS and adapted for CHERI-RISC-V implementations

TagController [37] Tag controller for emulating a tagged memory using a hierarchical in-memory table developed for CHERI-MIPS and adapted for CHERI-RISC-V implementations

Piccolo [36] CHERI-RISC-V CPU with a simple 3-stage pipeline, for low-end applications (e.g., embedded, IoT)

Flute [35] CHERI-RISC-V CPU with a simple 5-stage in-order pipeline, for low-end applications needing MMUs and some performance

Toooba [38] CHERI-RISC-V CPU with a superscalar, out-of-order pipeline and multi-core capable; based on RISCY-OOO from MIT

11.1 Capabilities in the Pipeline

Capability instructions in the CHERI architecture are modeled after integer operations and are almost entirely single-cycle in the open-source implementations. This makes it possible for a CHERI architecture to unify integer and capability registers and execution paths.

Register File

Capabilities may be stored in an extended, integer register file, or may use a separate, dedicated register file.

The CHERI-MIPS architecture and microarchitecture use a separate capability register file, enabling instructions to access capability operands in addition to two integer operands. CHERI-MIPS uses a dedicated module to perform CHERI operations, but this runs in lock-step with the main pipeline as many capability instructions have integer operands or results, and all legacy memory operations implicitly have **DDC** as a capability operand. This microarchitecture is described in [179]. A superscalar and out-of-order implementation in this style would dedicate execution units to capability operations with ports into the capability register file which would not be necessary in integer execution units, similar to specializations used for floating point execution units.

Our CHERI-RISC-V architecture and microarchitectures use a unified register file for both integers and capabilities. All integer execution units are extended to implement capability manipulation operations.

It should also be possible to implement a unified register-file architecture with a microarchitecturally split register file. One such option would split the physical register file between the lower half, which is used by integer operations, and the upper half which is consumed (and produced) exclusively by capability operations. This division could enable specialization of execution units to reduce the cost of capabilities to integer paths. This would also reduce total register file storage, as all integer operands and results would not require an entry in the capability register file, while complicating renaming due to operands being split between two renamed register files.

11.1.1 Capability Decoding

CHERI capabilities are compressed (see Section 11.2) and various microarchitectures may choose to decode capabilities in stages when there is opportunity in the pipeline. The open-source CHERI implementations, including CHERI-MIPS, Piccolo, Flute, and Toooba, use 3 stages of decompression. The first is the fully-compressed, architectural **in-memory** format. The second is a lightly-decoded **in-register** format that is produced on load from memory. The third is a **pipeline** format that is consumed by high-performance functions in the pipeline and is decoded on read from the register file.

The open-source cheri-cap-lib library (summarised in Appendix D) used in our open-source CHERI implementations expresses these three levels of compression using a typeclass; a microarchitectural "API" to capabilities which includes capability manipulation operations for each level of decompression. The in-register view extracts E (the exponent) into a dedicated field, reconstitutes the top two bits of the T field, and also extracts a_{mid} from the address into a dedicated field. This decompression is fast enough to be performed parallel to the byte-select in the general-purpose loads in the CHERI-MIPS pipeline [178]. In turn, this format is further decoded into the in-pipeline format by adding a few booleans and 2-bit offsets that locate the top and base with respect to the address. These fields are used to perform capability operations such as computing a full top or base, or fast representability checks.

While many variations of the capability typeclass could be useful for a CHERI implementation, these three design points provided in the library have enabled sufficient flexibility to implement an array of optimised hardware microarchitectures.

11.1.2 Program Counter Capability (PCC)

CHERI extends the program counter (PC) with bounds and permissions, which are together called **PCC**. PC is very performance sensitive and is predicted in most microarchitectures.

A processor requires the address of **PCC** at the earliest stage of the pipeline to initiate instruction fetch, but the bounds and permissions of **PCC** are needed only to decide exception conditions and can therefore be checked at any point in the pipeline. The CHERI-MIPS microarchitecture takes advantage of this distinction to speculate on only the address of **PCC** (*i.e.* the address of the instruction to be fetched) using the branch predictor, but forwards updates to **PCC** bounds to the execute stage of the pipeline. This microarchitecture is described in [179]. The in-order Piccolo and Flute microarchitectures share this design.

Forwarding in superscalar and out-of-order pipelines is more complex so we chose to predict the entirety of **PCC** in the Toooba microarchitecture. The BTB delivers both bounds and address, allowing the bounds to be checked early in the pipeline, with a branch misprediction resulting from a mismatch in either the address bits or the bounds and permissions. The Arm implementers of the Morello prototype observe that predicting the bounds, as is done in Toooba, is the optimal performance solution, though it is expensive microarchitecturally. In order to optimise prediction state, a microarchitecture may choose to predict the bounds separately from the address to take advantage of shared bounds and permissions between branch targets.

The Arm Morello design chose to predict the PC address but forward the bounds in the context of the superscalar, out-of-order Ares microarchitecture. In its base Ares design, any readers of the PC track which PC register file (PCrf) entry has their base address. For Morello, the PCrf also has a pointer to which **PCC** register file (PCCrf) entry has its bounds and permissions information. For commit-time bounds checking, Morello walks and broadcasts the PCrf entries that have a “resolved” (known) PCCrf entry so that we can resolve any **PCC** bounds exceptions. The issue queues can also use this broadcast PCrf to release any **PCC** readers who have dependencies without having to rename the PC or track any extra information.

Another way of handling the PC/**PCC** interlock would be to rename PC/**PCC** and do physical tag tracking for dependencies, as is done with any other renamed register.

11.1.3 Default Data Capability (DDC)

CHERI defines a **DDC** register which provides both an offset and bounds for non-capability-aware loads and stores such that the capability mechanism can constrain legacy executables. Both the offset and the bounds for integer addresses require special handling in the microarchitecture.

The CHERI-MIPS architecture added register-offset addressing to the base MIPS instruction set such that a capability-address store has two integer register operands (data and offset) as well as a capability register address operand. This path in the CHERI-MIPS microarchitecture was used to implement standard MIPS loads and stores which also offset through **DDC**.

The CHERI-RISC-V architecture does not introduce a new addressing mode and therefore does not require an additional register operand for addressing in any common case. Our CHERI-RISC-V microarchitectures implement **DDC** as a special, non-forwarded capability register. That is, CSpecialRW modifications of **DDC** traverse the back end of the pipeline alone to avoid consistency issues. **DDC** is then read directly in any place in the pipeline where it is needed directly from the register without forwarding.

In many implementations it may be desirable to optimise the address offset of **DDC**, as an extra add on the address-generation path may be problematic.

For memory operations with an immediate offset, microarchitectures without **DDC** forwarding can add the **DDC** offset to the immediate operand before the Execute stage so that we preserve two-operand address calculation on the critical path. As RISC-V and MIPS exclusively provide immediate-offset addressing, this optimization resolves the issue for these architectures where **DDC** forwarding is not implemented.

For register-offset addressing or for microarchitectures that forward **DDC**, the **DDC**-offset performance might be optimised by an architectural change that limits allowed alignments and possibly sizes of **DDC**. For example, if **DDC** were constrained to be aligned and sized to the same power of two, we may simply OR the **DDC** offset with the resulting address. If the access is in-bounds, the OR will be exactly equivalent to an ADD, and if not the instruction will be cancelled due to the exception. One could also imagine implementations that perform this optimization microarchitecturally such that power-of-two aligned-and-sized **DDC**s operate at full speed and other values of **DDC** fall back to a lower performance mode.

11.1.4 Capability Mode Bit

A CHERI ISA may choose to implement a capability *mode* bit in order to reuse legacy load and store encodings for capability-based loads and stores. This mode bit affects the decoding of instructions, determining the source of bounds for memory access and may also determine alternate decodings of other repurposed instructions.

CHERI-RISC-V specifies a mode bit that is defined in **PCC**. Our in-order open-source implementations, Piccolo and Flute, forward the mode bit along with the bounds to the Execute stage such that mode-bit-dependant decoding does not take place before Execute, which surprisingly fits well into these microarchitectures.

Our Toooba implementation predicts the mode bit with the entirety of **PCC** so that the mode bit is available anywhere in the pipeline including Decode.

If the mode bit is specified in a special register rather than being attached to **PCC**, the same design options should be available. The mode bit might cause a pipeline flush on modification so that the current mode might be accessed anywhere in the pipeline, might be forwarded to allow rapid mode switching, or might be predicted along with **PCC** to optimise repeated mode switches.

11.1.5 Bounds Checking

Many CHERI instructions must check bounds, including all memory-access instructions and many capability-manipulation instructions. The CHERI-MIPS microarchitecture implements all instructions with a single general-purpose bounds-check unit in the pipeline which is never used more than once by any instruction. In addition to this general-purpose bounds-check, there is a bounds-check on **PCC**.

The general-purpose bounds check generally produces exceptions and does not affect destination register values, so the bounds-check in CHERI-MIPS is set up in the Execute stage but is performed during Memory Access.

The general-purpose bounds-check unit must support not only less-than comparison for the top (the common case), but also less-than-or-equal-to for `CSetBounds`. `CSetBounds` also requires extended precision for the upper bound rather than wrap-around arithmetic used by addresses. The bounds check unit supports separate upper and lower address comparisons against the upper and lower bounds respectively in order to validate the highest and lowest byte accessed by a memory transaction, and also to support the TestSubset operation. This shared-bounds-check strategy is also used in the Piccolo and Flute microarchitectures, and the Toooba superscalar microarchitecture has one set of bounds-check logic per integer pipe and memory access pipe with some specialization to the two cases.

The **PCC** bounds check can be optimised in a number of ways.

The CHERI-MIPS and CHERI-RISC-V architectures are designed to remove the need to ever perform a *representable check* on **PCC** modification. While any legacy branch or jump to an integer register could be considered an add to the address of **PCC**, potentially requiring a representability check, the MIPS and RISC-V CHERI architectures avoid this condition by throwing an out-of-bounds exception on the branch using the general-purpose bounds-check for control flow instructions.

All of our open-source CHERI microarchitectures bounds check **PCC** for every instruction executed. As branch targets are guaranteed to be in-bounds, it should be possible for the bounds check on **PCC** to be elided for the majority of instructions, possibly checking **PCC** bounds in batches. Alternatively, we might calculate the number of instructions to the bound on each jump and assert that the distance-to-the-bound counter does not reach zero.

11.1.6 Special Capability Registers

The set of *Special Capability Registers* (SCRs) contain registers with special pipeline implications (such as **DDC** and **PCC**) and also registers to allow simplified privilege escalation which are gated by privilege ring. These registers are accessible only through explicit move-from and move-to GPR instructions.

The CHERI-MIPS implementation implements all SCRs besides **PCC**, **KCC**, and **EPCC** as forwarded general-purpose capability registers. **PCC** is predicted, and **KCC** and **EPCC** are used for swapping with **PCC** on exception and are broken out into dedicated registers without forwarding to enable single-cycle exceptions without the need to access the forwarded register file.

Piccolo, Flute, and Toooba implement all SCRs as non-forwarded registers, blocking the pipeline for the duration of the execution of any SCR modification.

11.2 Compressed Capability Optimizations

The compression scheme used in CHERI-128 and CHERI-64 is partially described in [178] and its key algorithms are implemented in the *cheri-cap-lib* repository. The key algorithms are reproduced in the Appendix D. These algorithms are a significant contributions to the community as they enable reasonably efficient microarchitectural implementations of CHERI.

11.2.1 Decompressing Bounds

Decompressing the bounds of capabilities is highly optimised. Bounds decompression requires detecting the relative positioning between the top and bottom with respect to the address, as described in [178].

The *GetTop* function is more complex than the *GetBase* function, as described in Section D.1, due to the requirement to discern between top being 0 or 2^{64}. This algorithm is a noted contribution as it is non-trivial to develop a correct algorithm that is fast enough for common use in pipelines.

11.2.2 Bounds Checking

There are two types of bounds check in CHERI microarchitectures: precise and representable. Precise bounds checks assert that an address is between the bounds of the capability, but a representable check asserts that a transformation on the address of a compressed capability does not change its bounds due to the limitations of compression.

Precise Bounds Check

Cheri-cap-lib provides *CapInBounds* (listed in Section D.2) which is an algorithm to check that an encoded capability is within bounds without decompressing the bounds of the capability.

We have not required an optimised bounds check which integrates an offset to the address (*e.g.* for offset addressing) as these bounds checks are not on the critical path in our designs, but generally produce exceptions. Precise bounds checks with an offset are usually checked against fully decoded bounds (usually in the next pipeline stage), but could use the *IncOffset* function discussed below followed by *CapInBounds*.

Fast Representability Checks

When the address of a capability is being modified, the algorithm must assert that the resulting capability will still decode the same bounds as the original capability. Custom *fast representability checks* that operate directly on the compressed fields of the encoding are required for each single-cycle operation that modifies the address to avoid dependence on decompressed values which generally require the majority of a cycle to calculate.

The *IncOffset* and *SetOffset* operations are supported by a single function listed in Section D.3 and implement the function described in Section 3.5.4. This shared function allows a single circuit to support both operations.

The *SetAddress* operation must detect if an arbitrary new address is within the representable limits of the capability, and has a distinct implementation listed in Section D.4.

11.2.3 Setting Bounds

The *SetBounds* function listed in Section D.5 and described in Section 3.5.4 provides a single, shared, high-speed function that returns a single data structure containing a capability with the new bounds (to implement `CSetBounds`), a flag indicating if rounding was necessary (to facilitate `CSetBoundsExact`), a mask that could be applied to a pointer to align it with the supplied length (to facilitate `CRepresentableAlignmentMask`), as well as the length that was actually achieved after rounding (to facilitate `CRoundRepresentableLength`).

It is a challenge to implement a *SetBounds* circuit that rounds only when precisely necessary while achieving single-cycle execution. The example algorithm is described in detail in the comments of the listing, and includes pre-computing all fields for both the rounded and unrounded cases while simultaneously detecting if rounding will occur, followed by a select of the correct return values. The rounding detection logic is sophisticated and uses a *smear-right* technique to generate a mask to select bits in the address and length relative to the most significant set bit of the length without waiting for the result of *CountLeadingZeros*. A case matrix is then constructed to detect carry ins to an arbitrary region of the new top based on masked values rather than waiting for the result of full adds.

11.3 Loading and Storing Capabilities

CHERI requires atomic memory access to capability-wide words (*e.g.* 129-bit words).

11.3.1 Capability Width

All open-source CHERI implementations have widened the memory interface between the core and the caches to the capability width. The CHERI-MIPS microarchitecture required that all memory paths were at least capability-width at least to the TagController. For Flute and Piccolo we allow memory paths past the L1 to be half the width of a capability, but never split a capability between bursts. These implementations duplicate the tag bit on the two flits of the capability. The *tag* bit is added to the USER fields of the data channels of AXI (RDATA/WDATA) in these implementations, so it was not possible to transfer less than one tag bit with each flit.

In addition to the cache interfaces, the Toooba microarchitecture enlarged all load and store buffers to at least 129 bits along with all other memory forwarding paths.

The Arm Morello architecture defines store capability pair instructions (*ST{L}XP*) which perform a 32-byte store. Morello implements this in a 16-byte store buffer design which required cracking the operation into two store μops. This adds complexity to handle correct success or failure for two μops instead of one to ensure that the result is atomic such that both succeed or both fail. New logic was required to handle one tag/address for both stores, to handle translation atomically, and to handle writing data from both or none at all. Morello associates adjacent pairs of merge buffer entries so that they atomically retire and write together or fail together.

Alternative solutions to store pairs of capabilities may be preferable in other microarchitectures. To simplify, a design could make the entire store data path 32 bytes wide within the processor including the store buffers and writes to the cache or memory subsystem. Other solutions are highly dependent on individual microarchitectures including how and where they track pass or fail of exclusive instructions, how they write to the data cache and memory subsystem, or how they preserve store order for release semantics.

11.3.2 Capability Permission Complexity

The CHERI architecture requires data-dependent faults such that the address and data must be available for inspection before a store can be issued. Specifically, the architecture defines a PERMIT_STORE_LOCAL_CAPABILITY bit on an address that may trigger a fault if the GLOBAL bit is not set on capability data that is being stored.

The open-source CHERI implementations are based on microarchitectures that do not issue stores unless both operands are available, and so are able to trivially inspect both operands and mark a store for exception if necessary. Most high performance processors will separate address and data issue to release the store address as soon as possible, but might need to delay the store address issue until store data is available in order to capture both the tag and global bit for fault detection. A few other options could be considered if the PERMIT_STORE_LOCAL_CAPABILITY is no longer required in the architecture.

11.4 Tagged Memory

The CHERI capability model requires one extra bit per capability word, *e.g.* CHERI-128 requires 129-bit memory words. This requires changes to the microarchitecture of the memory subsystem to widen structures where possible, or emulate wider memory where it is not.

11.4.1 Tagging Data Caches

Data banks and interfaces of caches can simply be widened to accommodate tagged words. The capability tags can either be stored in the data banks, or the capability tags for a line might be aggregated and stored separately, *e.g* into the record for that line in the cache-tag bank. We have used both approaches in open-source implementations, with the second facilitating `CLoadTags`.

Alternatively, tags could be stored in a separate cache structure that could reduce on-chip storage using compression. This design point would need to solve problems with coherency and would need to integrate into the pipeline such that tag and data pairs are always accessed atomically.

Efficient Tagged Memory [70] discusses each of the above design points in detail.

11.4.2 Tagging Memory

External memory has become a commodity, so there are strong pressures to build systems that support industry-standard interfaces.

Tag Controller with Cache

Our primary approach, described in [70], is a tag controller that allows an external memory controller to emulate a memory holding tagged words. This tag controller maintains a tag table in the external memory and provides the tag bits for each line requested from the remainder of external memory. The tag controller contains a cache of lines from the tag table to reduce tag table accesses. Furthermore, the tag table can be hierarchical such that each bit of a root level indicates whether any bits are set in a block of the leaf table. This structure significantly reduces cache pressure, as a single line of the root table can potentially replace many lines of the leaf (or flat) table.

We have implemented this approach in the open-source TagController project which is used in all of our open-source implementations. This tag controller is parametrisable for arbitrary hierarchy depths and arbitrary block sizes at each level of the hierarchy.

Wide Memory

Commodity external memory may hold memory words wider than its processor word size (*e.g.* ECC memory). A capability system may choose such a memory type and use these bits to hold capability tags alongside data in external memory. As ECC bits typically provide more storage than necessary to hold capability tags, a CHERI system using ECC memory should be able to support in-word capability tags and as well as error detection and correction at a some level.

Dedicated Memory

It is also possible to design a system with a dedicated memory channel for tags. For example, a system with a 1024-bit memory interface (*e.g.* HBM) might add an 8-bit memory interface for accessing tags.

11.4.3 Loading Tags

The CHERI architecture includes a `CLoadTags` instruction to load the tags for a cache line into a register. `CLoadTags` is expected to be cache coherent, so it is not possible to bypass data caches completely, and it is complex to allow greater-than-cache-line granularity.

The CHERI-MIPS implementation will opportunistically return tags from the cache if the line is present, but will not trigger a cache fill based on a miss due to `CLoadTags`, but will forward the request to the next level of cache hierarchy, ultimately hitting the TagController if all caches miss. If the request hits the TagController, it will respond to the request directly, performing an ordinary tag table lookup and caching any results.

The Arm Morello project found that the implementation of `CLoadTags` (*LDCT*) that triggers a cache fill requires special care to wait for all the data to return before responding. This is of concern if the tag bits are stored in the same banks as the data they are associated with and if the data returned is sent in multiple beats (*e.g.* 2). One way to do this is to force the access to be in the middle of the cache line, forcing the load operation to wait until the entire cache line is returned.

Performance for the Arm design would be more ideal if tag loading were tracked separately from data lines to avoid special handling. An alternative, simpler design of LDCT would be to crack the instruction into one microarchitectural tag load for each capability word (*e.g.* 4 tag load μops) and merge their results.

11.5 Speculative Side-channel Precautions

We recommend that CHERI implementations take reasonable precautions to prevent data access through speculative side channels. Many of these observations are explored in [173]. CHERI microarchitectures should reasonably be expected to constrain memory access in speculation to capabilities that are architecturally available to the program. That is, no capability should exist in registers and forwarding paths beyond capabilities in the latest committed architectural state of the register file, those transitively reachable through them, and less-powerful capabilities derived from these. This property, which we may call Speculative Capability Constraint (*SCC*), requires a few microarchitectural features.

11.5.1 Capability-Guarded Cache Access

Permissions and bounds of a memory access should be verified before any cache access is initiated. This is generally reasonable, as these checks are simpler than TLB translation which must also occur before the cache can take action on behalf of a memory access. This property is necessary to support *SCC* by preventing unreachable capabilities from being introduced into the pipeline from memory. All of our open-source implementations have this behavior.

11.5.2 PCC Bounds Forwarding (Not Prediction)

SCC may be violated by a design that predicts the bounds of **PCC**. For example, the Toooba implementation predicts the bounds of **PCC**, storing the entire **PCC** in the branch target buffer. On any jump, it is possible for a powerful **PCC** to be predicted, introducing read rights to new addresses that were not implied by the latest-committed state of the register file. The Morello implementation chooses to forward the bounds of **PCC** rather than predict them, so the **PCC** capability cannot be used in a data memory access unless it is legally sourced form another register in the pipeline. CHERI-MIPS, Piccolo, and Flute share this design choice, though they are of less note as their simple pipelines do not allow speculative read gadgets.

11.5.3 Speculative Forgery Prevention

SCC may also be violated if capabilities can be forged in speculation. CHERI capability manipulation instructions should not speculatively produce capabilities with privilege greater than their operands provide. For example, `CBuildCap` should not forward tagged bits to its consumers while waiting for the result of its bounds check. `CSetBounds` and `CUnseal` share similar concerns. It is believed that all open-source CHERI implementations may currently forward unsafe values for these instructions, and the Toooba microarchitecture is likely vulnerable to speculative execution attacks through this vector. This concern might be more systematically alleviated by an architecture that clears tags rather than throws exceptions for operations that manipulate the privilege of a capability. The ARM Morello architecture generally clears tags rather than throws exceptions for capability manipulation instructions, and we expect the Morello microarchitecture to be immune to this class of attacks for this reason.

In addition, any speculation in a microarchitecture that could synthesise a value rather than deriving it from architecturally-defined bits (*e.g.* value speculation) should not produce valid capability values. This could include not only a capability predicted to be loaded from a memory location, but also a predicted integer value that is used to bound a capability.

If these restrictions become performance-limiting, one could imagine deploying Speculative Taint Tracking [188] to ensure that speculatively-forged capabilities do not affect cache state.

11.5.4 Compartment ID (*CID*) Enforcement

Even if *SCC* is enforced, we may have code paths that manipulate powerful capabilities that must be protected from speculative execution attacks. The CID is described in Section 2.5 to provide an architectural means to convey trust boundaries to the microarchitecture. The CID should be used to tag microarchitectural state to prevent instructions in disparate compartments influencing each other's execution in much the same way as we might hope a modern microarchitecture might prevent user space speculative state influencing kernel execution. For example, the branch target buffer should tag entries with the CID such that targets learned in one compartment would not be used when speculating in another compartment, allowing an attacker to redirect branches that expose powerful capabilities to side-channel gadgets. The branch history table may also be tagged, as well as prefetchers and any other structure that holds state that influences prediction.

The CID itself may be large and require compression in the microarchitecture. For example, a microarchitecture may introduce a table that holds several active CIDs while attaching table indices to all state used for speculation. Such a microarchitecture requires a means to flush state belonging to old table values before installing a new value at an index.

If domain crossing is to be highly optimised, the application of the CID may be imprecise (*e.g.* allowing use of old CID state until the new CID install commits) or the CID itself may be predicted. Either of these may be very difficult to allow without introducing speculative-side-channel vulnerabilities.

Chapter 12

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 its ISA instantiations in MIPS, RISC-V, and ARMv8-A.

12.1 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* [84] provides a detailed history of segment- and capability-based designs through the early 1990s [84]. However, it leaves off just as the transition to microkernel-based capability systems such as Mach [1], L4 [85], and, later, seL4 [76], as well as capability-influenced virtual machines such as the Java Virtual Machine [54], begins. Chapter 13 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 [95] and LLVM [82].

- 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 [1], 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 [153]) 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.

12.1.1 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.

12.1.2 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 [121] and Chromium [124], and more recently in our own Capsicum project [153], 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 13.

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.

12.2 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 [16].

problems, must also offer a longer-term architectural vision able to support further benefit as greater investment is made.

12.2.1 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 ISA.

12.2.2 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.

  For the first few years of the project, we worked with 64-bit MIPS, but we have more recently worked with RISC-V and ARMv8-A. These latter two ISAs have offered the opportunity to revisit our integration with the architecture "from scratch," allowing us to substantially refine our approach – e.g., by employing a merged register file.

- 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.

  In our 64-bit MIPS work, we utilized a single pipelined processor design. For RISC-V, we have employed three microarchitectures spanning a broad spectrum: 3-stage and 5-stage pipelined designs, and a superscalar design.

- 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.

12.3 Research and Development

Between 2010 and 2020, eight major versions of the CHERI model and its architectural instantiations – initially MIPS, and later also RISC-V and ARMv8-A – 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 12.1 including major publications. The major ISA versions, with their development focuses, are described in Table 12.3. 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 to the current MMU-based virtual-memory model, yet also compose with it cleanly [179]. 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

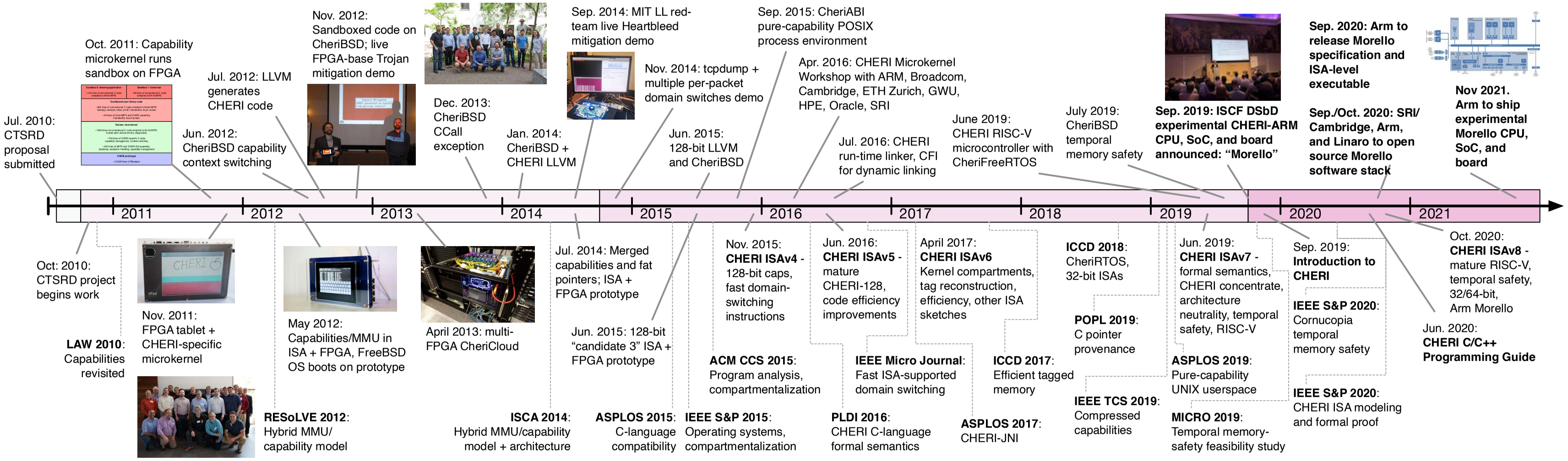

Figure 12.1: CHERI research and development timeline, 2010–2021

(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 [27, 98].

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 [69, 106, 108] 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 [149, 172].

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 [78], 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 [27, 40, 98]. 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 [70, 71]. 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.

2018–2020: Temporal Memory Safety

The potential to support strong and deterministic temporal memory safety was an aim of the CHERI architecture from inception, facilitated by several architectural features: tagged capabilities allowing accurate identification of pointers in memory; composition with an MMU to allow invalidation of portions of the virtual address space; and capability flow control via MMU and capability permission bits limiting where capabilities could be stored in memory. However, we had envisioned this primarily from the perspective of software compartmentalization, and hence a relatively low overall throughput of revoked or invalidated capabilities.

With an increased focus on supporting C and C++ memory safety by implementing all language-level pointers using architectural capabilities, we began to explore whether heap temporal memory safety could perform adequately using these techniques. While for some workloads, they were sufficient, we have expanded the set of architectural tools to support capability revocation through instructions to efficiently access tags across regions of memory, and MMU features to allow tracking of versioned capabilities permitting implementation of load-side barrier techniques similar to those found in garbage collection appearing in CHERI ISAv8. This remains an area of active ongoing research.

2014–2020: Architectural Neutrality

In our earlier research, CHERI was in all senses an extension to the 64-bit MIPS ISA. We took a "ground up" view on developing the model, prototyping in a tight hardware-software co-design loop around the architecture, microarchitecture, and software, as we brought together ideas about capability-system design and the baseline ISA. However, it became apparent that the evolving CHERI protection model in software related to ideas entirely portable across ISAs: C pointer implementation, fine-grained memory protection, and software encapsulation.

In 2014, we began a long-running and still ongoing collaboration with Arm Limited to explore generalizing the model across ISAs, and specifically to integrate it into the 64-bit ARMv8-A architecture. This led to the development of a portable architectural model: concept such as tagged capabilities are essential to CHERI, but not specific to MIPS, ARMv8-A, or other architectures. In 2017, in CHERI ISAv6, we sketched integrations of CHERI with the commercial x86-64 ISA and developing open-source RISC-V ISA. In CHERI ISAv7, we fully elaborated CHERI-RISC-V, and in CHERI ISAv8 we polished many aspects of this design based on the experience of implementing a full hardware-software stack, including three microarchitectures. In 2019, Arm announced Morello [7], both the integration of CHERI into ARMv8-A, and also an experimental board implementing the ISA in a high-end contemporary superscalar processor design.

12.4 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.

12.5 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.

12.6 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.

12.7 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).

12.8 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 32-bit and 64-bit RISC-V 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 8 on CHERI-x86-64 and Arm Morello [7]). However, we have taken a "ground-up" approach utilizing hardware-software co-design to ensure that concrete mapping exist that satisfies the practical engineering requirements of architecture, microarchitecture, compiler, operating system, and applications. At present, our research uses CHERI-RISC-V and Morello.

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.

CHERI-RISC-V and Arm Morello demonstrate that the abstract CHERI protection model, as well as our architectural approach, applies to a range of RISC architectures. 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 [143] or Google Native Client (NaCl) [186]). 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.

12.9 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.

In our CHERI-MIPS work, we developed two pipelined processor variations that incorporated our evolving CHERI-MIPS ISA. These prototypes allowed us to explore ISA design tradeoffs with moderate microarchitectural realism. We implement our prototypes in Bluespec SystemVerilog (BSV) [17], allowing us to create highly parameterizable designs, as well as perform rapid design-space exploration. Wherever possible, we open source our designs to allow reproduction and reuse by other researchers.

In addition to our BSV implementations, we have also implemented executable models using first the L3 ISA modeling language [52], and later SAIL [9]. The SAIL descriptions of CHERI-MIPS and CHERI-RISC-V are used for formal proof, SMT checking, and also directly incorporated into our CHERI ISA specification. We also use an adaptation of the QEMU fast ISA accelerator.

As the CHERI security model is necessarily a hardware-software model, we have also performed substantial experimentation with software stacks targeting all of our ISA instantiations of CHERI. We have an adaptation of the open-source Clang/LLVM compiler suite, LLD linker, and GDB debugger, which are able to generate code using CHERI capabilities. We have adaptations of the FreeRTOS and FreeBSD operating systems to CHERI, known as CheriFreeRTOS and CheriBSD. CheriBSD is portable across all of our CHERI-extended architectures: CHERI-RISC-V and Morello. We use CheriFreeRTOS and CheriBSD on our ISA simulations and also on FGPA.

Throughout, we consider metrics such as microarchitectural disruption, Power Area and Performance (PPA) on FPGA, dynamic benchmark performance, software language and source-code disruption, and security as part of our evaluation cycle.

Table 12.1: 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 [162]	Fat pointers + capabilities, compiler Instructions to optimize hybrid code Sealed capabilities, `CCall`/`CReturn`
2015	ISAv4 [167]	MMU-CHERI integration (TLB permissions) ISA support for compressed capabilities Hardware-accelerated domain switching Multicore instructions: `LL`/`SC` variants
2016	ISAv5 [168]	CHERI-128 compressed capability model Improved generated code efficiency Initial in-kernel privilege limitations
2017	ISAv6 [166]	Mature kernel privilege limitations Further generated code efficiency CHERI-x86 and CHERI-RISC-V sketches Jump-based protection-domain transition
2019	ISAv7 [165]	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
2020	ISAv8 [164]	Compressed capabilities in abstract model 32- and 64-bit address sizes Deployed sentry capabilities Fully elaborated CHERI-RISC-V MMU-assisted load-side-barrier revocation Richer microarchitectural exploration Synchronized with Arm Morello architecture [7]
2023	ISAv9 [163]	CHERI-RISC-V as primary reference platform CHERI-MIPS removed Capabilities stored in general-purpose registers Clear tags for non-montonic modifications DCC and PCC relocation disabled by default CHERI-x86-64 instruction descriptions

Chapter 13

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) [30], 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 [29, 58]. Dennis and Van Horn's 1965 *Programming Semantics for Multiprogrammed Computations* [42] 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 [132].

In 1975, Saltzer and Schroeder surveyed the rapidly expanding vocabulary of computer security in *The Protection of Information in Computer Systems* [133]. 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* [80], 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 [83], 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., [109, 112], among others).

The Security Research community also blossomed outside of MIT: Wulf's HYDRA operating system at Carnegie Mellon University (CMU) [28, 181], Needham and Wilkes' CAP Computer at Cambridge [176], SRI's Provably Secure Operating System (PSOS) [50, 109] hardware-software co-design that included strongly typed object capabilities, Rushby's security kernels supported by formal methods at Newcastle [131], 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 [79, 81]. 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 [128].

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 [4]. 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 [74]. 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 [15]. 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 [104] 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.

13.1 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 [79, 81].

Systems such as the CAP Computer at Cambridge [176] and

Ackerman's DEC PDP-1 architecture at MIT [2] 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 [42]. Levy provides a detailed exploration of segment- and capability-oriented computer system design through the mid-1980s in *Capability-Based Computer Systems* [84].

13.1.1 Objects of Authorization

Dennis and Van Horn's seminal text on capability systems [42] defines a capability as a structure that locates by means of [a unique code or effective name] some computing object, and indicates the actions that the computation may perform with respect to that object." One may then ask what exactly constitutes a "computing object," and one should not be surprised to learn that there have been several answers to that question.

Memory Capability Systems One answer, and perhaps the simplest, identifies a "computing object" with a mere span of memory. Such systems closely resemble traditional segmented memory architectures.

Software Object Capability Systems In so-called "object capability" ("ocap") systems, "computing objects" are identified with pairs of code and private data, as in "object-oriented programming." These systems may treat objects as entirely opaque, exposing only one or a series of "entry" capabilities, corresponding to object methods, or may, additionally, optionally offer more direct access to object data.

Perhaps the most common object capability systems are those built *without* dedicated hardware support. Therein, existing abstractions, such as a privileged supervisory program, are repurposed to provide the capability substrate.

Entry capability invocation ranges from fully synchronous, in which threads of control more or less directly transition from within one object to another, gaining and losing rights as required, to fully asynchronous, in which all invocations are done my means of message passing between other threads. In practice, systems tend to support both options, though they may take one or the other as their sole "primitive" operation.

Ackerman's architecture [2] 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 object capability systems.[1]

Hardware Object Capability Systems Some systems, including the ill-fated Intel iAPX 432 lineage (including the BiiN family of systems and the Intel i960MX), have attempted to move core aspects of the software object capability model into hardware. Exactly which features and object types are manifest in hardware depends on the particular system, but generally one

---

[1] CHERI lacks such a field within its capabilities; however, the `CInvoke` mechanism can be used to similar effect (recall Section 2.3.8).

should expect to see a rich notion of "context" or "domain" of execution and primitives for secure transition from one to another.

13.2 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 [69], and see increasing adaptation to off-the-shelf C programs with work such as Softbound [106], Hardbound [44], and CCured [108]. This flexibility permits a much broader range of common C idiom to be mapped into the capability-based memory-protection model.

13.3 Realizing Capability Systems

When a capability systems is encoded, as software or in hardware or with a mixture of the two, several decisions must be made about how the abstract objects and actions are to be made manifest. We highlight two of the major decisions which must be made.

13.3.1 Memory Layout

A reified capability system 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 *access 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 [176]), emphasising the homogeneous nature of the segment of memory referenced. Some other segmented architectures (notably including the HYDRA microkernel [28, 182]) have bifurcated segments, wherein each contiguous memory segment is effectively two: one containing capabilities and one containing data; in these systems, capabilities typically point at the boundary between the two regions and specify length of each. Naturally, such segments could carry exclusively capabilities or exclusively data.

One occasionally sees designs that straddle the distinction between segmented and tagged architectures. Therein, capabilities may authorize access to both data and other capabilities, but there is no requirement that data and capabilities be separated and contiguous within the authorized memory segment. For example, in the Coyotos microkernel [137], capabilities are stored in pages of memory accessible only to the capability kernel, but these "cap pages" are freely miscible with data pages (either accessible by the user program or solely by the kernel). This compromise allows Coyotos to run on commodity hardware and without a specialized language runtime, but requires indirection to capabilities in memory: data structures contain pointers to addresses that will fault if dereferenced in the user program, but these pointers may be passed to the capability kernel during system calls.

13.3.2 Indirection of Reference to Capabilities

Capability systems also differ in how they name and manipulate capabilities. In a traditional von Neumann architecture, the primitive indirection mechanism is the interpretation of an integer as an index into memory. Virtualization of this architecture is most often accomplished by inserting a mapping function from integer "virtual addresses" to integer "physical addresses" before the latter are given to the hardware's memory subsystem. Dennis and Van Horn built upon this primitive in their initial design of capability systems [42], and so many implementations of capability systems refer to capabilities by *integers*: the capabilities available to the current process are, at least logically, enumerated in a translation table (the "C-list" of the process) and the process uses integers to index into this table. Data fetch operations through capabilities in such systems specify the integer *index* (within this table) of the authorizing capability and the integer *offset* within that capability of the data desired, and the result is placed into an (integer) register. Capability fetch, on the other hand, takes a source index, an offset, and a *target index*: the loaded capability is placed at the target index within the translation table. Dually, while data stores transfer integers from registers to memory, capability stores transfer a capability from the translation table to memory.

This style of manipulation is especially popular in software capability systems, as integers abound and a plethora of key-value mapping data structures are well-understood. Adding another domain of interpretation to integers is even something of a time-honored tradition; the manipulation of "file descriptors" may be so habitual to UNIX programmers, for example, that pausing to reflect that they arise from *design decisions* may be seen as unusual. Software capability kernels can readily implement this design on behalf of their (user) programs, even on commodity hardware, by moving all capability interpretation into the kernel. As discussed earlier, Coyotos repurposes virtual addresses (with backing pages beyond the reach of user programs) as references to capabilities, conflating its capability translation with the translation structures used for address virtualization. One often sees small extensions added in the interest of amortizing the cost of transitions to and from the supervisor; for example, capability fetch and store operations may be willing to traverse *paths* (represented by *sequences* of integers), or multiple capability manipulations may be packaged together, possibly with other operations, into "channel programs" in domain-specific languages interpreted by the capability kernel.

Even in hardware, such designs are attractive, as they can be "drop-in compatible" with existing MMU-style translation systems, requiring no changes to the design of the CPU core. Indeed, the CAP family of computers [176] divides addresses, as generated by the CPU, into capability index and offset fields, much as other systems divide (virtual) addresses into virtual page number and intra-page offset. In systems where safe table manipulation operations are directly exposed to the user program, manipulation of the capability address space can be cheaper than in traditional MMU-style designs, where the MMU mapping structures are highly guarded and manipulated solely by the supervisor.

However, such indirection comes with costs. While the process as a whole may satisfy the principle of least authority, the use of integer indices still presents a challenge to the principle of intensional use. In the CAP systems, for example, indexing past the end of a maximally-sized segment, or, indeed, sufficiently past the end of any segment, will not result in a processor trap, but will, instead, be interpreted as an access within *a different capability*. Even when indices and offsets are maintained separately, there is nothing, architecturally, that ensures the *provenance* of the index as such: to confuse a program into acting using an unintended subset of its authority, it would suffice to corrupt the bits of an integer used as an index.

The use of integer indexes interpreted within a process may also complicate sharing between processes. When sending capabilities to another process, the sender must marshal those capabilities into a (to be) shared segment, as the indices by which the sender refers to these capabilities are useless to the recipient. (The situation is analogous to the passing of integer virtual addresses in shared memory segments: because the segments may be mapped at different offsets by participating processes, even if the addresses reference within the segment in one process, they may be meaningless to another.)

CHERI takes a subtly different approach, in which capabilities are loaded into CPU registers. The instruction stream combines in-register capabilities with offsets to make memory accesses. As with the systems employing separated index and offset fields, it is impossible for an out-of-bounds offset to shift the capability against which the offset is interpreted. Moreover, CHERI ensures the architectural validity of the derivation of the capability being used to authorize the access.[2]

---

[2] That is not to say that it is impossible for a CHERI program to have bugs or to otherwise give an attacker control over which capability is used in a given access. It does, however, rule out a large class of historically powerful attack vectors in which adversarial data is *directly* interpreted as an integer address.

13.4 Capabilities In Hardware

13.4.1 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 [31, 32, 118] and retrospective [10, 91] 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 [177]. 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 [118, 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 [31, 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 [31, ch. 7].

While concerned mostly with detection of software bugs, rather than any consideration of system security, Gumpertz's *Error Detection with Memory Tags* [61] 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" [61, 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.

13.4.2 Segmented Architectures

Cambridge CAP Computer

The family of Cambridge CAP computer designs[3] [176] 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.

---

[3] The CAP experiment seems to have produced one physical, heavily microprogrammed CPU design and at least three different microcode programs.

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]

13.5 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 [41]. 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) [119].

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 [28, 182] 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.[5] 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 [72].

Successors to HYDRA at CMU include Accent and Mach [1, 122], 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 sys-

---

[4] Though CHERI does have its IDC mechanism for compatibility with non-capability programs. Similar confusion 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.

[5] 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* [103]

tem 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 [81]. This further aligned with proposals by Andrews [5] and Rushby [131] for a *security kernel*, 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 [49].

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 [86].

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 [134] and Neumann's Provably Secure Operating System [109] design study were ground-up operating system designs based soundly in formal methodology.[6] In contrast, Schroeder's MLS kernel design for Multics [135], the DoD Kernelized Secure Operating System (KSOS) [94], and Bruce Walker's UCLA UNIX Security Kernel [144] 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 [145].

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 [19]. 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 [136]. 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 [142]. 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.

---

[6] PSOS's ground-up design included ground-up hardware, whereas Schiller's design revised only the software stack.

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 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 [105] – effectively, a composition of mandatory access control and capabilities in service to application security. This approach is embodied by Efstathopoulos et al.'s Asbestos [47] and Zeldovich et al.'s Histar [189] 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 [63], Shapiro's EROS [138] (now CapROS) and Coyotos [137] continued the investigation of higher-assurance software capability designs, and seL4 [76], 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 [6] (which uses Mach interprocess communication (IPC) objects as capabilities) and Cambridge's Capsicum [153] 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 [55, 56]. 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 [103], Wagner's Joe-E capability-safe subset of Java [99], and Miller's Caja capability-safe subset of JavaScript continue a language-level exploration of capability security [101].

13.6 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 [93]. 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 [14], and the DTrace instrumentation tool that, like BPF, uses a narrow virtual instruction set to implement the D language [22].

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 [187]. NaCl is closely related to Software Fault Isolation (SFI) [143], 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 [107]. 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 [69], and type-safe languages such as Ruby [129], 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 [95].

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 [126]; 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 [146]. Web browsers frequently incorporate many of these techniques including Java Virtual Machines and a JavaScript interpreter.

13.7 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 [49]), 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 [39, 117]. 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 [135]. 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 [109, 110]. 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.

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 [150]. 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 [151, 153]. 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.

13.8 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 14

Conclusion

The CHERI project is in its tenth year – an evolution described in detail in Chapter 12. Throughout the project, we have utilized hardware-software co-design, with an increasing focus on also co-designing with formal models. For the last several years, we have increased our focus on transition, including through a close collaboration with Arm in the development of their Morello architecture and hardware prototype [7]. Our contributions include:

1. We developed the CHERI protection model and reference CHERI-MIPS and CHERI-RISC-V Instruction-Set Architectures, which offer low-overhead fine-grained memory protection and support scalable software compartmentalization based on a hybrid capability model. Over several generations of the ISA, we refined integration with conventional RISCs ISA, composed the capability-system model with the MMU, pursued strong C-language compatibility, developed compartmentalization features based on an object-capability model, refined the architecture to improve performance and adoptability through features such as compressed 128-bit capabilities, introduced support for temporal memory safety, and developed the notion of a portable protection model that can be applied to further ISAs. We also explored 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. Employed 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. Elaborated the ISA feature set in CHERI to support real-world operating systems – primarily this has consisted of developing mature compositions of CHERI's concepts

with system features such as the MMU and exception model. 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. Created our CheriBSD and CheriFreeRTOS operating-system prototypes. These reference designs explore how CHERI integration with the OS can improve both OS and application security. We created prototype memory protection and software compartmentalization environments including CheriBSD's CheriABI process environment, and multiple approaches to sandboxing. We have applied CHERI to protection of key system libraries, but also the operating system kernel itself. This has included substantial baseline OS infrastructure work, including porting FreeBSD to the RISC-V architecture.

5. Prototyped, tested, and refined CHERI ISA extensions across multiple CPU architectures. We have open sourced reference MIPS and RISC-V processor designs, and the QEMU ISA-level emulator, on 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. Adapted 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).

7. Began 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.

8. Collaborated with Arm on the creation of Morello [7], a tight integration of CHERI into the ARMv8-A architecture as well as a reference industrial quality microarchitecture. Morello boards will ship for experimental use in late 2021 along with our complete CHERI software stack.

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/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 has been 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.

14.1 Future Work

We have made a strong beginning, but clearly there is still much to do in our remaining CHERI 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 both a high-level summary of prior CHERI ISA versions (Section A.2), and also a detailed change log for each version (Section A.2). This report was previously made available as the *CHERI Architecture Document*, but is now the *CHERI Instruction-Set Architecture*.

A.1 CHERI ISA Specification Version Summary

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 We more clearly differentiate an architecture-neutral CHERI protection model vs. architecture-specific instantiations in 64-bit MIPS, 64-bit RISC-V, and x86-64. We have defined a new capability compression scheme, CHERI Concentrate, and deprecated the previous CHERI-128 scheme. 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.

CHERI ISAv8 / UCAM-CL-TR-951 - 8.0 - October 2020 Capability compression is now part of the abstract model. Both 32-bit and 64-bit architectural address sizes are supported. Various previously experimental features, such as sentry capabilities and CHERI-RISC-V, are now considered mature. We have defined a number of new temporal memory-safety acceleration features including MMU assistance for a load-side-barrier revocation model. We have added a chapter on practical CHERI microarchitecture. CHERI ISAv8 is synchronized with Arm Morello.

CHERI ISAv9 / UCAM-CL-TR-987 - 9.0 - September 2023 CHERI-RISC-V has replaced CHERI-MIPS as the primary reference platform, and CHERI-MIPS has been removed from the specification. CHERI architectures now always use merged register files where existing general-purpose registers are extended to support capabilities. CHERI architectures have adopted two design decisions from Arm Morello: 1) CHERI architectures now clear tags rather than raising exceptions if an instruction attempts a non-monotonic modification of a capability; and 2) **DDC** and **PCC** no longer relocate legacy memory accesses by default. CHERI-RISC-V has received numerous updates to serve as a better baseline for an upstream standard proposal including a more mature definition of compressed instructions in capability mode. CHERI-x86-64 now includes details of extensions to existing x86 instructions and proposed new instructions in a separate ISA reference chapter along with various other updates.

A.2 Detailed CHERI ISA Version Change History

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 `[clbhwd]`CL, `[clbhwd]`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 `[csbhwd]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`, [cptrcmp]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 3.5.3.

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 load and store instructions, including `CLC`, `CSC`, `[clbhwd]`CL[BHWD][U], and `[csbhwd]`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, PERMIT_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, PERMIT_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 PERMIT_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 12. 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 C 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 9.

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 has been clarified to more specifically identify 64-bit MIPS privileged instructions, KSU bits, and general operation modified by the PERMIT_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 9.

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 PERMIT_ACCESS_SYSTEM_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, 64-bit RISC-V (Chapter 4), and x86-64 (Chapter 5).
- Our CHERI-RISC-V prototype has been substantially elaborated, and now includes an experimental encoding in Appendix B. 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 PERMIT_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 our recently released technical report, *Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks* [173], 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 various capability-relative load and store instructions, in which 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.
- We have added a new experimental instruction, Test that Capability is a Subset of Another (`CTestSubset`). 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.
- A description of an experimental *linear capability* model has been added (Section C.6). 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 C.7).
- An experimental appendix considering potential forms of compression for capability permissions has been added (Section C.4).
- 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 [70].

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 PERMIT_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 [40].

- 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 [9].
- 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 [97].
- 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 C.10). 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 C.2).
- 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 (later editions considered these no longer experimental, and so they are described in Section 3.9). These are similar to enter capabilities from the M-Machine [24], 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 C.11).

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 C.5).
- Pseudocode throughout the CHERI ISA specification is now generated from our Sail formal model of the CHERI-MIPS ISA [9].
- 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` which 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 B.
- 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. The CHERI-MIPS chapter has been shortened as a variety of content has been made architectural neutral.

- More detailed rationale is provided for our composition of CHERI with the MIPS exception-handling model.
- 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 3.7.
- We better describe the rationale for tagged capabilities in registers and memory, in contrast to cryptographic and probabilistic protections, in Section 9.2.
- We have made a number of improvements to the CHERI-x86-64 sketch, described in Chapter 5, 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 3.11.4. 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 (Section 2.3.5), which will hold state that can be changed without affecting privilege. Corresponding experimental `CGetFlags` and `CSetFlags` instructions have been added.
- 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 9.
- 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 3.4.2), 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 C.13.
- Initial discussion of a hybrid 64/128-bit capability design has been added to the experimental appendix, in Appendix C.12.
- We have added formal Sail instruction semantics for CHERI-RISC-V; this is currently in Appendix B.
- 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.

8.0 This version of the *CHERI Instruction-Set Architecture* is a full release of the Version 8 specification:

- We have performed modest updates to discuss Arm's Morello processor, System-on-Chip (SoC), and board. The authoritative reference to Morello is Arm's Morello specification [7].
- We have added a new chapter, Chapter 11, describing the impact on CHERI on practical microarchitecture at a high level. It considers topics such as the impact of capabilities on the pipeline and register file, efficient implementation of bounds compression and decompression, fast bounds checking, tagged memory, and the potential interaction with speculative execution. This includes insights gained during the building of Arm's Morello processor and SoC.
- Shift away from the idea that the fully precise 256-bit capabilities are the essential model for CHERI. Instead, describe capabilities as an architectural type made up of a set of architectural fields, which may be constrained in terms of precision, and that have an in-memory representation. The microarchitecture may hold capabilities in another format internally (e.g., when loaded in registers).
- The CHERI Concentrate description has been improved, including adding information about the Fast Representability Check. A number of constants have been updated.
- In several chapters, more care is taken when using the words "capability," "pointer," and "address," which are not interchangeable.
- Throughout, be more clear that CHERI applies to 32-bit architectures, not just 64-bit architectures. In Chapter 3, introduce `XLEN` and `CLEN` terminology previously only used for CHERI-RISC-V, to better abstract away from specific address lengths.
- We are now more clear that capabilities may describe physical as well as virtual addresses, and that virtual addressing may be implemented either using software-managed TLBs (as on MIPS) or via architectural page tables (as on RISC-V and ARMv8-A).

- The `CCall` instruction has been replaced with `CInvoke`, which does not have a selector mechanism and supports only exception-free domain transition. The prior exception-based mechanism was introduced as scaffolding used during domain-transition research, and the exception-free mechanism is our preferred approach. We have removed capability exception cause codes previously used for that purpose, and will likely deprecate `CSetCause`, which was used only for exception-based domain transition.
- The deprecated `CCheckPerm` and `CCheckType` instructions have been removed as they were intended to be used with the exception-based `CCall` mechanism.
- We have removed the experimental 64-bit capability format based on our older CHERI-128 compression model. We now document a 64-bit capability format as part of CHERI Concentrate in Section 3.4.
- We now advocate for a policy of specifying a minimum capability bounds precision, even with `CRRL` and `CRAM` instructions; see Section 9.23.3.
- We now more fully explore, and explain, exception throwing around capability load and store permissions on capabilities and on page-table / TLB entries. MMU-originated exceptions are now distinct from CHERI exceptions. Whereas a capability load through a capability without PERMIT_LOAD_CAPABILITY will always strip the tag, a capability load via a page without load capability permission will either strip the tag or throw an exception. Similarly, a store via a page without permission to store a capability will throw an exception if the tag bit is set on the capability being stored.
- We now discuss the use of per-page capability load barriers to enable efficient capability revocation and garbage collection in Section 3.11.3. Load barriers use additional MMU permissions to selectively trap on capability loads.
- `CPtrCmp` has been changed to compare only the addresses of the two capabilities, and no longer consider the tag bit, for non-exact comparisons. The previous behavior could result in surprising run-time failures of C/C++ programs.
- Sentry capabilities are no longer considered experimental.
- The instruction `CSealEntry` has been added to the ISA quick references, from which it was missing.
- We now more completely elaborate how sentry capabilities interact with exception handling, including automatic unsealing of sentries when installed in exception program counters, not just when they are jumped to.
- Two new instructions (`CGetPCCIncOffset` and `CGetPCCSetAddr`) have been added to improve code density when generating code to create **PCC**-derived capabilities for code or data access. This is of particular use when utilizing sentry capabilities.
- The `CRRL` and `CRAM` instructions, which assist with capability bounds alignment and padding, are no longer considered experimental.
- The `CLoadTags` instruction is no longer considered experimental, and has now also been defined for CHERI-RISC-V.

- We have removed a note on `CSeal` regarding representable unsealed capabilities being unrepresentable as sealed capabilities: all representable capabilities can now be sealed.
- We have a removed a note on `CIncOffset` regarding a special case in which offset of 0 is permitted to operate on sealed capabilities, allowing `CIncOffset` to be used to implement `CMove` as a pseudo-instruction. This is no longer required, as `CMove` is now its own instruction to avoid this special casing.
- `CIncOffset` is no longer specified to clear the base and length if the bounds become unrepresentable, as we guarantee that the cursor will hold the arithmetic result rather than the offset.
- `CSetFlags` no longer incorrectly notes that an exception is thrown if the argument is untagged.
- We have added a new chapter, Chapter 7, listing the semantics and encodings of CHERI-RISC-V instructions.
- Capability flags, and their associated instructions, `CGetFlags` and `CSetFlags`, are no longer considered experimental. On CHERI-RISC-V, a capability flag is used to control what instruction endcoding mode is active. For details, see Sections 2.3.5, 3.4.2 and 4.3.7.
- Certain CHERI-RISC-V SCRs are now defined to exist only when their corresponding extensions (e.g., the N extension) is present.
- In CHERI-RISC-V, when a special capability register is written using `CSpecialRW`, and some bits in the register are defined as WARL (i.e., may be modified during the write), the tag bit will be cleared if the capability value is sealed.
- A new Unaligned Base CHERI exception code has been defined, allowing CHERI-RISC-V to throw an exception if the installed **PCC** value has an unaligned base.
- CHERI-RISC-V encodings are now defined for `CRRL` and `CRAM`, and encoding space is reserved for a future implementation of `CClear` when using a split register file.
- CHERI-RISC-V encodings have been added for the `CSetEqualExact`, `CLoadTags`, and `CClearTags` instructions.
- The CHERI-RISC-V exception cause code has changed from 0x20 to 0x1c due to a collision with encoding space reserved for future use.
- We now define a set of CHERI-RISC-V atomic instructions corresponding to the equivalent base RISC-V atomic instructions.
- We now document which RISC-V CSRs and FCSRs are white listed to not require PERMIT_ACCESS_SYSTEM_REGISTERS, such as the cycle counter.
- Capability exception cause codes are now properly architecture neutral, but the mechanism for obtaining them is more accurately architecture specific.
- CHERI-RISC-V now reports capability-related exception information via the existing RISC-V Trap Value CSRs, x`tval`, rather than through a new capability cause register (a design inherited from CHERI-MIPS that is less consistent with the baseline RISC-V design).

- We added two new exception codes for CHERI-related MMU faults to CHERI-RISC-V. For these exceptions, *x*`tval` holds the address of the faulting memory reference as with existing MMU faults on RISC-V.
- We added additional PTE bits to CHERI-RISC-V to support capability revocation. `CD` provides a capability dirty bit to track pages holding capabilities. `CRM` and `CRG` enable per-page capability load barriers.
- We document the CHERI-RISC-V `CRET`, `CJR`, and `CJALR` assembly aliases.
- We have added a CHERI-RISC-V assembly programming section to the CHERI-RISC-V ISA quick reference.
- There have been numerous updates to our CHERI-x86-64 architectural sketch. The page table permission bits have been adjusted to avoid conflicting with the Protection Keys extension. Violations of the CHERI page table permissions now raise a page fault rather than a CHERI fault. The read capability permission now faults if violated rather than stripping tags. We have removed an ambiguous suggestion for handling RIP-relative addressing.
- The interaction of CHERI with existing memory versioning schemes (e.g., Arm MTE) in Appendix C.5 is now more fully articulated and includes support for integer-pointer versioning instructions as well as a new atomic instruction to manipulate version values in memory.
- A new experimental instruction, `CLCNT`, has been proposed to perform a non-temporal (streaming) load of a capability via a capability. This instruction may prove useful when scanning memory for capabilities in order to implement revocation. We have not yet validated this approach through a full-stack implementation.
- A new experimental instruction, `CClearTags`, has been proposed to perform fast zeroing of multiple tags in memory, and will not allocate cache lines if data is not already present in a cache. This instruction may prove useful when rapidly clearing capabilities for revocation purposes, in the absence of data confidentiality requirements. We have not yet validated this approach through a full-stack implementation.
- A new experimental protection-domain transition mechanism, *sealed indirect enter capabilities*, has been proposed to allow a sealed entry capability to carry not just a pointer to domain-specific code, but also to domain-specific data, by adding an additional level of indirection. A new instruction, `CInvokeInd`, would be used to invoke a sealed indirect enter capability. We have not yet validated this approach through a full-stack implementation. This is described in Section C.8.
- A new *ephemeral capability* type is proposed. Ephemeral capabilities can be held in registers but not stored to memory, and we consider the implications for fast cross-domain calls, and an explicitly maintained delegation tree to provide prompt revocation of delegation sub-trees. We have not yet validated this approach through a full-stack implementation. This is described in Section C.3.
- A new *anti-tamper seal* mechanism is proposed, to allow validation that a delegated capability that has been returned has not been modified (other than its address) while

in use. The suggested use case is around memory allocators, to ensure that a pointer passed to free is consistent with the pointer originally allocated. We have not yet validated this approach through a full-stack implementation. This is described in Section C.9.

- We now describe potential capability-related prefetch instructions in Section 3.11.5, with specific consideration of side-channel attacks. We also explicitly specify **DDC** bounds checking on the MIPS `PREF` prefetch instruction.
- We now reference our papers *CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety* (IEEE MICRO 2019), *Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process* (IEEE SSP 2020), and *Cornucopia: Temporal Safety for CHERI Heaps* (IEEE SSP 2020).

9.0 This version of the *CHERI Instruction-Set Architecture* is a full release of the Version 9 specification:

- We have shifted to CHERI-RISC-V as our primary reference platform instead of CHERI-MIPS. This included several changes to Chapter 2 and Chapter 3 to replace MIPS-specific details with more architectural-neutral concepts. Section 3.8 was also moved to Chapter 3.
- The privileged architecture portions of CHERI-RISC-V are now defined as an extension to version 1.11 of the RISC-V privileged architecture specification.
- CHERI-RISC-V reports capability exception details in *x*`tval` rather than *x*`ccsr`.
- The RISC-V `JAL` and `JALR` instructions are now mode-dependent meaning that they use capability register operands in capability mode rather than always using integer registers. The capability mode version of these instructions are named `CJAL` and `CJALR`. The previous `CJALR` instruction has been renamed to `JALR.CAP`. In addition, `JALR.PCC` has been added to permit integer jump and links in capability mode.
- Section 4.3.8 has been rewritten to reflect an initial implementation of CHERI-RISC-V compressed instructions in capability encoding mode.
- Opcode encodings have been reserved for CHERI-RISC-V memory versioning instructions as well as `CRelocate`.
- CHERI-RISC-V always uses a merged register file and the architecture-neutral chapters now assume a merged register file on all CHERI architectures. This included removing the dirty bit from *x*`ccsr` as well as the `CGetAddr`, `Clear`, and `CSub` instructions.
- CHERI-RISC-V clears tags rather than raising exceptions for non-monotonic modifications to capabilities.
- Added `CGetHigh` and `CSetHigh` to retrieve and modify the upper half of a capability.
- Added `CGetTop` to retrieve the upper limit of a capability.

- **DDC** and **PCC** no longer relocate legacy memory accesses. These registers still constrain legacy memory accesses. This included deprecating `CFromPtr` and `CToPtr`.
- Removed CHERI-MIPS from the specification as it is deprecated and no longer actively developed.
- Added a new section in Chapter 2 describing potential uses of capabilities to protect physical addresses.
- CHERI-RISC-V now enables/disables CHERI extensions via a bit in the `menvcfg` and `senvcfg` CSRs rather than *x*`ccsr`.
- CHERI-RISC-V *x***ScratchC** capability registers now extend the existing *x*`scratch` registers.
- We have expanded the CHERI-x86-64 sketch in Chapter 5 to include details on extensions to existing instructions to support operations on capabilities as well as details for new instructions in a new ISA reference in Chapter 8.
- Added a description of the 64-bit CHERI Concentrate capability format.

Appendix B

CHERI-RISC-V ISA Quick Reference

B.1 Primary New Instructions

The RISC-V specification reserves 4 major opcodes for extensions: 11 (0xb / 0b0001011), 43 (0x2b / 0b0101011), 91 (0x5b / 0b1011011), and 123 (0x7b / 0b1111011). The proposed CHERI encodings use major opcode 0x5b for all capability instructions.
All register-register operations use the RISC-V R-type or I-type encoding formats.

B.1.1 Capability-Inspection Instructions

31 25	24 20	19 15	14 12	11 7	6 0	
0x7f	0x0	cs1	0x0	rd	0x5b	`CGetPerm` rd, cs1
0x7f	0x1	cs1	0x0	rd	0x5b	`CGetType` rd, cs1
0x7f	0x2	cs1	0x0	rd	0x5b	`CGetBase` rd, cs1
0x7f	0x3	cs1	0x0	rd	0x5b	`CGetLen` rd, cs1
0x7f	0x4	cs1	0x0	rd	0x5b	`CGetTag` rd, cs1
0x7f	0x5	cs1	0x0	rd	0x5b	`CGetSealed` rd, cs1
0x7f	0x6	cs1	0x0	rd	0x5b	`CGetOffset` rd, cs1
0x7f	0x7	cs1	0x0	rd	0x5b	`CGetFlags` rd, cs1
0x7f	0x17	cs1	0x0	rd	0x5b	`CGetHigh` rd, cs1
0x7f	0x18	cs1	0x0	rd	0x5b	`CGetTop` rd, cs1

B.1.2 Capability-Modification Instructions

31 25	24 20	19 15	14 12	11 7	6 0	
0xb	cs2	cs1	0x0	cd	0x5b	`CSeal` cd, cs1, cs2
0xc	cs2	cs1	0x0	cd	0x5b	`CUnseal` cd, cs1, cs2

0xd	rs2	cs1	0x0	cd	0x5b	CAndPerm cd, cs1, rs2
0xe	rs2	cs1	0x0	cd	0x5b	CSetFlags cd, cs1, rs2
0xf	rs2	cs1	0x0	cd	0x5b	CSetOffset cd, cs1, rs2
0x10	rs2	cs1	0x0	cd	0x5b	CSetAddr cd, cs1, rs2
0x11	rs2	cs1	0x0	cd	0x5b	CIncOffset cd, cs1, rs2
imm[11:0]		cs1	0x1	cd	0x5b	CIncOffsetImm cd, cs1, imm
0x8	rs2	cs1	0x0	cd	0x5b	CSetBounds cd, cs1, rs2
0x9	rs2	cs1	0x0	cd	0x5b	CSetBoundsExact cd, cs1, rs2
uimm[11:0]		cs1	0x2	cd	0x5b	CSetBoundsImm cd, cs1, uimm
0x16	rs2	cs1	0x0	cd	0x5b	CSetHigh cd, cs1, rs2
0x7f	0xb	cs1	0x0	cd	0x5b	CClearTag cd, cs1
0x1d	cs2	cs1	0x0	cd	0x5b	CBuildCap cd, cs1, cs2
0x1e	cs2	cs1	0x0	cd	0x5b	CCopyType cd, cs1, cs2
0x1f	cs2	cs1	0x0	cd	0x5b	CCSeal cd, cs1, cs2
0x7f	0x11	cs1	0x0	cd	0x5b	CSealEntry cd, cs1

B.1.3 Pointer-Arithmetic Instructions

31 25	24 20	19 15	14 12	11 7	6 0	
0x12	cs2	cs1	0x0	rd	0x5b	CToPtr rd, cs1, cs2
0x13	rs2	cs1	0x0	cd	0x5b	CFromPtr cd, cs1, rs2
0x7f	0xa	cs1	0x0	cd	0x5b	CMove cd, cs1

B.1.4 Pointer-Comparison Instructions

0x20	cs2	cs1	0x0	rd	0x5b	CTestSubset rd, cs1, cs2
0x21	cs2	cs1	0x0	rd	0x5b	CSetEqualExact rd, cs1, cs2

B.1.5 Control-Flow Instructions

0x7f	0xc	cs1	0x0	cd	0x5b	JALR.CAP cd, cs1
0x7f	0x14	rs1	0x0	rd	0x5b	JALR.PCC rd, rs1

0x7e	cs2	cs1	0x0	0x1	0x5b	CInvoke cs1, cs2

B.1.6 Special Capabilty Register Access Instructions

0x1	scr	cs1	0x0	cd	0x5b	CSpecialRW cd, scr, cs1

B.1.7 Fast Register-Clearing Instructions

31 25	24 20	19 18	17 15	14 12	11 7	6 0	
0x7f	0xe	q	$m_{[7:5]}$	0x0	$m_{[4:0]}$	0x5b	CClear q(uarter), m(ask)
0x7f	0x10	q	$m_{[7:5]}$	0x0	$m_{[4:0]}$	0x5b	FPClear q(uarter), m(ask)

B.1.8 Adjusting to Compressed Capability Precision Instructions

0x7f	0x8	rs1	0x0	rd	0x5b	CRoundRepresentableLength rd, rs1
0x7f	0x9	rs1	0x0	rd	0x5b	CRepresentableAlignmentMask rd, rs1

B.1.9 Tag-Memory Access Instructions

0x7f	0x12	cs1	0x0	rd	0x5b	CLoadTags rd, cs1
0x7e	0x0	cs1	0x0	0x1f	0x5b	CClearTags cs1

B.1.10 Memory Loads with Explicit Address Type Instructions

These memory load instructions explicitly expect either capability addresses or integer addresses, with bounds coming either from cs1 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 constrained. 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 constrained. 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	0x0	rd	0x5b	`LB.DDC` rd, rs1
0x7d	0x01	rs1	0x0	rd	0x5b	`LH.DDC` rd, rs1
0x7d	0x02	rs1	0x0	rd	0x5b	`LW.DDC` rd, rs1
0x7d	0x03	rs1	0x0	cd	0x5b	`LC.DDC` cd, rs1 *(RV32)*
0x7d	0x03	rs1	0x0	rd	0x5b	`LD.DDC` rd, rs1 *(RV64/128)*
0x7d	0x17	rs1	0x0	cd	0x5b	`LC.DDC` cd, rs1 *(RV64)*
0x7d	0x17	rs1	0x0	rd	0x5b	`LQ.DDC` rd, rs1 *(RV128)*
0x7d	0x04	rs1	0x0	rd	0x5b	`LBU.DDC` rd, rs1
0x7d	0x05	rs1	0x0	rd	0x5b	`LHU.DDC` rd, rs1
0x7d	0x06	rs1	0x0	rd	0x5b	`LWU.DDC` rd, rs1 *(RV64/128)*
0x7d	0x07	rs1	0x0	rd	0x5b	`LDU.DDC` rd, rs1 *(RV128)*
0x7d	0x08	cs1	0x0	rd	0x5b	`LB.CAP` rd, cs1
0x7d	0x09	cs1	0x0	rd	0x5b	`LH.CAP` rd, cs1
0x7d	0x0a	cs1	0x0	rd	0x5b	`LW.CAP` rd, cs1
0x7d	0x0b	cs1	0x0	cd	0x5b	`LC.CAP` cd, cs1 *(RV32)*
0x7d	0x0b	cs1	0x0	rd	0x5b	`LD.CAP` rd, cs1 *(RV64/128)*
0x7d	0x1f	cs1	0x0	cd	0x5b	`LC.CAP` cd, cs1 *(RV64)*
0x7d	0x1f	cs1	0x0	rd	0x5b	`LQ.CAP` rd, cs1 *(RV128)*
0x7d	0x0c	cs1	0x0	rd	0x5b	`LBU.CAP` rd, cs1
0x7d	0x0d	cs1	0x0	rd	0x5b	`LHU.CAP` rd, cs1
0x7d	0x0e	cs1	0x0	rd	0x5b	`LWU.CAP` rd, cs1 *(RV64/128)*
0x7d	0x0f	cs1	0x0	rd	0x5b	`LDU.CAP` rd, cs1 *(RV128)*
0x7d	0x10	rs1	0x0	rd	0x5b	`LR.B.DDC` rd, rs1
0x7d	0x11	rs1	0x0	rd	0x5b	`LR.H.DDC` rd, rs1

0x7d	0x12	rs1	0x0	rd	0x5b	`LR.W.DDC` rd, rs1
0x7d	0x13	rs1	0x0	cd	0x5b	`LR.C.DDC` cd, rs1 *(RV32)*
0x7d	0x13	rs1	0x0	rd	0x5b	`LR.D.DDC` rd, rs1 *(RV64/128)*
0x7d	0x14	rs1	0x0	cd	0x5b	`LR.C.DDC` cd, rs1 *(RV64)*
0x7d	0x14	rs1	0x0	rd	0x5b	`LR.Q.DDC` rd, rs1 *(RV128)*
0x7d	0x18	cs1	0x0	rd	0x5b	`LR.B.CAP` rd, cs1
0x7d	0x19	cs1	0x0	rd	0x5b	`LR.H.CAP` rd, cs1
0x7d	0x1a	cs1	0x0	rd	0x5b	`LR.W.CAP` rd, cs1
0x7d	0x1b	cs1	0x0	cd	0x5b	`LR.C.CAP` cd, cs1 *(RV32)*
0x7d	0x1b	cs1	0x0	rd	0x5b	`LR.D.CAP` rd, cs1 *(RV64/128)*
0x7d	0x1c	cs1	0x0	cd	0x5b	`LR.C.CAP` cd, cs1 *(RV64)*
0x7d	0x1c	cs1	0x0	rd	0x5b	`LR.Q.CAP` rd, cs1 *(RV128)*

B.1.11 Memory Stores with Explicit Address Type Instructions

These memory store instructions explicitly expect either capability addresses or integer addresses, with bounds coming either from cs1 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 constrained. 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	0x0	0x00	0x5b	`SB.DDC` rs2, rs1
0x7c	rs2	rs1	0x0	0x01	0x5b	`SH.DDC` rs2, rs1
0x7c	rs2	rs1	0x0	0x02	0x5b	`SW.DDC` rs2, rs1
0x7c	cs2	rs1	0x0	0x03	0x5b	`SC.DDC` cs2, rs1 *(RV32)*
0x7c	rs2	rs1	0x0	0x03	0x5b	`SD.DDC` rs2, rs1 *(RV64/128)*
0x7c	cs2	rs1	0x0	0x04	0x5b	`SC.DDC` cs2, rs1 *(RV64)*
0x7c	rs2	rs1	0x0	0x04	0x5b	`SQ.DDC` rs2, rs1 *(RV128)*
0x7c	rs2	cs1	0x0	0x08	0x5b	`SB.CAP` rs2, cs1

0x7c	rs2	cs1	0x0	0x09	0x5b	SH.CAP rs2, cs1
0x7c	rs2	cs1	0x0	0x0a	0x5b	SW.CAP rs2, cs1
0x7c	cs2	cs1	0x0	0x0b	0x5b	SC.CAP cs2, cs1 *(RV32)*
0x7c	rs2	cs1	0x0	0x0b	0x5b	SD.CAP rs2, cs1 *(RV64/128)*
0x7c	cs2	cs1	0x0	0x0c	0x5b	SC.CAP cs2, cs1 *(RV64)*
0x7c	rs2	cs1	0x0	0x0c	0x5b	SQ.CAP rs2, cs1 *(RV128)*
0x7c	rd/rs2	rs1	0x0	0x10	0x5b	SC.B.DDC rs2, rs1
0x7c	rd/rs2	rs1	0x0	0x11	0x5b	SC.H.DDC rs2, rs1
0x7c	rd/rs2	rs1	0x0	0x12	0x5b	SC.W.DDC rs2, rs1
0x7c	cd/cs2	rs1	0x0	0x13	0x5b	SC.C.DDC cs2, rs1 *(RV32)*
0x7c	rd/rs2	rs1	0x0	0x13	0x5b	SC.D.DDC rs2, rs1 *(RV64/128)*
0x7c	cd/cs2	rs1	0x0	0x14	0x5b	SC.C.DDC cs2, rs1 *(RV64)*
0x7c	rd/rs2	rs1	0x0	0x14	0x5b	SC.Q.DDC rs2, rs1 *(RV128)*
0x7c	rd/rs2	cs1	0x0	0x18	0x5b	SC.B.CAP rs2, cs1
0x7c	rd/rs2	cs1	0x0	0x19	0x5b	SC.H.CAP rs2, cs1
0x7c	rd/rs2	cs1	0x0	0x1a	0x5b	SC.W.CAP rs2, cs1
0x7c	cd/cs2	cs1	0x0	0x1b	0x5b	SC.C.CAP cs2, cs1 *(RV32)*
0x7c	rd/rs2	cs1	0x0	0x1b	0x5b	SC.D.CAP rs2, cs1 *(RV64/128)*
0x7c	cd/cs2	cs1	0x0	0x1c	0x5b	SC.C.CAP cs2, cs1 *(RV64)*
0x7c	rd/rs2	cs1	0x0	0x1c	0x5b	SC.Q.CAP rs2, cs1 *(RV128)*

B.2 Memory-Access via Capability with Offset Instructions

B.2.1 Memory-Access Instructions

When using 64-bit capabilities in RV32, the RV64 instructions LD and SD are reused to behave as LC and SC respectively.

31 25	24 20	19 15	14 12	11 7	6 0	
imm		rs1	0x3	cd	0x3	LC cd, rs1, imm *(RV32)*
imm[11:5]	cs2	rs1	0x3	imm[4:0]	0x23	SC cs2, rs1, imm *(RV32)*

When using 128-bit capabilities in RV64, the RV128 instructions `LQ` and `SQ` *(anticipated encoding)* are reused to behave as `LC` and `SC` respectively.

31 25	24 20	19 15	14 12	11 7	6 0	
imm		rs1	0x2	cd	0xf	`LC` cd, rs1, imm *(RV64)*
imm[11:5]	cs2	rs1	0x4	imm[4:0]	0x23	`SC` cs2, rs1, imm *(RV64)*

B.2.2 Atomic Memory-Access Instructions

When using 64-bit capabilities in RV32, the RV64A instructions `LR.D`, `SC.D` and `AMOSWAP.D` are reused to behave as `LR.C`, `SC.C` and `AMOSWAP.C` respectively.

31 27	26	25	24 20	19 15	14 12	11 7	6 0	
0x2	aq	rl	0x0	rs1	0x3	cd	0x2f	`LR.C` cd, rs1 *(RV32)*
0x3	aq	rl	cs2	rs1	0x3	rd	0x2f	`SC.C` rd, cs2, rs1 *(RV32)*
0x1	aq	rl	cs2	rs1	0x3	cd	0x2f	`AMOSWAP.C` cd, cs2, rs1 *(RV32)*

When using 128-bit capabilities in RV64, the RV64A instructions `LR.Q`, `SC.Q` and `AMOSWAP.Q` *(anticipated encoding)* are reused to behave as `LR.C`, `SC.C` and `AMOSWAP.C` respectively.

31 27	26	25	24 20	19 15	14 12	11 7	6 0	
0x2	aq	rl	0x0	rs1	0x4	cd	0x2f	`LR.C` cd, rs1 *(RV64)*
0x3	aq	rl	cs2	rs1	0x4	rd	0x2f	`SC.C` rd, cs2, rs1 *(RV64)*
0x1	aq	rl	cs2	rs1	0x4	cd	0x2f	`AMOSWAP.C` cd, cs2, rs1 *(RV64)*

We do not provide any of the other AMOs at this point when operating on capability values, as they generally make sense only when operating on integer values.

Since capabilities have precise bounds, sub-word atomics cannot be implemented using word-sized atomics. To avoid unnecessary complexity compared with a non-CHERI RISC-V implementation, we define only `LR.B`, `SC.B`, `LR.H` and `SC.H`, without any of the corresponding AMOs. We also require these to be present only in capability mode, but implementations may choose to always provide them for simplicity.

31 27	26	25	24 20	19 15	14 12	11 7	6 0	
0x2	aq	rl	0x0	rs1	0x0	rd	0x2f	`LR.B` rd, rs1
0x3	aq	rl	rs2	rs1	0x0	rd	0x2f	`SC.B` rd, rs2, rs1
0x2	aq	rl	0x0	rs1	0x1	rd	0x2f	`LR.H` rd, rs1
0x3	aq	rl	rs2	rs1	0x1	rd	0x2f	`SC.H` rd, rs2, rs1

B.3 Assembly Programming

B.3.1 Capability Register ABI Names

Table B.1 lists the ABI names of the capability registers. The ABI names follow from the ABI names of the RISC-V **x** registers. All capability registers are Caller-Save in the hybrid ABI. Capability registers follow the same save requirements as **x** registers in the purecap ABI.

Register	ABI Name	Description	Hybrid Saver	Purecap Saver
c0	cnull	NULL pointer	-	-
c1	cra	Return address	Caller	Caller
c2	csp	Stack pointer	Caller	Callee
c3	cgp	Global pointer	-	-
c4	ctp	Thread pointer	-	-
c5	ct0	Temporary/alternate link register	Caller	Caller
c6-7	ct1-2	Temporaries	Caller	Caller
c8	cs0/cfp	Saved register/frame pointer	Caller	Callee
c9	cs1	Saved register	Caller	Callee
c10-11	ca0-1	Function arguments/return values	Caller	Caller
c12-17	ca2-7	Function arguments	Caller	Caller
c18-27	cs2-11	Saved registers	Caller	Callee
c28-31	ct3-6	Temporaries	Caller	Caller

Table B.1: Assembler mnemonics for CHERI RISC-V capability registers

B.3.2 Capability Encoding Mode Instructions

Table B.2 lists uncompressed instructions which change semantics under capability mode. Table B.3 lists compressed instructions which change semantics under capability mode. Table B.4 lists psuedoinstructions removed in capability mode. Table B.5 lists psuedoinstructions added in capability mode.

Integer Instruction	Capability Instruction
`l{b|h|w|d}[u] rd, offset(rs1)`	`cl{b|h|w|d}[u] rd, offset(cs1)`
`lc cd, offset(rs1)`	`clc cd, offset(cs1)`
`s{b|h|w|d} rs2, offset(rs1)`	`cs{b|h|w|d} rs2, offset(cs1)`
`sc rs2, offset(rs1)`	`csc cs2, offset(cs1)`
`fl{h|w|d|q} fd, offset(rs1)`	`cfl{h|w|d|q} fd, offset(cs1)`
`fs{h|w|d|q} fs2, offset(rs1)`	`cfs{h|w|d|q} fs2, offset(cs1)`
`lr.{b|h|w|d} rd, (rs1)`	`clr.{b|h|w|d} rd, (cs1)`
`lr.c cd, (rs1)`	`clr.c cd, (cs1)`
`sc.{b|h|w|d} rd, rs2, (rs1)`	`csc.{b|h|w|d} rd, rs2, (cs1)`
`sc.c cd, cs2, (rs1)`	`csc.c cd, cs2, (cs1)`
`amo<op>.{w|d}[.order] rd, rs2, (rs1)`	`camo<op>.{w|d}[.order] rd, rs2, (cs1)`
`amo<op>.c[.order] cd, cs2, (rs1)`	`camo<op>.c[.order] cd, cs2, (cs1)`
`auipc rd, offset`	`auipcc cd, offset`

Table B.2: Uncompressed Instructions Dependent on Encoding Mode

Integer Instruction	Capability Instruction	ISA
`c.addi16sp sp, offset`	`c.cincoffsetimm16csp csp, offset`	-
`c.addi4spn rd, sp, offset`	`c.cincoffsetimm4cspn cd, csp, offset`	-
`c.jal offset`	`c.cjal offset`	RV32
`c.jr rs1`	`c.cjr cs1`	-
`c.jalr rs1`	`c.cjalr cs1`	-
`c.l{w|d} rd, offset(rs1)`	`c.cl{w|d} rd, offset(cs1)`	-
`c.l{w|d}sp rd, offset(sp)`	`c.cl{w|d}csp rd, offset(csp)`	-
`c.s{w|d} rs2, offset(rs1)`	`c.cs{w|d} rs2, offset(cs1)`	-
`c.s{w|d}sp rs2, offset(sp)`	`c.cs{w|d}csp rs2, offset(csp)`	-
`c.flw fd, offset(rs1)`	`c.clc cd, offset(cs1)`	RV32
`c.flwsp fd, offset(sp)`	`c.clccsp cd, offset(csp)`	RV32
`c.fsw fs2, offset(rs1)`	`c.csc cs2, offset(cs1)`	RV32
`c.fswsp fs2, offset(sp)`	`c.csccsp cs2, offset(csp)`	RV32
`c.fld fd, offset(rs1)`	`c.cfld fd, offset(cs1)`	RV32
`c.fldsp fd, offset(sp)`	`c.cfldcsp fd, offset(csp)`	RV32
`c.fsd fs2, offset(rs1)`	`c.cfsd fs, offset(cs1)`	RV32
`c.fsdsp fs2, offset(sp)`	`c.cfsdcsp fs, offset(csp)`	RV32
`c.fld fd, offset(rs1)`	`c.clc cd, offset(cs1)`	RV64
`c.fldsp fd, offset(sp)`	`c.clccsp cd, offset(csp)`	RV64
`c.fsd fs2, offset(rs1)`	`c.csc cs, offset(cs1)`	RV64
`c.fsdsp fs2, offset(sp)`	`c.csccsp cs, offset(csp)`	RV64

Table B.3: Compressed Instructions Dependent on Encoding Mode

Pseudoinstruction	Meaning
`la rd, symbol`	Load address
`lla rd, symbol`	Load local address
`l{b|h|w|d} rd, symbol`	Load global
`s{b|h|w|d} rd, symbol, rt`	Store global
`fl{w|d} rd, symbol, rt`	Floating-point load global
`fs{w|d} rd, symbol, rt`	Floating-point store global
`call symbol`	Call far-away subroutine
`tail symbol`	Tail call far-away subroutine

Table B.4: Pseudoinstructions Removed in Capability Mode

Pseudoinstruction	Base Instruction(s)	Meaning
`clgc cd, sym`	`1:  auipcc cd, %captab_pcrel_hi(sym)` `clc cd, %pcrel_lo(1b)(cd)`	Load from capability table
`cllc cd, sym`	`1:  auipcc cd, %pcrel_hi(sym)` `cincoffset cd, cd, %pcrel_lo(1b)`	Load PCC-relative capability
`cjr cs`	`cjalr cnull, cs`	Jump to capability
`cjalr cs`	`cjalr cra, cs`	Jump to capability and link
`cret`	`cjalr cnull, cra`	Return to capability
`cspecialr cd, scr`	`cspecialrw cd, scr, cnull`	Read special capability register
`cspecialw scr, cs`	`cspecialrw cnull, scr, cs`	Write special capability register

Table B.5: Pseudoinstructions Added in Capability Mode

B.4 Encoding Summary

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

Top-level encoding allocation (funct3 field)

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

Two Source & Dest encoding allocation (funct7 field)

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

func	rs2/cs2	cs1	0x0	cd	0x5b

	00	01	10	11
00000	-	CSpecialRW	CSetVersion[*]	CAmoCDecVersion[*]
00001	-	-	-	-
00010	CSetBounds	CSetBoundsExact	-	CSeal
00011	CUnseal	CAndPerm	CSetFlags	CSetOffset
00100	CSetAddr	CIncOffset	CToPtr[‡]	CFromPtr[‡]
00101	CSub[†]	CRelocate[*]	CSetHigh	-
00110	-	-	-	-
00111	-	CBuildCap	CCopyType	CCSeal
01000	CTestSubset	CSEQX	-	-
01001	-	-	-	-
01010	-	-	-	-
01011	-	-	-	-
01100	-	-	-	-
01101	-	-	-	-
01110	-	-	-	-
01111	-	-	-	-
10000	-	-	-	-
10001	-	-	-	-
10010	-	-	-	-
10011	-	-	-	-
10100	-	-	-	-
10101	-	-	-	-
10110	-	-	-	-
10111	-	-	-	-
11000	-	-	-	-
11001	-	-	-	-
11010	-	-	-	-
11011	-	-	-	-
11100	-	-	-	-
11101	-	-	-	-
11110	-	-	-	-
11111	Stores	Loads	Two Source	Source & Dest

[†] Previously used by a removed instruction.

[‡] Deprecated (may be removed in a future version).

[*] Reserved for future use.

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/cs2	rs1/cs1	0x0	func	0x5b

	00	01	10	11
000	SB.DDC	SH.DDC	SW.DDC	SD.DDC
001	SQ.DDC	-	-	-
010	SB.CAP	SH.CAP	SW.CAP	SD.CAP
011	SQ.CAP	-	-	-
100	SC.B.DDC†	SC.H.DDC†	SC.W.DDC†	SC.D.DDC†
101	SC.Q.DDC†	-	-	-
110	SC.B.CAP†	SC.H.CAP†	SC.W.CAP†	SC.D.CAP†
111	SC.Q.CAP†	-	-	-

†The SC.{B, H, W, D, Q}.{DDC, CAP} instructions are available only 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/cs1	0x0	rd/cd	0x5b

	00	01	10	11
000	LB.DDC	LH.DDC	LW.DDC	LD.DDC
001	LBU.DDC	LHU.DDC	LWU.DDC	LDU.DDC‡
010	LB.CAP	LH.CAP	LW.CAP	LD.CAP
011	LBU.CAP	LHU.CAP	LWU.CAP	LDU.CAP‡
100	LR.B.DDC†	LR.H.DDC†	LR.W.DDC†	LR.D.DDC†
101	LR.Q.DDC†	-	-	LQ.DDC
110	LR.B.CAP†	LR.H.CAP†	LR.W.CAP†	LR.D.CAP†
111	LR.Q.CAP†	-	-	LQ.CAP

†The LR.{B, H, W, D, Q}.{DDC, CAP} instructions are available only when the RISC-V A extension (atomic) is present.
‡LDU.{DDC, CAP} instructions are available only in RV128.

Two Source encoding allocation (rd field)

Two Source instructions are of the following form:

31 25	24 20	19 15	14 12	11 7	6 0
0x7e	rs2/cs2	rs1/cs1	0x0	func	0x5b

	00	01	10	11
000	-	CInvoke	CStoreVersion[†]	-
001	-	-	-	-
010	-	-	-	-
011	-	-	-	-
100	-	-	-	-
101	-	-	-	-
110	-	-	-	-
111	-	-	-	One Source

[†] Reserved for future use.

One Source encoding allocation (rs2 field)

One Source instructions are of the following form:

31 25	24 20	19 15	14 12	11 7	6 0
0x7e	func	rs1/cs1	0x0	0x1f	0x5b

	00	01	10	11
000	CClearTags[†]	-	-	-
001	-	-	-	-
010	-	-	-	-
011	-	-	-	-
100	-	-	-	-
101	-	-	-	-
110	-	-	-	-
111	-	-	-	-

[†] Reserved for future use.

Source & Dest encoding allocation (rs2 field)

Source & Dest instructions are of the following form:

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	func	rs1/cs1	0x0	rd/cd	0x5b

	00	01	10	11
000	CGetPerm	CGetType	CGetBase	CGetLen
001	CGetTag	CGetSealed	CGetOffset	CGetFlags
010	CRRL	CRAM	CMove	CClearTag
011	JALR.CAP	Clear†	CClear	CGetAddr†
100	FPClear	CSealEntry	CLoadTags	CGetVersion‡
101	JALR.PCC	CLoadVersions‡	CLoadVersion‡	CGetHigh
110	CGetTop	-	-	-
111	-	-	-	Dest-Only

†Previously used by a removed instruction.

‡Reserved for future use.

Dest-Only encoding allocation (rs1 field)

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

31 25	24 20	19 15	14 12	11 7	6 0
0x7f	0x1f	func	0x0	rd	0x5b

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

Appendix C

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.

C.1 Additional Architectural Assistance For Revocation

C.1.1 CClearTags

Typically, allocators allow for the return of "uninitialized" memory (e.g., `malloc` vs. `calloc`). In the context of temporal safety, this proves to be problematic unless the allocator is type- and use-specicialized: data and pointers may unintentionally flow into the possession of the holder of a new allocation, violating both confidentiality and, in a non-architectural sense, provenance integrity. Most treatments of security-conscious allocators therefore always zero memory. CHERI can directly probe at the difference between confidentiality of data and provenance integrity of pointers. Towards this end, we introduce a `CClearTags` instruction, which permits bulk zeroing of capabilities within a cacheline (i.e., at the same granularity as `CLoadTags`).

While `CClearTags` may be a useful alternative to `bzero`-ing memory when confidentiality is not required, it has at least one additional use in the context of revocation specifically. `CClearTags` should accelerate sweeping by allowing allocators to optimistically, even if not perfectly, remove capabilities within freed regions, removing comparatively expensive look-aside checks of validity during sweeps, at a lower cost than zeroing.

`CClearTags` is intended to be *non-allocating* in the caches, as with its counterpart `CLoadTags`. If a cacheline is not present in the cache fabric, the induced store should always transit to the tag cache rather than pulling the line data into the caches.

C.1.2 Non-Temporal (Streaming) CLC

During revocation, whenever a capability is identified via `CLoadTags`, it is fetched from memory for analysis. A large fraction of these capabilities are expected to be valid, and so will not cause additional activity within their cache line.[1] Being able to hint to the cache that the revoker is *streaming* through memory, and so should not pave over the caches, seems like a worthwhile objective. We therefore introduce a non-temporal, streaming `CLC` analogue, `CLCNT` with architectural semantics exactly matching those of `CLC`, but as a separate opcode to hint to the microarchitecture.

While one implementation of such a streaming `CLC` would be to never promote lines in the cache hierarchy (i.e., leave them in place, in DRAM or the LLC when loading), analysis done as part of the Efficient Tagged Memory [70, §VI.B] suggests that if a cacheline has one capability, it is relatively likely to have another.[2] This suggests that `CLCNT` should be caching, but restrictedly so. Naïvely, we suggest that ensuring misses usually allocate below some top k lines of the MRU queue for the cache "way" activated will result in it being evicted relatively soon and without introducing too much contention with the application. When a `CLCNT` hits in the cache, it should not trigger promotion to the front of the MRU queue. Whether this policy is a good one, and, if so, what the correct values of k and "usually" are, remain open questions.

C.2 Recursive Mutable Load Permission

Several software capability systems have exploited the use of immutable data structured to facilitate safe sharing (e.g., Joe-E [99]). CHERI capabilities can provide references through 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

---

[1] Those capabilities that are found to be revoked are then subjected to a `CLR.C`/`CSC.C` sequence for atomic replacement with their revoked image.

[2] Specifically, the ratio of the probability of a capability being found in memory with a "grouping factor 8" (corresponding to our FPGA's caches for CHERI Concentrate capabilities) to an independent sampling of eight "grouping factor 1" binomials (analysing on a word-by-word basis) is between 0.97 and 0.32, implying clustering of capabilities.

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.[3] 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 [13].[4]

Interaction With Sealed Capabilities A question arises about loads of *sealed* capabilities bearing (for example) PERMIT_STORE through a capability lacking PERMIT_RECURSIVE_MUTABLE_LOAD: in some sense, this sealed capability is authorizing mutation; are we to clear its PERMIT_STORE, despite the seal? We view the immutability of sealed capabilities as taking precence, and so preserve PERMIT_STORE under seal even in this scenario. Beyond aesthetics, we conjecture that this interpretation is convenient for software: PERMIT_RECURSIVE_MUTABLE_LOAD can be cleared to create read-only collections of sealed handles to software objects.[5] Nevertheless, software must be aware that mutation authority under seal is not stripped by PERMIT_RECURSIVE_MUTABLE_LOAD.

C.3 Hierarchical Revocation From Ephemeral Capabilities

The "revocation" work atop CHERI to date has been about revoking all (non-TCB) access to resources (usually, virtual address space). However, "Capability Revocation" more typically means the ability for any agent, which has delegated access to some resource, to revoke some (or all) of its delegations while retaining access and the ability to further delegate access

---

[3] The concept of such *transitively* read-only capabilities appears to have been first developed in KeyKOS, where such capabilities were termed 'sensory keys' [63]. 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 [138]. In the successor system Coyotos, 'weak' was once again made to imply read-only access [46, 139].

[4] Readers may be familiar with the infamous proof of Boebert [18] that "an unmodified capability machine" is unable to enforce this property. As CHERI distinguishes between capabilities and data, the proof is not directly applicable [100], and, indeed, one could imagine using trusted intermediate software to emulate the effects of PERMIT_RECURSIVE_MUTABLE_LOAD, as proposed by Miller [102]. Despite that, PERMIT_RECURSIVE_MUTABLE_LOAD is still of practical utility, as it is a light-weight, architecturally enforceable mechanism that avoids indirection.

[5] By way of example, software can create an immutable collection c of *arbitrary* unsealed capabilities by sealing each capability logically held in c to a type with an *ambiently available* unsealing right and referencing c itself through PERMIT_RECURSIVE_MUTABLE_LOAD-clear capabilities. A layer of indirection suffices to permit arbitrary *sealed* capabilities within c as well. Software can restrict this model by using compartmentalized unsealers rather than ambient authority.

[123]. Efforts to develop such capacity atop CHERI would likely rely upon sealed capabilities or domain transitions (with or without exceptions) to mediate delegation, as these are the most apparent mechanisms available to us to prevent unchecked duplication of usable authority. However, these come with somewhat large costs: domain transitions impose cycle overheads, and either strategy would seem to require that the set of *operations* on the delegated resource be fixed by its constructor. For example, the original source of a revokably delegated data resource would likely have to provide specialized `memcpy` implementations for moving data out of or in to the resource's memory. In this section, we explore the implementation of directly-accessible revokable delegation assuming the existence of *ephemeral* capabilities, which cannot be stored to memory once loaded into a register file.

C.3.1 Ephemeral Capabilities

The basic primitive of an ephemeral capability is a degenerate generalization of the GLOBAL / STORE_LOCAL_CAPABILITY mechanism of CHERI (recall Section 3.4.2) and the "capability coloring" proposal of Appendix C.10. Ephemeral capabilities are constructed via a new *store* instruction, `CStoreEphemeral` which, given a non-ephemeral capability in a register, places an ephemeral version in memory, which may thereafter be loaded with `CLC`.[6] If an attempt is made to store, via `CSC`, an ephemeral capability to memory, an un-tagged version is stored instead. Thus, once loaded, an ephemeral capability is confined to the register file and even a context switch will destroy it. Similarly, a (transitive) callee's attempt to spill such a capability to the stack will instead detag it, and so these ephemeral capabilities must be considered lost across general procedure calls. (While ephemeral capabilities can still be passed in registers as arguments, in general, we suggest passing a non-ephemeral capability to the ephemeral one instead.)

Software consuming these ephemeral capabilities must be prepared to deal with revocation and the *appearance* of revocation, by attempting to reload a tagged ephemeral capability from memory. Software must be careful to preserve access to the memory locus of the ephemeral capability loaded to the register file. Operations done against such revokably delegated resources should be idempotent (which may just mean that they are precisely resumable).[7]

C.3.2 Revocation

Armed with such a primitive capability form, resource revocation still needs to remove (a hierarchy of) ephemeral capabilities from memory, but the issuing authority has ensured that access to the delegated resource has not spread unchecked in memory. Having removed all the targeted ephemeral capabilities from memory, a single context switch on all cores suffices to ensure that

---

[6] We propose the use of a reserved **otype** value, rather than a new permission bit, to designate capabilities of this form (but without interpreting this value as *sealed*). As we intend these non-storable capabilities to be ephemeral, there will not be enough of them present in the system at any moment to justify the use of a permission bit. Moreover, while the use of an **otype** means that we cannot seal these non-storable capabilities, even in registers, this does not seem to be a loss.

[7] It is possible to imagine architectural assistance beyond trapping on untagged capabilities, should the "trap-and-reload" approach sketched here be unduely onerous.

there is no retained access. These context switches may be actively driven (with IPIs) or passively observed (as with epoch-revocation schemes). We propose that a *compartment* within the TCB oversee the construction and (hierarchical) delegation of revokable delegate intermediaries.

In order to efficiently revoke a subtree of the delegation relationship, we will need to construct that relationship explicitly in memory in a way that its subtrees are easily enumerated. The design we propose herein makes heavy use of sealed capabilities to small regions of memory, directly storing the delegation relationship metadata with the delegated, ephemeral capabilities themselves. These small regions of memory are used once and so may be reclaimed by the existing *global* revocation mechanisms after their purpose has been served.[8] Such a *delegation box* contains:

- An ephemeral capability to the delegated resource
- A capability to the progenitor delegation box, if any.
- A pair of capabilities forming a doubly-linked list of this box's delegation *siblings*.
- A capability to one of its child delegation boxes, if any.

Straightforward rose tree [140] operations suffice to maintain the hierarchical delegation structure using these boxes, and sealing ensures that we can safely give out the rights to manipulate a box (constructing a new child or revoking the subtree of which it is the root). A separate grant of access directly to the ephemeral capability allows the direct use of the delegated resource until the delegation box's revocation.

C.4 Compressed Permission Representations

The model of Section 3.4.2 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).

---

[8] Users of this mechanism must therefore be prepared for faults while using the ephemeral capability as well as when attempting to reload it. Fortunately, this seems straightforward.

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.[9] 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 INVOKE is *checked* only as part of `CInvoke`'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 `CInvoke` (just with `CUnseal`). The utility of such regions is perhaps not readily apparent, but any shift to make INVOKE apply only to object capabilities would require modification of the `CSeal` instruction 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 EXECUTE implied the absence of STORE, STORE_CAPABILITY, and STORE_LOCAL_CAPABILITY (see Appendix C.4.3). Further, granting LOAD_CAPABILITY effectively implies granting LOAD: `CLC` and `CLR.C` would trap without the latter, but more substantially, a capability load of an un-tagged (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, STORE_LOCAL_CAPABILITY implies STORE_CAPABILITY, which, in turn, implies STORE. Taking all of these implications into consideration, one finds that there are 15 consistent states of the six virtual-address-space rights (EXECUTE, LOAD, LOAD_CAPABILITY, STORE, STORE_CAPABILITY, STORE_LOCAL_CAPABILITY) considered, enabling a four-bit compressed representation.[10]

Consider the powerful PERMIT_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 *inter-*

---

[9] 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.

[10] If one restricts consideration to just the five bits of LOAD, LOAD_CAPABILITY, STORE, STORE_CAPABILITY, and STORE_LOCAL_CAPABILITY, one finds 12 valid states, requiring four bits. A straightforward reduced encoding then leaves LOAD and LOAD_CAPABILITY unaltered but can use two bits to indicate which of $\emptyset$, {STORE}, {STORE_CAPABILITY, STORE}, or {STORE_LOCAL_CAPABILITY, STORE_CAPABILITY, STORE} is present.

preting (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.

C.4.1 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 **μperms** 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 C.1.

In all capabilities, we reserve three bits for uninterpreted user permissions, and four bits for the flow control detailed in Section C.10.[11] 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: EXECUTE (Ex), LOAD (L), STORE (St), LOAD_CAPABILITY (LC), STORE_CAPABILITY (SC), INVOKE (I),[12] 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.

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.[13]

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 C.10). 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), PER-

---

[11] Absent the use of this experimental coloring scheme, these reserved bits can instead be used to carry the GLOBAL and STORE_LOCAL_CAPABILITY bits, with two bits remaining reserved.

[12] While any capability type can, in principle, be sealed and could be unsealed at `CInvoke` time, `CInvoke` unseals only two capabilities, installing them as PCC and IDC. As such, it seems sensible to restrict `CInvoke` to operating only on VA capabilities, and so PERMIT_INVOKE is defined only therein.

[13] 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**								
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 C.1: Bit-level representations of a type-segregated metadata-bit-packing scheme.

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 C.2: A variant of packed metadata including multiple sealed forms.

MIT_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.

C.4.2 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.

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 C.2. 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.

C.4.3 `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_CAPABILITY (SLC) bit and its unused slot in the RX form would vanish.

C.5 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 [68] and Arm's MTE [8]. 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 embed a four-bit version field[14] in every memory-authorizing capability, either using reserved bits or by shrinking the address field from 64 to 60 bits.[15] Further, we pair the same number of bits of version with each '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.) 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 passing the existing CHERI permissions and bounds checks as well as any MMU permissions checks) the granule and intra-capability versions are

---

[14] There is nothing special about the value four; even a one-bit versioning scheme has practical utility, while more bits reduce likelihood of collision in stochastic schemes and delay revocation in deterministic schemes (see Appendix C.5.3). Four simply seems to be a popularly acceptable value.

[15] Practically, most modern systems do not make use of their entire 64-bit virtual address space and require that all such addresses be sign-extended values derived from (typically) 40-bit to 57-bit values, depending on the architecture. We can therefore repurpose some of these bits with only modest, localized changes to system software.

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 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; the canonical example is, of course, memory allocators, which will hold unversioned capabilities internally and give out (and take back) versioned capabilities.

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.

C.5.1 Legacy Memory Versioning Behaviors

When adding CHERI to an architecture that already has memory versioning support (e.g., SPARC or Arm), it may be desirable to retain compatibility with existing mechanism in hybrid or legacy code. That is, we may wish, assuming the system has enabled memory versioning and has provided a non-NULL DDC, for legacy (i.e., integer pointer using) load and store instructions to continue to specify the intended memory version and legacy version manipulation instructions to continue to function. (Recall that all such legacy instructions have their integer addresses interposed by DDC.)

We therefore propose that the interposed integer offsets arising from legacy instructions be interpreted subject to existing architectural address handling rules. Arm’s MTE, for example, requires the use of Top Byte Ignore (TBI), which partitions the 64-bit address into an 8-bit metadata field and a 56-bit address; we propose that Arm processors with CHERI and MTE continue to claim the top 8 bits of any integer offset within a capability as a metadata field.[16] When combined with an *unversioned* capability, the integer offset specifies the memory version used for a memory transaction; a *versioned* capability instead overrides the requested version from the offset.[17] This policy may also be applicable to *capability-authorized* instructions with integer offset register operands, which may simplify capability-aware supervisory software that must operate on versioned integer addresses. (It seems unlikely that there is utility to permitting offset *immediate* operands to influence memory version fields.)

---

[16] Because integer offsets often come about through arithmetic, which may not be aware of the 8+56 partition in the semantics of the bits being manipulated, it may be useful to slightly tweak the encoding of versions. Instead of directly taking the top 8 bits as the source of the version value, it may be useful to XOR them with the top bit of the remaining 56-bit offset. Thus, the 64-bit 2’s-complement values of 1 and -1 would be interpreted as 56-bit 1 and -1, respectively, but both with a version field of zero.

[17] As memory transactions are already opportunities for traps in most architectures, it may be worth trapping if the integer offset calls for a non-zero version field in combination with a versioned capability. On the other hand, it is likely acceptable from a security policy perspective if the discrepancy is ignored.

C.5.2 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.[18]

- `CLoadVersions` loads version fields for an entire cache-line of memory granules into an integer register, akin to `CLoadTags`. It is intended as an optimization for system software paging virtual memory.

Atomics

In addition to the above, we desire a means for *atomic* update of the version of a memory granule (as well as up to a capability-sized word within it). Unfortunately, our desires brush up against (micro)architectural limits. A version-manipulating, capability-sized (and -aligned) store-conditional instruction, for example, should take four operands: ① an unversioned capability authorizing access to the target, ② the data/capability to store to memory, ③ the desired new version, and ④ the destination register indicating success or failure of the store. However, it is challenging to fit so many register indices into a single instruction and this may also exceed the port availability of the processor's register file. (A general compare-and-swap instruction is even worse, adding both the expected value of memory and the expected memory version.) With these constraints in mind, we propose two possibly feasible subsets:

- `CSCAndUnversion` takes an unversioned memory capability authorizing the store, a capability register to store, and the output register. (It is, therefore, rather like an ordinary `CSC`.) It fixes the desired new version to the unversioned value. Thus, on successful store, the memory version granule is inaccessible to any versioned capability, and the same authority used with this instruction can be used with `CStoreVersion` to subsequently update the target granule's version.

---

[18] It may be sensible to always clear the tag or always trap, as well. We do not have a use case for the tagged result when-equal case.

- `CSCWithVersion` is similar, but reads the desired version *from the output register* before storing back the success indication. This works around the encoding space problem, but may still require an excess of access to the register file.

C.5.3 Use With System Software

We envision that software will make use of memory versions *monotonically*. That is, versions of memory granules will be altered to revoke *all* access by any existing versioned capability inclusive of that granule rather than to *restore* access at some earlier version. Thus, we believe that `CSCAndUnversion` is sufficiently atomic for software's needs. Despite the observable transition of the granule to an unversioned state before any subsequent transition to a version not yet held anywhere in the system, the net authority in the system remains the same.

Because there are only finitely many versions available, we further envision that the *system software* will provide a *revocation* mechanism (in the style of Cornucopia [51]) to de-tag or otherwise remove authority from 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.

C.5.4 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.

C.6 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.

C.6.1 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 `AUIPCC`, 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.

C.6.2 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.

C.6.3 Related Work in Linear Capabilities

Skorstengaard et al. have concurrently developed ideas about linear capabilities [141], 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 [25] also propose that *linear pointers* could contribute to reasoning about concurrent memory use.

C.7 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.

C.7.1 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.

C.7.2 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.

C.8 Indirect Sentry Capabilities

While sentry capabilities facilitate the construction of capabilities that grant the right to run code from a fixed entry point, if that code is intended to run in a particular (register) context, software must use trampolines (e.g., the PLT stubs) to ensure that this context is constructed correctly. These trampolines must intermingle data and code, as the trampoline has amplified access, relative to its caller, only to the region of its instruction pointer. The trampolines must, as well, be *per-context* (e.g., library instance), which necessitates duplication of the trampoline code sequence for each context.

C.8.1 Points-to-PCC

Herein, we propose yet another architecturally-understood form of sealed capability, the *indirect* sentry capability, which is a curious hybrid of a sentry capability (of Section 3.9) and a special case of an indirect capability (recall Appendix C.7). Where sentry capabilities point directly at the code to be run (and expose the entire region bounded by PCC to the callee), these indirect capabilities point at a capability to be installed into PCC (which, in turn, points to the code to be run). Upon invoking such a capability, it is unsealed and installed into the IDC (capability) register and the pointed-to capability is installed into PCC; thereby, the callee is granted access to both regions of memory.[19] The unsealing and IDC register writeback is not separable from the load from memory and change of PCC: either both registers are updated and the instruction completes, or neither are updated and the instruction traps. We propose a `CInvoke`-like, single-operand instruction for such invocations, `CInvokeInd`; we intend this to be a separate instruction from `CJR` so that there is no need for a conditional load in the microarchitecture. We do not envision a version of `CInvokeInd` that writes a link address, but see below

---

[19] If this pointed-to capability is, itself, a sentry capability, it should be unsealed as part of the load into PCC. We do not currently believe that *requiring* this capability to be a sentry capability has any meaningful impact on the security properties of the system, and so we do not.

for discussion of making function calls and returns with `CInvokeInd`.

Any capability authorizing capability load may be made into a sealed indirect entry capability, for which we propose reserving the **otype** $2^{64} - 4$.[20] A new, two-operand instruction is required for this action, which we call `CSealIndEntry`.

It is straightforward to adapt the designs of Section 3.9 to this instruction so that, for example, the PLT stub *code* can be relocated to the common, read-only section, leaving a kind of data-only trampoline which contains capabilities to the (also shared) code to be run and the per-instance RW data. Each entrypoint requires one capability, rather than a full PLT stub. This enables unifying the per-instance PLT stub and per-instance data regions into a single per-instance region which continues to not need execute permission.

Additionally, this mechanism could be suitable for decreasing the information exposure between caller and callee functions. If, rather than exposing a return (sentry) capability to the callee, the caller were to spill its return capability to the stack and expose a sealed indirect entry capability derived from the stack, the callee can have its access to the caller's stack completely removed. Upon return, the caller's original stack capability would be available in IDC. Spilling the return address will involve storing a capability derived from PCC but pointing past the `CInvokeInd` instruction. All told, we expect this kind of function call to require ten instructions on call rather than the one `CJALR`:

- three (`AUIPCC`, `CIncOffsetImm`, `CSC`) to compute and spill the return address,
- two to move the stack pointer (`CRepresentableAlignmentMask`, `CAndAddr`),
- four to bound the stack pointer (`CGetOffset`, `CSetOffset` (to zero), `CSetBounds`, `CSetOffset` (back)),
- one to seal indirect sentry capability into the link register (`CSealIndEntry`), and
- one to transfer control (`CJR` or `CInvokeInd`).

There is likely opportunity for additional, specialized instructions here; some plausible examples include:

- An instruction which set the *limit* (i.e., $\mathbf{base} + \mathbf{length}$) of a capability to the cursor and left the base alone could replace the four instruction sequence bounding the stack pointer.
- `CRepresentableAlignmentMask` and `CAndAddr` could be fused into a dedicated instruction for aligning the capability's offset appropriately.

C.8.2 Points to Pair

Another option, for architectures open to multi-word transactions in their memory subsystems, is an indirect sentry capability which points to the pair of PCC and IDC. Invocation of such a sentry performs two capability loads through an ephemeral (architecturally invisible), unsealed

---

[20] While we do not anticipate comingling code and data within the authorized region, we do not see much benefit in enforcing a lack of PERMIT_EXECUTE on the original capability nor in shedding it as part of sealing.

copy of the given sentry and then, with both capabilities in hand, installs both into the register file atomically. There is no requirement that the two capabilities pointed at be sealed. Because these capabilities reside in memory, the instruction constructing these "points to pair" indirect sentries likely cannot perform any validation on their contents.

C.9 Anti-tamper Seals

When implementing allocators such as the C language's malloc and free, it is common to require that the caller only pass values to free that were previously returned to malloc (according to ISO C, doing otherwise is undefined behavior.) As is typical of C, run-time programmers exploit this and do not perform checks that the passed pointer is in fact an allocated pointer and the implementation may not retain sufficent information to confirm this. We could greatly reduce the number of check in the free path if we could be certain that the passed capability was exactly the one we handed out.

To address this need, we propose a new variant of sealing: anti-tamper seals. A portion of the otype space would be reserved for anti-tamper seals and capabilities sealed with an anti-tamper otype would have the following properties:

- The capability can be dereferenced or jumped to as though it were unsealed.
- Address modifying instructions (e.g. CSetOffset) work as though the capability where unsealed.
- CAndPerm, CSetBounds, and CSetBoundsExact unseal the capability (setting its otype to -1).

The justification for allowing address adjustments is similar to that for allowing capabilities to stray out of bounds. We want to allow for the case that a programmer alters the address of a capability before restoring its address using some seperate state (e.g. buffer length) and freeing it. It's unclear how common such code is, but intuitively such patterns will be difficult to detect statically.

C.10 Compact Capability Coloring

As noted above, the GLOBAL permission described in the model of Section 3.4.2 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 [120]. 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’.[21]

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).[22]

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

- 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.

---

[21] 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 C.2.

[22] 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.

- 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.

C.11 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.

C.11.1 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.[23] 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.

C.11.2 Shared VTables with Sentry Capabilities and Type Tokens

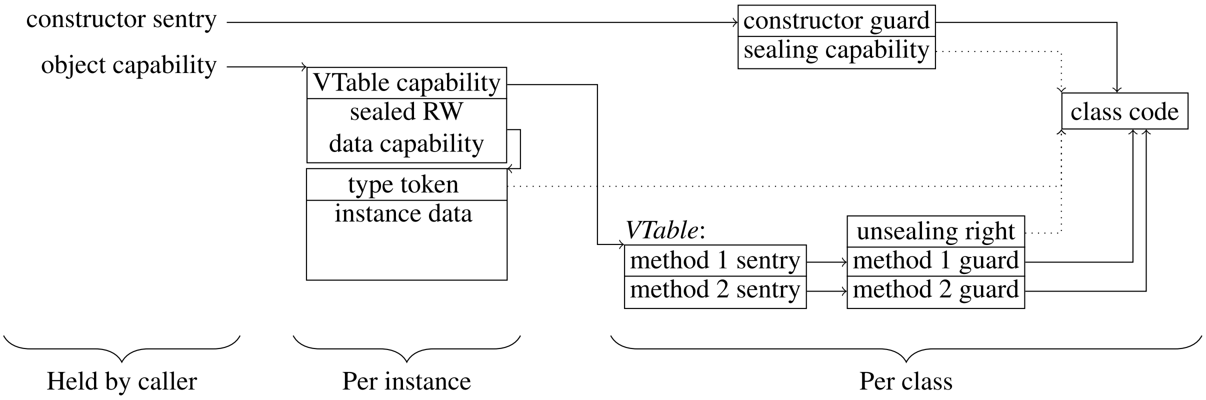

Figure C.3: Schematic representation of a shared VTable design for a base class. The user directly holds a sentry capability to the object constructor guard, which uses the adjacent Permit_Create_Type_Token-bearing capability to stamp object instances. Each object instance is held by the user through 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 sentry 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.

Sentry capabilities (recall Section 3.9) give software the ability to ensure that control flow can enter a given region at a particular address: the bearer of a sentry capability can jump to it but cannot adjust its offset. However, unlike the existing `CInvoke` mechanism, sentry capabilities when invoked transition only the PCC register. To transition other registers as a

---

[23] 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 could gain the ability to reconstruct tags given the sealing authority.

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 sentry 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 C.3 we show a schematic representation of using in-memory type tokens to guard method invocation of a multiply instantiated (C++) object.

Combined with sentry 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 a sentery capability to a region containing a capability bearing PERMIT_SEAL. The non-constructor capabilities in the VTable are sentry 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.

C.11.3 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[24] capability whose cursor will form the type tag.[25] 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 expect 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

---

[24] 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.

[25] 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.

cursor of the authorizing capability matches the virtual address stored in the type token at the base of the sealed object,[26] then `CUnsealTyT` produces an unsealed version of the sealed capability. Microarchitecturally, `CUnsealTyT` 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.

C.11.4 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.

C.11.5 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 C.11.2 above, having one sentry capability per procedure and gating permission by possession of the procedure's guard's sentry 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).

C.11.6 Token-mediated CInvoke

`CInvoke` poses something of a challenge for in-memory type tags: a single instruction must,

---

[26] This load is why `CSealTyT` required PERMIT_LOAD of its to-be-sealed capability.

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 `CInvoke` 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, `CInvoke` 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.

C.11.7 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.[27] Such **otype** bits would also let software create sealed objects other than enter capabilities without memory footprint.

C.12 Windowed 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 not impacted. We introduce new, purpose-made instructions for loading and storing these short representations of capabilities; stores especially may fail if translation is not possible.

---

[27] 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.

C.12.1 Restricting Capabilities to 32-bit Windows

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[28] 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).[29] 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.[30]

C.12.2 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_SET_CID 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.[31] All told, these implications eliminate five existing permission bits from short capabilities' representations.

---

[28] It would be sufficient to store a de-tagged word, but trapping is more likely programmer friendly. While this is a data-dependent action, as it requires a comparison between the (untranslated, virtual) target address and the capability from the register file, this is not the only data dependence in the short capability store instruction.

[29] 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.

[30] 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.

[31] 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.

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.

C.12.3 Tag Bits and Representation for Shared Memory

Short "capabilities" could plausably be left untagged in the architecture and used only as forgeable fat pointers which are lifted into the capability space on coversion. If we were to tag short capabilities, we require more bits for distinguishing mixed capability widths from data. In a 128-bit-sized and -aligned region of memory, there are five possible options, assuming that 128-bit capabilities must remain 128-bit-aligned: 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-byte cache lines, requires 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 C.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	0	1	Forbidden
1	1	1	X	A 128-bit capability

Table C.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.

With Relaxed Alignment Requirements

It may be more natural to permit *all* capabilities, both 64-bit and 128-bit, to be stored at 64-bit alignment. In such a case, within a 128-bit-sized and -aligned region, there are now these 10 possibilities: ① One 128-bit capability, spanning the whole region. ② The tail of a 128-bit capability, followed by the head of a 128-bit capability. ③ The tail of a 128-bit capability, followed by a 64-bit capability. ④ The tail of a 128-bit capability, followed by data. ⑤ The head of a 128-bit capability, preceeded by a 64-bit capability. ⑥ The head of a 128-bit capability, preceeded by data. ⑦ Two 64-bit capabilities. ⑧ One 64-bit capability, followed by data. ⑨ One 64-bit capability, preceded by data. ⑩ Only data.

C.12.4 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 capabilities 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.

C.13 Capabilities For Physical Addresses

C.13.1 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* (PTEs). 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 time. As a result, software must ensure that capabilities are not transmissible improperly[32] 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. Note that these extensions may be added to a system which supports CHERI for virtual address pointers with no impact to most user mode software. These extensions most affect the interfaces between firmware, hypervisors, and operating system kernels.

C.13.2 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 [148, §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

---

[32] 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.

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 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.[33] 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.

C.13.3 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

---

[33] This obviates the RISC-V `mstatus` MPRV mechanism for toggling address translation.

(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.

C.13.4 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 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 Section 2.3.5 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.

C.13.5 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.

C.13.6 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.

C.14 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.

C.14.1 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.

C.14.2 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.

C.14.3 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)
```

C.14.4 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.

C.14.5 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.

C.14.6 Implementation outline

For a minimalist implementation, the following might be done:

1. Choose a representation for the bits that indicate a color within a capability
2. Implement a CSR that configures the current color of a processor core. The permissions on this register are up for debate but access control might be similar to that of the IOMMU page table base register - potentially set externally rather than internal to a PE
3. In user mode, loading or dereferencing capabilities checks the color matches the current one, and causes an exception if mismatched
4. A privileged instruction takes two capabilities, an input capability and an authorization capability. The output of the instruction is the input capability with the color changed to match the authorization capability.
5. Checks may be necessary to verify the input capability is a subset of the authorization capability, bearing in mind they may be in different address spaces without a contiguous mapping.
6. It may be necessary for a PE that wishes to change the color of its capability to call out to a more trustworthy component, such as a bridge or another processor with more authority. This could be modeled by the PE not possessing a suitable authorization capability, and thus making an API call to another PE or ATPU. ATPUs may implement such translation in hardware to make it efficient.

C.15 Compartment ID Sealing

C.15.1 Motivation

Compartment identification is essential. Code and data associated with a compartment must be somehow identified and validated when they are being accessed. The most natural way seems to be to give them identification numbers. This has traditionally been done in software, e.g., processes are assigned process IDs (PIDs) in operating systems. In some cases, a software ID can correspond to a hardware ID, e.g., PIDs can correspond to address space identifiers (ASIDs). However, that is not necessarily the case. In this proposal, we suggest an architectural ID for compartments. We call these numbers compartment IDs (CIDs). That means that one CID is mapped to one compartment at a time. An architecturally defined CID needs to be implemented by hardware.

C.15.2 Storage

The first question to tackle is where to store a CID. We envision multiple options for that. First, one can use a dedicated register for compartment identification as done by the Morello architecture. Second, one can import CIDs into the capability format, which is what we propose in this document. Having this solution leads to an atomic change of CID and code capability – a property we envision to be useful for secure compartmentalisation. Future research has to show if one approach of storing CIDs is preferable over the other.

The CID will substitute the otype bits (18 bits) in the 128-bit capability format. The otype bits are currently not well-used. The otype field indicates whether a capability is sealed and if so which “type” it has. There exists currently two fixed values that are used: one for unsealed capabilities and one for sealed entry (sentry) capabilities. There are 14 more reserved fixed values, which are not currently used. All other values are used as values for sealed capabilities. However, there is also the possibility to combine otypes and CIDs. One potential approach is to subdivide the 18 otype bits such that there is space for a few otypes and the remainder of the bits is dedicated to CIDs. This would allow sentries and CID sealing to be combined.

In the following text, we propose to completely substitute otypes. All values but the zero CID (CID==0) are valid IDs. Furthermore, every CID sealed code capability already is a sealed entry capability. Please note that every capability within a compartment can be manipulated by the compartment itself. This also includes CID sealed code capabilities. In the CHERI world up to now, sentries cannot be manipulated.

An extension would be to add an additional bit indicating whether a capability is a sentry. This will make capabilities immutable even within a compartment. The only way to unseal a sentry is to jump to it.

At the moment, this CID sealing proposal is limited to the 128-bit capability format. However, the underlying mechanism to add CID bits to the meta information bits of the capability works for every format. These CIDs are referred to as the Architectural CIDs (ACIDs). In the 128-bit capability format, we propose to allocate 10 bits to the compartment ID, but we envision any number of bits fitting in the meta information bits to be a valid implementation of CID Sealing. This length is also known as the ACID_LENGTH. In this proposal, we allow for 1024 CIDs to be encoded, and keep the remaining 8 bits reserved. Software may choose to virtualise these and can create Software CIDs (SCIDs), which are a concept similar to PIDs. A SCID may consist of the corresponding ACID added to other bits in order to create a virtual identifier or can be fully independent of the ACID it is mapped to.

Compartment IDs come with two new instructions. One instruction for reading the CID into a general purpose register and one instruction for setting the CID of a capability. Like capabilities themselves, CIDs are not considered secret. Therefore, the CID reading instruction is not privileged, but the writing instruction is restricted by the PERMIT_SET_CID permission. The PERMIT_SET_CID bit is a hardware permission bit that, if set in a PCC, allows manipulating the ACID. Like all hardware permission bits in CHERI, PERMIT_SET_CID is constrained by monotonicity. One security policy is that a code capability with this bit set should only be available to supervisor code. We suggest encodings for the instructions in order to demonstrate the adaptability of CID to the currently existing CHERI-RISC-V ISA.

- `CSetCID cd, rs0`:
  set the CID in cd to the value in rs0. This instruction needs the PERMIT_SET_CID bit set, otherwise it throws an exception. This instruction uses the ACID_LENGTH lower bits of rs0 and ignores the upper bits.

  Possible encoding (31:0): 0x7f, 0x19, rs0, 0x0, cd, 0x5b (assign a random free funct5)

- `CGetCID rd, cs1`:
  extract the CID of cs1 and store it in rd

  Possible encoding (31:0): 0x7f, 0x1, cs1, 0x0, rd, 0x5b (this is the same encoding as CGetType, which could be obsolete with this proposal)

An alternative way of enabling sealing not just constrained by the PERMIT_SET_CID bit is to use non-memory capabilities. Comparable to the mechanism already present for otypes in the CHERI-RISC-V ISA, we envision a more fine-grained mechanism. We propose to create capabilities that authorise for another capability to be sealed with a CID being in a certain range of CIDs. This authorising capability has the same fields as a conventional CHERI memory capability, but uses its fields differently. The address field is interpreted as the CID, and the bounds define a range of CIDs. This allows for code to be granted a range of CIDs it can use to seal other capabilities with. One possible extension to this is an instruction that retrieves the next CID from the authorising capability and atomically increments the CID field.

C.15.3 Sealing

The current CID (the CID of the current compartment) is determined by the CID in the PCC, which is also a capability. If the PCC changes and the new PCC has a different CID, this constitutes a compartment change (see Section C.15.4).

With an ACID being in all capabilities, we can establish a concept referred to as CID Sealing. All capabilities are implicitly sealed by their CID. Capabilities are considered unsealed if and only if: their CID matches `PCC.CID` or their CID is 0. All other capabilities are implicitly sealed. An implicitly sealed capability can be inspected (its fields can be read), but it cannot be manipulated nor can it be used to reference memory.

CID sealing allows a register file to hold capabilities from different compartments without allowing capability leaks. The currently executing compartment can only make use of its own capabilities or explicitly unsealed ones.

In this proposal, CID sealing completely replaces the sealing mechanism currently present in CHERI. A CID sealed capability can only be manipulated by the compartment it is owned by. Therefore, it can be securely handed out to other compartments, e.g., as a code pointer back.

However, CID sealing can also co-exist with current sealing mechanisms in place, e.g., sentries, as discussed in Section C.15.2. In comparison to conventional CHERI-RISC-V sealing, CID sealing cannot produce code pointers that are immutable within a compartment, but only across compartments.

C.15.4 Compartment Change

A compartment change is done when the CID is changed. The CID can be changed by installing a code capability with a different CID into the PCC register. Installing a new PCC can be facilitated in many ways, e.g., by jumping to a capability using the CJALR instruction. We envision there to exist multiple ways of changing the PCC and the CID sealing mechanism is independent of the concrete way. Once the new PCC is installed, the new compartment needs to bootstrap. We envision two ways for that, which only differ in the way they retrieve their initial data capability.

First, the new PCC can have further capabilities in its global space, which can be loaded into the register file. This can be facilitated by the AUIPCC instruction. The following instruction sequence loads a capability at an example fixed offset:

```
auipcc ct0, 2
clc cs0, 0x100(ct0)
```

Second, a new data capability can be installed alongside the new PCC. This gives the compartment a capability ready to use in the register file. One option for this kind of compartment transition would be indirect sealed capability pairs (see Section C.8). The following code example shows the brevity of this approach. The instruction sequence was shrunk to just one instruction reading from the new data capability(ct6).

```
clc cs0, 0(ct6)
```

C.15.5 Sharing

Sealing implicitly forbids sharing capabilities between compartments. However, sometimes this behaviour is desired by software. We have designed two mechanisms for sharing capabilities between compartments:

- Explicit unsealing: This will explicitly unseal a capability, and it can be shared either via the register file or via memory. Another compartment can pick up this capability and seal it again and use it. This does not protect against transitive capability leaks - an unsealed capability can be passed on to a third compartment and thus be leaked.
- CID spaces: CIDs can be separated into CID spaces (see Section C.15.7). One approach is to allow unsealing within a CID space. This allows all compartments in that space to use that capability, but no other compartment can unseal the capability. This protects from transitive capability leaks outside of the CID space. As an addition, we envision a bit that – if set – allows sharing within the CID space.

Furthermore, we also envision the possibility of *resealing* capabilities. A capability belonging to one compartment can be resealed to another compartment. This effectively means that code can seal one or more of its own capabilities for another compartment. The main advantages are that this avoids capabilities to be unsealed in the open as done with explicit unsealing as well as potential performance improvements avoiding the need to unseal and seal capabilities. However, resealing brings the disadvantage for the receiving compartment of not knowing whether a capability was sealed by itself or another compartment. Therefore, we would expect the need for additional validation checks.

C.15.6 Explicit Unsealing

We need to reserve a CID value that represents an unsealed capability instead of a valid compartment ID. We can choose any value to represent an unsealed capability. In this proposal, a capability is unsealed if and only if its ACID is set to 0. This ACID is also known as the zero

ACID. A capability with the zero ACID can be CID sealed by any compartment. This introduces two new instructions (in comparison to the three-operand operations currently present in the CHERI-RISC-V, our proposed operations have two operands. The third implicit operand is PCC):

- `CCIDUnseal cd, cs1`
  If cs1.CID and PCC.CID match, then cs1 will be assigned to cd with cd.CID=0. Otherwise, clear the tag of cs1 and assign it to cd.

  Possible encoding(31:0): 0x7f, 0x1a, rs0, 0x0, cd, 0x5b (assign a random free funct5)

- `CCIDSeal cd, cs1`
  If cs1.CID==0, then cs1 will be assigned cd with cd.CID=PCC.CID. Otherwise, clear the tag of cs1 and assign it to cd.

  Possible encoding(31:0): 0x7f, 0x1b, rs0, 0x0, cd, 0x5b (assign a random free funct5)

A capability with CID==0 can be used by any compartment. An unsealed capability can be misused and the system can become victim to transitive capability leaks, e.g., by code that carelessly stores capabilities in shared memory.

Having the zero CID represent an unsealed capability seems intuitive and allows code unaware of CID sealing to operate correctly. In contrast, with otypes, the unsealed capability is represented by -1, which expands to all 1s in the otype bits.

C.15.7 CID Spaces

We envision that CIDs can build subspaces in order to model relationships between compartments. One possibility is to put compartments with identical upper bits into the same group, e.g., $CID - Space$ = 0b10101010xx. This would lead to a CID space of size 4.

CID spaces can express trust relationships between compartments, e.g., capabilities within a CID space are implicitly unsealed in any compartment of that CID space.

A further refinement to this is to make the mask configurable with which the CID spaces are defined. This would use another field of size log_2(ACID_LENGTH) in order to specify how many of the most significant ACID bits are the mask.

Another improvement might be to add a bit that determines whether a capability is allowed to be used within CID spaces. This would add fine-grained control over which capabilities can be shared within CID spaces and which are private to the owning compartment.

Alternatively to capabilities being implicitly unsealed by the capability in the PCC register, we also envision a system with an implicit unseal register. On every capability access, this register is checked in parallel. The address and bounds information span a range of CIDs which the currently executing compartment can unseal.

C.15.8 Code and Data Compartments

A common use case for compartments is to have one code base that operates on multiple data sets as found often in web browsers. We envision this to be modelled by CID sealing and present our preliminary model for code and data compartments in the following paragraph:

The capabilities for different data sets are separated into compartments. There will be one code compartment for each data, but each code capability maps to the same code. This means that all code capabilities are identical in all fields, except for the CID field. Depending on whether the different compartments trust each other, the compartments can be placed into the same CID space.

It is also possible to use conventional CHERI compartmentalisation where each data compartment has the same CID, but the data capabilities are non-overlapping sets between the compartments. In this case, the supervisor code needs to be careful that two data capabilities from different compartments are never accessible to one capability at the same time.

C.15.9 Revocation

When employing more compartments than storable in the ACID space, software will need to virtualise CIDs. This will lead to one or more ACIDs needing to be reused. In order to maintain safety and security, every capability from the old compartment needs to be made inaccessible. This prevents leaking capabilities from the old to the new compartment where both of them have the same ACID, but different SCIDs.

We have come up with a preliminary revocation mechanism, which we will sketch in the following paragraphs. Please note that this mechanism likely incurs a substantial performance penalty.

When the supervisor code, e.g., the operating system, has run out of available ACIDs, it needs to revoke a ACID currently in use. We currently propose to pick this ACID by random. However, one could imagine keeping information in the OS that would enable different strategies, e.g., implementing a least recently used (LRU) policy. The OS saves all capabilities of the old compartment in its own space as they are. During this sweep, the OS will make all of these capabilities unusable, e.g., by marking it with an *unusable* bit. If code tries to legitimately use this capability, the OS will need to jump in and assign a new ACID to this SCID and update all of its pointers to make it usable again. For example, a legitimate use case would be waking up the compartment after a longer phase of not invoking it.

C.15.10 Performance Implications

Using CID sealing can lead to performance improvements. When changing compartments, the calling compartment no longer has to invalidate its own capabilities, but can rely on the sealing mechanism to prevent another compartment from using its capabilities. This saves the calling compartment from using multiple instructions to zero out the register file provided it does not contain sensitive information (with CHERI, a compartment can also use cclear instructions, which can zero out a quarter of the register file on CHERI-RISC-V). CID sealing does help with confidentiality because another compartment cannot dereference the CID sealed capabilities left in the register file. However, CID sealed capabilities are still readable bit patterns and therefore can leak secrets in the integer portion, e.g., keys.

One potential improvement is that short compartment calls with only a few instructions do not poison many registers. After returning from a short-instruction callee compartment, many registers will still hold the original value of the calling compartment. This can potentially be used to enhance performance even more because the calling compartment does not have to completely re-instantiate its register state.

Appendix D

CHERI Concentrate Listings

The cheri-cap-lib [33] open-source library has been used in all of our open-source [34, 35, 36, 38] implementations and contains certain notable algorithms which have been highly optimised and verified to a significant degree. These algorithms are well documented by their contents, so several are reproduced in their current form[1] here to serve as reference to anyone implementing CHERI Concentrate compression in any model or microarchitecture.

D.1 GetTop

This *GetTop* function for compressed capabilities is very similar to the *GetBase* function which uses *baseBits* rather than *topBits*. There is a significant divergence from line 11 where we discern between a top of 0 and a top of 2^{64}. It has been non-trivial to develop an algorithm that is both correct and fast enough for all uses in our implementations.

```
 1 function CapAddrPlus1 getTopFat(CapFat cap, TempFields tf);
 2   // First, construct a full length value with the top bits and the
 3   // correction bits above, and shift that value to the appropriate spot.
 4   CapAddrPlus1 addTop = signExtend({pack(tf.topCorrection), cap.bounds.topBits}) <<
         cap.bounds.exp;
 5   // Build a mask on the high bits of a full length value to extract the high
 6   // bits of the address.
 7   Bit#(TSub#(TAdd#(CapAddrW,1),MW)) mask = ~0 << cap.bounds.exp;
 8   // Extract the high bits of the address (and append the implied zeros at the
 9   // bottom), and add with the previously prepared value.
10   CapAddrPlus1 ret = {truncateLSB({1'b0,cap.address})&mask,0} + addTop;
11   // If the bottom and top are more than an address space away from eachother,
12   // invert the 64th/32nd bit of Top.  This corrects for errors that happen
13   // when the representable space wraps the address space.
14   Bit#(2) topTip = truncateLSB(ret);
15   // Calculate the msb of the base.
16   // First assume that only the address and correction are involved...
```

---

[1] https://github.com/CTSRD-CHERI/cheri-cap-lib/tree/618e844

```
  Bit#(TSub#(CapAddrW,MW)) bot = truncateLSB(cap.address) + (signExtend(pack(tf.baseCorrection)) << cap.bounds.exp);
  Bit#(2) botTip = {1'b0, msb(bot)};
  // If the bit we're interested in are actually coming from baseBits, select
  // the correct one from there.
  // exp == (resetExp - 1) doesn't matter since we will not flip unless
  // exp < resetExp-1.
  if (cap.bounds.exp == (resetExp - 2)) botTip = {1'b0, cap.bounds.baseBits[valueOf(MW)-1]};
  // Do the final check.
  // If exp >= resetExp - 1, the bits we're looking at are coming directly from
  // topBits and baseBits, are not being inferred, and therefore do not need
  // correction. If we are below this range, check that the difference between
  // the resulting top and bottom is less than one address space.  If not, flip
  // the msb of the top.
  if (cap.bounds.exp<(resetExp-1) && (topTip - botTip) > 1)
    ret[valueOf(CapAddrW)] = ~ret[valueOf(CapAddrW)];
  return ret;
endfunction
```

D.2 CapInBounds

CapInBounds detects if the current address of a capability is within its bounds. This function does not decode the Top and Base of the capability, but operates directly on compressed fields saving both time and logic.

```
function Bool capInBounds(CapFat cap, TempFields tf, Bool inclusive);
  // Check that the pointer of a capability is currently within the bounds
  // of the capability
  Bool ptrVStop = inclusive ? cap.addrBits <= cap.bounds.topBits
                            : cap.addrBits <  cap.bounds.topBits;
  // Top is ok if the pointer and top are in the same alignment region
  // and the pointer is less than the top.  If they are not in the same
  // alignment region, it's ok if the top is in Hi and the bottom in Low.
  Bool topOk  = (tf.topHi  == tf.addrHi) ? ptrVStop : tf.topHi;
  Bool baseOk = (tf.baseHi == tf.addrHi) ? cap.addrBits >= cap.bounds.baseBits
                                         : tf.addrHi;
  return topOk && baseOk;
endfunction
```

D.3 IncOffset

The IncOffset function from cheri-cap-lib is shared between the IncOffset operation and the SetOffset operation as these two can be made to largely share logic. The IncOffset function is almost entirely composed of the *fast representable check* [178], as the only change to the capability in a non-faulting case is to add the increment to the address. This check determines if the resulting capability will decode as having the same bounds after the address modification, invalidating the capability if this might not be the case. As this check is conservative, the boundary conditions are specified in the CHERI architecture in Section 3.5.4. This check is intended to run alongside the add of the address in execute unit.

```
function VnD#(CapFat) incOffsetFat( CapFat cap
                                  , CapAddr pointer
                                  , CapAddr offset // this is the increment in inc offset, and the offset in set offset
                                  , TempFields tf
                                  , Bool setOffset);
// NOTE:
// The 'offset' argument is the "increment" value when setOffset is false, and
// the actual "offset" value when setOffset is true.
//
// For this function to work correctly, we must have
// 'offset' = 'pointer'-'cap.address'.
// In the most critical case we have both available and picking one or the
// other is less efficient than passing both.  If the 'setOffset' flag is set,
// this function will ignore the 'pointer' argument and use 'offset' to set the
// offset of 'cap' by adding it to the capability base. If the 'setOffset' flag
// is not set, this function will increment the offset of 'cap' by replacing
// the 'cap.address' field with the 'pointer' argument (with the assumption
// that the 'pointer' argument is indeed equal to 'cap.address'+'offset'.  The
// 'cap.addrBits' field is also updated accordingly.
  CapFat ret = cap;
  Exp e = cap.bounds.exp;
  // Updating the address of a capability requires checking that the new
  // address is still within representable bounds. For capabilities with big
  // representable regions (with exponents >= resetExp-2), there is no
  // representability issue.
  // For the other capabilities, the check consists of two steps:
  // - A "inRange" test
  // - A "inLimits" test

  // The inRange test
  // ----------------
  // Conceptually, the inRange test checks the magnitude of 'offset' is less
  // then the representable region's size S. This ensures that the inLimits
  // test result is meaningful. The test succeeds if the absolute value of
  // 'offset' is less than S, that is -S < 'offset' < S. This test reduces to a
  // check that there are no significant bits in the high bits of 'offset',
  // that is they are all ones or all zeros.
  CapAddr offsetAddr = offset;
  Bit#(TSub#(CapAddrW,MW)) signBits       = signExtend(offset[valueOf(TSub#(CapAddrW,1))]);
```

```
  Bit#(TSub#(CapAddrW,MW)) highOffsetBits = truncateLSB(offsetAddr);
  Bit#(TSub#(CapAddrW,MW)) highBitsfilter = -1 << e;
  highOffsetBits = (highOffsetBits ^ signBits) & highBitsfilter;
  Bool inRange = (highOffsetBits == 0);

  // The inLimits test
  // -----------------
  // Conceptually, the inLimits test ensures that neither the of the edges of
  // the representable region have been crossed with the new address. In
  // essence, it compares the distance 'offsetBits' added (on MW bits) with the
  // distance 'toBounds' to the edge of the representable space (on MW bits).
  // - For a positive or null increment
  //   inLimits = offsetBits < toBounds - 1
  // - For a negative increment:
  //   inLimits = (offsetBits >= toBounds) and ('we were not already on the
  //   bottom edge') (when already on the bottom edge of the representable
  //   space, the relevant bits of the address and those of the representable
  //   edge are the same, leading to a false positive on the i >= toBounds
  //   comparison)

  // The sign of the increment
  Bool posInc = msb(offsetAddr) == 1'b0;

  // The offsetBits value corresponds to the appropriate slice of the
  // 'offsetAddr' argument
  Bit#(MW) offsetBits  = truncate(offsetAddr >> e);

  // The toBounds value is given by substracting the address of the capability
  // from the address of the edge of the representable region (on MW bits) when
  // the 'setOffset' flag is not set. When it is set, it is given by
  // substracting the base address of the capability from the edge of the
  // representable region (on MW bits).  This value is both the distance to the
  // representable top and the distance to the representable bottom (when
  // appended to a one for negative sign), a convenience of the two's
  // complement representation.

  // NOTE: When the setOffset flag is set, toBounds should be the distance from
  // the base to the representable edge. This can be computed efficiently, and
  // without relying on the temporary fields, as follows: equivalent to
  // (repBoundBits - cap.bounds.baseBits):
  Bit#(MW) toBounds_A   = {3'b111,0} - {3'b000,truncate(cap.bounds.baseBits)};
  // equivalent to (repBoundBits - cap.bounds.baseBits - 1):
  Bit#(MW) toBoundsM1_A = {3'b110,~truncate(cap.bounds.baseBits)};
  /*
  XXX not sure if we still care about that
  if (toBoundsM1_A != (toBounds_A-1)) $display("error %x", toBounds_A[15:13]);
  */
```

```
// When the setOffset flag is not set, we need to use the temporary fields
// with the upper bits of the representable bounds
Bit#(MW) repBoundBits = {tf.repBoundTopBits,0};
Bit#(MW) toBounds_B   = repBoundBits - cap.addrBits;
Bit#(MW) toBoundsM1_B = repBoundBits + ~cap.addrBits;
// Select the appropriate toBounds value
Bit#(MW) toBounds   = setOffset ? toBounds_A   : toBounds_B;
Bit#(MW) toBoundsM1 = setOffset ? toBoundsM1_A : toBoundsM1_B;
Bool addrAtRepBound = !setOffset && (repBoundBits == cap.addrBits);

// Implement the inLimit test
Bool inLimits = False;
if (posInc) begin
  // For a positive or null increment
  // SetOffset is offsetting against base, which has 0 in the lower bits, so
  // we don't need to be conservative.
  inLimits = setOffset ? offsetBits <= toBoundsM1
                       : offsetBits <  toBoundsM1;
end else begin
  // For a negative increment
  inLimits = (offsetBits >= toBounds) && !addrAtRepBound;
end

// Complete representable bounds check
// -----------------------------------
Bool inBounds = (inRange && inLimits) || (e >= (resetExp - 2));

// Updating the return capability
// ------------------------------
if (setOffset) begin
  // Get the base and add the offsetAddr. This could be slow, but seems to
  // pass timing.
  ret.address = getBotFat(cap,tf) + offsetAddr;
  // Work out the slice of the address we are interested in using MW-bit
  // arithmetics.
  Bit#(MW) newAddrBits = cap.bounds.baseBits + offsetBits;
  // Ensure the bits of the address slice past the top of the address space
  // are zero
  Bit#(2) mask = (e == resetExp) ? 2'b00 : (e == resetExp-1) ? 2'b01 : 2'b11;
  ret.addrBits = {mask, ~0} & newAddrBits;
end else begin
  // In the incOffset case, the 'pointer' argument already contains the new
  // address
  ret.address  = pointer;
  ret.addrBits = truncate(ret.address >> e);
end
// Nullify the capability if the representable bounds check has failed
```

```
  if (!inBounds) ret.isCapability = False;

  // return updated / invalid capability
  return VnD {v: inBounds, d: ret};
endfunction
```

D.4 SetAddress

This *SetAddress* function assigns a new address to the address field of a capability. It is almost entirely composed of a *fast representable check* which asserts that the bounds will continue to decode to the same value after the assignment without actually decoding either of the bounds, which is slow. If the bounds would change, the capability is invalidated.

```
function VnD#(CapFat) setAddress(CapFat cap, CapAddr address, TempFields tf);
  CapFat ret = setCapPointer(cap, address);
  Exp e = cap.bounds.exp;
  // Calculate what the difference in the upper bits of the new and original
      addresses must be if
  // the new address is within representable bounds.
  Bool newAddrHi  = truncateLSB(ret.addrBits) < tf.repBoundTopBits;
  Bit#(TSub#(CapAddrW,MW)) deltaAddrHi = signExtend({1'b0,pack(newAddrHi)} - {1'b0,
      pack(tf.addrHi)}) << e;
  // Calculate the actual difference between the upper bits of the new address and
      the original address.
  Bit#(TSub#(CapAddrW,MW)) mask = -1 << e;
  Bit#(TSub#(CapAddrW,MW)) deltaAddrUpper = (truncateLSB(address)&mask) - (
      truncateLSB(cap.address)&mask);
  Bool inRepBounds = deltaAddrHi == deltaAddrUpper;
  if (!inRepBounds) ret.isCapability = False;
  return VnD {v: inRepBounds, d: ret};
endfunction
```

D.5 SetBounds

The *SetBounds* function sets a new base and length of a capability, performing and necessary rounding. This function actually returns a data structure which includes not only the new capability, but a flag indicating if rounding was necessary (to facilitate *CSetBoundsExact*), a mask that could be applied to a pointer to align it with the supplied length (to facilitate *CRepresentableAlignmentMask*), as well as the length that was actually achieved after rounding (to facilitate *CRoundRepresentableLength*).

```
function SetBoundsReturn#(CapFat, CapAddrW) setBoundsFat(CapFat cap, Address
    lengthFull);
  CapFat ret = cap;
  // Find new exponent by finding the index of the most significant bit of the
```

```
// length, or counting leading zeros in the high bits of the length, and
// substracting them to the CapAddr width (taking away the bottom MW-1 bits:
// trim (MW-1) bits from the bottom of length since any length with a
// significance that small will yield an exponent of zero).
CapAddr length = truncate(lengthFull);
Bit#(TSub#(CapAddrW,TSub#(MW,1))) lengthMSBs = truncateLSB(length);
Exp zeros = zeroExtend(countZerosMSB(lengthMSBs));
// Adjust resetExp by one since it’s scale reaches 1-bit greater than a
// 64-bit length can express.
Bool maxZero = (zeros==(resetExp-1));
Bool intExp = !(maxZero && length[fromInteger(valueOf(TSub#(MW,2)))]==1’b0);
// Do this without subtraction
//fromInteger(valueof(TSub#(SizeOf#(Address),TSub#(MW,1)))) - zeros;
Exp e = (resetExp-1) - zeros;
// Derive new base bits by extracting MW bits from the capability address
// starting at the new exponent’s position.
CapAddrPlus2 base = {2’b0, cap.address};
Bit#(TAdd#(MW,1)) newBaseBits = truncate(base>>e);

// Derive new top bits by extracting MW bits from the capability address +
// requested length, starting at the new exponent’s position, and rounding up
// if significant bits are lost in the process.
CapAddrPlus2 len = {2’b0, length};
CapAddrPlus2 top = base + len;

// Create a mask with all bits set below the MSB of length and then masking
// all bits below the mantissa bits.
CapAddrPlus2 lmask = smearMSBRight(len);
// The shift amount required to put the most significant set bit of the len
// just above the bottom HalfExpW bits that are taken by the exp.
Integer shiftAmount = valueOf(TSub#(TSub#(MW,2),HalfExpW));

// Calculate all values associated with E=e (e not rounding up)
// Round up considering the stolen HalfExpW exponent bits if required
Bit#(TAdd#(MW,1)) newTopBits = truncate(top>>e);
// Check if non-zero bits were lost in the low bits of top, either in the ’e’
// shifted out bits or in the HalfExpW bits stolen for the exponent
// Shift by MW-1 to move MSB of mask just below the mantissa, then up
// HalfExpW more to take in the bits that will be lost for the exponent when
// it is non-zero.
CapAddrPlus2 lmaskLor = lmask>>fromInteger(shiftAmount+1);
CapAddrPlus2 lmaskLo  = lmask>>fromInteger(shiftAmount);
// For the len, we’re not actually losing significance since we’re not
// storing it, we just want to know if any low bits are non-zero so that we
// will know if it will cause the total length to round up.
Bool lostSignificantLen  = (len&lmaskLor)!=0 && intExp;
Bool lostSignificantTop  = (top&lmaskLor)!=0 && intExp;
```

```
// Check if non-zero bits were lost in the low bits of base, either in the
// 'e' shifted out bits or in the HalfExpW bits stolen for the exponent
Bool lostSignificantBase = (base&lmaskLor)!=0 && intExp;

// Calculate all values associated with E=e+1 (e rounding up due to msb of L
// increasing by 1) This value is just to avoid adding later.
Bit#(MW) newTopBitsHigher = truncateLSB(newTopBits);
// Check if non-zero bits were lost in the low bits of top, either in the 'e'
// shifted out bits or in the HalfExpW bits stolen for the exponent Shift by
// MW-1 to move MSB of mask just below the mantissa, then up HalfExpW more to
// take in the bits that will be lost for the exponent when it is non-zero.
Bool lostSignificantTopHigher  = (top&lmaskLo)!=0 && intExp;
// Check if non-zero bits were lost in the low bits of base, either in the
// 'e' shifted out bits or in the HalfExpW bits stolen for the exponent
Bool lostSignificantBaseHigher = (base&lmaskLo)!=0 && intExp;
// If either base or top lost significant bits and we wanted an exact
// setBounds, void the return capability

// We need to round up Exp if the msb of length will increase.
// We can check how much the length will increase without looking at the
// result of adding the length to the base.  We do this by adding the lower
// bits of the length to the base and then comparing both halves (above and
// below the mask) to zero.  Either side that is non-zero indicates an extra
// "1" that will be added to the "mantissa" bits of the length, potentially
// causing overflow.  Finally check how close the requested length is to
// overflow, and test in relation to how much the length will increase.
CapAddrPlus2 topLo = (lmaskLor & len) + (lmaskLor & base);
CapAddrPlus2 mwLsbMask = lmaskLor ^ lmaskLo;
// If the first bit of the mantissa of the top is not the sum of the
// corrosponding bits of base and length, there was a carry in.
Bool lengthCarryIn = (mwLsbMask & top) != ((mwLsbMask & base)^(mwLsbMask & len));
Bool lengthRoundUp = lostSignificantTop;
Bool lengthIsMax        = (len & (~lmaskLor)) == (lmask ^ lmaskLor);
Bool lengthIsMaxLessOne = (len & (~lmaskLor)) == (lmask ^ lmaskLo);

Bool lengthOverflow = False;
if (lengthIsMax && (lengthCarryIn || lengthRoundUp)) lengthOverflow = True;
if (lengthIsMaxLessOne && lengthCarryIn && lengthRoundUp) lengthOverflow = True;

if(lengthOverflow && intExp) begin
  e = e+1;
  ret.bounds.topBits = lostSignificantTopHigher ? newTopBitsHigher + 'b1000
                                                : newTopBitsHigher;
  ret.bounds.baseBits = truncateLSB(newBaseBits);
end else begin
  ret.bounds.topBits = lostSignificantTop ? truncate(newTopBits + 'b1000)
                                          : truncate(newTopBits);
```

```
    ret.bounds.baseBits = truncate(newBaseBits);
  end
  Bool exact = !(lostSignificantBase || lostSignificantTop);

  ret.bounds.exp = e;
  // Update the addrBits fields
  ret.addrBits = ret.bounds.baseBits;
  // Derive new format from newly computed exponent value, and round top up if
  // necessary
  if (!intExp) begin // If we have an Exp of 0 and no implied MSB of L.
    ret.format = Exp0;
  end else begin
    ret.format = EmbeddedExp;
    Bit#(HalfExpW) botZeroes = 0;
    ret.bounds.baseBits = {truncateLSB(ret.bounds.baseBits), botZeroes};
    ret.bounds.topBits  = {truncateLSB(ret.bounds.topBits),  botZeroes};
  end

  // Begin calculate newLength in case this is a request just for a
  // representable length:
  CapAddrPlus2 newLength = {2'b0, length};
  CapAddrPlus2 baseMask = -1; // Override the result from the previous line if
                              // we represent everything.
  if (intExp) begin
    CapAddrPlus2 oneInLsb = (lmask ^ (lmask>>1)) >> shiftAmount;
    CapAddrPlus2 newLengthRounded = newLength + oneInLsb;
    newLength        = (newLength        & (~lmaskLor));
    newLengthRounded = (newLengthRounded & (~lmaskLor));
    if (lostSignificantLen) newLength = newLengthRounded;
    baseMask = (lengthIsMax && lostSignificantTop) ? ~lmaskLo : ~lmaskLor;
  end

  // Return derived capability
  return SetBoundsReturn { cap:    ret
                         , exact:  exact
                         , length: truncate(newLength)
                         , mask:   truncate(baseMask) };
endfunction
```

Appendix E

CHERI-128 Alternative Compression Formats (Deprecated)

On the path to developing our current capability compression scheme, CHERI Concentrate (see Section 3.5.4), 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

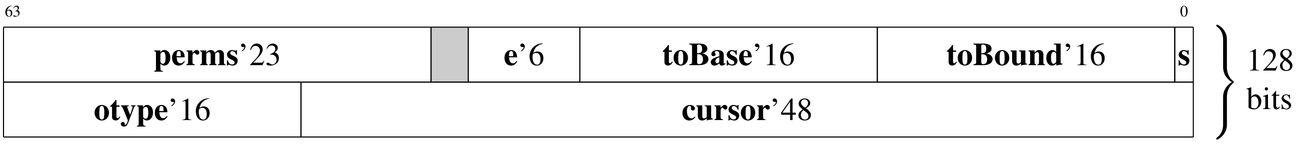

Figure E.1: CHERI-128 c1 memory representation of a capability

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.

$$\mathbf{base} = \{\mathbf{cursor}[63 : \mathbf{e} + 16] + \mathbf{correction}, \mathbf{baseBits}\} \ll \mathbf{e}$$

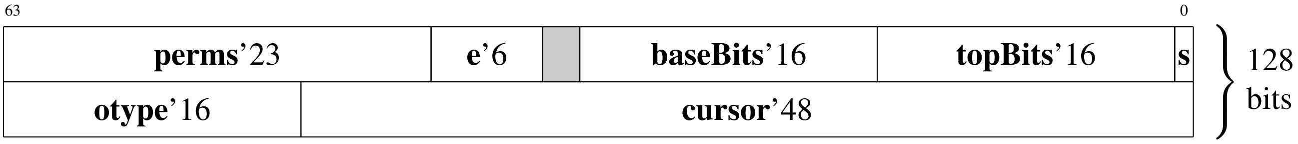

Figure E.2: CHERI-128 c2 memory representation of a capability

The bits above $(\mathbf{e}+16)$ in **cursor** may differ from **base** by at most 1, i.e.

$$\mathbf{correction} = f(\mathbf{baseBits}, \mathbf{topBits}, \mathbf{cursor}[\mathbf{e}+15:\mathbf{e}]) = (1, 0, \mathit{or} - 1)$$

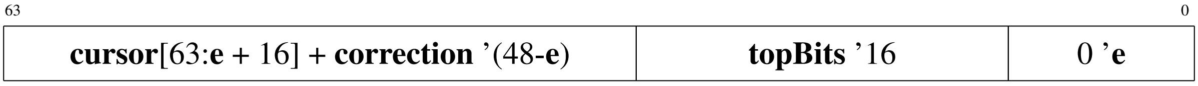

Figure E.3: CHERI-128 c2 base construction

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:

$$\mathbf{correction} = f(\mathbf{baseBits}, \mathbf{topBits}, \mathbf{cursor}[\mathbf{e}+15:\mathbf{e}]) = (1, 0, \mathit{or} - 1)$$

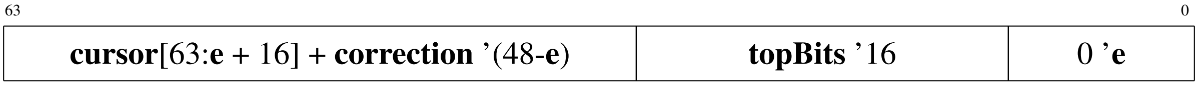

Figure E.4: CHERI-128 c2 top bound construction

Candidate 2 Notes Candidate 2 is inspired by "Low-fat pointers" [78], 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 $2^{(\mathbf{e}+16)}$-sized region. Candidate 2 sets the edges of this range to a fixed $1024^{\mathbf{e}}$ 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 [178], detailed in Section 3.5.4.

Alternative exponents The CHERI-128 scheme treats the exponent (**e**) as a $2^{\mathbf{e}}$ 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 $16^{\mathbf{e}[5:2]}$. 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 [71]. 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.

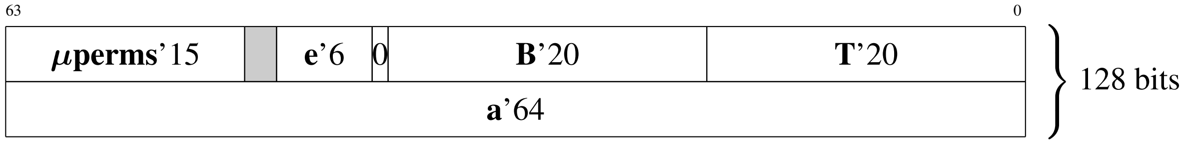

Figure E.5: Unsealed CHERI-128 memory representation of a capability

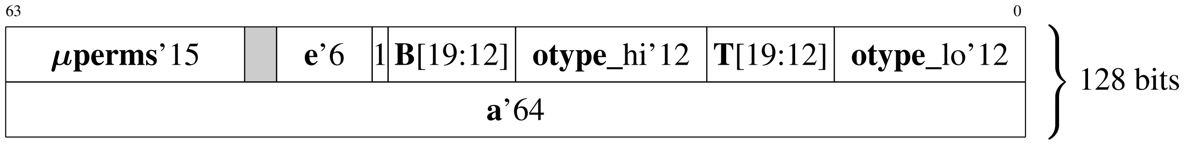

Figure E.6: Sealed CHERI-128 memory representation of a capability

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 $\mathbf{base} + \mathbf{offset}$.

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_INVOKE
perms[9]	9	PERMIT_UNSEAL
perms[10]	10	PERMIT_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** $(\mathbf{base} + \mathbf{length})$. 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:

  $$\mathbf{maxLength}(n) \geq \textbf{rlength}\text{where}\mathbf{maxLength}(n) = \left\lfloor \frac{2^{n+20}}{1 + 2^{-6}} \right\rfloor$$

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 $\mathbf{c_b}$ 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 $\mathbf{c_t}$ 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-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 or a suitably-aligned word of 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 CInvoke instruction, which may unseal (and jump to) suitably checked sealed capabilities..

capability object type In addition to fat-pointer metadata such as capability bounds and capability permissions, capabilities also contain an integer object type. The object type space is partitioned into a range of non-reserved and reserved capability object type types. The reserved capability object types are hardware-interpreted and include unsealed capabilities or sealed entry capabilities. If the object type is one of the non-reserved capability object types, the capability is a sealed capability with an object type. For sealed capabilities with object types, 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 CInvoke to implement a software object model or to implement software-defined tokens of authority.

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. A capability register might be a dedicated register intended primarily for capability-related operations (e.g., the default data capability (**DDC**)), or a general-purpose integer register that has been extended with capability metadata (such as the program-counter capability (**PCC**). 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 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.

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.

CInvoke The `CInvoke` instruction is a source of controlled non-monotonicity in the CHERI-RISC-V and CHERI-x86-64 ISAs. It can directly enter any userspace domain described by a pair of sealed capabilities with the *Permit_Invoke* 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. 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.

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 roughly 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.

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 code capability An architecture-defined capability which holds 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.

exception data capability An architecture-defined capability which holds 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..

exception program-counter capability An architecture-specific location into which the running program-counter capability (**PCC**) is stored on an exception, and whose value is loaded 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 CInvoke. 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 object's internal state.

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.

Morello An application of the CHERI protection model to the ARMv8-A architecture.

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. 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-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 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 saved in the exception program-counter capability, and the value of the exception code capability moved into the program-counter capability. On exception return, the value of the exception program-counter capability will be restored to 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.

reserved capability object type Certain capability object types are not available for software use and instead have hardware-defined semantics. On CHERI-RISC-V and CHERI-x86-64, all negative capability object types are reserved: unsealed capabilities use the value $2^{64} - 1$ and sealed entry capabilities have an object type of $2^{64} - 2$. The remaining capability object types are used for sealed capabilities with object types.

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 not equal to the unsealed object type ($2^{64} - 1$ for CHERI-RISC-V and CHERI-x86-64). A sealed capability's address, capability bounds, capability permissions, and other fields are immutable – i.e., cannot be modified using capability-based instructions. A sealed capability cannot be directly dereferenced using the instruction set, and must be unsealed before it can be used. This can be used to implement non-monotonic domain transition, as a sealed capability may carry rights not otherwise present in the capability registers. Two types exist: sealed capabilities with object types and sealed entry capabilities. They have different properties catering to different use cases.

sealed capability with an object type A sealed capability whose capability object type is not one of the reserved capability object types. These sealed capability 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. Unsealing can be performed using the CInvoke instruction, or using the `CUnseal` instruction combined with a suitable sealing capability. Sealed capabilities with object types provide the necessary architectural encapsulation support to efficiently implement fine-grained compartmentalization using an object-oriented model.

sealed entry capability A sealed entry capability (also known as sentry capability) is a sealed capability whose capability object type is set to the sentry reserved capability object type ($2^{64} - 2$ for CHERI-RISC-V and CHERI-x86-64). Sealed entry capabilities are commonly referred to as sentry capabilities. Sealed entry capabilities are do not support linking sealed code and data capabilities, unlike sealed capabilities with object types. A sealed entry capability is unsealed by jumping to it using a regular capability jump instruction.

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.

sentry capability Sentry capability is a convenient shorthand for a sealed entry capability.

software compartmentalization The configuration of code capabilities and data capabilities available via accessible 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 CInvoke 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 default data capability (**DDC**) as well as additional architecture-specific capability registers. Not all registers are accessible at all times; for example, some may be available only in certain rings, or when **PCC** has PERMIT_ACCESS_SYSTEM_REGISTERS.

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 registers, 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.

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 the unsealed object type ($2^{64} - 1$ for CHERI-MIPS and CHERI-RISC-V). 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, as well as capability bounds relative to the address. The integer addresses passed to legacy load and store instructions are checked with CHERI using the default data capability (**DDC**). The interpretation – physical or virtual addresses – is not changed by CHERI. Similarly, the integer addresses passed to legacy branch and jump instructions are checked using the program-counter capability (**PCC**).

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] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian and M. Young. *Mach: A New Kernel Foundation for UNIX Development*. Tech. rep. Computer Science Department, Carnegie Mellon University, Aug. 1986.

[2] 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*. ACM, 1967, pp. 5.1–5.10. DOI: 10.1145/800001.811666.

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

[4] J. Anderson. *Computer Security Technology Planning Study*. Tech. rep. ESD-TR-73-51. U.S. Air Force Electronic Systems Division, Oct. 1972.

[5] G. R. Andrews. *Partitions and Principles for Secure Operating Systems*. Tech. rep. Cornell University, 1975.

[6] Apple Inc. *Mac OS X Snow Leopard*. http://www.apple.com/macosx/. 2010.

[7] *Arm Architecture Reference Manual Supplement: Morello for A-profile Architecture*. Arm Limited. Oct. 2020.

[8] *ARM® A64 Instruction Set Architecture ARMv8, for ARMv8-A architecture profile*. Arm Limited. 2020.

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

[10] 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'. In: *Annals of the History of Computing* 9.1 (Jan. 1987), pp. 37–92. DOI: 10.1109/MAHC.1987.10006.

[11] T. Bauereiss, B. Campbell, T. Sewell, A. Armstrong, L. Esswood, I. Stark, G. Barnes, R. N. M. Watson and P. Sewell. 'Verified Security for the Morello Capability-enhanced Prototype Arm Architecture'. In: *31st European Symposium on Programming, ESOP 2022*. Ed. by I. Sergey. Vol. 13240. Springer, 2022, pp. 174–203. DOI: `10.1007/978-3-030-99336-8\_7`. URL: `https://doi.org/10.1007/978-3-030-99336-8%5C_7`.

[12] T. Bauereiss, B. Campbell, T. Sewell, A. Armstrong, L. Esswood, I. Stark, G. Barnes, R. N. M. Watson and P. Sewell. *Verified security for the Morello capability-enhanced prototype Arm architecture*. Tech. rep. UCAM-CL-TR-959. University of Cambridge, Computer Laboratory, Sept. 2021. DOI: `10.48456/tr-959`. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-959.pdf`.

[13] D. Bell and L. L. Padula. *Secure Computer System: Unified Exposition and Multics Interpretation*. Tech. rep. ESD-TR-75-306. The Mitre Corporation, Mar. 1976.

[14] 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*. ACM, 1995, pp. 267–283. DOI: `10.1145/224056.224077`.

[15] R. Bisbey II and D. Hollingworth. *Protection Analysis: Project final report*. Tech. rep. USC Information Sciences Institute (ISI), 1978.

[16] 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*. USENIX Association, 2008, pp. 309–322. URL: `http://www.cs.ucl.ac.uk/staff/m.handley/papers/wedge.pdf`.

[17] *Bluespec SystemVerilog Version 3.8 Reference Guide*. Bluespec, Inc. Nov. 2004.

[18] W. E. Boebert. 'On the Inability of an Unmodified Capability Machine to Enforce the *-Property'. In: *Proc. 7th DoD/NBS Computer Security Conference*. DoD/NBS Computer Security Conference. Sept. 1984, pp. 291–293. URL: `https://csrc.nist.gov/CSRC/media/Publications/conference-paper/1984/09/24/7th-dod-nbs-computer-security-conference/documents/1984-7th-conference-proceedings.pdf`.

[19] M. Branstad and J. Landauer. 'Assurance for the Trusted Mach Operating System'. In: *Proceedings of the Fourth Annual Conference on Computer Assurance COMPASS '89*. IEEE. June 1989, pp. 9–13.

[20] J. V. Bulck, M. Minkin, O. Weisse, 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'. In: 27th USENIX Security Symposium. USENIX Association, Aug. 2018, pp. 991–1008. ISBN: 978-1-939133-04-5. URL: `https://www.usenix.org/conference/usenixsecurity18/presentation/bulck`.

[21] B. Campbell and I. Stark. 'Randomised testing of a microprocessor model using SMT-solver state generation'. In: *Sci. Comput. Program.* 118 (2016), pp. 60–76. DOI: `10.1016/j.scico.2015.10.012`.

[22] 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*. USENIX Association, 2004.

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

[24] N. P. Carter, S. W. Keckler and W. J. Dally. 'Hardware Support for Fast Capability-Based Addressing'. In: *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA). ACM, Nov. 1994, pp. 319–327. DOI: `10.1145/195473.195579`. URL: `https://www.cs.utexas.edu/users/skeckler/pubs/asplos94.pdf`.

[25] 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)*. 2013, pp. 570–576.

[26] 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. Watson. 'CHERI JNI: Sinking the Java Security Model into the C'. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China). ACM, Apr. 2017, pp. 569–583. DOI: `10.1145/3037697.3037725`. URL: `https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201704-asplos-cherijni.pdf`.

[27] D. Chisnall, C. Rothwell, B. Davis, R. N. M. Watson, J. Woodruff, S. W. Moore, P. G. Neumann and M. Roe. 'Beyond the PDP-11: Architectural support for a memory-safe C abstract machine'. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015.

[28] E. Cohen and D. Jefferson. 'Protection in the Hydra Operating System'. In: *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, 1st Nov. 1975, pp. 141–160. DOI: `10.1145/800213.806532`.

[29] 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*. ACM, 1965, pp. 185–196. DOI: `10.1145/1463891.1463912`.

[30] 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*. ACM, 1962, pp. 335–344. DOI: `10.1145/1460833.1460871`.

[31] B. Corporation. *B 6700 Information Processing System Reference Manual*. 1972. URL: `http://bitsavers.trailing-edge.com/pdf/burroughs/B6500_6700/1058633_B6700_RefMan_May72.pdf`.

[32] B. A. Creech. 'Architecture of the B-6500'. In: *Software Engineering: Proceedings of the Third Symposium on Computer and Information Sciences*. Ed. by J. T. Tou. Vol. 1. Dec. 1970, pp. 29–43. ISBN: 9780323157445.

[33] *CTSRD-CHERI/cheri-cap-lib: A library of specific implementations of cheri and providing an abstract interface to those implementations*. URL: https://github.com/CTSRD-CHERI/cheri-cap-lib.

[34] *CTSRD-CHERI/cheri-cpu: CHERI-MIPS implementation in a 6-stage pipeline with associative caches and multi-core support*. URL: https://github.com/CTSRD-CHERI/cheri-cpu.

[35] *CTSRD-CHERI/Flute: RISC-V CPU, simple 5-stage in-order pipeline, for low-end applications needing MMUs and some performance*. URL: https://github.com/CTSRD-CHERI/Flute.

[36] *CTSRD-CHERI/Piccolo: RISC-V CPU, simple 3-stage pipeline, for low-end applications (e.g., embedded, IoT)*. URL: https://github.com/CTSRD-CHERI/Piccolo.

[37] *CTSRD-CHERI/TagController: Multi-level tag controller for emulating a tagged memory using an in-memory table*. URL: https://github.com/CTSRD-CHERI/TagController.

[38] *CTSRD-CHERI/Toooba: RISC-V Core; superscalar, out-of-order, multi-core capable; based on RISCY-OOO from MIT*. URL: https://github.com/CTSRD-CHERI/Toooba.

[39] R. C. Daley and P. G. Neumann. 'A General-Purpose File System for Secondary Storage'. In: *AFIPS Conference Proceedings, Fall Joint Computer Conference*. Spartan Books, Nov. 1965, pp. 213–229.

[40] B. Davis, R. N. M. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka, A. Joannou, B. Laurie, A. T. Markettos, J. E. Maste, A. Mazzinghi, E. T. Napierala, R. M. Norton, M. Roe, P. Sewell, S. Son and J. Woodruff. 'CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment'. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2019 (Providence, RI, USA). ACM, Apr. 2019, pp. 379–393. DOI: 10.1145/3297858.3304042. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201904-asplos-cheriabi.pdf.

[41] P. J. Denning. 'Fault Tolerant Operating Systems'. In: *ACM Computing Surveys* 8.4 (1976), pp. 359–389. DOI: 10.1145/356678.356680.

[42] J. B. Dennis and E. C. Van Horn. 'Programming semantics for multiprogrammed computations'. In: *Communications of the ACM* 9.3 (1966), pp. 143–155. DOI: 10.1145/365230.365252.

[43] *Department of Defense Trusted Computer System Evaluation Criteria (TCSEC)*. DoD 5200.28-STD, Orange Book. 26th Dec. 1985.

[44] J. Devietti, C. Blundell, M. M. K. Martin and S. Zdancewic. 'Hardbound: Architectural Support for Spatial Safety of the C Programming Language'. In: *SIGPLAN Not.* 43.3 (Mar. 2008), pp. 103–114. DOI: 10.1145/1353536.1346295.

[45] M. G. Dodson. *Capability-based access control for cyber physical systems*. Tech. rep. UCAM-CL-TR-963. University of Cambridge, Computer Laboratory, Oct. 2021. DOI: `10.48456/tr-963`. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-963.pdf`.

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

[47] 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'. In: *SIGOPS Oper. Syst. Rev.* 39 (5 Oct. 2005), pp. 17–30. DOI: `10.1145/1095809.1095813`.

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

[49] R. S. Fabry. 'The case for capability based computers (Extended Abstract)'. In: *SOSP '73: Proceedings of the Fourth ACM Symposium on Operating System Principles*. ACM, 1973, p. 120. DOI: `10.1145/800009.808060`.

[50] R. J. Feiertag and P. G. Neumann. 'The Foundations of a Provably Secure Operating System (PSOS)'. In: *Proceedings of the National Computer Conference*. AFIPS Press, 1979, pp. 329–334.

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

[52] A. Fox. 'Improved Tool Support for Machine-Code Decompilation in HOL4'. In: *Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. Ed. by C. Urban and X. Zhang. Springer International Publishing, 2015, pp. 187–202. ISBN: 978-3-319-22102-1.

[53] F. A. Fuchs, J. Woodruff, S. W. Moore, P. G. Neumann and R. N. Watson. 'Developing a Test Suite for Transient-Execution Attacks on RISC-V and CHERI-RISC-V'. In: *Workshop on Computer Architecture Research with RISC-V*. 2021.

[54] L. Gong. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, Reading, Massachusetts, 1999.

[55] 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*. Dec. 1997.

[56] J. Gosling, B. Joy and G. L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN: 0201634511.

[57] S. Govindavajhala and A. Appel. 'Using Memory Errors to Attack a Virtual Machine'. In: *Proceedings of the 2003 Symposium on Security and Privacy*. IEEE Computer Society. May 2003, pp. 154–165.

[58] R. Graham. 'Protection in an Information Processing Utility'. In: *Communications of the ACM* 11.5 (May 1968).

[59] 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)*. Oct. 2015.

[60] 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)*. Tech. rep. UCAM-CL-TR-873. University of Cambridge, Computer Laboratory, Dec. 2015. URL: `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-873.pdf`.

[61] R. H. Gumpertz. 'Error Detection with Memory Tags'. PhD thesis. Carnegie Mellon University, Dec. 1981. URL: `http://reports-archive.adm.cs.cmu.edu/anon/scan/CMU-CS-84-122.pdf`.

[62] B. Gutstein. *Memory safety with CHERI capabilities: security analysis, language interpreters, and heap temporal safety*. Tech. rep. UCAM-CL-TR-975. University of Cambridge, Computer Laboratory, Nov. 2022. DOI: `10.48456/tr-975`. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-975.pdf`.

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

[64] N. Hardy. 'The Confused Deputy (or why capabilities might have been invented)'. In: *ACM SIGOPS Operating Systems Review* 22.4 (Oct. 1988).

[65] *Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z*. 325383. Version 070US. Intel Corporation. May 2019.

[66] International Standards Organization. *Common Criteria for Information Technology Security Evaluation – Part 3: Security assurance components*. Sept. 2012.

[67] International Standards Organization. *Common Criteria for Information Technology Security Evaluation – Part 1: Introduction and General Model*. Apr. 2017.

[68] *Introduction to SPARC M7 and Application Data Integrity (ADI)*. `https://swisdev.oracle.com/_files/What-Is-ADI.html`.

[69] 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*. USENIX Association, 2002, pp. 275–288. ISBN: 1-880446-00-6.

[70] 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)*. Nov. 2017.

[71] A. J. P. Joannou. *High-performance memory safety: optimizing the CHERI capability machine*. Tech. rep. UCAM-CL-TR-936. University of Cambridge, Computer Laboratory, May 2019. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-936.pdf`.

[72] 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*. Institut de Recherche d’Informatique. 13th–14th Aug. 1974, pp. 121–135.

[73] P. Karger. ‘Limiting the Damage Potential of Discretionary Trojan Horses’. In: *Proceedings of the 1987 Symposium on Security and Privacy*. IEEE Computer Society. Apr. 1987, pp. 32–37.

[74] P. Karger and R. Schell. ‘Multics Security Evaluation: Vulnerability Analysis’. In: *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC), Classic Papers section*. Dec. 2002.

[75] 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)*. Oct. 2003.

[76] 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’. In: *Communications of the ACM* 53 (6 June 2009), pp. 107–115. DOI: `10.1145/1743546.1743574`.

[77] 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’. In: *ArXiv e-prints* (Jan. 2018). arXiv: `1801.01203`.

[78] 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*. Nov. 2013.

[79] B. W. Lampson. ‘Dynamic protection structures’. In: *AFIPS ’69 (Fall): Proceedings of the November 18-20, 1969, Fall Joint Computer Conference*. ACM, 1969, pp. 27–38. DOI: `10.1145/1478559.1478563`.

[80] B. Lampson. ‘Redundancy and Robustness in Memory Protection’. In: *Information Processing 74 (Proceedings of the IFIP Congress 1974)*. Vol. Hardware II. North-Holland, Amsterdam, 1974, pp. 128–132.

[81] B. W. Lampson. ‘Protection’. In: *SIGOPS Operating Systems Review* 8.1 (1974), pp. 18–24. DOI: 10.1145/775265.775268.

[82] C. Lattner and V. Adve. ‘LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation’. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California). IEEE Computer Society, Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

[83] N. G. Leveson and W. Young. ‘An Integrated Approach to Safety and Security Based on System Theory’. In: *Communications of the ACM* 57.2 (Feb. 2014), pp. 31–35.

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

[85] J. Liedtke. ‘On Microkernel Construction’. In: *SOSP’95: Proceedings of the 15th ACM Symposium on Operating System Principles*. Dec. 1995. URL: http://l4ka.org/publications/.

[86] 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*. ACM, 1974, pp. 973–980. DOI: 10.1145/1500175.1500361.

[87] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom and M. Hamburg. ‘Meltdown’. In: *ArXiv e-prints* (Jan. 2018). arXiv: 1801.01207.

[88] A. Margaritov, D. Ustiugov, E. Bugnion and B. Grot. ‘Prefetched address translation’. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 1023–1036.

[89] A. T. Markettos, J. Baldwin, R. Bukin, P. G. Neumann, S. W. Moore and R. N. M. Watson. ‘Position Paper: Defending Direct Memory Access with CHERI Capabilities’. In: *HASP@MICRO 2020: Hardware and Architectural Support for Security and Privacy, Virtual Event, Greece, 17 October 2020*. Ed. by J. Szefer, W. Shi and R. B. Lee. ACM, 2020, 7:1–7:9. DOI: 10.1145/3458903.3458910. URL: https://doi.org/10.1145/3458903.3458910.

[90] A. J. Mashtizadeh, A. Bittau, D. Boneh and D. Mazières. ‘CCFI: Cryptographically Enforced Control Flow Integrity’. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 12th Oct. 2015, pp. 941–951. DOI: 10.1145/2810103.2813676.

[91] A. J. W. Mayer. ‘The Architecture of the Burroughs B5000: 20 Years Later and Still Ahead of the Times?’ In: *ACM SIGARCH Computer Architecture News* 10.4 (June 1982), pp. 3–10. DOI: 10.1145/641542.641543.

[92] 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)*. July 2018.

[93] 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*. USENIX Association, 1993.

[94] E. McCauley and P. Drongowski. 'KSOS: The Design of a Secure Operating System'. In: *National Computer Conference*. AFIPS Conference Proceedings. 1979, pp. 345–353.

[95] 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.

[96] K. Memarian. *The Cerberus C semantics*. Tech. rep. UCAM-CL-TR-981. University of Cambridge, Computer Laboratory, May 2023. DOI: 10.48456/tr-981. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-981.pdf.

[97] 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. DOI: 10.1145/3290380.

[98] 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*. ACM, June 2016.

[99] 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*. ACM, 2010, pp. 1–7. DOI: 10.1145/1814217.1814224.

[100] M. S. Miller, K.-P. Yee and J. Shapiro. 'Capability Myths Demolished'. en. In: (), p. 15. URL: http://srl.cs.jhu.edu/pubs/SRL2003-02.pdf.

[101] M. S. Miller, M. Samuel, B. Laurie, I. Awad and M. Stay. *Caja: Safe active content in Sanitized JavaScript*. May 2008. URL: http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf.

[102] M. S. Miller and J. S. Shapiro. 'Paradigm Regained: Abstraction Mechanisms for Access Control'. In: (2003). Ed. by V. A. Saraswat, pp. 224–242. URL: http://www.erights.org/talks/asian03/.

[103] M. S. Miller. 'Robust composition: towards a unified approach to access control and concurrency control'. PhD thesis. Johns Hopkins University, 2006.

[104] J. H. Morris Jr. 'Protection in programming languages'. In: *Communications of the ACM* 16.1 (1973), pp. 15–21. DOI: 10.1145/361932.361937.

[105] A. C. Myers and B. Liskov. 'A decentralized model for information flow control'. In: *SIGOPS Oper. Syst. Rev.* 31 (5 Oct. 1997), pp. 129–142. DOI: 10.1145/269005.266669.

[106] 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*. ACM, 2009, pp. 245–258. DOI: 10.1145/1542476.1542504.

[107] 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*. ACM, 1996, pp. 229–243. DOI: 10.1145/238721.238781.

[108] 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*. ACM, 2002, pp. 128–139. DOI: 10.1145/503272.503286.

[109] P. G. Neumann, R. Boyer, R. Feiertag, K. Levitt and L. Robinson. *A Provably Secure Operating System: The System, Its Applications, and Proofs*. Tech. rep. Computer Science Laboratory, SRI International, Menlo Park, California, May 1980.

[110] P. G. Neumann and R. J. Feiertag. 'PSOS Revisited'. In: *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), Classic Papers section*. IEEE Computer Society. Dec. 2003, pp. 208–216.

[111] P. G. Neumann. 'Fundamental Trustworthiness Principles in CHERI'. In: *New Solutions for Cybersecurity*. Ed. by H. Shrobe, D. L. Shrier and A. Pentland. MIT Press/-Connection Science, 2018. Chap. 6.

[112] P. G. Neumann. 'Holistic Systems'. In: *ACM Software Engineering Notes* 31.6 (Nov. 2006), pp. 4–5.

[113] 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*. U.S. Air Force Cryptographic Modernization Office and AFRL. Dec. 2010.

[114] K. Nienhuis, A. Joannou, T. Bauereiss, A. Fox, M. Roe, B. Campbell, M. Naylor, R. M. Norton, S. W. Moore, P. G. Neumann, I. Stark, R. N. M. Watson and P. Sewell. 'Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process'. In: *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP)*. May 2020, pp. 1007–1024. DOI: 10.1109/SP40000.2020.00055.

[115] K. Nienhuis, A. Joannou, A. Fox, M. Roe, T. Bauereiss, B. Campbell, M. Naylor, R. M. Norton, S. W. Moore, P. G. Neumann, I. Stark, R. N. M. Watson and P. Sewell. *Rigorous engineering for hardware security: formal modelling and proof in the CHERI design and implementation process*. Tech. rep. UCAM-CL-TR-940. University of Cambridge, Computer Laboratory, Sept. 2019. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-940.pdf.

[116] R. M. Norton. *Hardware support for compartmentalisation*. Tech. rep. UCAM-CL-TR-887. University of Cambridge, Computer Laboratory, May 2016. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-887.pdf.

[117] E. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, Massachusetts, 1972.

[118] E. I. Organick. 'Computer System Organization'. In: (May 1973). URL: `http://bitsavers.org/pdf/burroughs/B5000_5500_5700/Organick_Computer_System_Organization_The_B5700_B6700_Series_1973.pdf`.

[119] 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*. IEEE Computer Society Press, 1981, pp. 443–457.

[120] 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)*. June 1979, p. 355. DOI: `10.1109/AFIPS.1979.128`.

[121] N. Provos, M. Friedl and P. Honeyman. 'Preventing Privilege Escalation'. In: *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, 2003.

[122] R. Rashid and G. Robertson. 'Accent: A Communications Oriented Network Operating System Kernel'. In: *Proceedings of the Eighth ACM Symposium on Operating System Principles*. Dec. 1981, pp. 64–75.

[123] D. Redell. 'Naming and Protection in Extendible Operating Systems'. PhD thesis. University of California at Berkeley, 1974.

[124] 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*. ACM, 2009, pp. 219–232. DOI: `10.1145/1519065.1519090`.

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

[126] 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*. ACM, 1997, pp. 40–53. DOI: `10.1145/263699.263707`.

[127] A. Richardson. *Complete spatial safety for C and C++ using CHERI capabilities*. Tech. rep. UCAM-CL-TR-949. University of Cambridge, Computer Laboratory, June 2020. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-949.pdf`.

[128] D. M. Ritchie and K. Thompson. 'The UNIX time-sharing system'. In: *Communications of the ACM* 17.7 (1974), pp. 365–375. DOI: `10.1145/361011.361061`.

[129] Ruby Users Group. *Ruby Programming Language*. `http://www.ruby-lang.org/`. Oct. 2010.

[130] P. D. Rugg. *Efficient spatial and temporal safety for microcontrollers and application-class processors*. Tech. rep. UCAM-CL-TR-984. University of Cambridge, Computer Laboratory, July 2023. DOI: `10.48456/tr-984`. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-984.pdf`.

[131] J. Rushby. 'The Design and Verification of Secure Systems'. In: *Proceedings of the Eighth ACM Symposium on Operating System Principles*. Dec. 1981, pp. 12–21. URL: `http://www.csl.sri.com/~%7B%7Drushby/abstracts/sosp81`.

[132] J. Saltzer. ‘Protection and the Control of Information Sharing in Multics’. In: *Communications of the ACM* 17.7 (July 1974), pp. 388–402.

[133] J. H. Saltzer and M. D. Schroeder. ‘The Protection of Information in Computer Systems’. In: *Proceedings of the IEEE* 63.9 (Sept. 1975), pp. 1278–1308. DOI: `10.1109/PROC.1975.9939`. URL: `http://cap-lore.com/CapTheory/ProtInf/`.

[134] W. Schiller. *The Design and Specification of a Security Kernel for the PDP-11/45*. Tech. rep. MTR-2934. Mitre Corporation, Mar. 1975.

[135] M. D. Schroeder. ‘Engineering a security kernel for Multics’. In: *SOSP ’75: Proceedings of the Fifth ACM Symposium on Operating Systems Principles*. ACM, 1975, pp. 25–32. DOI: `10.1145/800213.806518`.

[136] E. J. Sebes. ‘Overview of the architecture of Distributed Trusted Mach’. In: *Proceedings of the USENIX Mach Symposium*. USENIX Association, Nov. 1991, pp. 20–22.

[137] 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*. 2004, pp. 1–19. URL: `http://www.cs.jhu.edu/~swaroop/osverify-2004.pdf`.

[138] 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. URL: `http://portal.acm.org/citation.cfm?id=319151.319163`.

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

[140] D. Skillicorn. ‘Parallel Implementation of Tree Skeletons’. In: *Journal of Parallel and Distributed Computing* 39.2 (1996), pp. 115–125. DOI: `10.1006/jpdc.1996.0160`. URL: `http://www.sciencedirect.com/science/article/pii/S0743731596901604`.

[141] L. Skorstengaard, D. Devriese and L. Birkedal. ‘StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities’. In: *Proceedings of the ACM on Programming Languages* 3 (POPL Jan. 2019), 19:1–19:28. DOI: `10.1145/3290332`.

[142] 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*. USENIX Association, Aug. 1999, pp. 123–139.

[143] 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*. ACM, 1993, pp. 203–216. DOI: `10.1145/168619.168635`.

[144] B. J. Walker, R. A. Kemmerer and G. J. Popek. ‘Specification and verification of the UCLA Unix security kernel’. In: *Communications of the ACM* 23.2 (1980), pp. 118–131. DOI: `10.1145/358818.358825`.

[145] S. T. Walker. 'The advent of trusted computer operating systems'. In: *AFIPS '80: Proceedings of the May 19-22, 1980, national computer conference*. ACM, 1980, pp. 655–665. DOI: 10.1145/1500518.1500626.

[146] 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*. USENIX Association, 2009, pp. 417–432.

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

[148] A. Waterman and K. Asanović, eds. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*. June 2019. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf.

[149] 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'. In: *IEEE Micro* 36.5 (Sept. 2016), pp. 38–49. DOI: 10.1109/MM.2016.84.

[150] R. N. M. Watson. 'A Decade of OS Access-Control Extensibility'. In: *Communications of the ACM* 56.2 (1st Feb. 2013), pp. 52–63. DOI: 10.1145/2408776.2408792.

[151] R. N. M. Watson. *New Approaches to Operating System Security Extensibility*. Tech. rep. Ph.D. Thesis, University of Cambridge, Cambridge, UK, Oct. 2010.

[152] R. N. M. Watson. *New approaches to operating system security extensibility*. Tech. rep. UCAM-CL-TR-818. University of Cambridge, Computer Laboratory, Apr. 2012. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-818.pdf.

[153] R. N. M. Watson, J. Anderson, B. Laurie and K. Kennaway. 'Capsicum: Practical Capabilities for UNIX'. In: *Proceedings of the 19th USENIX Conference on Security*. USENIX Association, 11th Aug. 2010. URL: https://www.cl.cam.ac.uk/research/security/capsicum/papers/2010usenix-security-capsicum-website.pdf.

[154] R. N. M. Watson, G. Barnes, J. Clarke, R. Grisenthwaite, P. Sewell, S. W. Moore and J. Woodruff. *Arm Morello Programme: Architectural security goals and known limitations*. Tech. rep. UCAM-CL-TR-982. University of Cambridge, Computer Laboratory, July 2023. DOI: 10.48456/tr-982. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-982.pdf.

[155] 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*. Tech. rep. UCAM-CL-TR-853. University of Cambridge, Computer Laboratory, June 2014. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-853.pdf.

[156] 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*. Tech. rep. UCAM-CL-TR-869. University of Cambridge, Computer Laboratory, Apr. 2015. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-869.pdf.

[157] 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*. Tech. rep. UCAM-CL-TR-851. University of Cambridge, Computer Laboratory, June 2014. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-851.pdf.

[158] 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*. Tech. rep. UCAM-CL-TR-877. University of Cambridge, Computer Laboratory, Nov. 2015. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-877.pdf.

[159] R. N. M. Watson, S. W. Moore, P. Sewell and P. G. Neumann. *An Introduction to CHERI*. Tech. rep. UCAM-CL-TR-941. University of Cambridge, Computer Laboratory, Sept. 2019. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf.

[160] R. N. M. Watson, P. G. Neumann and S. W. Moore. 'Balancing Disruption and Deployability in the CHERI Instruction-Set Architecture (ISA)'. In: *New Solutions for Cybersecurity*. Ed. by H. Shrobe, D. L. Shrier and A. Pentland. MIT Press/Connection Science, 2018. Chap. 5.

[161] 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*. Tech. rep. UCAM-CL-TR-850. University of Cambridge, Computer Laboratory, June 2014. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-850.pdf.

[162] 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*. Tech. rep. UCAM-CL-TR-864. University of Cambridge, Computer Laboratory, Dec. 2014. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-864.pdf.

[163] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, F. A. Fuchs, R. Grisenthwaite, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son and H. Xia. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)*. Tech. rep. UCAM-CL-TR-987. University of Cambridge, Computer Laboratory, Sept. 2023. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.pdf.

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

[165] 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)*. Tech. rep. UCAM-CL-TR-927. University of Cambridge, Computer Laboratory, Oct. 2018. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf.

[166] 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)*. Tech. rep. UCAM-CL-TR-907. University of Cambridge, Computer Laboratory, Apr. 2017. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-907.pdf.

[167] 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*. Tech. rep. UCAM-CL-TR-876. University of Cambridge, Computer Laboratory, Nov. 2015. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-876.pdf.

[168] 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)*. Tech. rep. UCAM-CL-TR-891. University of Cambridge, Computer Laboratory, June 2016. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-891.pdf.

[169] R. N. M. Watson, A. Richardson, B. Davis, J. Baldwin, D. Chisnall, J. Clarke, N. Filardo, S. W. Moore, E. Napierala, P. Sewell and P. G. Neumann. *CHERI C/C++ Programming Guide*. Tech. rep. UCAM-CL-TR-947. University of Cambridge, Computer Laboratory, June 2020. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf.

[170] 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*. Tech. rep. UCAM-CL-TR-852. University of Cambridge, Computer Laboratory, June 2014. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-852.pdf.

[171] 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*. Tech. rep. UCAM-CL-TR-868. University of Cambridge, Computer Laboratory, Apr. 2015. URL: `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-868.pdf`.

[172] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. 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 2015 IEEE Symposium on Security and Privacy*. 2015 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, May 2015, pp. 20–37. DOI: `10.1109/SP.2015.9`. URL: `https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201505-oakland2015-cheri-compartmentalization.pdf`.

[173] 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*. Tech. rep. UCAM-CL-TR-916. University of Cambridge, Computer Laboratory, Feb. 2018. URL: `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-916.pdf`.

[174] 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)*. Mar. 2012.

[175] 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'. In: (Aug. 2018).

[176] M. Wilkes and R. Needham. *The Cambridge CAP Computer and Its Operating System*. Elsevier North Holland, New York, 1979.

[177] A. Wilkinson et al. 'A Penetration Study of a Burroughs Large System'. In: *ACM Operating Systems Review* 15.1 (Jan. 1981), pp. 14–25.

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

[179] 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: *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA). IEEE Press, June 2014, pp. 457–468. DOI: `10.1109/ISCA.2014.6853201`. URL: `https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201406-isca2014-cheri.pdf`.

[180] J. D. Woodruff. *CHERI: A RISC capability machine for practical memory safety*. Tech. rep. UCAM-CL-TR-858. University of Cambridge, Computer Laboratory, July 2014. URL: `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-858.pdf`.

[181] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson and F. Pollack. 'HYDRA: the kernel of a multiprocessor operating system'. In: *Communications of the ACM* 17.6 (1974), pp. 337–345. DOI: `10.1145/355616.364017`.

[182] W. Wulf, R. Levin and S. Harbison. *Hydra/C.mmp: An Experimental Computer System*. McGraw-Hill, New York, 1981.

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

[184] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. M. Watson and T. M. Jones. 'CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety'. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA). ACM, 12th Oct. 2019, pp. 545–557. DOI: `10.1145/3352460.3358288`. URL: `https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201910micro-cheri-temporal-safety.pdf`.

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

[186] 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*. IEEE Computer Society, 2009, pp. 79–93. DOI: `10.1109/SP.2009.25`. URL: `http://portal.acm.org/citation.cfm?id=1607723.1608126`.

[187] 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*. IEEE Computer Society, 2009, pp. 79–93. DOI: `10.1109/SP.2009.25`.

[188] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas and C. W. Fletcher. 'Speculative Taint Tracking (STT) A Comprehensive Protection for Speculatively Accessed Data'. In: *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (IEEE MICRO 2019)* (Columbus, Ohio, USA). Oct. 2019.

[189] 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*. USENIX Association, 2006, pp. 263–278. ISBN: 1-931971-47-1. URL: `http://portal.acm.org/citation.cfm?id=1298455.1298481`.