Technical Report

UCAM-CL-TR-891
ISSN 1476-2986

Number 891

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 5)

Robert N. M. Watson, Peter G. Neumann,
Jonathan Woodruff, Michael Roe,
Jonathan Anderson, David Chisnall,
Brooks Davis, Alexandre Joannou, Ben Laurie,
Simon W. Moore, Steven J. Murdoch,
Robert Norton, Stacey Son, Hongyan Xia

June 2016

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

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

Approved for public release; distribution is unlimited.
Sponsored by the Defense Advanced Research Projects
Agency (DARPA) and the Air Force Research Laboratory
(AFRL), under contracts FA8750-10-C-0237 ("CTSRD")
and FA8750-11-C-0249 ("MRC2") as part of the DARPA
CRASH and DARPA MRC research programs. The views,
opinions, and/or findings contained in this report are those of
the authors and should not be interpreted as representing the
official views or policies, either expressed or implied, of the
Department of Defense or the U.S. Government. Additional
support was received from St John's College Cambridge, the
Google SOAAP Focused Research Award, 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 Isaac Newton Trust, the UK Higher
Education Innovation Fund (HEIF), and Thales E-Security.

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

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

ISSN 1476-2986

Abstract

This technical report describes CHERI ISAv5, the fifth version of the Capability Hardware Enhanced RISC Instructions (CHERI) Instruction-Set Architecture (ISA)[1] being developed by SRI International and the University of Cambridge. This design captures six years of research, development, experimentation, refinement, formal analysis, and testing, and is a substantial enhancement to the ISA versions described in earlier technical reports. This version introduces the CHERI-128 "compressed" capability format, adds further capability instructions to improve code efficiency, and rationalizes a number of ISA design choices (such as system permissions) as we have come to better understand mappings from C programming-language and MMU-based operating-system models into CHERI. It also contains improvements to descriptions, explanations, and rationale.

The CHERI instruction set is a *hybrid capability-system architecture* that adds new capability-system primitives to a commodity 64-bit RISC ISA enabling software to efficiently implement *fine-grained memory protection* and a hardware-software *object-capability security model*. These extensions support incrementally adoptable, high-performance, formally based, programmer-friendly underpinnings for fine-grained software decomposition and compartmentalization, motivated by and capable of enforcing the principle of least privilege. Fine-grained memory protection provides direct mitigation of many widely deployed exploit techniques.

The CHERI system architecture purposefully addresses known performance and robustness gaps in commodity ISAs that hinder the adoption of more secure programming models centered around the principle of least privilege. To this end, CHERI blends traditional paged virtual memory with a per-address-space capability model that includes capability registers, capability instructions, and tagged memory that have been added to the 64-bit MIPS ISA. CHERI learns from the C-language fat-pointer literature: its capabilities describe fine-grained regions of memory and can be substituted for data or code pointers in generated code, protecting data and also providing Control-Flow Integrity (CFI). Strong monotonicity properties allow the CHERI capability model to express a variety of protection properties, from valid C-language pointer provenance and bounds checking to implementing the isolation and controlled communication structures required for higher-level models such as software compartmentalization.

CHERI's hybrid system approach, inspired by the Capsicum security model, allows incremental adoption of capability-oriented software design: software implementations that are more robust and resilient can be deployed where they are most needed, while leaving less critical software largely unmodified, but nevertheless suitably constrained to be incapable of having adverse effects. For example, we are focusing conversion efforts on low-level TCB components of the system: separation kernels, hypervisors, operating-system kernels, language runtimes, and userspace TCBs such as web browsers. Likewise, we see early-use scenarios (such as data compression, protocol parsing, image processing, and video processing) that relate to particularly high-risk software libraries, which are concentrations of both complex and historically vulnerability-prone code combined with untrustworthy data sources, while leaving containing applications unchanged.

[1] We have attempted to avoid confusion among three rather different uses of the word 'architecture'. The ISA specifies the interface between hardware and software, rather than describing either the (micro-)architecture of a particular hardware prototype, or laying out the total-system hardware-software architecture. We expect further documentation on the latter to emerge in the final year of the project.

Acknowledgments

The authors of this report thank other members of the CTSRD team, 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:

Ross J. Anderson	Graeme Barnes	Hadrien Barral	Stuart Biles
Matthias Boettcher	David Brazdil	Ruslan Bukin	Gregory Chadwick
Chris Dalton	Nirav Dave	Lawrence Esswood	Paul J. Fox
Richard Grisenthwaite	Khilan Gudka	Jong Hun Han	Alex Horsman
Asif Khan	Myron King	Chris Kitching	Wojciech Koszek
Patrick Lincoln	Anil Madhavapeddy	Ilias Marinos	A. Theodore Markettos
Alfredo Mazzinghi	Ed Maste	Dejan Milojic	Andrew W. Moore
Will Morland	Alan Mujumdar	Prashanth Mundkur	Philip Paeps
Alex Richardson	Colin Rothwell	John Rushby	Hassen Saidi
Hans Petter Selasky	Muhammad Shahbaz	Lee Smith	Andrew Turner
Richard Uhler	Munraj Vadera	Philip Withnall	Bjoern A. Zeeb

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

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

We would also like to acknowledge the late David Wheeler and Paul Karger, whose conversations with the authors about the CAP computer and capability systems contributed to our thinking on CHERI.

Finally, we are grateful to Howie Shrobe, MIT professor and past DARPA CRASH and MRC program manager, who has offered both technical insight and support throughout this work. We are also grateful to Robert Laddaga and Stu Wagner, who succeeded Howie in overseeing the CRASH program, to John Launchbury, DARPA I2O office director, and to Daniel Adams and Laurisa Goergen, SETAs supporting the CRASH and MRC programs.

Contents

Chapter 1

Introduction

CHERI (Capability Hardware Enhanced RISC Instructions) extends commodity RISC Instruction-Set Architectures (ISAs) with new capability-based primitives that improve software robustness to security vulnerabilities. The CHERI model is motivated by the *principle of least privilege*, which argues that greater security can be obtained by minimizing the privileges accessible to running software. 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 levels of abstraction. CHERI supports *architectural least privilege* through in-address-space *memory capabilities*, which replace integer virtual-address representations of code and data pointers to allow fine-grained protection of in-memory data, and also control-flow integrity. This application of least privilege provides strong protection against a broad range of memory- and pointer-based vulnerabilities and exploit techniques – buffer overflows, format-string attacks, pointer-corruption attacks, and so on. At a higher level of abstraction, CHERI also supports *application-level least privilege* through the robust and efficient implementation of highly scaleable in-address-space *software compartmentalization* using *object capabilities*. This application of least privilege provides strong mitigation of application-level vulnerabilities, such as logical errors, downloaded malicious code, or software Trojans inserted in the software supply chain.

CHERI is designed to support incremental adoption within current security-critical, C-language *Trusted Computing Bases (TCBs)*: operating-system (OS) kernels, key system libraries and services, language runtimes supporting higher-level type-safe languages, and applications such as web browsers and office suites. While CHERI builds on many historic ideas about capability systems (see Chapter 2), it is also a *hybrid capability-system architecture*, meaning that it also draws ideas from contemporary RISC ISAs, and is intended to compose well with current processor architectures and software designs. We hope that this will support gradual deployment of CHERI features in existing software, rather than obliging a clean-slate software design, offering a more gentle hardware-software adoption path.

CHERI has four 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:

1. *Fine-grained memory protection* improves software resilience to escalation paths that

allow software bugs to be coerced into more powerful software vulnerabilities; e.g., through remote code injection via buffer overflows and other memory-based techniques. Unlike MMU-based, CHERI's memory protection is intended to be driven by the compiler in protecting programmer-described data structures and references, rather than via coarse page-granularity protections. CHERI capabilities limit how pointers can be used by scoping the ranges of memory (via bounds) and operations that can be performed (via permissions). They also protect the integrity, provenance, and monotonicity of pointers in order to prevent inadvertent or inappropriate manipulation from leading to privilege escalation.

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

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

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

4. Finally, 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 will help ensure that we avoid creating a system that cannot meet our functional and security requirements. Formal methods can also help to avoid many of the characteristic design flaws that are common in both hardware and software. This desire requires us not only to perform

research into CPU and software design, but also to develop new formal methodologies and adaptations and extensions of existing ones.

We are concerned with 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 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.

Our selection of RISC as a foundation for the CHERI capability extensions is motivated by two factors. First, simple instruction set architectures are easier to reason about, extend, and implement. Second, RISC architectures (such as ARM and MIPS) are widely used in network embedded and mobile device systems such as firewalls, routers, smart phones, and tablets – markets with the perceived flexibility to adopt new CPU facilities, and also an immediate and pressing need for improved security. CHERI's new security primitives would also be useful in workstation and server environments, which face similar security challenges.

In its current incarnation, we have prototyped CHERI as an extension to the 64-bit MIPS ISA. However, our approach – and more generally the CHERI protection model – is intended to easily support other similar ISAs, such as 64-bit ARM and 64-bit RISC-V. The design principles would also apply to other non-RISC ISAs, such as 32-bit and 64-bit Intel and AMD, but require significantly more adaptation work, as well as careful consideration of the implications of the diverse set of CPU features found in more CISC-like architectures. We also consider the possibility that the syntax and semantics of the CHERI model might be implemented over conventional CPUs with the help of the compiler, static checking, and dynamic enforcement approaches found in software fault isolation techniques [92] or Google Native Client (NaCl) [117]. We do, however, hypothesize that some of the efficiencies and safety features found in the hardware implementation would be difficult to accomplish in a software transformation: for example, automatic and atomic clearing of tags for in-memory pointers during arbitrary memory writes might come at substantial expense in software, but are "free" in hardware.

Exploring, and iterating over, a substantial instruction-set design space has been considerably eased by our use of the Bluespec SystemVerilog [10] (BSV) Hardware Description Language (the BSV HDL) in prototyping. BSV has allowed rapid redesigns as our understanding of architectural, microarchitectural, and software requirements evolved – resulting from its use of modular abstractions, encapsulation, and hierarchicalization.

In the remainder of this chapter, we consider the motivations behind the CHERI design, introduce that design at a high level, explore the idea of a hybrid capability system and why it eases deployment, describe the CHERI threat model, summarize our efforts to use formal methodology to support our design and engineering process, describe our hardware and software prototypes, lay out our other publications containing more detailed evaluations of our approach, describe the changes appearing in this version of the CHERI ISA specification, and provide a guide to the remainder of this report. Those eager to move quickly to a high-level description of the CHERI protection model may wish to turn directly to Section 1.2 and then Chapter 3, with its mapping into the 64-bit MIPS ISA in Chapter 4. The Glossary at the end of the report contains stand-alone definitions of many key ideas and terms.

1.1 Motivation

The CHERI ISA provides a sound and formally based architectural foundation for the principled development of highly trustworthy systems. The CHERI instruction-set architecture and system architecture build on and extend 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.

- 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 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 [59] and LLVM [51].

- 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 the desktop, 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).

- Likewise, 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

[1] Levy's *Capability-Based Computer Systems* [53] provides a detailed history of segment- and capability-based designs through the early 1990s [53]. However, it leaves off just as the transition to microkernel-based capability systems such as Mach [2], L4 [54], and, later, SeL4 [45], as well as capability-influenced virtual machines such as the Java Virtual Machine [28], begins. Chapter 2 discuss historical influences on our work in greater detail.

FreeBSD, Mach [2], 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*, such as Capsicum [99], 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.

1.1.1 C-Language Trusted Computing Bases (TCBs)

Contemporary client-server and cloud computing are premised 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. This technique 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.

1.1.2 The Compartmentalization Problem

The *compartmentalization problem* arises from attempts to decompose security-critical applications 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 applications such as OpenSSH [78] and Chromium [80], and more recently in our own Capsicum project [99], 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 2.

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 applications 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 application 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.

1.2 The CHERI ISA Design

The CHERI ISA embodies two fundamental and closely linked technical goals to address vulnerability mitigation: first, fine-grained capability-oriented memory protection within address spaces, and second, primitives to support both scalable and programmer-friendly compartmentalization within address spaces based on the object-capability model. 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. Simultaneously, it will provide reasonable assurance of correctness and a realistic technology transition path from existing hardware and software platforms.

To this end, we have prototyped CHERI as an Instruction-Set Architecture (ISA) extension to the widely used 64-bit MIPS ISA; we are also considering the potential applicability of the

[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 [9].

underlying protection model to other RISC ISAs, such as ARM and RISC-V. CHERI adds the following features to the 64-bit MIPS ISA[3] to support granular memory protection and compartmentalization within address spaces:

- A set of *capability registers* describe the rights (*protection domain*) of the executing thread to memory that it can access, and to object references that can be invoked to transition between protection domains. We model these registers as a separate *capability register file*, supplementing the general-purpose file. Over time, we imagine that capability registers will displace general-purpose registers as the preferred means to describe data and object references. Capability registers contain a tag, sealed bit, permission mask, base, length, and offset (allowing the description of not just bounded regions, but also a pointer into that region, improving C-language compatibility). Capability registers are suitable for describing both data and code, and can hence protect both data integrity/confidentiality and control flow. Certain registers are reserved for use in exception handling; all others are available to be managed by the compiler using the same techniques used with conventional registers.

  Another potential integration into the ISA (which would maintain the same CHERI protection semantics) would be to extend the existing general-purpose registers so that they could also hold capabilities. This might reduce the hardware resources required to implement CHERI support. However, we selected our current approach for maintaining consistency with the MIPS ISA extension model (in which coprocessors have independent register files), and minimizing ABI disruption on boundaries between legacy and CHERI-aware code. We explore the potential space of mappings from the CHERI model into the ISA in greater detail in Section 4.6.

- New *capability instructions* allow executing code to create, constrain (e.g., by increasing the base, decreasing the length, or reducing 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 their fields are given additional integrity protections to provide encapsulation. Capability instructions implement *guarded manipulation*: invalid capability manipulations (e.g., to increase rights or length) and invalid capability dereferences (e.g., to access outside of a bounds-checked region) result in an exception that can be handled by the supervisor or language runtime. A key aspect of the instruction-set design is *intentional use of capabilities*: explicit capability registers, rather than ambient authority, are used to indicate exactly which rights should be exercised, to limit the damage that can be caused by exploiting bugs. Most capability instructions are part of the usermode ISA, rather than 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

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

memory without loss of integrity. 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. 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.

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 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.

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

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

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

files might permit existing instructions to be reused (with care) across integer and capability operations.

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

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

1.3 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.

1.4 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.

1.5 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.

1.6 Formal Methodology

Throughout this project, we apply formal methodology to help avoid system vulnerabilities. An important early observation is that existing formal methodology applied to software security has significant problems with multi-address-space security models; formal approaches have relied on the usefulness of addresses (pointers) as unique names for objects. Whereas this weakness in formal methods is a significant problem for traditional CPU designs, which offer security primarily through rings and address-space translation, CHERI's capability model is scoped within address spaces. This offers the possibility of applying existing software proof methodology in the context of hardware isolation (and other related properties) in a manner that was previously infeasible. We are more concretely (and judiciously) applying formal methodology in five areas:

1. We have developed a formal semantics for the CHERI 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 have also developed a more complete ISA model incorporating both MIPS and CHERI instructions using Cambridge's L3 instruction-set description language. Although we have not yet used this for automated theorem proving, we increasingly use the L3 description as a "gold model" of the instruction set 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 model to identify a number of bugs in multiple hardware implementations of CHERI, as well as to discover software dependences on undefined instruction-set behavior.

3. We have developed extensions to the BSV compiler to export an HDL description to SRI's PVS and SAL model checker. We have 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. These tools will allow us to verify low-level properties of the hardware design and use the power of model

checking and satisfiability solvers to analyze related properties. Ideally they will also help link ISA-level specifications with the CPU implementation.

4. We have proven a number of properties about our "compressed" 128-bit capability implementation to ensure that the protection and security properties present in the 256-bit reference semantics (e.g., capability monotonicity) hold of the compressed version – and that the compression and decompression algorithms are correct.

5. 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).

A detailed description of formal methods efforts relating to CHERI may be found in the emerging draft *CHERI Formal Methods Report* [74].

1.7 Hardware and Software Prototypes

As a central part of this research, we have developed reference prototypes of the CHERI ISA via several CHERI processor designs. These prototypes allow us to explore, validate, evaluate, and demonstrate the CHERI approach through realistic hardware properties and real-world software stacks. A detailed description of the current prototypes, both from architectural and practical use perspectives, may be found in our companion papers and technical reports, described in Section 1.8.

Our first prototype (CHERI1) is based on Cambridge's MAMBA research processor, and is a single-threaded, multi-core implementation intended to allow us to explore ISA design trade-offs with moderate microarchitectural realism. This prototype is implemented in the BSV HDL, a high-level functional programming language for hardware design. CHERI1 is a pipelined baseline processor implementing the 64-bit MIPS ISA, and incorporates an initial prototype of the CHERI capability coprocessor that includes capability registers and a basic capability instruction set.

Using the BSV hardware specification language and its Bluespec SystemVerilog, we are able to run the CPU in simulation, and synthesize the CHERI design to execute in field-programmable gate arrays (FPGAs). In our development work, we are targeting an Altera FPGAs on Terasic development boards. However, in our companion MRC2 project we have also targeted CHERI at the second-generation NetFPGA 10G and SUME research and teaching boards, which we hope to use in ongoing research into datacenter network fabrics. That work includes the development of Blueswitch, a BSV language implementation of an OpenFlow switch that can operate as a tightly coupled CHERI coprocessor. In the future, should it become desirable, we will be able to construct an ASIC design from the same BSV specification. We have released the CHERI soft core as *open-source hardware*, making it available for more widespread use in research. This should allow others, especially in the research community, to reproduce and extend our results.

We have also developed a second prototype (CHERI2), which is compatible with CHERI1 but has additional CPU features including fine-grained multi-threading. We have used this as a platform for early exploration of the synergy between compartmentalization and parallelism in multi-threaded processor designs. CHERI2 also employs a more stylized form of the BSV

language that is intended to considerably enhance our formal analysis of the hardware architecture.

In addition to the CHERI1 and CHERI2 implementations in BSV, we have implemented an executable model of CHERI in the L3 ISA modeling language [26], and a high-performance emulation in QEMU. The L3 and QEMU implementations support 256-bit capabilities and multiple forms of 128-bit capabilities including compressed capabilities and "magic" uncompressed capabilities, which are identical to 256-bit capabilities except for size. While intended primarily for formal modeling and use as a test oracle, we have also found the L3 ISA modeling language invaluable in practical design-space exploration.

As the CHERI security model is necessarily a hardware-software model, we have also performed substantial experimentation with software stacks targeting the CHERI ISA. We have created an adaptation of the commodity open-source FreeBSD operating system, CheriBSD, that supports a wide variety of peripherals on the Terasic tPad and DE4 FPGA development boards; we use these boards in both mobile tablet-style and network configurations. CheriBSD is able to manage the capability coprocessor, maintain additional thread state for capability-aware user applications, expose both hybrid and pure-capability system-call interfaces, and, increasingly, to use capability features for self protection against malicious userspace software. CheriBSD also implements exception-handler support for object-capability invocation, signal delivery when protection faults occur (allowing language runtimes to catch and handle protection violations), and error recovery for in-process sandboxes. We have adapted the Clang and LLVM compiler suite to allow language-level annotations in C to direct capability use in a hybrid ABI. Additionally, we have implemented a pure-capability compilation mode where all C pointers are capabilities. Using a mix of hybrid and pure-capability ABIs, we have developed a number of capability-enhanced applications to demonstrate fine-grained memory protection and in-process compartmentalization – to explore security, performance, and programmability tradeoffs.

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* [113]. 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* [13], 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* [110], 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* [35], 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* [60], 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.

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* [104, 105, 106, 107], describes the CHERI instruction set, both as a high-level, software-facing model and the specific mapping into the 64-bit MIPS instruction set. Successive versions have introduced improved C-language support, support for scalable compartmentalization, and compressed capabilities.

- The *Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide* [103] 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* [101].

- The *Bluespec Extensible RISC Implementation: BERI Hardware Reference* [108, 109] describes hardware aspects of our prototyping platform, including physical platform and practical user concerns.

- The *Bluespec Extensible RISC Implementation: BERI Software Reference* [100, 102] 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)* [34], 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 following technical reports are for PhD dissertations 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 Capsicum, which motivated our work on CHERI [96, 97].

- Jonathan Woodruff's PhD dissertation, *CHERI: A RISC capability machine for practical memory safety*, describes our CHERI1 prototype implementation [114].

- Robert Norton's PhD dissertation, *Hardware support for compartmentalisation*, describes how hardware support is provided for optimized domain transition using the CHERI2 prototype implementation [75].

As our research proceeded, and prior to our conference and journal articles, we published a number of workshop papers to lay 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) [73], shortly after the inception of the project.

- Mid-way through creation of both the BERI prototyping platform, and CHERI ISA model, we published *CHERI: A Research Platform Deconflating Hardware Virtualization and Protection* at the Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012) [111].

- 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* [73].

Further research publications and technical reports will be forthcoming.

1.9 CHERI ISA Version History

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

CHERI ISAv1 - 1.0–1.4 - 2010-2012 Early versions of the CHERI ISA explored the integration of capability registers and tagged memory – first in isolation from, and later in composition with, MMU-based virtual memory. CHERI 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 on only bounds, and not the pointer itself. Pointers are allowed to move outside of their defined bounds, but can only be dereferenced 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 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.

1.10 Changes in CHERI ISA 1.18

- This version of the CHERI ISA, *CHERI ISAv5*, has been prepared for publication as University of Cambridge technical report UCAM-CL-TR-891.

- 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 de-emphasized, as, while it refers correctly to CHERI's use of the MIPS coprocessor opcode extension space, some readers found it suggestive of an independent hardware unit rather than reflecting CHERI's tight integration into the main processor pipeline and memory subsystem.
- A reference has been added to Robert Norton's PhD dissertation on optimized CHERI domain switching.
- A reference has been added to our PLDI 2016 paper on C-language semantics and their interaction with the CHERI model.
- The object-type field in both 128-bit and 256-bit capabilities is now 24 bits, with Top and Bottom fields reduced to 8 bits for sealed capabilities. This reflects a survey of current object-oriented software systems, suggesting that 24 bits is a more reasonable upper bound than 20 bits.
- The assembly arguments to `CJALR` have been swapped for greater consistency with jump-and-link register instructions in the MIPS ISA.
- We have reduced the number of privileged permissions in the 256-bit capability model to a single privileged permission, Access_System_Registers, to match 128-bit CHERI. This is a binary-incompatible change.
- We have improved the description of the CHERI-128 model in a number of ways, including a new section on the CHERI-128 representable bounds check.
- The architecture chapter contains a more detailed discussion of potential ways to reduce the overhead of CHERI by reducing the number of capability registers, converging the general-purpose 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.

1.11 Document Structure

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

Chapter 1 introduces the CHERI instruction-set architecture, its motivations, goals, context, philosophy, and design.

Chapter 2 describes the historical context for this work, including past systems that have influenced our approach.

Chapter 3 describes the high-level model for the CHERI approach in terms of ISA features, software protection objectives, and software mechanism.

Chapter 4 provides a detailed description of the CHERI architecture, including its register and memory capability models, new instructions, procedure capabilities, and use of message-passing primitives.

Chapter 5 describes the CHERI capability coprocessor, its register file, tagged memory, and other ISA-related semantics with respect to a mapping into the 64-bit MIPS ISA.

Chapter 6 provides a detailed description of each new CHERI instruction, its pseudo-operations, and how compilers should handle floating-point loads and stores via capabilities.

Chapter 7 explores the composition of CHERI's ISA protection features and their impact on software protection models.

Chapter 8 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. A broader document on our use of formal methods relating to the CHERI total-system architecture is in draft form [74].

Chapter 9 briefly describes how we have used formal methodology to ensure correctness of aspects of the CHERI ISA.

Chapter 10 discusses our short- and long-term plans for the CHERI architecture, 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 ISA.

Appendix B is a quick reference for CHERI instructions and encodings, both current and proposed.

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 instruction-set architecture and its impact on software, as well as evaluation strategies and results. Additional information on our CHERI hardware and software implementations, as well as formal methods work, can be found in accompanying reports.

Chapter 2

Historical Context

As with many aspects of contemporary computer and operating system design, the origins of operating system security may be found at the world's leading research universities, but 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) [15], and continued over the next decade with MIT's Multics project, described many central tenets of computer security [16, 33]. Dennis and Van Horn's 1965 *Programming Semantics for Multiprogrammed Computations* [19] 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 [85].

In 1975, Saltzer and Schroeder surveyed the rapidly expanding vocabulary of computer security in *The Protection of Information in Computer Systems* [86]. 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* [48], which considered ways in which hardware memory protection addressed accidental and intentional types of failure: 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 [52], 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., [70, 71], among others).

The Security Research community also blossomed outside of MIT: Wulf's HYDRA operating system at Carnegie Mellon University (CMU) [115, 14], Needham and Wilkes' CAP Computer at Cambridge [112], SRI's Provably Secure Operating System (PSOS) [25, 71] hardware-software co-design that included strongly typed object capabilities, Rushby's security kernels supported by formal methods at Newcastle [84], 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 [49, 50]. 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 [82].

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 [42]. 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 [8]. 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 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 [64] 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.

2.1 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 [49, 50].

Systems such as the CAP Computer at Cambridge [112] and Ackerman's DEC PDP-1 architecture at MIT [3] 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 [19]. Levy provides a detailed exploration of segment- and capability-oriented computer system design through the mid-1980s in *Capability-Based Computer Systems* [53].

2.2 Microkernels

Denning has argued that the failures of capability hardware projects were classic failures of large systems projects, an underestimation of the complexity and cost of reworking an entire system design, rather than fundamental failures of the capability model [18]. 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) [77].

However, with a transition from complex instruction set computers (CISC) to reduced instruction set computers (RISC), and a shift away from microcode toward operating system implementation of complex CPU functionality, the attention of security researchers turned to microkernels.

Carnegie Mellon's HYDRA [14, 116] embodied this approach, in which microkernel message passing between separate tasks stood in for hardware-assisted security domain crossings at capability invocation. HYDRA developed a number of ideas, including the relationship between capabilities and object references, refined the *object-capability* paradigm, and further pursued the separation of policy and mechanism.[1] Jones and Wulf argue through the HYDRA design that the capability model allows the representation of a broad range of system policies as a result of integration with the OS object model, which in turn facilitates interposition as a means of imposing policies on object access [40].

Successors to HYDRA at CMU include Accent and Mach [79, 2], both microkernel systems intended to explore the decomposition of a large and decidedly un-robust operating system kernel. In microkernel designs, traditional OS services, such as the file system, are migrated out of ring 0 and into user processes, improving debuggability and independence of failure modes. They are also based on mapping of capabilities as object references into IPC pipes (*ports*), in which messages on ports represent methods on objects. This shift in operating system design went hand in hand with a related analysis in the security community: Lampson's model for capability security was, in fact, based on pure message passing between isolated processes [50]. This further aligned with proposals by Andrews [5] and Rushby [84] 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 [24].

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 [55].

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 pere-

[1] Miller has expanded on the object-capability philosophy in considerable depth in his 2006 PhD dissertation, *Robust composition: towards a unified approach to access control and concurrency control* [62]

grinations 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 [87] and Neumann's Provably Secure Operating System [27] design study were ground-up operating system designs based soundly in formal methodology.[2] In contrast, Schroeder's MLS kernel design for Multics [88], the DoD Kernelized Secure Operating System (KSOS) [58], and Bruce Walker's UCLA UNIX Security Kernel [93] 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 [94].

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 [11]. 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 [89]. 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 [91]. 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 upcalls and message passing (with associated context switches) to the security server. NSA and SCC eventually migrated FLASK to the FLUX microkernel developed by the University of Utah in the search for improved performance. Invigorated by the rise of microkernels and their congruence with security kernels, this flurry of operating system security research 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 [65] – effectively, a composition of mandatory access control and capabilities in service to application security. This approach is embodied by Efstathopoulos et al.'s Asbestos [22] and Zeldovich et al.'s Histar [119] 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 [36], Shapiro's EROS [90] (now CapROS) continues the investigation of higher-assurance software capability designs, seL4 [45], 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

[2] PSOS's ground-up design included ground-up hardware, whereas Schiller's design revised only the software stack.

X [6] (which uses Mach interprocess communication (IPC) objects as capabilities) and Cambridge's Capsicum [99] 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 [31, 29]. 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 [62], Wagner's Joe-E capability-safe subset of Java [61], and Miller's Caja capability-safe subset of JavaScript continue a language-level exploration of capability security [63].

2.3 Language and Runtime Approaches

Direct reliance on hardware for enforcement (which is central to both historic and current systems) is not the only approach to isolation enforcement. The notion that limits on expressibility in a programming language can be used to enforce security properties is frequently deployed in contemporary systems to supplement coarse and high-overhead operating-system process models. Two techniques are widely used: virtual-machine instruction sets (or perhaps physical machine instruction subsets) with limited expressibility, and more expressive languages or instruction sets combined with type systems and formal verification techniques.

The Berkeley Packet Filter (BPF) is one of the most frequently cited examples of the virtual machine approach: user processes upload pattern matching programs to the kernel to avoid data copying and context switching when sniffing network packet data [57]. 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 [7], and the DTrace instrumentation tool that, like BPF, uses a narrow virtual instruction set to implement the D language [12].

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 [118]. NaCl is closely related to Software Fault Isolation (SFI) [92], 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 [68]. 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 [39], and type-safe languages such as Ruby [83], 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 [59].

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 [81]; 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 [95]. Web browsers frequently incorporate many of these techniques including Java Virtual Machines and a JavaScript interpreter.

2.4 Bounds Checking and Fat Pointers

In contrast to prior capability systems, a key design goal for CHERI was to support mapping C-language pointers into capabilities. In earlier prototypes, we did this solely through base and bounds fields within capabilities, which worked well but required substantial changes to existing C software that often contained programming idioms that violated monotonic rights decrease for pointers. In later versions of the ISA, we adopt ideas from the C fat-pointer literature, which differentiate the idea of a delegated region from a current pointer: while the base and bounds are subject to guarded manipulation rules, we allow the offset to float within and beyond the delegated region. Only on dereference are protections enforced, allowing a variety of imaginative pointer operations to be supported. Many of these ideas originate with the type-safe C dialect Cyclone [39], and see increasing adaptation to off-the-shelf C programs with work such as Softbound [66], Hardbound [20], and CCured [69]. This flexibility permits a much broader range of common C idiom to be mapped into the capability-based memory-protection model.

2.5 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 [24]), 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 [76, 17]. 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. Multics was implemented in a stark subset 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, with encapsulated modular abstraction, interlayer state mappings, and abstract programs relating each layer to those on which it depended [71, 72]. 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 [98]. 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 [99, 96]. 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.

2.6 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 now is 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 earlier in Chapter 1, a transformed threat environment deriving from ubiquitous computing and networking, and the practical reality of widespread exploitation of software vulnerabilities, provides a strong motivation to investigate improved processor foundations for software security. This change in environment has coincided with improved hardware prototyping techniques and higher-level hardware definition languages that facilitate academic hardware-software system research at larger scales without which 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, and 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 3

The CHERI Protection Model

This chapter gives a high-level description of the protection model provided by the CHERI architecture, its use in software, and its impact on potential vulnerabilities. There are many potential concrete mappings of this protection model into an Instruction-Set Architecture (ISA), and so this chapter focuses on the structure and software-visible aims of the model, leaving a specific concrete mapping to later chapters. Whether used for memory protection or compartmentalization, CHERI's protection properties should hold with relative uniformity across underlying implementations (e.g., regardless of capability size, whether or not capabilities are stored in their own register file or in the general-purpose register file), and support common (and ideally portable) programming models and approaches.

3.1 CHERI Capabilities: Strong Protection for Pointers

The purpose of the CHERI ISA extensions is to provide strong protection for pointers within virtual address spaces, complementing existing virtual memory provided by Memory Management Units (MMUs). The rationale for this 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 and fine-grained bounds checking for pointers can be used to construct efficient *isolation* and *controlled communication*, foundations on which we can build scalable and programmer-friendly compartmentalization within address spaces.

This facilitates deploying fine-grained application sandboxing with greater ubiquity, in turn mitigating a broad range of logical programming errors higher in the software stack, as well as resisting future undiscovered vulnerability classes and exploit techniques.

Virtual memory also fails to address these problems, as (a) it scales poorly, paying an extremely high performance penalty as the degree of compartmentalization grows; and (b) it offers extremely poor programmability as the medium for sharing is the virtual-memory page rather than the pointer-based programming model used for 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 opaque (software-defined) pointers suitable to to implement higher-level protection models such as separation and efficient compartmentalization. Unlike virtual memory, whose functions are intended to be managed by low-level operating-system components such as kernels, hypervisors, and the C runtime, CHERI capabilities are targeted at compiler use, allowing program structure and dynamic memory allocation to direct their use. We anticipate CHERI protection being used within operating system kernels, both for memory protection and compartmentalization, as well as in userspace libraries and applications. Most source code will 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 virtual-address use.

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

3.2 Architectural Capability Features

In current systems, pointers are integer values that may be represented in two forms architecturally: in integer registers, or in memory. Capabilities are likewise stored in registers and memory, and contain integer values interpreted as virtual addresses; they also contain additional metadata to implement protection properties around pointers, such as bounds. Capabilities are therefore larger than the virtual addresses they protect – typically between $2\times$ (e.g., 128-bit compressed capabilities on a 64-bit architecture) and $4\times$ (e.g., 256-bit uncompressed capabilities on a 64-bit architecture). Most of the capability is stored in addressable memory, as is the case for current integer pointers; however, there is also a 1-bit tag that may be inspected via the instruction set, but is not visible via byte-wise loads and stores. In the remainder of this section, we describe the high-level protection properties that capabilities grant pointers:

- Tags for pointer integrity and provenance

- Bounds to limit the dereferenceable range of a pointer
- Permissions to limit the use of a pointer
- Capability monotonicity and guarded manipulation to prevent privilege escalation
- Sealed capabilities with object types to implement software encapsulation
- Capability control flow to limit pointer propagation
- Capability compression to reduce the in-memory overhead of pointer metadata
- Capability hybridization with C-language pointers
- Capability hybridization with MMU-based virtual memory

3.2.1 Tags for Pointer Integrity and Provenance

Each capability (whether held in a register or stored in memory) has an associated 1-bit tag that consistently tracks pointer validity. In-memory tags are maintained by the memory subsystem as 1 bit for each capability-sized capability-aligned unit of memory (either 128 bits or 256 bits, depending on the ISA variant); they are not directly addressable. Other metadata associated with capabilities (e.g., bounds and permissions) are stored in addressable memory and protected by the corresponding tag bits. Tags follow capabilities into and out of capability registers with corresponding loads and stores.

Capabilities are valid for dereference – for load, store, and instruction fetch – only if the tag is set. Dereferencing an untagged capability (i.e., one without a tag set) will cause a hardware exception. Tagged capabilities can be constructed only by deriving them from existing tagged capabilities, which ensures *pointer provenance*. Attempts to overwrite all or a portion of a capability in memory will automatically (and atomically) clear the tag. For example, this prevents arbitrary data received over the network from ever being directly dereferenced as a pointer.

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

3.2.2 Bounds on Pointers

Capabilities contain lower and upper bounds for each pointer; while the pointer may move out of bounds (and back in again), attempts to dereference an out-of-bounds pointer will throw a hardware exception. This prevents exploitation of buffer overflows on the heap and stack, as well as out-of-bounds execution. Allowing pointers to sometimes be out-of-bounds with respect to their buffers – without faulting – is important for C-language compatibility. The

256-bit capability variant allows pointers to stray arbitrarily out of bounds, but the 128-bit scheme imposes some restrictions (see Section 5.9 for details).

Bounds originate in allocation events. The operating system places bounds on pointers to initial address-space allocations during process startup (e.g., via ELF auxiliary arguments), and on an ongoing basis as new address-space mappings are made available (e.g., via `mmap` system calls). Most bounds originate in userspace allocations, including the heap allocator, and compiler-generated 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).

3.2.3 Permissions on Pointers

Capabilities extend pointers with a permissions mask controlling how a pointer may be used; for example, the compiler may set the permissions so that pointers to data cannot be reused as code pointers, or so that pointers to code cannot be used to store data. Further permissions control the ability to load and store capabilities themselves, allowing the compiler to implement policies such as *code and data pointers cannot be loaded from character strings.* Permissions can also be made accessible to higher-level aspects of the runtime and programmer model, offering dynamic enforcement of concepts similar to **`const`**.[1]

Permissions, as with bounds, are linked to allocation events. Initial permissions for memory mappings will be introduced by the kernel via process startup and as new memory mappings are introduced. The run-time linker will be responsible for creating hierarchies of executable capabilities representing code pointers, from which (implicitly), other forms of runtime addresses will be derived, such as return addresses. Read-only and readable/writable capabilities will originate with the run-time linker, heap allocator, and stack allocator. A separate hierarchy of type capabilities, used in sealing, will likely be provided independently via a system-call interface, to prevent arbitrary code and data capabilities being used for this purpose. Languages may provide further facilities to allow programmer-directed refinement of permissions.

3.2.4 Capability Monotonicity via Guarded Manipulation

A new capability can be derived from another capability only via valid manipulations that may narrow (but never broaden) rights ascribed to the new capability. This property of *capability monotonicity* prevents broadening the bounds on pointers, increasing the rights on pointers, and so on, eliminating many manipulation attacks and inappropriate pointer reuses. This property 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 un-delegated areas of memory.

[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.

3.2.5 Sealed Capabilities

Capability *sealing* allows capabilities to be marked as *immutable* and *non-dereferenceable*, causing hardware exceptions to be thrown if attempts are made to modify or dereference them, or to jump to them. This enables capabilities to be used as unforgeable tokens of authority for higher-level software constructs grounded in *encapsulation*, while still allowing them to fit within the pointer-centric framework offered by CHERI capabilities. Sealed capabilities are the foundation for building the CheriBSD *object-capability model* supporting in-address-space compartmentalization; they 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.

3.2.6 Capability Object Types

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

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

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

3.2.7 Capability Flow Control

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

Capability TLB bits extend the existing load and store permissions on TLB entries (or, in architectures with hardware page-table walkers, page-table entries) with new permissions

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

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

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

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

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

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

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

3.2.8 Capability Compression

The 256-bit in-memory representation of CHERI capabilities provides full accuracy for pointer lower bounds, and upper bounds, as well as a large *object type space* with software-defined permissions. The 128-bit implementation of CHERI uses floating-point-like *fat-pointer compression techniques* that rely on redundancy between the three 64-bit virtual addresses. The compressed representation exchanges stronger alignment requirements (proportional to object size) for a more compact representation. The CHERI-128 compression model (see Section 5.9) maintains the monotonicity inherent in the 256-bit model: no ISA manipulation of a capability can grant increased rights, and when unrepresentable cases are generated (e.g., a pointer substantially out of bounds, or a very unaligned object), the pointer becomes un-dereferenceable. Memory allocators already implement alignment requirements for heap and stack allocations (word, pointer, page, and superpage alignments), and these algorithms require only minor extension to ensure fully accurate bounds for large memory allocations. (Small allocations $< 1MiB$ require no additional alignment.) Relative to a 64-bit pointer, the 128-bit design reduces per-pointer memory overhead (with a strong influence on cache footprint for some software designs) by roughly two thirds, compared to the 256-bit representation.

3.2.9 Hybridization with Integer Pointers

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

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

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

Programs compiled to use capabilities to represent pointers (whether implicitly or via explicit program annotations) will not use the default data capability, instead employing capability registers and capability-based instructions for pointer operations and indirection. The program-counter capability will be used regardless of code model is employed, although capability-aware code generation will employ constrained program-counter bounds and permissions to implement control-flow integrity 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.

3.2.10 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. 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.

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

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

3.3 Software Protection Using CHERI

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

3.3.1 C/C++ Language Support

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

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

Temporal safety CHERI provides instruction-set foundations for higher-level *temporal safety* properties such as non-reuse of heap allocations via garbage collection and revocation, and compiler clearing of return addresses on the stack.

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 the:

Compiler and linker responsible for generating code that manipulates and dereferences code and data pointers, compile-time linkage, and also 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 information flow-control mechanism, and used to enforce higher-level policies such as *stack capabilities cannot be written to global data structures.*

Code-Pointer Protection

Again with support of the compiler and linker, CHERI capabilities can be used to implement forms of *Control-Flow Integrity (CFI)* that prevent code pointers from being misused. This can limit a number of 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.

3.3.2 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 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.

3.3.3 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 (i.e., grounded in address-space randomization) that often yield to determined attackers.

3.3.4 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 RISC 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 RISC 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 calls 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 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.

Entire software systems need not utilize only one code-generation or calling-convention model. For example, a kernel compiled with conventional RISC 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 RISC 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 RISC code can execute within compartments such they are sandboxed from 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.

3.3.5 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, the kernel itself, or some combination of both. Typical kernel deployment patterns, some of which are orthogonal and may be used in combination, might be:

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

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

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

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

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

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

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

Chapter 4

Mapping the CHERI Protection Model into Architecture

Having considered the software-facing semantics of the CHERI protection model in the previous chapter, we turn to the high-level architectural implications of CHERI capabilities within a contemporary 64-bit RISC ISA. We attempt to remain at some distance from the specifics of the 64-bit MIPS ISA on which we have prototyped CHERI in the hope of accomplishing generality. However, MIPS differs substantially from other RISC ISAs in several areas – especially in its use of a software-managed Translation Lookaside Buffer (TLB), and in the details of its exception mechanism. In those cases, we necessarily take a MIPS-oriented perspective.

In this chapter, we describe our high-level design goals, the hybrid capability-system approach as applied in an instruction set, the implications for the software stack, and the CHERI capability model. We also describe the implications of CHERI for exceptions, tagged memory, and peripheral devices. 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. In Chapters 5 and 6, we consider in detail an instantiation of the CHERI protection model as an extension to the 64-bit MIPS ISA.

4.1 Design Goals

The key observation motivating the CHERI design is that virtual-memory-based protection techniques, nearly universal in commodity CPUs, are neither sufficiently expressive nor sufficiently efficient to support fine-grained memory protection or scalable software compartmentalization. Virtual addressing, implemented by a memory management unit (MMU) and translation look-aside buffer (TLB), clearly plays an important role by disassociating physical memory allocation and address-space management, facilitating software features such as strong separation, OS virtualization, and virtual-memory concepts such as swapping and paging. However, with a pressing need for scalable and fine-grained separation, the overheads and programmability difficulties imposed by virtual addressing as the sole primitive actively deter employment of the principle of least privilege at an architectural level (i.e., in instruction generation, the representation of pointers, etc.) and also at a software abstraction level (in representing isolation and controlled communication required for compartmentalization).

The security goals identified in Section 3.3 (spatial safety, temporal safety, software compartmentalization, and enforcement of language-level properties), combined with observations about TLB performance and a desire to compartmentalize existing single-address-space applications, led us to the conclusion that new instruction set primitives for memory and object control *within an address space* would usefully complement existing address-space-based separation. In this view, security state associated with a thread should be captured as a set of registers that can be explicitly managed by code, and be preserved and restored cheaply on either side of security domain transitions – in effect, part of a thread's register file. In the parlance of contemporary CPU and OS design, this establishes a link between hardware threads (OS threads) and security domains, rather than address spaces (OS processes) and security domains.

Because we wish to consider delegation of memory and object references within an address space as a first-class operation, we choose to expose these registers to the programmer (or, more desirably, the compiler) so that they can be directly manipulated and passed as arguments. Previous systems built along these principles have been referred to as *capability systems*, a term that also usefully describes CHERI.

CHERI's capability model represents an explicit capability system, in which common capability manipulation operations are unprivileged instructions and transfer of control to a supervisor during regular operations is avoided. In historic capability systems, microcode (or even the operating system) was used to implement complex capability operations, some of which were privileged. In contemporary RISC CPU designs, the intuitive functional equivalent has an exception that triggers the supervisor. However, entrance to a supervisor usually remains an expensive operation, and hence one to avoid in high-performance paths. In keeping with the RISC design philosophy, we are willing to delegate significant responsibility for safety to the compiler and run-time linker to minimize hardware knowledge of higher-level language constructs.

CHERI capabilities may refer to *regions of memory,* with bounded memory access (as in *segments*). Memory capabilities will frequently refer to programmer-described data structures such as strings of bytes, structures consisting of multiple fields, and entries in arrays, although they might also refer to larger extents of memory (e.g., the entire address space). While compatibility features in the CHERI ISA allow programmers to continue to use pointers in legacy code, we anticipate that capabilities will displace pointer use as code is migrated to CHERI code generation, providing stronger integrity for data references, bounds checking, permission checking, and so on. In our prototype extensions to the C language, programmers can explicitly request that capabilities be used instead of pointers, providing stronger protection, or in some cases rely on the compiler to automatically generate capability-aware code – for example, when code accessing the stack is compiled with a suitable application binary interface (ABI). We are exploring further static analysis and compilation techniques that will allow us to automate deployment of capability-aware code to a greater extent, minimizing disruption of current source code while allowing programs to experience protection improvements.

Alternatively, capabilities may refer to *objects* that can be *invoked*, which allows the implementation of *protected subsystems* – i.e., services that execute in a security domain other than the caller's. At the moment of object invocation, caller capabilities are *sealed* to protect them from inappropriate use by the callee, and the invoked object is *unsealed* to allow the object callee to access private resources it requires to implement its services. The caller and callee experience a controlled delegation of resources across object invocation and return. For

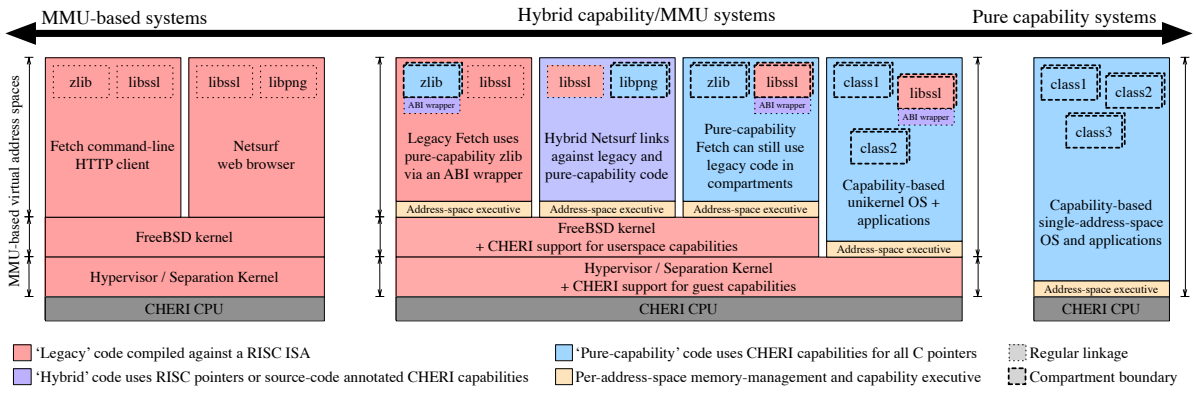


Figure 4.1: CHERI's hybrid capability architecture: initially, legacy software components execute without capability awareness, but security-sensitive TCB elements or particularly risky code bases are converted. In the long term, all packages are converted, implementing least privilege throughout the system.

example, the caller might delegate access to a memory buffer, and the callee might then write a Unicode string to the buffer describing the contents of the protected object, implementing call-by-reference.[1] A key goal has been to allow capabilities passed across protection-domain boundaries to refer to ordinary C data on the stack or heap, allowing easier adaptation of existing programs and libraries to use CHERI's features. The semantics of capabilities are discussed in greater detail later in this and the following chapter.

4.2 A Hybrid Capability-System Architecture

Despite our complaints about the implications of virtual addressing for compartmentalization, we feel that virtual memory is a valuable hardware facility: it provides a strong separation model; it makes implementing facilities such as swapping and paging easier; and by virtue of its virtual layout, it can significantly improve software maintenance and system performance. CHERI therefore adopts a *hybrid capability-system model*: we retain support for a commodity virtual-memory model, implemented using an MMU with a TLB, while also introducing new primitives to permit multiple security domains within address spaces (Figure 4.1). Each address space becomes its own decomposition domain, within which protected subsystems can interact using both hierarchical and non-hierarchical security models. In effect, each address space is its own *virtual capability machine*.

To summarize our approach, CHERI draws on two distinct, and previously uncombined, designs for processor architecture:

- *Page-oriented virtual memory systems* allow an executive (often the operating system kernel) to create a *process abstraction* via the MMU. In this model, the kernel is responsible for maintaining separation using this relatively coarse tool, and then providing system calls that allow spanning process isolation, subject to access control. Systems such

[1] CHERI does not implement implicit rights *amplification*, a property of some past systems including HYDRA. Callers across protected subsystem boundaries may choose to pass all rights they hold, but it is our expectation that they will generally not do so – otherwise, they would use regular function calls within a single protected subsystem.

as this make only weak distinctions between code and data, and in the mapping from programming language to machine code discard most typing and security information.

- *Capability systems*, often based on a single global address space, map programming-language type information and protection constraints into instruction selection. Code at any given moment in execution exists in a protection domain consisting of a dynamic set of rights whose delegation is controlled by the flow of code. (These *instantaneous rights* are sometimes referred to as *spheres of protection* in the operating system and security literature.) Such a design generally offers greater assurance, because the *principle of least privilege* can be applied at a finer granularity.

Figure 4.1 illustrates the following alternative ways in which the CHERI architecture might be used. In CHERI, even within an address space, existing and capability-aware code can be hybridized, as reads and writes via general-purpose MIPS registers are automatically indirected through a reserved capability register before being processed by the MMU. This allows a number of interesting compositions, including the execution of capability-aware, and hence significantly more robust, libraries within a legacy application. Another possibility is a capability-aware application running one or more instances of capability-unaware code within sandboxes, such as legacy application components or libraries – effectively allowing the trivial implementation of the Google Native Client model.

Finally, applications can be compiled to be fully capability-aware, i.e., able to utilize the capability features for robustness and security throughout their structure. The notion of a capability-aware *executive* also becomes valuable – likely as some blend of the run-time linker and low-level system libraries (such as `libc`): the executive will set up safe linkage between mutually untrusting components (potentially with differing degrees of capability support, and hence differing ABIs), and ensure that memory is safely managed to prevent memory-reuse bugs from escalating to security vulnerabilities.[2] Useful comparison might also be made between our notion of an in-address-space executive and a microkernel, as the executive will similarly take responsibility for configuring protection and facilitating controlled sharing of data. As microkernels are frequently capability-based, we might find that not only are ideas from the microkernel space reusable, but also portions of their implementations. This is an exciting prospect, especially considering that significant effort has been made to apply formal verification techniques to microkernels.

4.3 The CHERI Software Stack

The notion of hybrid design is key to our adoption argument: CHERI systems are able to execute today's commodity operating systems and applications with few modifications. Use of capability features can then be selectively introduced in order to raise confidence in the robustness and security of individual system components, which are able to fluidly interact with other unenhanced components. This notion of hybrid design first arose in Cambridge's Capsicum [99] (which blends the POSIX Application Programming Interface (API), as implemented in the FreeBSD operating system) with a capability design by allowing processes to execute in hybrid mode or in *capability mode*. Traditional POSIX code can run along side capability-mode

[2] Similar observations about the criticality of the run-time linker for both security and performance in capability systems have been made by Karger [43].

processes, allowing the construction of sandboxes; using a capability model, rights can be delegated to these sandboxes by applications that embody complex security policies. One such example from our USENIX Security 2010 Capsicum paper [99] is the Chromium web browser, which must map the distributed World Wide Web security model into local OS containment primitives.

CHERI's software stack will employ hybrid design principles from the bottom up: capability-enhanced separation kernels will be able to implement both conventional virtual-machine interfaces to guest operating systems, or directly host capability-aware operating systems or applications, ensuring robustness. This would provide an execution substrate on which both commodity systems built on traditional RISC instruction models (such as FreeBSD) can run side by side with a pure capability-oriented software stack, such as capability-adapted language runtimes. Further, CheriBSD, a CHERI-enhanced version of the FreeBSD operating system, and its applications, will be able to employ CHERI features in their own implementations. For example, key data-processing libraries, such as image compression or video decoding, might use CHERI features to limit the impact of programming errors through fine-grained memory protection, but also apply compartmentalization to mitigate logical errors through the principle of least privilege. We have extended the existing Clang/LLVM compiler suite to support C-language extensions for capabilities, allowing current code to be recompiled to use capability protections based on language-level annotations, but also to link against unmodified code.

To this end, the CHERI ISA design allows software context to address memory either via legacy MIPS ISA load and store instructions, which implicitly indirect through a reserved capability register configurable by software, or via new capability load and store instructions that allow the compiler to explicitly name the object to be used. In either case, access is permitted to memory only if it is authorized by a capability that is held in the register file (or, by transitivity, any further capability that can be retrieved using those registers and the memory or objects that it can reach). New ABIs and calling conventions are defined to allow transition between (and across) CHERI-ISA and MIPS-ISA code to allow legacy code to invoke capability-aware code, and vice versa. For example, in this model CheriBSD might employ capability-oriented instructions in the implementation of risky data manipulations (such as network-packet processing), while still relying on traditionally written and compiled code for the remainder of the kernel. Similarly, within the Chromium web browser, the JavaScript interpreter might be implemented in terms of capability-oriented instructions to offer greater robustness, while the remainder of Chromium would use traditional instructions.

One particularly interesting property of our hardware design is that capabilities can take on different semantics within different address spaces, with each address-space's executive integrating memory management and capability generation. In the CheriBSD kernel, for example, virtual addressing and capability use can be blended; the compiler and kernel memory allocator can use capabilities for certain object types, but not for others. In various userspace processes, a hybrid UNIX / C runtime might implement limited pools of capabilities for specially compiled components, but another process might use just-in-time (JIT) compilation techniques to map Java bytecode into CHERI instructions, offering improved performance and a significantly smaller and stronger Java TCB.

Capabilities supplement the purely hierarchical ring model with a non-hierarchical mechanism – as rings support traps, capabilities support protected subsystems. One corollary is that the capability model could be used to implement rings within address spaces. This offers some interesting opportunities, not least the ability to implement purely hierarchical models where

desired; for example, a separation kernel might use the TLB to support traditional OS instances, but only capability protections to constrain an entirely capability-based OS. A further extreme is to use the TLB only for paging support, and to implement a single-address-space operating system as envisioned by the designers of many historic capability systems.

This hybrid view offers a vision for a gradual transition to stronger protections, in which individual libraries, applications, and even whole operating systems can incrementally adopt stronger hardware memory protections without sacrificing the existing software stack. Discussion of these approaches also makes clear the close tie between memory-oriented protection schemes and the role of the memory allocator, an issue discussed in greater depth later in this chapter.

4.4 Architectural Goals

Given the pointer-centric objectives of the CHERI protection model, along with our compatibility and performance objectives, we identified the following architecture goals in identifying mappings into a contemporary RISC instruction-set architecture:

1. Extensions should subscribe to the RISC design philosophy: a load-store instruction set intended to be targeted by compilers, with more complex instructions motivated by quantitative analysis. While current page-table structures (or in the case of MIPS, simply TLB mechanisms) should be retained for compatibility, new table-oriented structures should be avoided in describing new security primitives. In general, instructions that do not access memory should be single cycle.

2. 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.

3. Protection primitives must be common-case, not exceptional, occurring in performance-centric code paths such as stack and heap allocation, on pointer arithmetic, and on pointer load and store, rather than being infrequent amortizable costs.

4. New primitives, such as tagged memory and capabilities, must be efficiently representable in contemporary hardware designs (e.g., superscalar processors and buses) while offering substantial semantic and performance improvements that would be difficult or impossible to support on current architectures.

5. Flexibility must exist 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.

6. When used to implement isolation and controlled communication in support of compartmentalization, CHERI's communication primitives should 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 scale with data size,

nor trigger TLB aliasing which 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).

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

8. When implementing efficient protection-domain switching, hardware should support 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.

9. CHERI should compose 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.

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

11. 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.

4.5 Capability Model

Chapters 5 and 6 provide detailed descriptions of CHERI's capability registers, capabilities in tagged memory, changes to the interpretation of current instructions, new capability instructions, exception delivery, and so on. These concepts are briefly introduced here.

4.5.1 Capabilities are for Compilers

Throughout, we stress the distinction between the notion of the hardware security model and the programming model; unlike in historic CISC designs, and more in keeping with historic RISC designs, CHERI instructions are intended to support the activities of the compiler, rather than be directly programmed by application authors. While there is a necessary alignment between programming language models for computation (and in the case of CHERI, security) and the hardware execution substrate, the purpose of CHERI instructions is to make it possible for the compiler to *cleanly* and *efficiently* implement higher-level models, and not implement them directly. As such, we differentiate the idea of a *hardware capability type* from a *programming language type* – the compiler writer may choose to conflate the two, but this is an option rather than a requirement.

4.5.2 Capabilities

Capabilities are unforgeable tokens of authority through which programs access all memory and services within an address space. Capabilities may be held in capability registers, where they can be manipulated or dereferenced using capability coprocessor instructions, or in memory. Capabilities themselves may refer to memory (unsealed capabilities) or objects (sealed capabilities). Memory capabilities are used as arguments to load and store instructions, to access either data or further capabilities. Object capabilities may be invoked to transition between protection domains using call and return instructions.

Unforgeability is implemented by two means: tag bits and controlled 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 instructions automatically clear the tag bit. When data is loaded into a capability register, its tag bit is also loaded; while data without a valid tag can be loaded into a capability 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.

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. In general, we expect that languages will not expose capability registers to management by programmers, instead using them for instruction operands and as a cache of active values, as is the case for general-purpose registers 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 compile correctness for non-TCB code.

4.5.3 Capability Registers

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

There are also several implicit capability registers associated with each hardware thread, including a memory capability that corresponds to the instruction pointer, and capabilities used during exception handling. This is structurally congruent to implied registers and system control coprocessor (CP0) registers found in the base MIPS ISA.

Unlike general-purpose registers, capability registers are structured, consisting of a 1-bit tag and a 256-bit set of architectural fields with defined semantics and constrained values. Capability instructions retrieve and set these fields by moving values in and out of general-purpose registers, enforcing constraints on field manipulation.

Microarchitectural and in-memory representations of capabilities may differ substantially from the architectural representation in terms of size and contents, but these differences will not be exposed via instructions operating on capability-register fields. We define two variants

with 256-bit and 128-bit in-memory representations of a conceptual 256-bit capability register, with the latter employing capability compression (Section 5.9) to reduce the register-file and memory footprint.

The ISA-visible capability fields are:

Sealed bit If the sealed bit is unset, the capability describes a *memory segment* that is accessible via load and store instructions. If it is set, the capability describes an *object capability*, which can be accessed only via *object invocation*.

Permissions The permissions mask controls operations that may be performed using the capability; some permissions are hardware-defined, controlling instructions that may be used with the capability; others are software-defined and intended for use with the object-capability mechanism.

Object type Notionally, a software-managed *object type* used to ensure that corresponding code and data capabilities for an object are used together correctly.

Base This is the base address of a memory region.

Length This defines the length of a memory region.

Offset 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, but an exception will be thrown if a requested access is out of range.

Reserved fields These bits are reserved for future experimentation.

Tag bit The tag bit is not part of the base 256 bits. It indicates whether or not the capability register holds a valid capability; this allows non-capability values to be moved via capability registers, making it possible to implement software functions that, for example, copy memory oblivious to capabilities being present.

We have discussed a number of schemes to reduce overhead implied by the quite sizeable capability register file:

- Having 32 capability registers is nicely symmetric with the MIPS ISA, but in practice leads to substantial overhead; this could be reduced to a smaller number such as 16, or even 8.

- Modeling our approach on the MIPS coprocessor opcode allocation, we have chosen to implement capabilities as an independent register file from the general-purpose register file. In fact, these could be combined, extending some or all existing registers to incorporate capability metadata. This would reduce or eliminate the need for additional control logic, and substantially reduce the overall size of a software context.

- 256-bit capabilities offer complete precision in representing the virtual address and bounds in a capability, as well as object type and a large set of software-defined permissions. By reducing the size of the representable virtual address (e.g., from 64 bits to 40 bits), as well as the sizes of other fields, a 128-bit capability could be accomplished.

- Similarly, fat-pointer compression schemes (e.g., [47] can be exploited to reduce overhead of maintaining bounds, which often contain significant redundancy relative to a capability's virtual address. Combined with other field reductions, and stronger alignment requirements, this can also accomplish a 128-bit capability; we description such a scheme in Section 5.9.

- It is also plausible to implement capabilities of multiple sizes – for example, a larger object-capability size, and smaller memory-capability size, using a full 256-bit representation in the register file, but different load and store instructions for what are effectively different in-memory types. This approach is less appealing as it will expose the differences in types to the software toolchain – e.g., requiring multiple pointer sizes.

Once again, hardware specifications written in the BSV language allow considerable flexibility and ease of understanding among these options, which will be particularly valuable as we get further into detailed experimentation, simulation, and modeling.

Object invocation is a central operation in the CHERI ISA, as it implements protected subsystem domain transitions that atomically update the set of rights (capabilities) held by a hardware thread, and that provide a trustworthy return path for later use. When an object capability is invoked, its data and code capabilities are *unsealed* to allow access to per-object instance data and code execution. Rights may be both acquired and dropped on invocation, allowing non-hierarchical security models to be implemented. Strong typing and type checking of objects, a notion first introduced in PSOS's *type enforcement,* serves functions both at the ISA level – providing object atomicity despite the use of multiple independent capabilities to describe an object – and to support language-level type features. For example, types can be used to check whether additional object arguments passed to a method are as they should be. As indicated earlier, the hardware capability type may be used to support language-level types, but should not be confused with language-level types.

4.5.4 Memory Model

In the abstract, capabilities are unforgeable tokens of authority. In the most reductionist sense, the CHERI capability namespace is the virtual address space, as all capabilities name (and authorize) actions on addresses. CHERI capabilities are unforgeable by virtue of capability register semantics and tagged memory, and act as tokens of authority by virtue of memory segments and object capability invocation.

However, enforcement of uniqueness over time is a property of the software memory allocation policy. More accurately, it is a property of virtual address-space allocation and reuse, which rests in a memory model composed from the capability mechanism, virtual address space configuration, and software language-runtime memory allocation.

This issue has presented a significant challenge in the design of CHERI: how can we provide sufficient mechanism to allow memory management, fundamentally a security operation in capability systems, while not overly constraining software runtimes regarding the semantics they can implement? Should we provide hardware-assisted garbage collection along the lines of the Java Virtual Machine's garbage collection model? Should we implement explicit revocation functionality, along the lines of Redell's capability revocation scheme (effectively, a level of indirection for all capabilities, or selectively when the need for revocation is anticipated)?

We have instead opted for dual semantics grounded in the requirements of real-world low-level system software: CHERI lacks a general revocation scheme; however, in coordination with the software stack, it can provide for both strict limitations on the extent of hardware-supported delegation periods, and software-supported generalized revocation using interposition. The former is intended to support the brief delegation of arguments from callers to callees across object-capability invocation; the latter allows arbitrary object reference revocation at a greater price.

4.5.5 Local Capabilities and Revocation

To this end, capabilities may be further tagged as *local*, which allows them to be processed in registers, stored in constrained memory regions, and passed on via invocation of other objects. The goal of local capabilities is to introduce a limited form of *revocation* that is appropriate for temporary delegation across protected subsystem invocations, which are not permitted to persist beyond that invocation. Among other beneficial properties, local capabilities allow the brief delegation of access to arguments passed by reference, such as regions of the caller's stack (a common paradigm in C language programming).

In effect, *local capabilities* inspire a single-bit information-flow model, bounding the potential spread of capabilities for ephemeral objects to capability registers and limited portions of memory. The desired protection property can be enforced through appropriate memory management by the address-space executive: that is, local capabilities can be limited to a particular thread, with bounded delegation time down the (logical) stack.[3]

Generalized revocation is not supported directly by the CHERI ISA; instead, we rely on the language runtime to implement either a policy of *virtual address non-reuse* or *garbage collection*. A useful observation is that address space non-reuse is not the same as memory non-reuse: the meta-data required to support sparse use of a 64-bit address space scales with actual allocation, rather than the span of consumed address space. For many practical purposes, a 64-bit address space is *virtually* infinite[4], so causing the C runtime to not reuse address space is now a realistic option. Software can, however, make use of interposition to implement revocation or other more semantically rich notions of privilege narrowing, as proposed in HYDRA.

4.5.6 Notions of Privilege

In operating-system design, *privileges* are a special set of rights exempting a component from the normal protection and access-control models – perhaps for the purposes of system bootstrapping, system management, or low-level functionality such as direct hardware access. In CHERI, three notions of privilege are defined – two in hardware, and a new notion of privilege in software relating to the interactions of capability security models between rings.

Ring-based privilege is derived from the commodity hardware notion that a series of successively higher-level rings provides progressively fewer rights to manage hardware protection

[3]It has been recommended that we substitute a generalized generation count-based model for an information flow model. This would be functionally identical in the local capability case, used to protect per-stack data. However, it would also allow us to implement protection of thread-local state, as well as garbage collection, if desired. The current ISA does not yet reflect this planned change.

[4]As is 640K of memory. It has also not escaped our notice that there is a real OS cost to maintaining the abstraction of virtual memory; one merit to our approach is that it will deemphasize the virtual memory as a protection system, potentially reducing that overhead.

features, such as TLB entries – and consequently potentially greater integrity, reliability, and resilience overall (as in Multics). Attempts to perform privileged instructions will trap to a lower ring level, which may then proceed with the operation, or reject it. CHERI extends this notion of privilege into the new capability coprocessor, authorizing certain operations based on the ring in which a processor executes, and potentially trapping to the next lower ring if an operation is not permitted. The trap 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 trap handler.

Hardware capability context privilege is a new notion of privilege that operates within rings, and is managed by the capability coprocessor. When a new address space is instantiated, code executing in the address space is provided with adequate initial capabilities to fully manage the address space, and derive any required capabilities for memory allocation, code linking, and object-capability type management. In CHERI, all capability-related privileges are captured by capabilities, and capability operations never refer to the current processor ring to authorize operations, although violation of a security property (i.e., an attempt to broaden a memory capability) will lead to a trap, and allow a software supervisor in a lower ring to provide alternative semantics. This approach follows the spirit of Paul Karger's paper on limiting the damage potential of discretionary Trojan horses [41], and extends it further.

Supervisor-enforced capability context privilege is a similar notion of privilege that may also be implemented in software trap handlers. For example, an operating system kernel may choose to accept system-call traps only from appropriately privileged userspace code (e.g., by virtue of holding a capability with full access to the userspace address space, rather than just narrow access, or that has a reserved user-defined permission bit set), and therefore can check the capability registers of the saved context to determine whether the trap was from an appropriate execution context. This might be used to limit system call invocation to a specific protected subsystem that imposes its own authorization policy on application components by safely wrapping system calls from userspace.

4.5.7 Traps, Interrupts, and Exception Handling

As in MIPS, traps and interrupts remain the means by which ring transitions are triggered in CHERI. They are affected in a number of ways by the introduction of capability features:

New exceptions 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, attempts to read a capability from memory using a capability without the read capability permission will trigger a trap.

Reserved capability registers for exception handling New exception-handling functionality is required to ensure that exception handlers themselves can execute properly. We reserve several capability registers for use both by the exception-handling mechanism itself (describing the rights that the exception handler will run with) and for use by software exception handlers (a pair of reserved registers that can be used safely during context switching). This approach is not dissimilar from the current notion of exception-handling registers in the MIPS ABI, which reserves two general-purpose registers for this purpose. However, whereas the MIPS ABI simply dictates that user code cannot rely on the two reserved exception registers being

preserved, CHERI requires that access is blocked, as capability registers delegate rights and also hold data. We currently grant access to exception-related capability registers by virtue of special permission bits on the capability that describe the currently executing code; attempting to access reserved registers without suitable permission will trigger an exception.

Saved program-counter capability Exception handlers must also be able to inspect exception state; for example, as **PC**, the program counter, is preserved today in a control register, **EPC**, the program counter capability must be preserved as **EPCC** so that it can be queried.

Implications for pipelining Another area of concern in the implementation is the interaction between capability registers and pipelining. Normally, writing to TLB control registers in CP0 occurs only in privileged rings, and the MIPS ISA specifies that a number of no-op instructions follow TLB register writes in order to flush the pipeline of any inconsistent or intermediate results. Capability registers, on the other hand, may be modified from unprivileged code, which cannot be relied upon to issue the required no-ops. This case can be handled through the squashing of in-flight instructions, which may add complexity to pipeline processing, but is required to avoid the potential for serious vulnerabilities.

4.5.8 Tagged Memory

As with general-purpose registers, storage capability register values in memory is desirable – for example, to push capabilities onto the stack, or manipulate arrays of capabilities. To this end, each capability-aligned and capability-sized word in memory has an additional *tag bit*. The bit is set whenever a capability is atomically written from a register to an authorized memory location, and cleared if a write occurs to any byte in the word using a general-purpose store instruction. Capabilities may be read only from capability-aligned words, and only if the tag bit is set at the moment of load; otherwise, a capability load exception is thrown. Tags are associated with physical memory locations, rather than virtual ones, such that the same memory mapped at multiple points in the address space, or in different address spaces, will have the same tags.

Tags require strong coherency with the data they protect, and it is expected that tags will be cached with the memory they describe within the cache hierarchy. Strong atomicity properties are required such that it is not possible to partially overwrite a capability value in memory while retaining the tag. This proves a set of properties that falls out naturally from current coherent memory-subsystem designs.

Additional bits are present in TLB entries to indicate whether a given memory page is configured to have capabilities loaded or stored for the pertinent address space identifier (ASID). For example, this allows the kernel to set up data sharing between two address spaces without permitting capability sharing (which, as capability interpretation is scoped to address spaces, might lead to undesirable security or programmability properties). Special instructions allow the supervisor to efficiently extract and set tag bits for ranges of words within a page for the purpose of more easily implemented paging of capability memory pages. Use of these instructions is conditioned on notions of ring and capability context privilege.

4.5.9 Capability Instructions

Various newly added instructions are documented in detail in Chapter 5. Briefly, these instructions are used to load and store via capabilities, load and store capabilities themselves, manage capability fields, invoke object capabilities, and create capabilities. Where possible, the structure and semantics of capability instructions have been aligned with similar core MIPS instructions, similar calling conventions, and so on. The number of instructions has also been minimized to the extent possible.

4.5.10 Object Capabilities

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

In traditional capability designs, invocation of an object capability triggered microcode responsible for state management. Initially, we have implemented `CCall` and `CReturn` as software exception handlers in the kernel, but are now exploring optimizations in which `CCall` and `CReturn` perform a number of checks and transformations to minimize software overhead. In the longer term, we hope to investigate the congruence of object-capability invocation with message-passing primitives between hardware 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 and capability registers, then message passing may offer a way to provide significantly enhanced performance.[5] In this view, hardware 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.

[5]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.

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.

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

4.5.11 Peripheral Devices

As described in this chapter, our capability model is a property of the instruction set architecture of a CHERI CPU, and imposed on code executing on the CPU. However, in most computer systems, Direct Memory Access (DMA) is used by 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 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.

This raises a number of interesting questions. Should DMA be forced to pass through the capability equivalent of an I/O MMU in order to be appropriately constrained? How might this change the interface to peripheral devices, which currently assume that physical addresses are passed to them? Certainly, reuse of current peripheral networking and video devices with CHERI CPUs while maintaining desired security properties is desirable.

For the time being, device drivers continue to hold the privilege for DMA to use arbitrary physical memory addresses, although hybrid models – such as allowing DMA access only to specific portions of physical memory – may prove appropriate. Similar problems have plagued virtualization in commodity CPUs, where guest operating systems require DMA memory performance but cannot be allowed arbitrary access to physical memory. Exploring I/O MMU-like models and their integration with capabilities is high on our todo list; one thing is certain, however: a combination of hardware- and software-provided cache and memory management must ensure that tags are suitably cleared when capability-oblivious devices write to memory, in order to avoid violation of capability integrity properties.

In the longer term, one quite interesting idea is embedding CHERI support in peripheral devices themselves, to require the device to implement a CHERI-aware TCB that would synchronize protection information with the host OS. This type of model appeals to ideas from heterogeneous computing, and is one we hope to explore in greater detail in the future. This will require significant thinking on both how CHERI protection applies to other computation models, and also how capability state (e.g., the tag bit and its implied atomicity) will be exposed via the bus architecture.

4.6 Deep Versus Surface Design Choices

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

- Capabilities can be used to implement pointers into virtual address spaces;
- Tags on registers determine whether they are valid pointers for loading, fetching, or jumping to;
- Tagged registers can contain both data and capabilities, allowing (for example) capability-oblivious memory copies;
- Tags on pointer-sized, pointer-aligned units of memory preserve validity (or invalidity) across loads and stores to memory;
- Tags are associated with physical memory locations – i.e., if the same physical memory is mapped at two different virtual addresses, the same tags will be used;
- Attempts to store data into memory that has a valid tag will atomically clear the tag;
- Capability loads and stores to memory offer strong atomicity with respect to capability values and tags preventing races that might yield combinations of different capability values, or the tag remaining set when a corrupted capability is reloaded;
- Pointers are extended to include bounds and permissions; the "pointer" is able to float freely within (and to varying extents, beyond) the bounds;
- Permissions are sufficient to control both data and control-flow operations;
- Guarded manipulation implements monotonicity: rights can be reduced but not increased through valid manipulations of pointers;
- Invalid manipulations of pointers violating guarded-manipulation rules lead to an exception or clearing of the valid tag, whether in a register or in memory, with suitable atomicity;
- Loads via, stores via, and jumps to capabilites are constrained by their permissions and bounds, throwing exceptions on a violation;
- Capability exceptions, in general, are delivered with greater priority than MMU exceptions;
- Permissions on capabilities include the ability to not just control loading and storing of data, but also loading and storing of capabilities;

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

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

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

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

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

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

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

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

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

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

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

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

- The number of capability registers present;

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

- How capabilities representing escalated privilege for exception handlers are stored;
- Whether specific capability-related failures (in particular, operations violating guarded manipulation) lead to an immediate exception, or simply clearing of the tag and a later exception on use;
- How tags are stored in the memory subsystem – e.g., whether close to the DRAM they protect or in a partition of memory – as long as they are presented with suitable protections and atomicity up the memory hierarchy;
- How the instruction-set opcode space is utilized – e.g., via coprocessor reservations in the opcode space, reuse of existing instructions controlled by a mode, etc; and
- 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;
- 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.

Chapter 5

CHERI Instruction-Set Architecture

This chapter describes an application of the CHERI approach to the 64-bit MIPS ISA. New instructions are implemented as a MIPS coprocessor – coprocessor 2 – an encoding space reserved for ISA extensions. In addition to adding new instructions, some behaviors have been modified in CHERI – notably, those of some standard MIPS instructions, TLBs, and exception handling. For example, existing memory load and store instructions are now implicitly indirected through a capability in order to enforce permissions, rebasing, and bounds checking on legacy code.

5.1 Capability Registers

Table 5.1 illustrates capability registers defined by the capability coprocessor. CHERI defines 28 general-purpose capability registers, which may be named using most capability register instructions. These registers are intended to hold the working set of rights required by in-execution code, intermediate values used in constructing new capabilities, and copies of capabilities retrieved from **EPCC** and **PCC** as part of the normal flow of code execution, which is congruent with current MIPS-ISA exception handling via coprocessor 0. Four capability registers have special functions and are accessible only if allowed by the permissions field **C0**. Note that **C0** and **C27** (**IDC**) also have hardware-specific functions, but are otherwise general-purpose capability registers.

Each capability register also has an associated tag indicating whether it currently contains a valid capability. Any load and store operations via an invalid capability will trap.

5.1.1 Capability Register Conventions / Application Binary Interface (ABI)

We are developing a set of ABI conventions regarding use of the other software-managed capability registers similar to those for general-purpose registers: caller-save, callee-save, a stack capability register, etc. Code can be compiled using a number of ABIs:

MIPS ABI The MIPS n64 ABI: capability registers and capability instructions are unused. Generated code relies on MIPS compatibility features to interpret pointers with respect to the program-counter and default-data capabilities.

Hybrid ABI Capability-aware code makes selective use of capability registers and instructions, but can transparently interoperate with MIPS-ABI code when capability arguments

Register(s)	Description
PCC	Program counter capability (**PCC**): the capability through which **PC** is indirected by the processor when fetching instructions.
DDC (**C0**)	Capability register through which all non-capability load and store instructions are indirected. This allows legacy MIPS code to be controlled using the capability coprocessor.
C1...**C25**	General-purpose capability registers referenced explicitly by capability-aware instructions.
IDC (**C26**)	Invoked data capability: the capability that was unsealed at the last protected procedure call. This capability holds the unlimited capability at boot time.
KR1C (**C27**)	A capability reserved for use during kernel exception handling.
KR2C (**C28**)	A capability reserved for use during kernel exception handling.
KCC (**C29**)	Kernel code capability: the code capability moved to **PCC** when entering the kernel for exception handling.
KDC (**C30**)	Kernel data capability: the data capability containing the security domain for the kernel exception handler.
EPCC (**C31**)	Capability register associated with the exception program counter (**EPC**) required by exception handlers to save, interpret, and store the value of **PCC** at the time the exception fired.

Table 5.1: Capability registers defined by the capability coprocessor.

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.

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

- **C1**-**C2** are caller-save. During a cross-domain call, these are used to pass the **PCC** and **IDC** values, respectively. In the invoked context, they are always available as temporaries, irrespective of whether the function was invoked as the result of a cross-domain call.
- **C3**-**C10** are used to pass arguments and are not preserved across calls.
- **C11**-**C16** are caller-save registers.
- **C17**-**C24** are callee-save registers.
- **C25** is currently unused.

In all ABIs, the following convention also applies:

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

The pure-capability ABI, used within compartments or for pure-capability ("CheriABI") applications, implements the following further conventions for capability use:

- **C11**, in the pure-capability ABI, contains the stack capability (congruent to MIPS **$sp**).
- **C12**, in the pure-capability ABI, contains the jump register (congruent to MIPS **$t9**).
- **C17**, in the pure-capability ABI, contains the link register (congruent to MIPS **$ra**).

When calling (or being called) across protection domains, there is no guarantee that a non-malicious caller or callee will abide by these conventions. Thus, all registers should be regarded as caller-save, and callees cannot depend on caller-set capabilities for the stack and jump registers. Additionally, all capability registers that are not part of the explicit argument or return-value sets should be cleared via explicit assignment or via the `CClearHi` and `CClearLo` instructions. This will prevent leakage of rights to untrustworthy callers or callees, as well as

accidental use (e.g., due to a compiler bug). Where rights are explicitly passed between domains, it may be desirable to clear the global bit that will (in a suitably configured runtime) limit further propagation of the capability. Similar concerns apply to general-purpose registers, or capability registers holding data, which should be preserved by the caller if their correct preservation is important, and cleared by the caller or callee if they might leak sensitive data. Optimized clearing instructions `ClearHi` and `ClearLo` are available to efficiently clear general-purpose registers.

5.1.2 Cross-Domain Procedure Calls

A cross-domain procedure call, instruction `CCall`, escapes to a handler that takes a sealed executable ("code") and sealed non-executable ("data") capability with matching types. If the types match, the unsealed code capability is placed in **PCC** and the unsealed data capability is placed in **IDC**. The handler will also push the previous **PCC**, **IDC**, **PC** + 4, and **SP** to a stack pointed to by **TSC**. The stack pointer **TSC** may be implemented either as a hardware register or as a variable internal to a software implementation of `CCall`. The caller should invalidate all registers that are not intended to be passed to the callee before the call.

A cross-domain procedure return, instruction `CReturn`, also escapes to a handler that pops the code and data capabilities from the stack at **TSC** and places them in **PCC** and **IDC**, respectively; it likewise pops **PC** and **SP**. The callee should invalidate all registers that are not intended to be passed to the caller before the return.

The caller is responsible for ensuring that its protection domain is entirely embodied in the capability in **IDC** so that it can restore its state upon return.

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

5.1.3 Capabilities and Exception Handling

KCC and **KDC** hold the code capability and data capability that describe the protection domain of the system exception handler. When an exception occurs, the victim **PCC** is copied to **EPCC** so that the exception may return to the correct address, and **KCC**, excepting its **offset** field, which will be set to the appropriate MIPS exception-vector address, is moved to **PCC** to grant execution rights for kernel code. When an exception handler returns with `ERET`, **EPCC**, possibly after having been updated by the software exception handler, is moved into **PCC**. **KDC** may be manually installed by the exception handler if needed, and will typically be

moved into **DDC** in order to allow otherwise unmodified MIPS exception handlers to be used. This may also need to be restored before returning from the exception.

5.2 Capabilities

Each capability register contains the following fields:

- Tag bit ("**tag**", 1 bit)
- Sealed bit ("**s**", 1 bit)
- Permissions mask ("**perms**")
- User-defined permissions mask ("**uperms**")
- Object type ("**otype**", 24 bits)
- Offset ("**offset**", 64 bits)
- Base virtual address ("**base**", 64 bits)
- Length in bytes ("**length**", 64 bits)

A capability register can be in one of four possible states:

- It contains a valid capability. The **tag** bit is set, and all of the above fields contain defined values.
- It contains all of the fields of a capability (as defined above), but the capability is not valid for use (for example, because the program that set the values has not yet demonstrated that it is authorized to create a valid capability with those values). The **tag** bit is not set.
- It contains a 64-bit integer stored in the **offset** field. The **tag** bit is not set, and the **offset** field contains the integer.
- It contains N bits of non-capability data that have been loaded from memory. (Where N is the size of a capability, 256 bits for the 256-bit representation). The **tag** bit is not set. Programs should not rely on there being any particular relationship between the bytes that have been loaded from memory and the **offset** field, although see below for the current in-memory representation of capabilities.

The CHERI ISA can be implemented with several different in-memory representations of capabilities. A 256-bit format is described in Section 5.8. A 128-bit compressed format is described in Section 5.9.

5.2.1 Tag Bit

The **tag** bit indicates whether a capability register contains a capability or normal data. If **tag** is set, the register contains a capability. If **tag** is cleared, the rest of the register contains 256 bits of normal data.

5.2.2 Sealed Bit

The **s** flag indicates whether a capability is usable for general-purpose capability operations. If this flag is set, the capability is sealed and it may be used only by a `CCall` instruction. If the `CCall` instruction receives a sealed executable capability and a sealed non-executable capability with matching **otype** fields, both capabilities will have their **s** flag cleared and control flow branches to the code indicated by the code capability, thus entering a new security domain.

5.2.3 Permission Bits

The **perms** bit vector governs the permissions of the capability including read, write and execute permissions. Bits 0–7 and 10 of this field, which control use and propagation of the capability, and also limit access to exception-handling features, are described in Table 5.2.

5.2.4 User-Defined Permission Bits

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

User-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, user-defined permissions on code capabilities could be employed by a userspace runtime to allow the kernel to determine whether a particular piece of user code is authorized to perform system calls. Similarly, user permissions on sealed data capabilities might authorize use of specific methods (or sets of methods) on object capabilities, allowing different references to objects to authorize different software-defined behaviors. By clearing all hardware-defined permissions, software-defined capabilities might be used as unforgeable tokens authorizing use of in-application or kernel services.

5.2.5 Object Type

Th 24-bit **otype** field holds the "type" of a sealed capability; this field allows an unforgeable link to be created between associated data and object capabilities. While earlier versions of the CHERI ISA interpreted this field as an address, recent versions treat this as an opaque software-managed value.

5.2.6 Offset

The 64-bit **offset** field holds an offset between the base of the capability and a byte of memory that is currently of interest to the program that created the capability. Effectively, it is a convenient place for a program to take an index into an array and store it inside a capability that grants access to the array. No bounds checks are performed on **offset** when its value is set by the `CSetOffset` instruction: programs are free to set its value beyond the end of the capability

Value	Name
0	Global
1	Permit_Execute
2	Permit_Load
3	Permit_Store
4	Permit_Load_Capability
5	Permit_Store_Capability
6	Permit_Store_Local_Capability
7	Permit_Seal
8	*reserved*
9	*reserved*
10	Access_System_Registers

Table 5.2: Memory permission bits for the **perms** capability field

as defined by the **length** field. (Bounds checks are, however, performed when a program attempts to use the capability to access memory at the address given by **base** + **offset** + register offset.)

5.2.7 Base

The 64-bit **base** field is the base virtual address of the segment described by a capability.

5.2.8 Length

The 64-bit **length** field is the length of the segment described by a capability.

5.3 Capability Permissions

Table 5.2 shows the definition of bits 0–7 of the **perms** field.

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.

Permit_Store_Capability Allow this capability to be used as a pointer for storing other capabilities.

Permit_Load_Capability Allow this capability to be used as a pointer for loading other capabilities.

Permit_Store Allow this capability to be used as a pointer for storing data from general-purpose registers.

Permit_Load Allow this capability to be used as a pointer for loading data into general-purpose registers.

Permit_Store_Local_Capability Allow this capability to be used as a pointer for storing local capabilities.

Permit_Seal Allow this capability to be used to seal or unseal capabilities that have the same **otype**.

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.

In general, permissions on a capability relate to its implicit or explicit use in authorizing an instruction – e.g., in fetching the instruction via **PCC**, branching to a code capability, loading or storing explicitly via a data capability, performing sealing operations, or controlling 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 fit the "capability as an operand" model:

Access_System_Registers Allow access to **EPCC**, **KDC**, **KCC**, **KR1C** and **KR2C** when this capability is in **PCC**.

5.4 Capability Exceptions

In the future, we anticipate increasing the coverage of this permission to include further privileged ISA features, such as control of the MMU, certain cache operations, ability to return from an exception, control over the interrupt mechanism, and so on, with the aim of permitting compartmentalized code within an otherwise privileged ring in the ISA.

Many of the capability instructions can cause an exception (e.g., if the program attempts a load or a store that is not permitted by the capability system). When the cause of an exception is that the attempted operation is prohibited by the capability system, the *ExcCode* field within the **cause** register of coprocessor 0 are set to 18 (*C2E*, coprocessor 2 exception), **PCC** and **EPCC** are set as described in Section 5.5 and **capcause** is set as described below.

5.4.1 Capability Cause Register

The capability coprocessor has a **capcause** register that gives additional information on the reason for the exception. It is formatted as shown in Figure 5.1. The possible values for the *ExcCode* of **capcause** are shown in Table 5.4.1. If the last instruction to throw an exception did not throw a capability exception, then the *ExcCode* field of **capcause** will be *None*. *ExcCode*

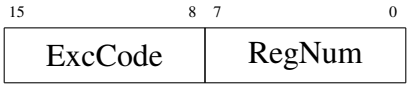


Figure 5.1: Capability Cause Register

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

Table 5.3: Capability Exception Codes

values from 128 to 255 are reserved for use by application programs. (A program can use `CSetCause` to set *ExcCode* to a user-defined value).

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

The `CGetCause` instruction can be used by an exception handler to read the **capcause** register. `CGetCause` will raise an exception if **PCC**.**perms**.*Access_System_Registers* is not set, so the operating system can prevent user space programs from reading **capcause** directly by not granting them *Access_System_Registers* permission.

5.4.2 Exception Priority

If an instruction throws more than one capability exception, **capcause** is set to the highest priority exception (numerically lowest priority number) as shown in Table 5.4. The *RegNum* field of **capcause** is set to the register which caused the highest priority exception.

All capability exceptions (C2E) have higher priority than address error exceptions (AdEL, AdES).

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

Some of these priority rules are security critical. In particular, an exception caused by a register being reserved must have priority over other capability exceptions (e.g., AdEL and AdES) to prevent a process from discovering information about the contents of a register that it is not allowed to access.

Other priority rules are not security critical, but are defined by this specification so that exception processing is deterministic.

An operating system might implement unaligned loads and stores by catching the AdEL and AdES exceptions and emulating the load or store. As capability exceptions have higher priority than alignment exceptions, this exception handler would not need to check the permissions (and base, length, etc.) of the capability before emulating the load/store, because it would be guaranteed that all capability checks had already been done by the hardware, and had passed.

5.4.3 Exceptions and Indirect Addressing

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

5.4.4 Hardware-Software Implementation of CCall and CReturn

In the current hardware implementation of CHERI, the `CCall` and `CReturn` instructions always raise an exception, so that the details of the call or return operation can be implemented

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

Table 5.4: Exception Priority

in software by a trap handler. This exception uses a different trap handler vector, at 0x100 above the general purpose exception handler. To accelerate the software handler, the instructions perform a number of checks in hardware – for example, on `CCall`, that the tag bits, sealed bits, and object types of the passed code and data capabilities correspond suitably. The exception cause will be set to *C2E*. If all checks are successful, then **capcause** will be set to *Call Trap* for `CCall`, and *Return Trap* for `CReturn`. If one or more checks fail, then a suitable exception code for the failure, such as *Tag Violation*, *Sealed Violation*, or *Type Violation*, will be set instead. This design balances a desire for a flexible software implementation with the performance benefits of parallel checking in hardware.

5.5 CPU Reset

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

- The **tag** bit is set.
- The **s** bit is unset.
- **offset** = 0 (except for **PCC**.**offset**, which will be initialized to the boot vector address)
- **base** = 0
- **length** = $2^{64} - 1$

- **otype** = 0

- All available permission bits are set. (When the 256-bit capability representation is used, 31 permission bits are available, including 16 user-defined permissions. When the 128-bit capability representation is used, 19 permission bits are available, including 4 user-defined permissions; the remaining 12 user-defined permissions at the most-significant end of the permissions field are not available, and are set to zero. Permission bits 8 and 9 are currently reserved for future use; these are included in the the 31 (or 19) permission bits that are set on reset).

- All unused bits are cleared.

The initial values of **PCC** and **KCC** will allow the system to initially execute code relative to virtual address 0. The initial value of **DDC** will allow general-purpose loads and stores to all of virtual memory for the bootstrapping process.

5.6 Changes to Standard MIPS ISA Processing

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

Instruction fetch The MIPS-ISA program counter (**PC**) is extended to a full program-counter capability (**PCC**), which incorporates the historic **PC** as **PCC.offset**. Instruction fetch is controlled by the *Permit_Execute* permission, as well as bounds checks, tag checks, and a requirement that the capability not be sealed. Failures will cause a coprocessor 2 exception (*C2E*) to be thrown. If an exception occurs during instruction fetch (e.g., AdEL, or a TLB miss) then *BadVAddr* is set equal to **PCC.base** + **PCC.offset**, providing the absolute virtual address rather than a **PCC**-relative virtual address to the supervisor, avoiding the need for capability awareness in TLB fault handling.

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

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

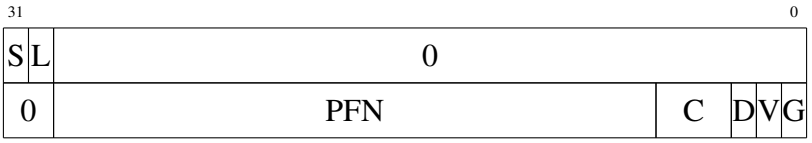


Table 5.5: EntryLo Register

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

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

Exceptions The MIPS exception program counter (**EPC**) is extended to a full exception program-counter capability (**EPCC**), which incorporates the historic **EPC** as **EPCC**.**offset**. Once **PCC** has been preserved, the contents of the *kernel code capability* (**KCC**), excluding **KCC**.**offset**, are moved into **PCC**. **EPCC**.**offset** (**EPC**) will be set to the exception vector address normally used by MIPS. This allows the exception handler to run with the permissions granted by **KCC**, which may be greater than the permissions granted by **PCC** before the exception occurred.

If capability compression is being used (see Section 5.9), and the value of **EPC** is sufficiently far outside the bounds of **EPCC** that a capability with those bounds and offset is not representable (e.g., when the exception was caused by a branch far outside the range of **PCC**), then **EPCC**.**offset** is set to **EPC**, **EPCC**.**tag** is cleared, **EPCC**.**base** is set to zero and **EPCC**.**length** is set to zero.

On return from an exception (`eret`), **PCC** is restored from **EPCC**, which will include **EPCC**.**offset** (also visible as **EPC**) [1] This allows exception handlers that are not aware of capabilities to continue to work on a CPU with the CHERI extensions. The result of `eret` is **UNDEFINED** if **EPCC**.**tag** is not set or **EPCC**.**s** is set. Similarly, the result of an exception or interrupt is **UNDEFINED** if **KCC**.**tag** is not set or **KCC**.**s** is set.

5.7 Changes to the Translation Lookaside Buffer (TLB)

CHERI adds two new fields to the EntryLo register, shown as L and S in Table 5.5, to the conventional MIPS Translation Lookaside Buffer (TLB).

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

[1] In our current CHERI1 prototype, for reasons of critical path length, **EPCC**.**offset** will be updated to be the value of the MIPS **EPC** on exception entry, but writes to **EPCC**.**offset** will not be propagated to **PCC**.**offset** on exception return. As described later in this chapter, we have proposed shifting **EPCC** out of the ordering capability register file and instead using special registers in order to eliminate this problem. Our L3 formal model of the CHERI ISA implements the specified behavior. CheriBSD is able to operate on both as it is careful to update both **EPC** and **EPCC** before returning.

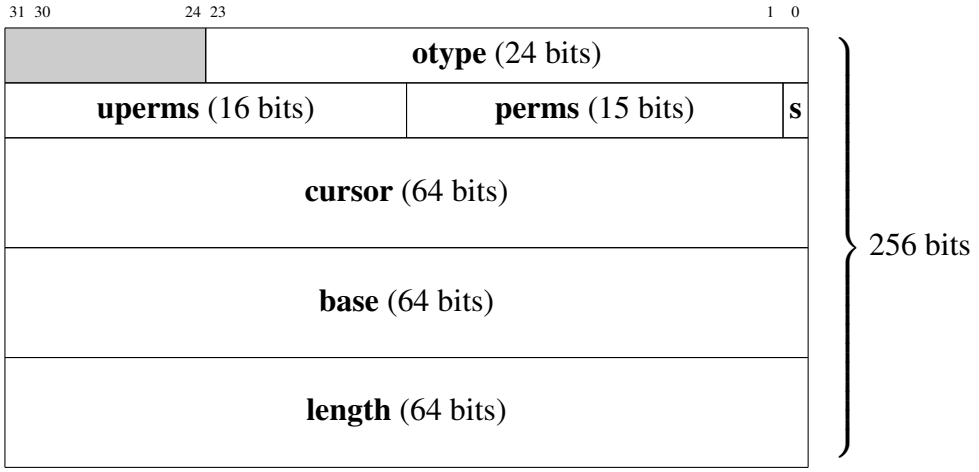


Figure 5.2: 256-bit memory representation of a capability

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

At with other TLB-related exceptions, *BadVAddr* will be set to the absolute virtual address that has triggered the fault.

5.8 256-bit Capability Format

A 256-bit format for representing capabilities is shown in Figure 5.2. This is the format that is currently used by the 256-bit versions of the Bluespec implementation, the L3 formal model, and the CHERI-enabled QEMU emulator. Programs should not rely on this memory representation, as there are alternative capability representations (see, for example, the 128-bit format in Section 5.9), and it may change in future. Instead, programs should access the fields through the instructions provided in the ISA.

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

In this representation, **uperms** is a 16 bit field and **perms** is 15-bit field.

5.9 128-bit Capability Compression

256-bit capabilities offer high levels of precision and software compatibility, but at a cost: quadrupling the size of pointers. This has significant software and micro-architectural costs to cache footprint, memory bandwidth, and also in terms of the widths of memory paths in the

design. However, the CHERI ISA is designed to be largely agnostic to the in-memory representation, permitting alternative "compressed" representations while retaining largely compatible 256-bit software behavior. Compression is possible because the base, length, and pointer values in capabilities are frequently redundant, which can be exploited by increasing the alignment requirements on bounds associated with a pointer (while retaining full precision for the pointer itself). Space can further be recovered by enforcing stronger alignment requirements on sealed capabilities than for data capabilities (as only sealed capabilities require an object type), and by reducing the number of permission and reserved bits.

Using this approach, it is possible to usefully represent capabilities via a compressed 128-bit in-memory representation, while retaining a 256-bit architectural view. Compression results in a loss of precision, exposed as a requirement for stronger bounds alignment, for larger memory allocations. Because of the representation, we are able to vary the requirement for alignment based on the size of the allocation, and for small allocations ($< \frac{3}{4}MiB$), 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 $\frac{3}{4}MiB$ 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 $1GiB$ 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 5.9.4.

Different representations are 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.

5.9.1 CHERI-128 Implementation

The compressed in-memory formats for CHERI-128 unsealed and sealed capabilities are depicted in Figures 5.3 and 5.4.

μperms Hardware permissions for this format are trimmed from those listed in Table 5.2 by consolidating system registers. The condensed format is listed in Table 5.6

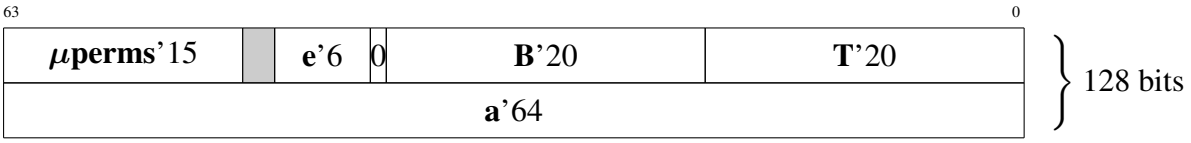


Figure 5.3: Unsealed CHERI-128 memory representation of a capability

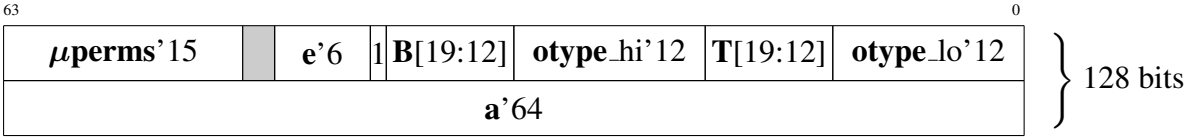


Figure 5.4: Sealed CHERI-128 memory representation of a capability

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.

s Indicates if a capability is sealed or not, listed simply as 0 or 1 in Figures 5.3 and 5.4 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:

$$\mathbf{B} = \left\lfloor \frac{\mathbf{base\_req}}{2^{\mathbf{e}}} \right\rfloor \bmod 2^{20}$$

Which can be rewritten as a bit-manipulation:

$$\mathbf{B} = \mathbf{base\_req}[19 + \mathbf{e} : \mathbf{e}]$$

For sealed capabilities, $\mathbf{B}[11:0] = 0$

architectural bit#	**μperms** 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
–	8–9	Reserved
perms[10]	10	Access_System_Registers
uperms[15–18]	11–14	Software-defined permissions

Table 5.6: Permission bit mapping

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:

$$\mathbf{T} = \left\lceil \frac{\textbf{base\_req} + \textbf{rlength}}{2^{\mathbf{e}}} \right\rceil \bmod 2^{20}$$

Rewritten as bit manipulations:

$$\mathbf{T} = \begin{cases} (\textbf{base\_req} + \textbf{rlength})[19 + \mathbf{e} : \mathbf{e}], & \text{if } (\textbf{base\_req} + \textbf{rlength})[\mathbf{e} - 1 : 0] = 0 \\ (\textbf{base\_req} + \textbf{rlength})[19 + \mathbf{e} : \mathbf{e}] + 1, & \text{otherwise} \end{cases}$$

otype The 24-bit **otype** field (concatenation of the two **otype** fields of Figure 5.4) corresponds directly to the **otype** bit vector but is defined only when the capability is sealed. These bits are not allocated in an unsealed capability, and the **otype** of an unsealed capability is 0.

The hardware computes **e** according to the following formula:

$$\mathbf{e} = \left\lceil \mathbf{plog_2} \left(\frac{(\textbf{rlength}) \cdot (1 + 2^{-6})}{2^{20}} \right) \right\rceil \text{where} \mathbf{plog_2}(x) = \begin{cases} 0, & \text{if } x < 1 \\ \mathbf{log_2}(x), & \text{otherwise} \end{cases}$$

which is equivalent to the following bit manipulation:

$$\mathbf{e} = \mathbf{idxMSNZ}((\textbf{rlength} + (\textbf{rlength} \gg 6)) \gg 19)$$

where:

- $\mathbf{idxMSNZ}(x)$ 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:

  $$\textbf{rlength} + 8\text{KiB} \leq 2^{\mathbf{e}+20}$$

  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:

  $$\mathbf{maxLength}(n) \geq \textbf{rlength} \text{where} \mathbf{maxLength}(n) = \left\lfloor \frac{2^{n+20}}{1 + 2^{-6}} \right\rfloor$$

5.9.2 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 only done 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:

$$inLimits = \begin{cases} I_{mid} < (R - \mathbf{a}_{mid} - 1), & \text{if } i \geq 0 \\ I_{mid} \geq (R - \mathbf{a}_{mid}) \wedge R \neq \mathbf{a}_{mid}, & \text{if } i < 0 \end{cases}$$

We must conservatively subtract one from the representable limit when we are incrementing upwards to account for any carry that may propagate up from the lower bits of the full pointer add. This is simplified to a single comparison and an equivalence check in our implementation.

One final test is required that ensure 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:

$$representable = (inRange \wedge inLimits) \vee (\mathbf{e} \geq 44)$$

5.9.3 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 5.7:

$$\begin{aligned} \mathbf{base}[63 : 20 + \mathbf{e}] &= \mathbf{a}[63 : 20 + \mathbf{e}] + \mathbf{c_b} \\ \mathbf{base}[19 + \mathbf{e} : \mathbf{e}] &= \mathbf{B} \\ \mathbf{base}[\mathbf{e} - 1 : 0] &= 0 \end{aligned}$$

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 5.7:

$$\begin{aligned} \mathbf{top}[64 : 20 + \mathbf{e}] &= \mathbf{a}[63 : 20 + \mathbf{e}] + \mathbf{c_t} \\ \mathbf{top}[19 + \mathbf{e} : \mathbf{e}] &= \mathbf{T} \\ \mathbf{top}[\mathbf{e} - 1 : 0] &= 0 \end{aligned}$$

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 5.7: Calculating $\mathbf{c_b}$ and $\mathbf{c_t}$

5.9.4 Alignment Requirements

Unsealed capabilities: Compressed capabilities impose alignment requirements on software if precise bounds are required. The calculation of **e** determines the alignment requirement (see section 5.9.1):

$$alignment = 2^{\mathbf{e}}$$

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:

$$\textbf{rlength} < 2^{\mathbf{e}} \cdot \tfrac{3}{4} MiB$$

So the conservative approximation of **e** can be computed as follows (or the precise version used from section 5.9.1), noting that **e** is also rounded up to ensure the bottom two bits are zero:

$$\mathbf{e} = \left\lceil plog_2 \left(\frac{\textbf{rlength}}{\frac{3}{4} MiB} \right) \right\rceil$$

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 $2^4 \cdot \frac{3}{4} MiB = 12 MiB$, etc. Page alignment (4KiB pages) is only required when objects are between 1GiB and 3GiB.

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 $4KiB$ (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 $2^4 \cdot \frac{3}{4} = 12MiB$ are aligned on 64KiB boundaries. Similarly, a region of length $1GiB$ to $3GiB$ will be $16MiB$ aligned.

5.10 Potential Future Changes to the CHERI ISA

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

- Migrate our reserved capabilities, such as **DDC**, **EPCC**, and so on, from our capability register file (e.g., **C0**) to control registers accessed via special `get` and `set` instructions. This would simplify control logic and critical paths in the pipeline. We have already introduced pseudo-ops to `get` and `set` the Default Data Capability, and should do so for others as well, easing the ISA-level transition. This would also make it easier to experiment with changes in register count.

- If all control registers are removed from the capability register file, consider using **C0** as a NULL capability register similar to the MIPS `$zero`, rather than using `CFromPtr` to construct NULL capabilities.

- Introduce an explicit `CMove` instruction, rather than using a pseudo-op of `CIncOffset`, which requires special casing handling of sealed capabilities in that instruction.

- Provide a `CCMove` conditional-move instruction in order to avoid branches when conditionally setting a capability register. This would mirror similar non-capability MIPS `movn` and `movz` instructions.

- 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.

- 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 user-defined permission bits on a sealed capability. The other permission bits on a sealed capability can be regarded as the permissions to access memory that the called protected subsystem will gain when `CCall` is invoked on the sealed capability; these should not be modifiable by the caller. On the other hand, the user-defined capability bits can be regarded as application-specific permissions that the caller has for the object that the sealed capability represents, and the caller might want to restrict these permissions before passing the sealed capability to another subsystem.

- Provide an "import capability" instruction that takes as arguments an untagged capability *cb* and a tagged capability *ct*. The result is *cb* with the tag bit set, provided that the

access granted by the result is a subset of the access granted by *ct*; an exception is raised otherwise. This instruction does not need to be privileged, because it does not grant its caller any access rights that its caller did not already have via *ct*. An example of where this instruction might be used is when the operating system swaps memory back in to memory from a storage device. The operating system could use this instruction to efficiently restore the tag bit on a capability; the current approach is that the operating system has to reconstruct the capability using a sequence of several instructions. This instruction might be named `CSetTag`.

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

- Provide a conditional branch instruction that branches depending on whether a capability is equal or not equal to NULL. Checking the tag bit with `CBTU` is not the same as checking for equality with NULL. In the current ISA, several instructions are needed to do the latter.

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

- The TLB load-capability bit selects between "allow capabilities to be loaded with tags" and "strip tags when loading into a capability register", avoiding an exception when asserting a pure data interpretation of memory. The capability load-capability permission, in contrast, defines throwing an exception if a tagged capability is loaded. It is desirable to modify the semantics of `CLC` to match TLB semantics by stripping the tag rather than throwing an exception, which would permit a capability-oblivious memory copy to read from a capability without the load-capability permission while ensuring no exception will be thrown due to a tag. This would also make implementation of an efficient tag-clearing memory copy straight forward: the copy routine would start by clearing the load-capability permission on the source, rather than needing to clear the tag on each individual loaded value.

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

- Allow `CReturn` to accept code/data capability arguments, which might be ignored for the time being – or simply make `CReturn` a variation on `CCall`. Consider adding a `CTailCall` variation that will not push an additional frame onto the trusted stack. Variations on `CCall` that specifically target asymmetric distrust (e.g., by eliding register clearing), may also offer opportunities for improved performance.

- Consider further the effects of combining general-purpose and capability register files, which would avoid adding a new register file, but make some forms of ABI compatibility more challenging.

- Provide a `CTailCall` instruction that combines `CCall` and `CReturn`. This is a performance optimization for tail-recursive functions that end by doing a `CCall` and then a `CReturn` of the result.

- Investigate a three-capability variation on object capabilities for the 128-bit version of CHERI. This would provide more bits to be used in describing classes and objects, and avoid requiring storing the object type in pointer bits.
- Extend the function of the system permission bit(s) to cover not just access to exception-related capabilities, but also other privileged aspects of the underlying ISA. For example, the permission bit could also control use of MIPS CP0 registers, TLB manipulation instructions, and exception-related instructions. This would allow compartmentalization of code within privileged rings on MIPS.
- Introduce a *Perm_Unseal* permission that can be used to unseal sealed capabilities of a type – without necessarily authorizing sealing.
- Introduce support for a userspace exception handler for `CCall` and `CReturn`, allowing more privileged user code (rather than kernel code) to implement the semantics of domain switching, provide memory for use in trusted stacks (if any), and so on. This would allow application environments to provide their own object models without needing to depend on highly privileged kernel code.

Chapter 6

CHERI Instruction-Set Reference

CHERI's instructions express a variety of operations affecting capability and general-purpose registers as well as memory access and control flow. A key design concern is *guarded manipulation*, which provides a set of constraints across all instructions that ensure monotonic non-increase in rights through capability manipulations. These instructions also assume, and specify, the presence of *tagged memory*, described in the previous chapter, which protects in-memory representations of capability values. Many further behaviors, such as reset state and exception handling (taken for granted in these instruction descriptions), are also described in the previous chapter.

The instructions fall into a number of categories: instructions to copy fields from capability registers into general-purpose registers so that they can be computed on, instructions for refining fields within capabilities, instructions for memory access via capabilities, instructions for jumps via capabilities, instructions for sealing capabilities, and instructions for capability invocation. In this chapter, we specify each instruction via both informal descriptions and pseudocode. To allow for more succinct pseudocode descriptions, we rely on a number of common definitions also described in this chapter.

6.1 Notation Used in Pseudocode

The pseudocode in the rest of this chapter uses the following notation:

- **not**, **or**, **and**, **true**, **false**

  Boolean operators.

- =, $\neq$

  Comparison.

- $+$, $-$, $*$, $/$, mod

  Arithmetic operators. Operations are over the (infinite range) mathematical integers, with no wrap-around on overflow. In cases where wrap-around on overflow is intended, this is explicitly indicated with a mod operator. All of the CHERI instructions can be implemented with finite-range (typically 64 bit) arithmetic; the hardware does not need to

implement bignum arithmetic. For example, a formal verification based on this specification could prove that every integer value computed will always fit within the finite range of the variable to which it is being assigned.

If $b > 0$, then $a \bmod b$ is in the range 0 to $b - 1$. (So it is an unsigned value).

- a^b

  Integer exponentiation. Defined only for $b \geq 0$.

- $<, \leq, >, \geq$

  Integer comparison.

- <*expression*>.<*field*>

  Selection of a field within a structure.

- <*expression*> **with** <*field*> $\leftarrow$ <*expression*>

  A structure with the named field replaced with another value.

- <*expression*>[<*expression*>]

  Selection of an element within an array.

- <*expression*> [<*expression*> .. <*expression*>]

  A slice of an array.

- <*expression*> **with** [<*expression*>] $\leftarrow$ <*expression*>

  An array with the selected element replaced with a new value.

- <*expression*> **with** [<*expression*> .. <*expression*>] $\leftarrow$ <*expression*>

  An array with the selected slice replaced with a new slice.

- <*expression*> $\cap$ <*expression*>

  The bitwise **and** of two arrays of booleans.

- <*expression*> $\cup$ <*expression*>

  The bitwise **or** of two arrays of booleans.

- $\emptyset$

  A boolean array in which every element is set to **false**.

- <*variable*> $\leftarrow$ <*expression*>

  Assignment of a new value to a mutable variable. If a mutable variable is a structure, a new value can be assigned to an individual field. If a mutable variable is an array. a new value can be assigned to an individual array element, or to a slice of the array.

- **if** ... **then** ... **else if** ... **else** ... **endif**

  Conditional branch.

or
and
not
$=, \neq, <, \leq, >, \geq$
with
$+, -$
$*, /, \mathrm{mod}$
a^b

Figure 6.1: Operator precedence in pseudocode

	128-bit	256-bit
`capability_size`	16	32
`max_otype`	$2^{24} - 1$	$2^{24} - 1$
`max_uperm`	3	15

Figure 6.2: Constants in pseudocode

- <*identifier*> (<*expression*> [, <*expression*>]*)

  Function invocation.

- <*identifier*> (<*expression*> [, <*expression*>]*)

  Procedure call.

The precedence of the operators used in the pseudocode is shown in table 6.1.

6.2 Common Constant Definitions

The constants used in the pseudo-code are show in table 6.2; their value depends on whether the 128-bit or 256-bit representation of capabilities is being used.

The null capability is defined as follows:

null_capability = int_to_cap(0)

6.3 Common Variable Definitions

The following variables are used in the pseudocode:

cb	:	Capability
cd	:	Capability
cs	:	Capability
ct	:	Capability
rd	:	Unsigned64
rs	:	Unsigned64
rt	:	Unsigned64
mask	:	Unsigned16
offset	:	Signed16

6.4 Common Function Definitions

The following functions are used in the pseudocode for more than one instruction, and are collected here for convenience. The & notation means use the number of a register, rather than its contents.

function REGISTER_INACCESSIBLE(cb)

return
not PCC.access_system_registers **and**
(&cb = KDC **or**
&cb = KCC **or**
&cb = KR1C **or**
&cb = KR2C **or**
&cb = EPCC)

end function

`to_signed64` converts an unsigned 64 bit integer into a signed 64 bit integer:

function TO_SIGNED64(x)
if x $< 2^{32}$ **then**

return x
else
return x $- 2^{64}$
end if
end function

`zero_extend` converts a sequence of bytes into an unsigned 64 bit integer. CHERI uses a big-endian byte ordering.

`sign_extend` converts a sequence of bytes into a signed 64 bit integer (i.e. if the most significant bit of the first byte is set, the result is negative).

`bytes_to_cap` converts a sequence of bytes into a capability.

`cap_to_bytes` converts a capability into a sequence of bytes.

`int_to_cap` converts a 64-bit integer into a capability that holds the integer in its offset field. It has the following properties:

- forall x: Unsigned64 int_to_cap(x).offset = x
- forall x: Unsigned64 int_to_cap(x).tag = **false**

- forall x: Unsigned64 int_to_cap(x).base = 0

The contents of other fields of `int_to_cap` depends on the capability compression scheme in use (e.g. 256-bit capabilities or 128-bit compressed capabilities). In particular, with 128 bit compressed capabilities, **length** is not always zero. The length of a capability created via `int_to_cap` is not semantically meaningful, and programs should not rely on it having any particular value.

`raise_c2_exception` is used when an instruction raises an exception. The following pseudocode omits the details of MIPS exception handling (branching to the address of the exception hander, etc.)

procedure RAISE_C2_EXCEPTION(cause, reg)
    cp0.cause ← 18
    capcause.reg ← &reg
    capcause.cause ← cause
    ...
end procedure

procedure RAISE_C2_EXCEPTION_NOREG(cause)
    cp0.cause ← 18
    capcause.reg ← 0xff
    capcause.cause ← cause
    ...
end procedure

`execute_branch` is used when an instruction branches to an address. The MIPS ISA includes branch delay slots, so the instruction in the branch delay slot will be executed before the branch is taken; this is omitted in the following pseudocode:

procedure EXECUTE_BRANCH(pc)
    ...
end procedure

`execute_branch_pcc` is used when an instruction branches to an address and changes PCC. The change to PCC does not take effect until after the instruction in the branch delay slot has been executed.

procedure EXECUTE_BRANCH_PCC(pc)
    ...
end procedure

6.5 Table of CHERI Instructions

Tables 6.3 and 6.4 list available capability coprocessor instructions.

6.6 Details of Individual Instructions

The following sections provide a detailed description of each CHERI ISA instructions. Each instruction description includes the following information:

- Instruction opcode format number

Mnemonic	Description
CGetBase	Move base to a general-purpose register
CGetOffset	Move offset to a general-purpose register
CGetLen	Move length to a general-purpose register
CGetTag	Move tag bit to a general-purpose register
CGetSealed	Move sealed bit to a general-purpose register
CGetPerm	Move permissions field to a general-purpose register
CGetType	Move object type field to a general-purpose register
CToPtr	Capability to pointer
CPtrCmp	Compare capability pointers
CClearRegs	Clear multiple registers
CIncBase	*Instruction removed*
CIncOffset	Increase offset
CSetBounds	Set bounds
CSetBoundsExact	Set bounds exactly
CSetLen	*Instruction removed*
CClearTag	Clear the tag bit
CAndPerm	Restrict permissions
CSetOffset	Set cursor to an offset from base
CGetPCC	Move PCC to capability register
CGetPCCSetOffset	Get PCC with new offset
CFromPtr	Create capability from pointer
CSub	Subtract capabilities
CSC	Store capability register
CLC	Load capability register
CL[BHWD][U]	Load via capability register
CS[BHWD]	Store via capability register
CLLC	Load linked capability via capability register
CLL[BHWD][U]	Load linked via capability register
CSC[BHWD]	Store conditional via capability register
CSCC	Store conditional capability via capability
CBTU	Branch if capability tag is unset
CBTS	Branch if capability tag is set
CJR	Jump capability register
CJALR	Jump and link capability register

Figure 6.3: Capability coprocessor instruction summary

Mnemonic	Description
CCheckPerm	Raise exception on insufficient permission
CCheckType	Raise exception if object types do not match
CSeal	Seal a capability
CUnseal	Unseal a sealed capability
CCall	Call into another security domain
CReturn	Return to the previous security domain
CGetCause	Move the capability exception cause register to a general-purpose register
CSetCause	Set the capability exception cause register

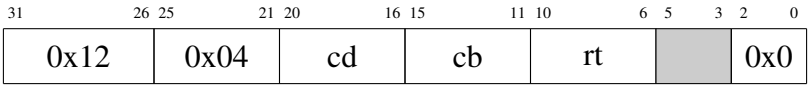
Figure 6.4: Capability coprocessor instruction summary, continued

- Assembly language syntax
- Bitwise figure of the instruction layout
- Text description of the instruction
- Pseudo-code description of the instruction
- Enumeration of any exceptions that the instruction can trigger

CAndPerm: Restrict Permissions

Format

CAndPerm cd, cb, rt



Description

Capability register *cd* is replaced with the contents of capability register *cb* with the **perms** field set to the bitwise AND of its previous value and the contents of general-purpose register *rt*.

Pseudocode

```
if register_inaccessible(cd) then
    raise_c2_exception(exceptionAccessSystem, cd)
else if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if cb.sealed then
    raise_c2_exception(exceptionSealed, cb)
else
    cd ← cb
    cd.perms ← cb.perms ∩ rt[0 .. 14]
    cd.uperms ← cb.uperms ∩ rt[15 .. 15 + max_uperm]
end if
```

TO DO: The size of the perms and uperms fields are variable.

Exceptions

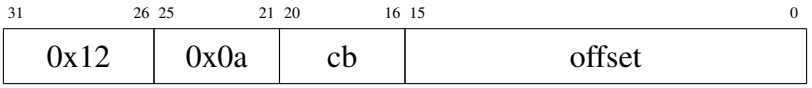
A coprocessor 2 exception is raised if:

- *cb* or *cd* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cb*.**tag** is not set.
- *cb*.**s** is set.

CBTS: Branch if Capability Tag is Set

Format

CBTS cb, offset



Description

Sets the **PC** to **PC** + 4**offset* + 4, where *offset* is sign extended, if *cb.tag* is set.

The instruction following the branch, in the delay slot, is executed before branching.

Pseudocode

```
if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else if cb.tag then
    execute_branch(PC + 4*sign_extend(offset) + 4)
end if
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.

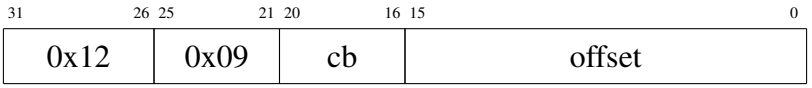
Notes

- Like all MIPS branch instructions, `CBTS` has a branch delay slot. The instruction after it will always be executed, regardless of whether the branch is taken or not.
- This instruction is intended to resemble the conditional branch instructions from the MIPS ISA. In particular, the shift left of the offset by 2 bits and adding 4 is the same as MIPS conditional branches.
- `CBTS` does not check that the branch is outside the range of **PCC**, but the bounds check performed during instruction fetch will catch out of range branches.

CBTU: Branch if Capability Tag is Unset

Format

CBTU cb, offset



Description

Sets the **PC** to **PC** + 4**offset* + 4, where *offset* is sign extended, if *cb.tag* is not set.

The instruction following the branch, in the delay slot, is executed before branching.

Pseudocode

```
if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else if not cb.tag then
    execute_branch(PC + 4*sign_extend(offset) + 4)
end if
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.

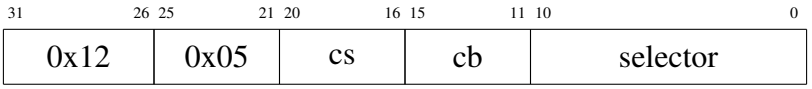
Notes

- Like all MIPS branch instructions, `CBTU` has a branch delay slot. The instruction after it will always be executed, regardless of whether the branch is taken or not.
- This instruction is intended to resemble the conditional branch instructions from the MIPS ISA. In particular, the shift left of the offset by 2 bits and adding 4 is the same as MIPS conditional branches.
- `CBTU` does not check that the branch is outside the range of **PCC**, but the bounds check performed during instruction fetch will catch out of range branches.

CCall: Call into Another Security Domain

Format

CCall cs, cb[, selector]



Description

`CCall` is used to make a call into a protected subsystem (which may have access to a different set of capabilities than its caller). *cs* contains a code capability for the subsystem to be called, and *cb* contains a sealed data capability that will be unsealed for use by the called subsystem. In terms of object-oriented programming, *cb* is a capability for an *object* and *cs* is a capability for the methods of the object's class.

This instruction can be implemented in a number of ways split over hardware and software; we have experimented with several. A simple implementation might have `CCall` throw a software exception, with all other behavior implemented via a software trap handler. A hybrid implementation could perform various checks in hardware, deferring only trusted stack manipulation (or other behaviors, such as asynchronous calling conventions) to the software trap handler. Further defensive coding conventions (beyond instruction semantics) may also sensibly be shifted to the exception handler in order to avoid redundancy – e.g., the clearing of the same registers to prevent leaks in either direction. A significant tension exists in the hardware optimization of this instruction between using a flexible calling convention and semantics versus exploiting hardware optimization opportunities. Authors of compilers or assembly language programs should not rely on `CCall` being implemented in any particular blend of hardware and software.

1. **PCC** (with its **offset** field set to the program counter (**PC**) + 4) is pushed onto the trusted system stack.

2. **IDC** is pushed onto the trusted system stack.

3. *cs* is unsealed and the result placed in **PCC**.

4. *cb* is unsealed and the result placed in **IDC**.

5. The program counter is set to *cs*.**offset**. (i.e. control branches to virtual address *cs*.**base** + *cs*.**offset**, because the program counter is relative to **PCC**.**base**).

The *selector* field can be omitted in assembly and, in the current version of the ISA, must be 0 if explicitly provided. Issuing a `CCall` instruction with any other value is undefined behavior.

TO DO: In CheriBSD, `CCall` also checks that the *global* bit is set, to prevent local capabilities being leaked via `CCall`. Some more discussion of this part of the security model is needed.

Pseudocode (software)

```
if register_inaccessible(cs) then
    raise_c2_exception(exceptionAccessSystem, cs)
else if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else if not cs.tag then
    raise_c2_exception(exceptionTag, cs)
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if not cs.sealed then
    raise_c2_exception(exceptionSealed, cs)
else if not cb.sealed then
    raise_c2_exception(exceptionSealed, cb)
else if cs.otype ≠ cb.otype then
    raise_c2_exception(exceptionType, cs)
else if not cs.perms.Permit_Execute then
    raise_c2_exception(exceptionPermitExecute, cs)
else if cb.perms.Permit_Execute then
    raise_c2_exception(exceptionPermitExecute, cb)
else if cs.offset≥ cs.length then
    raise_c2_exception(exceptionLength, cs)
else
    PCC.offset ← PC + 4
    TSS ← TSS − capability_size
    mem[TSS .. TSS + capability_size − 1] ← PCC
    tags[toTag(TSS)] ← PCC.tag
    TSS ← TSS − capability_size
    mem[TSS .. TSS + capability_size − 1] ← IDC
    tags[toTag(TSS)] ← IDC.tag
    PCC ← cs
    PCC.sealed ← false
    IDC ← cb
    IDC.sealed ← false
    PC ← cs.offset
end if
```

Pseudocode (hardware, no optimization)

```
raise_c2_exception(exceptionCall, cs)
```

Pseudocode (hardware, permissions check in hardware)

```
if register_inaccessible(cs) then
    raise_c2_exception(exceptionAccessSystem, cs)
else if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
```

else if not cs.tag **then**
    raise_c2_exception(exceptionTag, cs)
else if not cb.tag **then**
    raise_c2_exception(exceptionTag, cb)
else if not cs.sealed **then**
    raise_c2_exception(exceptionSealed, cs)
else if not cb.sealed **then**
    raise_c2_exception(exceptionSealed, cb)
else if cs.otype $\neq$ cb.otype **then**
    raise_c2_exception(exceptionType, cs)
else if not cs.perms.Permit_Execute **then**
    raise_c2_exception(exceptionPermitExecute, cs)
else if cb.perms.Permit_Execute **then**
    raise_c2_exception(exceptionPermitExecute, cb)
else if cs.offset$\geq$ cs.length **then**
    raise_c2_exception(exceptionLength, cs)
else
    raise_c2_exception(exceptionCall, cs)
end if

Exceptions

A coprocessor 2 exception will be raised so that the desired semantics can be implemented in a trap handler.

The capability exception code will be 0x05 and the handler vector will be 0x100 above the general purpose exception handler.

A further coprocessor 2 exception raised if:

- *cs* or *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cs*.**s** is not set.
- *cb*.**s** is not set.
- *cs*.**otype** $\neq$ *cb*.**otype**
- *cs*.**perms**.*Permit_Execute* is not set.
- *cb*.**perms**.*Permit_Execute* is set.
- *cs*.**offset** $\geq$ *cs*.**length**.
- The trusted system stack would overflow (i.e., if **PCC** and **IDC** were pushed onto the system stack, it would overflow the bounds of **TSC**).

Notes

- From the point of view of security, `CCall` needs to be an atomic operation (i.e. the caller cannot decide to just do some of it, because partial execution could put the system into an insecure state). From the point of view of hardware design, `CCall` needs to write two capabilities to memory, which might take more than one clock cycle. One possible way to satisfy both of these constraints is to make `CCall` cause a software trap, and the trap handler uses its access to **KCC** and **KDC** to implement `CCall`.

- Implementations may choose to restrict the register numbers that may be passed as *cs* and *cb* in order to avoid the need to decode the instruction and identify the register arguments. The software implementation in CheriBSD at the time of writing requires the *cs* be **C1**, and that *cb* be **C2**, consistent with the CHERI ABI.

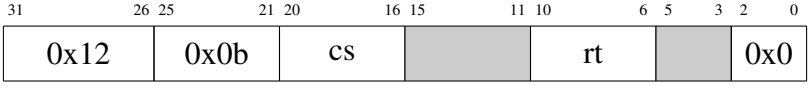
- In the current implementation, where `CCall` is implemented by the hardware raising an exception, `CCall` does not have a branch delay slot.

- The 10 bit selector in the `CCall` instruction is reserved for future iterations to provide different domain transition semantics, for example an asynchronous message send rather than a synchronous call.

CCheckPerm: Raise Exception on Insufficient Permission

Format

CCheckPerm cs, rt



Description

A exception is raised (and the capability cause set to "user defined permission violation") if there is a bit set in *rt* that is not set in *cs*.**perms** (i.e. *rt* describes a set of permissions, and an exception is raised if *cs* does not grant all of those permissions).

Pseudocode

if register_inaccessible(cs) **then**
    raise_c2_exception(exceptionAccessSystem, cs)
else if not cs.tag **then**
    raise_c2_exception(exceptionTag, cs)
else if cs.perms $\cap$ rt[0 .. 14] $\neq$ rt[0 .. 14] **then**
    raise_c2_exception(exceptionUserDefined, cs)
else if cs.uperms $\cap$ rt[15 .. 15 + max_uperm] $\neq$ rt[15 .. 15 + max_uperm] **then**
    raise_c2_exception(exceptionUserDefined, cs)
else if rt[16 + max_perm .. 63] $\neq$ 0 **then**
    raise_c2_exception(exceptionUserDefined, cs)
end if

TO DO: The size of the perms and uperms fields are variable. When we rationalize the exception codes, we might want to reconsider which exception is raised here.

Exceptions

A coprocessor 2 exception is raised if:

- *cs* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cs*.**tag** is not set.
- There is a bit that is set in *rt* and is not set in *cs*.**perms**.
- There is a bit that is set in *rt* and is not set in *cs*.**uperms**.

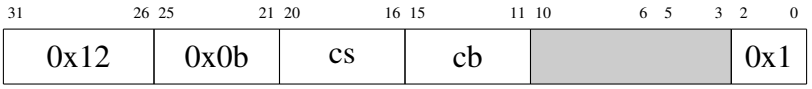
Notes

- If *cs*.**tag** is not set, then *cs* does not contain a capability, *cs*.**perms** might not be meaningful as a permissions field, and so a *tagViolation* exception is raised.
- This instruction can be used to check the permissions field of a sealed capability, so the instruction does not check *cs*.**s**.

CCheckType: Raise Exception if Object Types Do Not Match

Format

CCheckType cs, cb



Description

An exception is raised if *cs*.**otype** is not equal to *cb*.**otype**.

Pseudocode

if register_inaccessible(cs) **then**
    raise_c2_exception(exceptionAccessSystem, cs)
else if register_inaccessible(cb) **then**
    raise_c2_exception(exceptionAccessSystem, cb)
else if not cs.tag **then**
    raise_c2_exception(exceptionTag, cs)
else if not cb.tag **then**
    raise_c2_exception(exceptionTag, cb)
else if not cs.sealed **then**
    raise_c2_exception(exceptionSealed, cs)
else if not cb.sealed **then**
    raise_c2_exception(exceptionSealed, cb)
else if cs.otype $\neq$ cb.otype **then**
    raise_c2_exception(exceptionType, cs)
end if

Exceptions

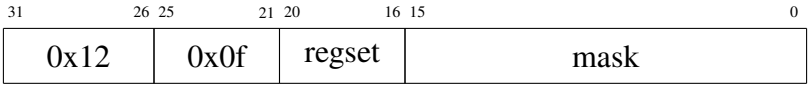
A coprocessor 2 exception is raised if:

- *cs* or *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cs*.**tag** is not set.
- *cb*.**tag** is not set.
- *cs*.**s** is not set.
- *cs*.**s** is not set.
- *cs*.**otype**$\neq$ *cb*.**otype**.

CClearRegs: Clear Multiple Registers

Format

ClearLo mask
ClearHi mask
CClearLo mask
CClearHi mask



Description

The registers in the target register set, *regset*, corresponding to the set bits in the immediate *mask* field are cleared. That is, if bit 0 of *mask* is set, then the lowest numbered register in *regset* is cleared, and so on. The following values are defined for the *regset* field:

Mnemonic	*regset*	Affected registers
ClearLo	0	**R0–R15**
ClearHi	1	**R16–R31**
CClearLo	2	**C0–C15**
CClearHi	3	**C16–C31**
FPClearLo	4	**F0–F15**
FPClearHi	5	**F16–F31**

For general purpose registers clearing means setting to zero, and for capability registers to the NULL capability (**base** = 0, **length** = 0, **offset** = 0, all permissions false).

FIXME: The null capability depends on the compression scheme.

Exceptions

- A Reserved Instruction exception is raised for unknown or unimplemented values of *regset*.
- `CClearHi` raises a CP2 exception (with *CapCause.ExcCode* set to *Access_System_Registers Violation*) if one or more of the capability registers to be cleared are reserved registers, and **PCC**.*Access_System_Registers* is not set.
- `CClearLo` and `CClearHi` raise a coprocessor unusable exception if the capability co-processor is disabled.
- `FPClearLo` and `FPClearHi` raise a coprocessor unusable exception if the floating point unit is disabled.

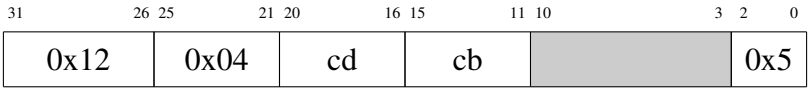
Notes

- These instructions are designed to accelerate the register clearing that is required for secure domain transitions. It is expected that they can be implemented efficiently in hardware using a single 'valid' bit per register that is cleared by the ClearRegs instruction and set on any subsequent write to the register.
- **TO DO**: The mnemonic for the general-purpose register instruction doesn't make it very clear what the instruction does. It would be nice to have a more descriptive mnemonic.

CClearTag: Clear the Tag Bit

Format

CClearTag cd, cb



Description

Capability register *cd* is replaced with the contents of *cb*, with the tag bit cleared.

Pseudocode

```
if register_inaccessible(cd) then
    raise_c2_exception(exceptionAccessSystem, cd)
else if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else
    cd ← cb with tag ← false
end if
```

Exceptions

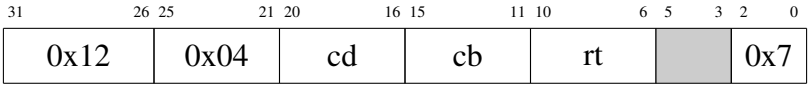
A coprocessor 2 exception is raised if:

- *cb* or *cd* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.

CFromPtr: Create Capability from Pointer

Format

CFromPtr cd, cb, rt



Description

rt is a pointer using the C-language convention that a zero value represents the NULL pointer. If *rt* is zero, then *cd* will be the NULL capability (tag bit not set, all other bits also not set). If *rt* is non-zero, then *cd* will be set to *cb* with the **offset** field set to *rt*.

Pseudocode

if register_inaccessible(cd) **then**
    raise_c2_exception(exceptionAccessSystem, cd)
else if register_inaccessible(cb) **then**
    raise_c2_exception(exceptionAccessSystem, cb)
else if rt = 0 **then**
    cd ← null_capability
else if not cb.tag **then**
    raise_c2_exception(exceptionTag, cb)
else if cb.sealed **then**
    raise_c2_exception(exceptionSealed, cb)
else if not representable(cb.sealed, cb.base, cb.length, rt) **then**
    cd ← int_to_cap((cb.base + rt) $\mathrm{mod} 2^{64}$)
else
    cd ← cb **with** offset ← rt
end if

Exceptions

A coprocessor 2 exception is raised if:

- *cb* or *cd* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cb*.**tag** is not set and $rt \neq 0$.
- *cb*.**s** is set and $rt \neq 0$.

Notes

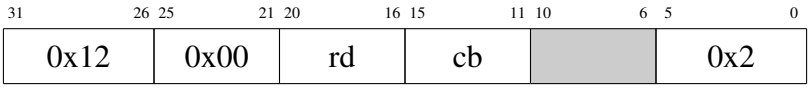
- `CSetOffset` doesn't raise an exception if the tag bit is unset, so that it can be used to implement the *intcap_t* type. `CFromPtr` raises an exception if the tag bit is unset: although it would not be a security violation to to allow it, it is an indication that the program is in error.

- The encodings of the NULL capability are chosen so that zeroing memory will set a capability variable to NULL. This holds true for compressed capabilities as well as the 256-bit version.

CGetBase: Move Base to a General-Purpose Register

Format

CGetBase rd, cb



Description

General-purpose register *rd* is set equal to the **base** field of capability register *cb*.

Pseudocode

```
if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else
    rd ← cb.base
end if
```

Exceptions

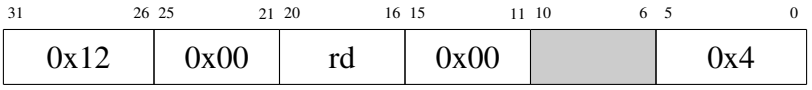
A coprocessor 2 exception is raised if:

- *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.

CGetCause: Move the Capability Exception Cause Register to a General-Purpose Register

Format

CGetCause rd



Description

General-purpose register *rd* is set equal to the capability cause register.

Pseudocode

```
if not PCC.perms.Access_System_Registers then
    raise_c2_exception_noreg(exceptionAccessSystem)
else
    rd[0 .. 7] ← capcause.reg
    rd[8 .. 15] ← capcause.cause
    rd[16 .. 63] ← 0
end if
```

Exceptions

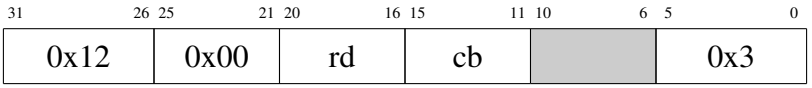
A coprocessor 2 exception is raised if:

- **PCC**.**perms**.*Access_System_Registers* is not set.

CGetLen: Move Length to a General-Purpose Register

Format

CGetLen rd, cb



Description

General-purpose register *rd* is set equal to the **length** field of capability register *cb*.

Pseudocode

if register_inaccessible(cb) **then**
    raise_c2_exception(exceptionAccessSystem, cb)
else if cb.length $\geq 2^{64}$ **then**
    rd $\leftarrow 2^{64} - 1$
else
    rd $\leftarrow$ cb.length
end if

Exceptions

A coprocessor 2 exception is raised if:

- *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.

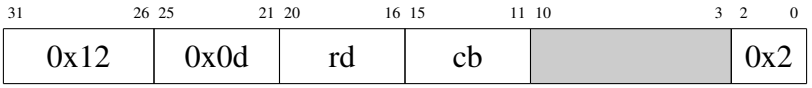
Notes

- With the 256-bit representation of capabilities, **length** is a 64-bit unsigned integer and can never be greater than $2^{64} - 1$. With the 128-bit compressed representation of capabilities, the result of decompressing the length can be 2^{64}; `CGetLen` will return the maximum value of $2^{64} - 1$ in this case.

CGetOffset: Move Offset to a General-Purpose Register

Format

CGetOffset rd, cb



Description

General-purpose register *rd* is set equal to the **offset** fields of capability register *cb*.

Pseudocode

```
if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else
    rd ← cb.offset
end if
```

Exceptions

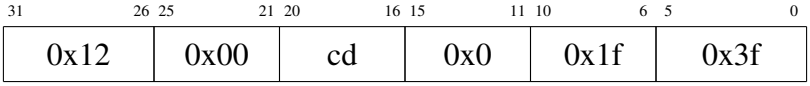
A coprocessor 2 exception is raised if:

- *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.

CGetPCC: Move PCC to Capability Register

Format

CGetPCC cd



Description

Capability register *cd* is set equal to the **PCC**, with cd.**offset** set equal to **PC**.

Pseudocode

```
if register_inaccessible(cd) then
    raise_c2_exception(exceptionAccessSystem, cd)
else
    cd ← PCC with offset ← PC
end if
```

Exceptions

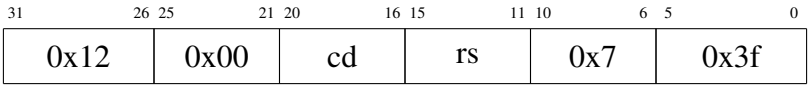
A coprocessor 2 exception is raised if:

- *cd* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.

CGetPCCSetOffset: Get PCC with new offset

Format

CGetPCCSetOffset cd, rs



Description

Capability register *cd* is set equal to the **PCC**, with cd.**offset** set equal to *rs*.

Pseudocode

if register_inaccessible(cd) **then**
    raise_c2_exception(exceptionAccessSystem, cd)
else if not representable(PCC.sealed, PCC.base, PCC.length, rs) **then**
    cd ← int_to_cap((PCC.base + rs) $\mod 2^{64}$)
else
    cd ← PCC **with** offset ← rs
end if

Exceptions

A coprocessor 2 exception is raised if:

- *cd* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.

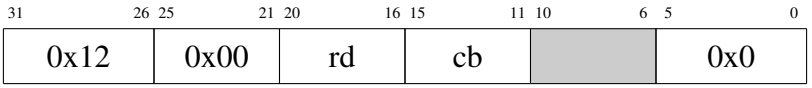
Notes

- This instruction is a peformance optimization; a similar effect can be achieved with `CGetPCC` followed by `CSetOffset`.

CGetPerm: Move Permissions Field to a General-Purpose Register

Format

CGetPerm rd, cb



Description

The least significant 31 bits (bits 0 to 30) of general-purpose register *rd* are set equal to the **perms** field of capability register *cb*. The other bits of *rd* are set to zero.

Pseudocode

if register_inaccessible(cb) **then**
    raise_c2_exception(exceptionAccessSystem, cb)
else
    rd ← 0 **with** [0 .. 14] ← cb.perms, [15 .. 15 + max_uperm] ← cb.uperms
end if
TO DO: The size of the perms and uperms fields are variable.

Exceptions

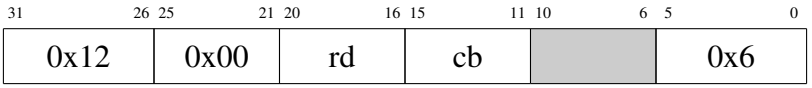
A coprocessor 2 exception is raised if:

- *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.

CGetSealed: Move Sealed Bit to a General-Purpose Register

Format

CGetSealed rd, cb



Description

The low-order bit of *rd* is set to *cb*.**s**. All other bits of *rd* are cleared.

Pseudocode

```
if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else
    rd[0] ← cb.sealed
    rd[1 .. 63] ← 0
end if
```

Exceptions

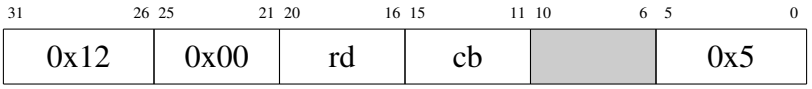
A coprocessor 2 exception is raised if:

- *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.

CGetTag: Move Tag Bit to a General-Purpose Register

Format

CGetTag rd, cb



Description

The low bit of *rd* is set to the tag value of *cb*. All other bits are cleared.

Pseudocode

```
if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else
    rd[0] ← cb.tag
    rd[1 .. 63] ← 0
end if
```

Exceptions

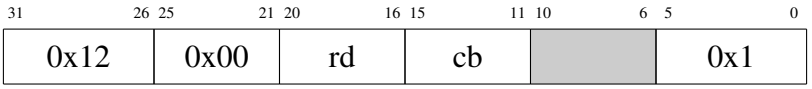
A coprocessor 2 exception is raised if:

- *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.

CGetType: Move Object Type Field to a General-Purpose Register

Format

CGetType rd, cb



Description

General-purpose register *rd* is set equal to the **otype** field of capability register *cb*.

Pseudocode

```
if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else
    rd[0 .. 23] ← cb.otype
    rd[24 .. 63] ← 0
end if
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.

CIncBase: *Instruction Removed*

Format

Instruction removed

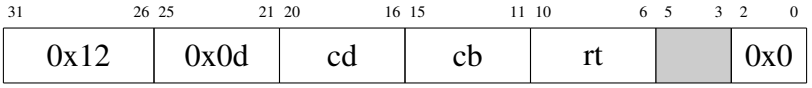
Description

This instruction has been removed from the CHERI ISA in favor of `CSetBounds`. The opcode remains reserved.

CIncOffset: Increase Offset

Format

CIncOffset cd, cb, rt
CMove cd, cb



Description

Capability register *cd* is set equal to capability register *cb* with its **offset** field replaced with *cb*.**offset** + *rt*.

If capability compression is in use, and the requested **base**, **length** and **offset** cannot be represented exactly, then *cd*.**tag** is cleared, *cd*.**base** and *cd*.**length** are set to zero, *cd*.**perms** is cleared, and *cd*.**offset** is set equal to *cb*.**base** + *cb*.**offset** + *rt*.

Pseudocode

if register_inaccessible(cd) **then**
    raise_c2_exception(exceptionAccessSystem, cd)
else if register_inaccessible(cb) **then**
    raise_c2_exception(exceptionAccessSystem, cb)
else if cb.tag **and** cb.sealed **and** rt $\neq 0$ **then**
    raise_c2_exception(cb, exceptionSealed)
else if not representable(cb.sealed, cb.base, cb.length, (cb.offset + rt) $\mod 2^{64}$) **then**
    cd $\leftarrow$ int_to_cap((cb.base + cb.offset + rt) $\mod 2^{64}$)
else
    cd $\leftarrow$ cb **with** offset $\leftarrow$ (cb.offset + rt) $\mod 2^{64}$
end if

Exceptions

A coprocessor 2 exception is raised if:

- *cb* or *cd* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cb*.**tag** and *cb*.**s** are both set.

Notes

- For security reasons, `CIncOffset` must not change the offset of a sealed capability.
- As a special case, we allow `CIncOffset` with an offset of zero to work on sealed capabilities; this is so that `CIncOffset` can be used as a capability move instruction.

- If the tag bit is not set, and the offset is being used to hold an integer, then `CIncOffset` should still increment the offset. This is so that `CIncOffset` can be used to implement increment of a `intcap_t` type. In this case, the bit in the position corresponding to the sealed bit will typically not be set.

- If the tag bit is not set, the capability register contains arbitrary non-capability data, and the bit in the position corresponding to the sealed bit is set, we allow the operation to succeed. (Although the effect on the non-capability data will depend on which binary representation of capabilities is being used).

- If the tag bit is not set, and capability compression is in use, the arbitrary data in *cb* might not decompress to sensible values of the **base** and **length** fields, and there is no guarantee that retaining these values of **base** and **length** while changing **offset** will result in a representable value.

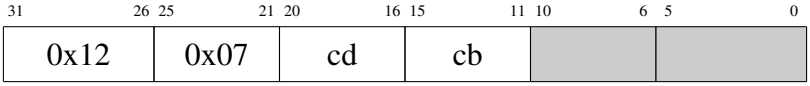
  From a software perspective, the requirement is that incrementing **offset** on an untagged capability will work if **base** and **length** are zero. (This is how integers, and pointers that have lost precision, will be represented). If **base** and **length** have non-zero values (or *cb* cannot be decompressed at all), then the values of **base** and **length** after this instruction are **UNPREDICTABLE**.

- In assembly language, `CMove` *cd*, *cb* is a pseudo-instruction that the assembler converts to `CIncOffset` *cd*, *cb*, *$zero*.

CJALR: Jump and Link Capability Register

Format

CJALR cb, cd



Description

The current **PCC** (with an offset of the current **PC** + 8) is saved in *cd*. **PCC** is then loaded from capability register *cb* and **PC** is set from its offset.

Pseudocode

```
if register_inaccessible(cd) then
    raise_c2_exception(exceptionAccessSystem, cd)
else if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if cb.sealed then
    raise_c2_exception(exceptionSealed, cb)
else if not cb.perms.Permit_Execute then
    raise_c2_exception(exceptionPermitExecute, cb)
else if not cb.perms.Global then
    raise_c2_exception(exceptionGlobal, cb)
else if cb.offset + 4 > cb.length then
    raise_c2_exception(exceptionLength, cb)
else if align_of(cb.base + cb.offset) < 4 then
    raise_exception(exceptionAdEL)
else
    cd ← PCC with offset← PC + 8
    execute_branch_pcc(cb.offset, cb)
end if
```

Exceptions

A coprocessor 2 exception will be raised if:

- *cd* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cb.***tag** is not set.

- *cb.***s** is set.
- *cb.***perms**.*Permit_Execute* is not set.
- *cb.***perms**.*Global* is not set.
- *cb.***offset** + 4 is greater than *cb.***length**.

An address error exception will be raised if

- *cb.***base** + *cb.***offset** is not 4-byte word aligned.

Notes

- `cjalr` has a branch delay slot.
- The change to **PCC** does not take effect until the instruction in the branch delay slot has been executed.

CJR: Jump Capability Register

Format

CJR cb

Description

PCC is loaded from *cb*, and **PC** is loaded from *cb.***offset**.

Pseudocode

```
if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if cb.sealed then
    raise_c2_exception(exceptionSealed, cb)
else if not cb.perms.Permit_Execute then
    raise_c2_exception(exceptionPermitExecute, cb)
else if not cb.perms.Global then
    raise_c2_exception(exceptionGlobal, cb)
else if cb.offset + 4 > cb.length then
    raise_c2_exception(exceptionLength, cb)
else if align_of(cb.base + cb.offset) < 4 then
    raise_exception(exceptionAdEL)
else
    execute_branch_pcc(cb.offset, cb)
end if
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cb*.**tag** is not set.
- *cb.***s** is set.
- *cb.***perms**.*Permit_Execute* is not set.
- *cb.***perms**.*Global* is not set.
- Register *cb.***offset** + 4 is greater than *cb.***length**.

An address error exception is raised if:

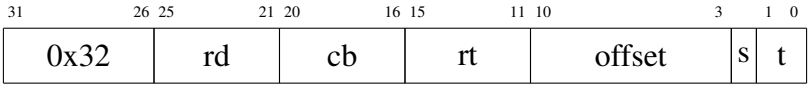
- *cb.***base** + *cb.***offset** is not 4-byte word aligned.

*cb.***base** and *cb.***length** are treated as unsigned integers, and the result of the addition does not wrap around (i.e., an exception is raised if *cb.***base**+*cb.***offset** is greater than maxaddr).

Load via Capability Register

Format

CLB rd, rt, offset(cb)
CLH rd, rt, offset(cb)
CLW rd, rt, offset(cb)
CLD rd, rt, offset(cb)
CLBU rd, rt, offset(cb)
CLHU rd, rt, offset(cb)
CLWU rd, rt, offset(cb)
CLBR rd, rt(cb)
CLHR rd, rt(cb)
CLWR rd, rt(cb)
CLDR rd, rt(cb)
CLBUR rd, rt(cb)
CLHUR rd, rt(cb)
CLWUR rd, rt(cb)
CLBI rd, offset(cb)
CLHI rd, offset(cb)
CLWI rd, offset(cb)
CLDI rd, offset(cb)
CLBUI rd, offset(cb)
CLHUI rd, offset(cb)
CLWUI rd, offset(cb)



Purpose

Loads a data value via a capability register, and extends the value to fit the target register.

Description

The lower part of general-purpose register *rd* is loaded from the memory location specified by *cb*.**base** + *cb*.**offset** + *rt* + $2^t * offset$. Capability register *cb* must contain a valid capability that grants permission to load data.

The size of the value loaded depends on the value of the *t* field:

0 byte (8 bits)

1 halfword (16 bits)

2 word (32 bits)

3 doubleword (64 bits)

The extension behavior depends on the value of the *s* field: 1 indicates sign extend, 0 indicates zero extend. For example, CLWU is encoded by setting *s* to 0 and *t* to 2, CLB is encoded by setting *s* to 1 and *t* to 0.

Pseudocode

```
if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if cb.sealed then
    raise_c2_exception(exceptionSealed, cb)
else if not cb.perms.Permit_Load then
    raise_c2_exception(exceptionPermitLoad, cb)
else
    if t = 0 then
        size ← 1
    else if t = 1 then
        size ← 2
    else if t = 2 then
        size ← 4
    else if t = 3 then
        size ← 8
    end if
    cursor ← (cb.base + cb.offset) mod 2^64
    addr ← (cursor + rt + size * sign_extend(offset)) mod 2^64
    if addr + size > cb.base + cb.length then
        raise_c2_exception(exceptionLength, cb)
    else if addr < cb.base then
        raise_c2_exception(exceptionLength, cb)
    else if align_of(addr) < size then
        raise_exception(exceptionAdEL)
    else if s = 0 then
        rd ← zero_extend(mem[addr .. addr + size − 1])
    else
        rd ← sign_extend(mem[addr .. addr + size − 1])
    end if
end if
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cb*.**tag** is not set.
- *cb*.**s** is set.
- *cb*.**perms**.*Permit_Load* is not set.

- $addr + size > cb.\textbf{base} + cb.\textbf{length}$

  NB: The check depends on the size of the data loaded.

- $addr < cb.\textbf{base}$

An AdEL exception is raised if *addr* is not correctly aligned.

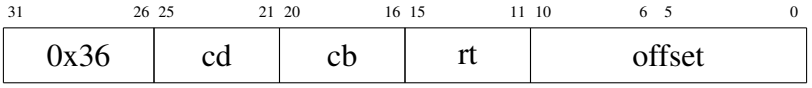
Notes

- This instruction reuses the opcode from the Load Word to Coprocessor 2 (`LWC2`) instruction in the MIPS Specification.
- *rt* is treated as an unsigned integer.
- *offset* is treated as a signed integer.
- BERI1 has a compile-time option to allow unaligned loads and stores. If BERI1 is built with this option, an unaligned load will only raise an exception if it crosses a cache line boundary.

CLC: Load Capability Register

Format

CLC cd, rt, offset(cb)
CLCR cd, rt(cb)
CLCI cd, offset(cb)



Description

Capability register *cd* is loaded from the memory location specified by *cb.***base** + *cb.***offset** + *rt* + *offset*. Capability register *cb* must contain a capability that grants permission to load capabilities. The virtual address *cb.***base** + *cb.***offset** + *rt* + *offset* must be *capability_size* aligned.

The bit in the tag memory corresponding to *cb.***base** + *cb.***offset** + *rt* + *offset* is loaded into the tag bit associated with *cd*.

Pseudocode

```
if register_inaccessible(cd) then
    raise_c2_exception(exceptionAccessSystem, cd)
else if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if cb.sealed then
    raise_c2_exception(exceptionSealed, cb)
else if not cb.perms.Permit_Load_Capability then
    raise_c2_exception(exceptionPermitLoadCapability, cb)
end if
cursor ← (cb.base + cb.offset) mod 2^64
addr ← cursor + rt + 16 * sign_extend(offset)
if addr + capability_size > cb.base + cb.length then
    raise_c2_exception(exceptionLength, cb)
else if addr < cb.base then
    raise_c2_exception(exceptionLength, cb)
else if align_of(addr) < capability_size then
    raise_exception(exceptionAdEL)
else if TLB(addr).L then
    cd ← bytes_to_cap(mem[addr .. addr + cap_size − 1]) with tag ← false
else
    cd ← bytes_to_cap(mem[addr .. addr + cap_size − 1] with tag ← tags[toTag(addr)]
end if
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb* or *cd* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and the corresponding bit in **PCC**.**perms** is not set.
- *cb*.**tag** is not set.
- *cb*.**s** is set.
- *cb*.**perms**.*Permit_Load_Capability* is not set.
- *addr* + *capability_size* $>$ *cb*.**base** $+$ *cb*.**length**.
- *addr* $<$ *cb*.**base**.

An address error during load (AdEL) exception is raised if:

- The virtual address *addr* is not *capability_size* aligned.

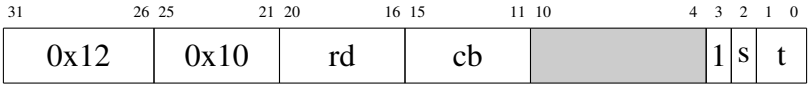
Notes

- This instruction reuses the opcode from the Load Doubleword to Coprocessor 2 (`LDC2`) instruction in the MIPS Specification.
- *offset* is interpreted as a signed integer.
- The `CLCI` mnemonic is equivalent to `CLC` with *cb* being the zero register ($zero). The `CLCR` mnemonic is equivalent to `CLC` with *offset* set to zero.
- Although the *capability_size* can vary, the offset is always in multiples of 16 bytes (128 bits).

Load Linked via Capability Register

Format

CLLB rd, cb
CLLH rd, cb
CLLW rd, cb
CLLD rd, cb
CLLBU rd, cb
CLLHU rd, cb
CLLWU rd, cb



Description

`CLL[BHWD][U]` and `CSC[BHWD]` are used to implement safe access to data shared between different threads. The typical usage is that `CLL[BHWD][U]` is followed (an arbitrary number of instructions later) by `CSC[BHWD]` to the same address; the `CSC[BHWD]` will only succeed if the memory location that was loaded by the `CLL[BHWD][U]` has not been modified.

The exact conditions under which `CSC[BHWD]` fails are implementation dependent, particularly in multicore or multiprocessor implementations). The following pseudocode is intended to represent the security semantics of the instruction correctly, but should not be taken as a definition of the CPU's memory coherence model.

Pseudocode

```
addr ← cb.base + cb.offset
size ← 2^t
if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if cb.sealed then
    raise_c2_exception(exceptionSealed, cb)
else if not cb.perms.Permit_Load then
    raise_c2_exception(exceptionPermitLoad, cb)
else if addr + size > cb.base + cb.length then
    raise_c2_exception(exceptionLength, cb)
else if addr < cb.base then
    raise_c2_exception(exceptionLength, cb)
else if align_of(addr) < size then
    raise_exception(exceptionAdEL)
else
    if s = 0 then
        rd ← zero_extend(mem[addr .. addr + size − 1])
    else
```

```
        rd ← sign_extend(mem[addr .. addr + size − 1])
      end if
      linkedFlag ← true
    end if
```

Exceptions

A coprocessor 2 exception is raised if:

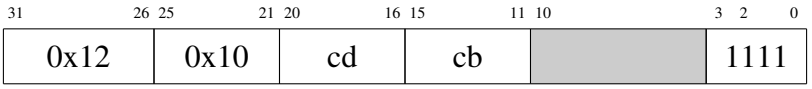
- *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cb*.**tag** is not set.
- *cb* is sealed.
- *cb*.**perms**.*Permit_Load* is not set.
- $addr + size > cb.\textbf{base} + cb.\textbf{length}$
- $addr < cb.\textbf{base}$

An AdEL exception is raised if *addr* is not correctly aligned.

CLLC: Load Linked Capability via Capability

Format

CLLC cd, cb



Pseudocode

addr ← (cb.base + cb.offset) $\mathrm{mod} 2^{64}$
if register_inaccessible(cd) **then**
    raise_c2_exception(exceptionAccessSystem, cd)
else if register_inaccessible(cb) **then**
    raise_c2_exception(exceptionAccessSystem, cb)
else if not cb.tag **then**
    raise_c2_exception(exceptionTag, cb)
else if cb.sealed **then**
    raise_c2_exception(exceptionSealed, cb)
else if not cb.perms.Permit_Load_Capability **then**
    raise_c2_exception(exceptionPermitLoadCapability, cb)
else if addr + capability_size > cb.base + cb.length **then**
    raise_c2_exception(exceptionLength, cb)
else if addr < cb.base **then**
    raise_c2_exception(exceptionLength, cb)
else if align_of(addr) < capability_size **then**
    raise_exception(exceptionAdEL)
else if TLB(addr).L **then**
    cd ← bytes_to_cap(mem[addr .. addr + capability_size]) **with** tag ← **false**
    linkedFlag ← **true**
else
    cd ← bytes_to_cap(mem[addr .. addr + capability_size]) **with** tag ← tags[toTag(addr)]
    linkedFlag ← **true**
end if

Exceptions

A coprocessor 2 exception is raised if:

- *cb* or *cd* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cb*.**tag** is not set.
- *cb* is sealed.
- *cb*.**perms**.*Permit_Load_Capability* is not set.

- $addr + capability\_size > cb.\textbf{base} + cb.\textbf{length}$
- $addr < cb.\textbf{base}$

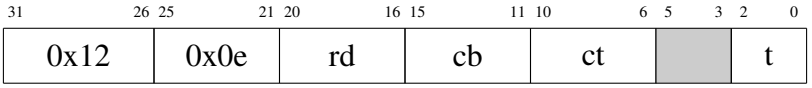
An AdEL exception is raised if:

- *addr* is not capability aligned.

CPtrCmp: CEQ, CNE, CL[TE][U], CEXEQ: Capability Pointer Compare

Format

CEQ rd, cb, ct
CNE rd, cb, ct
CLT rd, cb, ct
CLE rd, cb, ct
CLTU rd, cb, ct
CLEU rd, cb, ct
CEXEQ rd, cb, ct



Description

Capability registers *cb* and *ct* are compared, and the result of the comparison is placed in general purpose register *rd*. The rules for comparison are as follows:

- A capability with the **tag** bit unset is less than any capability with the **tag** bit set.
- Otherwise, the result of comparison is the result of comparing (**base** + **offset**) $\mathrm{mod} 2^{64}$ for the two capabilities. Numerical comparison is signed for `CLT` and `CLE`, and unsigned for `CLTU` and `CLEU`.
- `CExEq` compares all the fields of the two capabilities, including **tag** and the bits that are reserved for future use.

This instruction can be used to compare capabilities so that capabilities can replace pointers in C executables.

Mnemonic	*t*	Comparison
`CEQ`	0	=
`CNE`	1	$\neq$
`CLT`	2	$<$ (signed)
`CLE`	3	$\leq$ (signed)
`CLTU`	4	$<$ (unsigned)
`CLEU`	5	$\leq$ (unsigned)
`CEXEQ`	6	all fields are equal

Pseudocode

if t $>$ 5 **then**
    raise_exception(reservedInstruction)
else if register_inaccessible(cb) **then**

```
        raise_c2_exception(exceptionAccessSystem, cb)
    else if register_inaccessible(ct) then
        raise_c2_exception(exceptionAccessSystem, ct)
    else
        if cb.tag ≠ ct.tag then
            equal ← false
            if cb.tag then
                less ← false
                signed_less ← false
            else
                less ← true
                signed_less ← true
            end if
        else
            cursor1 ← (cb.base + cb.offset) mod 2^64
            cursor2 ← (ct.base + ct.offset) mod 2^64
            equal ← cursor1 = cursor2
            less ← cursor1 < cursor2
            signed_less ← to_signed64(cursor1) < to_signed64(cursor2)
        end if
        if t = 0 then
            rd ← equal
        else if t = 1 then
            rd ← not equal
        else if t = 2 then
            rd ← signed_less
        else if t = 3 then
            rd ← signed_less or equal
        else if t = 4 then
            rd ← less
        else if t = 5 then
            rd ← less or equal
        else if t = 6 then
            rd ← cb = ct
        else
            raise_exception(exceptionReservedInstruction)
        end if
    end if
```

Exceptions

A reserved instruction exception is raised if

- *t* does not correspond to comparison operation whose meaning has been defined.

A coprocessor 2 exception will be raised if:

- *cb* or *ct* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.

Notes

- `cltu` can be used by a C compiler to compile code that compares two non-null pointers (e.g., to detect whether a pointer to a character within a buffer has reached the end of the buffer). When two pointers to addresses within the same object (e.g., to different offsets within an array) are compared, the pointer to the earlier part of the object will be compared as less. (Signed comparison would also work as long as the object did not span address 2^{63}; the MIPS address space layout makes it unlikely that objects spanning 2^{63} will exist in user-space C code).

- Although the ANSI C standard does not specify whether a NULL pointer is less than or greater than a non-NULL pointer (clearly, they must not be equal), the comparison instructions have been designed so that when C pointers are represented by capabilities, NULL will be less than any non-NULL pointer.

- A C compiler may also use these instructions to compare two values of type `uintptr_t` that have been obtained by casting from an integer value. If the cast is compiled as a `CFromPtr` of zero followed by `CSetOffset` to the integer value, the result of `CPtrCmp` will be the same as comparing the original integer values, because `CFromPtr` will have set **base** to zero. Signed and unsigned capability comparison operations are provided so that both signed and unsigned integer comparisons can be performed on capability registers.

- A program could use pointer comparison to determine the value of **base**, by setting **offset** to different values and testing which values cause **base** + **offset** to wrap around and be less than **base** + a zero offset. This is not an attack against a security property of the ISA, because **base** is not a secret.

- One possible way in which garbage collection could be implemented is for the garbage collector to move an object and fix up all capabilities that refer to it. If there are appropriate restrictions on which capabilities the program has to start with, the garbage collector can be sure that the program does not have any references to the object stored as integers, and so can know that it is safe to move the object. With this type of garbage collection, comparing pointers by extracting their base and offset with `CGetBase` and `CGetOffset` and comparing the integer values is not guaranteed to work, because the garbage collector might have moved the object part-way through. `CPtrCmp` is atomic, and so will work in this scenario.

- Some compilers may make the optimization that if a check for ($a = b$) has succeeded, then b can be replaced with a without changing the semantics of the program. This optimization is not valid for the comparison performed by `CEq`, because two capabilities can point to the same place in memory but have different bounds, permissions etc. and so not be interchangeable. The `CExEq` instruction is provided for when a test for semantic equivalence of capabilities is needed; it compares all the fields, even the ones that are reserved for future use.

- Mathematically, `CEq` divides capabilities into *equivalence classes*, and the signed or unsigned comparison operators provide a *total ordering* on these equivalence classes. `CExEq` also divides capabilities into equivalence classes, but these are not totally ordered: two capabilities can be unequal according to `CExEq`, and also neither less or greater according to *CLT* (e.g., if they have the same **base** + **offset**, but different **length**).

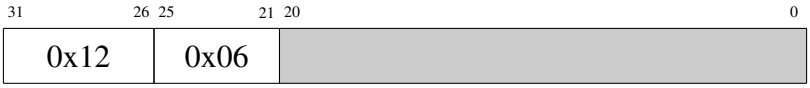
- There is an outstanding issue: when capability compression is in use, does `CExEq` compare the compressed representation or the uncompressed capability? There might be a difference between the two if there are multiple compressed representations that decompress to the same thing. If **tag** is false, then then capability register might contain non-capability data (e.g., an integer, or a string) and it might not decompress to anything sensible. Clearly in this case the in-memory compressed representation should be compared bit for bit. Is it also acceptable to compare the compressed representations when **tag** is true? This might lead to two capabilities that are sematically equivalent but have been computed by a different sequence of operations comparing as not equal. The consequence of this for programs that use `CExEq` is for further study.

- If a C compiler compiles pointer equality as `CExEq` (rather than `CEq`), it will catch the following example of undefined behavior. Suppose that *a* and *b* are capabilities for different objects, but *a* has been incremented until its **base** + **offset** points to the same memory location as *b*. Using `CExEq`, these pointers will not compare as equal because they have different bounds.

CReturn: Return to the Previous Security Domain

Format

CReturn



Description

`CReturn` is used by a protected subsystem to return to its caller.

1. **IDC** is popped off the trusted system stack.
2. **PCC** is popped off the trusted system stack.

In the current implementation of CHERI, `CReturn` is implemented by the hardware raising an exception, while the rest of the behavior is implemented in software by the exception handler. Later versions of CHERI may implement more of this instruction in hardware, for improved performance. Authors of compilers or assembly language programs should not rely on `CReturn` being implemented in either hardware or software.

TO DO: In CheriBSD, `CReturn` checks the *global* bit on capability registers so that `CReturn` cannot be used to leak local capabilities.

Pseudocode (hardware)

raise_c2_exception_noreg(exceptionReturn)

Pseudocode (software)

IDC ← mem[TSS .. TSS + capability_size − 1]
IDC.tag ← tags[toTag(TSS)]
TSS ← TSS + capability_size
PCC ← mem[TSS .. TSS + capability_size − 1]
PCC.tag ← tags[toTag(TSS)]
TSS ← TSS + capability_size
PC ← PCC.offset

Exceptions

The exception raised when `CReturn` is implemented in software is a coprocessor 2 exception (C2E) with the capability cause code set to 0x6 (exceptionReturn) and RegNum set to *cs*. The handler vector for this exception is 0x100 above the general purpose exception handler.

An additional coprocessor 2 exception is raised if:

- The trusted system stack would underflow.
- The tag bits are not set on the memory location that are popped from the stack into **IDC** and **PCC**.

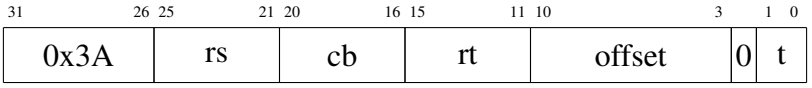
Notes

- The `CReturn` instruction will be removed in a future version of the ISA specification (though it will continue to exist as a pseudo-instruction in the assembler) and will be replaced by a specific selector in the `CCall` instruction.
- As with `CCall`, `CReturn` can have multiple implementation choices that span hardware and software. Pure software implementations can address more broad types of calling convention; pulling register clearing into the software handler provides guarantees that require less effort that might otherwise be performed redundantly by the caller and callee – althought it would also require greater knowledge (and hard coding) of the ABI.

Store via Capability Register

Format

CSB rs, rt, offset(cb)
CSH rs, rt, offset(cb)
CSW rs, rt, offset(cb)
CSD rs, rt, offset(cb)
CSBR rs, rt(cb)
CSHR rs, rt(cb)
CSWR rs, rt(cb)
CSDR rs, rt(cb)
CSBI rs, offset(cb)
CSHI rs, offset(cb)
CSWI rs, offset(cb)
CSDI rs, offset(cb)



Purpose

Stores some or all of a register into a memory location.

Description

Part of general-purpose register *rs* is stored to the memory location specified by *cb.***base** + *cb.***offset** + *rt* + $2^t * offset$. Capability register *cb* must contain a capability that grants permission to store data.

The *t* field determines how many bits of the register are stored to memory:

0 byte (8 bits)

1 halfword (16 bits)

2 word (32 bits)

3 doubleword (64 bits)

If less than 64 bits are stored, they are taken from the least-significant end of the register.

Pseudocode

```
if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if cb.sealed then
```

```
    raise_c2_exception(exceptionSealed, cb)
else if not cb.Permit_Store then
    raise_c2_exception(exceptionPermitStore, cb)
end if
if t = 0 then
    size ← 1
else if t = 1 then
    size ← 2
else if t = 2 then
    size ← 4
else if t = 3 then
    size ← 8
end if
cursor ← (cb.base + cb.offset) mod 2^64
addr ← (cursor + rt + size * sign_extend(offset)) mod 2^64
if addr + size > cb.base + cb.length then
    raise_c2_exception(exceptionLength, cb)
else if addr < cb.base then
    raise_c2_exception(exceptionLength, cb)
else if align_of(addr) < size then
    raise_exception(exceptionAdES)
else
    mem[addr .. addr + size − 1] ← rs[0 .. size − 1]
    tags[toTag(addr)] ← false
end if
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cb*.**tag** is not set.
- *cb*.**s** is set.
- *cb*.**perms**.*Permit_Store* is not set.
- *addr* + *size* > *cb*.**base** + *cb*.**length**.
- *addr* < *cb*.**base**

An address error during store (AdES) is raised if:

- *addr* is not aligned.

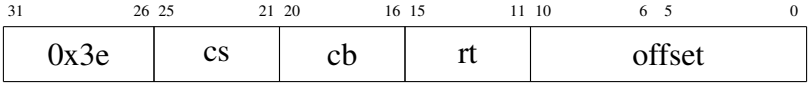
Notes

- This instruction reuses the opcode from the Store Word from Coprocessor 2 (`SWC2`) instruction in the MIPS Specification.
- *rt* is treated as an unsigned integer.
- *offset* is treated as a signed integer.
- BERI1 has a compile-time option to allow unaligned loads and stores. If BERI1 is built with this option, an unaligned store will only raise an exception if it crosses a cache line boundary.

CSC: Store Capability Register

Format

CSC cs, rt, offset(cb)
CSCR cs, rt(cb)
CSCI cs, offset(cb)



Description

Capability register *cs* is stored at the memory location specified by *cb.***base** + *cb.***offset** + *rt* + 16 ∗ *offset*, and the bit in the tag memory associated with *cb.***base** + *cb.***offset** + *rt* + 16 ∗ *offset* is set to the value of *cs.tag*. Capability register *cb* must contain a capability that grants permission to store capabilities. The virtual address *cb.***base** + *cb.***offset** + *rt* + 16 ∗ *offset* must be *capability_size* aligned.

When the 256-bit representation of capabilities is in use, the capability is stored in memory in the format described in Figure 5.2. **base**, **length** and **otype** are stored in memory with the same endian-ness that the CPU uses for double-word stores, i.e., big-endian. The bits of **perms** are stored with bit zero being the least significant bit, so that the least significant bit of the eighth byte stored is the **s** bit, the next significant bit is the *Global* bit, the next is *Permit_Execute* and so on.

Pseudocode

```
if register_inaccessible(cs) then
    raise_c2_exception(exceptionAccessSystem, cs)
else if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if cb.sealed then
    raise_c2_exception(exceptionSealed, cb)
else if not cb.perms.Permit_Store_Capability then
    raise_c2_exception(exceptionPermitStoreCapability, cb)
else if not cb.perms.Permit_Store_Local_Capability and cs.tag and not cs.perms.Global then
    raise_c2_exception(exceptionPermitStoreLocalCapability, cb)
end if
cursor ← (cb.base + cb.offset) mod 2^64
addr ← cursor + rt + 16 * sign_extend(offset)
if addr + capability_size > cb.base + cb.length then
    raise_c2_exception(exceptionLength, cb)
else if addr < cb.base then
    raise_c2_exception(exceptionLength, cb)
else if cs.tag and TLB(addr).S then
    raise_c2_exception(exceptionTLBStore, cs)
```

```
else if align_of(addr) < capability_size then
    raise_exception(exceptionAdES)
else
    mem[addr .. addr + capability_size − 1] ← cap_to_bytes(cs)
    tags[toTag(addr)] ← cs.tag
end if
```

Exceptions

A coprocessor 2 exception is raised if:

- *cs* or *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and the corresponding bit in **PCC**.**perms** is not set.
- *cb*.**tag** is not set.
- *cb*.**s** is set.
- *cb*.**perms**.*Permit_Store_Capability* is not set.
- *cb*.**perms**.*Permit_Store_Local* is not set and *cs.tag* is set and *cs*.**perms**.*Global* is not set.
- *addr* + *capability_size* > *cb*.**base** + *cb*.**length**.
- *addr* < *cb*.**base**. destination address is set.

A TLB Store exception is raised if:

- *cs*.**tag** is set and the *S* bit in the TLB entry for the page containing *addr* is not set.

An address error during store (AdES) exception is raised if:

- The virtual address *addr* is not *capability_size* aligned.

Notes

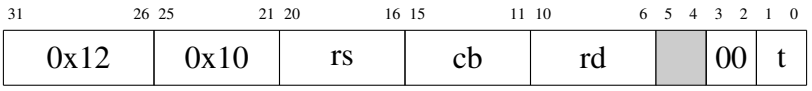
- If the address alignment check fails and one of the security checks fails, a coprocessor 2 exception (and not an address error exception) is raised. The priority of the exceptions is security-critical, because otherwise a malicious program could use the type of the exception that is raised to test the bottom bits of a register that it is not permitted to access.
- It is permitted to store a local capability with the tag bit unset even if the permit store local bit is not set in *cb*. This is because if the tag bit is not set then the permissions have no meaning.
- *offset* is interpreted as a signed integer.
- This instruction reuses the opcode from the Store Doubleword from Coprocessor 2 (`SDC2`) instruction in the MIPS Specification.

- The `CSCI` mnemonic is equivalent to `CSC` with *cb* being the zero register ($zero). The `CSCR` mnemonic is equivalent to `CSC` with *offset* set to zero.
- BERI1 has a compile-time option to allow unaligned loads and stores. `CSC` to an unaligned address will raise an exception even if BERI1 has been built with this option, because it would be a security vulnerability if an attacker could construct a corrupted capability with **tag** set by writing it to an unaligned address.
- Although the *capability_size* can vary, the offset is always in multiples of 16 bytes (128 bits).

CSC[BHWD]: Store Conditional via Capability

Format

CSCB rd, rs, cb
CSCH rd, rs, cb
CSCW rd, rs, cb
CSCD rd, rs, cb



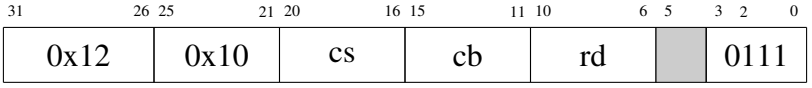
Pseudocode

```
addr ← cb.base + cb.offset
size ← 2^t
if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if cb.sealed then
    raise_c2_exception(exceptionSealed, cb)
else if not cb.perms.Permit_Store then
    raise_c2_exception(exceptionPermitStore, cb)
else if addr + size > cb.base + cb.length then
    raise_c2_exception(exceptionLength, cb)
else if addr < cb.base then
    raise_c2_exception(exceptionLength, cb)
else if align_of(addr) < size then
    raise_exception(exceptionAdES)
else if not linkedFlag then
    rd ← 0
else
    mem[addr .. addr + size − 1] ← rs[0 .. size − 1]
    tags[toTag(addr)] ← false
    rd ← 1
end if
```

CSCC: Store Conditional Capability via Capability

Format

CSCC rd, cs, cb



Pseudocode

addr ← (cb.base + cb.offset) $\mathrm{mod} 2^{64}$
if register_inaccessible(cs) **then**
    raise_c2_exception(exceptionAccessSystem, cs)
else if register_inaccessible(cb) **then**
    raise_c2_exception(exceptionAccessSystem, cb)
else if not cb.tag **then**
    raise_c2_exception(exceptionTag, cb)
else if cb.sealed **then**
    raise_c2_exception(exceptionSealed, cb)
else if not cb.perms.Permit_Store_Capability **then**
    raise_c2_exception(exceptionPermitStoreCapability, cb)
else if not cb.perms.Permit_Store_Local_Capability **and** cs.tag **and not** cs.perms.Global **then**
    raise_c2_exception(exceptionPermitStoreLocalCapability, cb)
else if addr + capability_size > cb.base + cb.length **then**
    raise_c2_exception(exceptionLength, cb)
else if addr < cb.base **then**
    raise_c2_exception(exceptionLength, cb)
else if cs.tag **and** TLB(addr).S **then**
    raise_c2_exception(exceptionTLBStore, cs)
else if align_of(addr) < capability_size **then**
    raise_exception(exceptionAdES)
else if not linkedFlag **then**
    rd ← 0
else
    mem[addr .. addr + capability_size − 1] ← cap_to_bytes(cs)
    tags[toTag(addr)] ← cs.tag
    rd ← 1
end if

Exceptions

A coprocessor 2 exception is raised if:

- *cs* or *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cb*.**tag** is not set.

- *cb*.**s** is set.
- *cb*.**perms**.*Permit_Store_Capability* is not set.
- *cb*.**perms**.*Permit_Store_Local_Capability* is not set and *cs*.**perms**.*Global* is not set.
- $addr + capability\_size > cb.\textbf{base} + cb.\textbf{length}$
- $addr < cb.\textbf{base}$

A TLB Store exception is raised if:

- The *S* bit in the TLB entry corresponding to virtual address *addr* is not set.

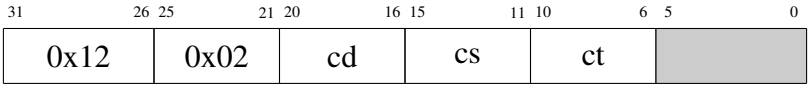
An address error during store (AdES) exception is raised if:

- *addr* is not correctly aligned.

CSeal: Seal a Capability

Format

CSeal cd, cs, ct



Description

Capability register *cs* is sealed with an **otype** of *ct*.**base** + *ct*.**offset** and the result is placed in *cd*:

- *cd*.**otype** is set to *ct*.**base** + *ct*.**offset**;
- *cd*.**s** is set;
- and the other fields of *cd* are copied from *cs*.

ct must grant *Permit_Seal* permission, and the new **otype** of *cd* must be between *ct*.**base** and *ct*.**base** + *ct*.**length** − 1.

Pseudocode

```
if register_inaccessible(cd) then
    raise_c2_exception(exceptionAccessSystem, cd)
else if register_inaccessible(cs) then
    raise_c2_exception(exceptionAccessSystem, cs)
else if register_inaccessible(ct) then
    raise_c2_exception(exceptionAccessSystem, ct)
else if not cs.tag then
    raise_c2_exception(exceptionTag, cs)
else if not ct.tag then
    raise_c2_exception(exceptionTag, ct)
else if cs.sealed then
    raise_c2_exception(exceptionSealed, cs)
else if ct.sealed then
    raise_c2_exception(exceptionSealed, ct)
else if not ct.perms.Permit_Seal then
    raise_c2_exception(exceptionPermitSeal, ct)
else if ct.offset ≥ ct.length then
    raise_c2_exception(exceptionLength, ct)
else if ct.base + ct.offset > max_otype then
    raise_c2_exception(exceptionLength, ct)
else if not representable(true, cs.base, cs.length, cs.offset) then
    raise_c2_exception(exceptionInexact, cs)
else
    cd ← cs with sealed ← true, otype ← ct.base + ct.offset
end if
```

Exceptions

A coprocessor 2 exception is raised if:

- *cd*, *cs*, or *ct* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and the corresponding bit in **PCC**.**perms** is not set.
- *cs*.**tag** is not set.
- *ct*.**tag** is not set.
- *cs*.**s** is set.
- *ct*.**s** is set.
- *ct*.**perms**.*Permit_Seal* is not set.
- *ct*.**offset** $\geq$ *ct*.length
- *ct*.**base** + *ct*.**offset** $>$ *max_otype*
- The bounds of *cb* cannot be represented exactly in a sealed capability.

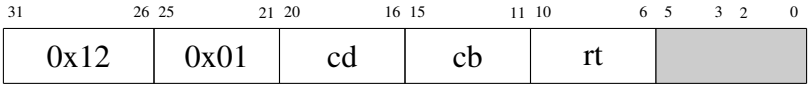
Notes

- If capability compression is in use, the range of possible (**base**, **length**, **offset**) values might be smaller for sealed capabilities than for unsealed capabilities. This means that `CSeal` can cause loss of precision; this isn't show in the above pseudocode.

CSetBounds: Set Bounds

Format

CSetBounds cd, cb, rt



Description

Capability register *cd* is replaced with a capability that:

- Grants access to a subset of the addresses authorized by *cb*. That is, *cd*.**base** $\geq$ *cb*.**base** and *cd*.**base** + *cd*.**length** $\leq$ *cb*.**base** + *cb*.**length**.
- Grants access to at least the addresses *cb*.**base** + *cb*.**offset** . . . *cb*.**base** + *cb*.**offset** + *rt* − 1. That is, *cd*.**base** $\leq$ *cb*.**base** + *cb*.**offset** and *cd*.**base** + *cd*.**length** $\geq$ *cb*.**base** + *cb*.**offset** + *rt*.
- Has an **offset** that points to the same memory location as *cb*'s **offset**. That is, *cd*.**offset** = *cb*.**offset** + *cb*.**base** - *cd*.**base**.
- Has the same **perms** as *cb*, that is, *cd*.**perms** = *cb*.**perms**.

When the hardware uses a 256-bit representation for capabilities, the bounds of the destination capability *cd* are exactly as requested. When the hardware uses a smaller (compressed) representation of capabilities in which not all combinations of **base** and **length** are representable, then *cd* may grant access to a range of memory addresses that is wider than requested, but is still guaranteed to be within the bounds of *cb*.

Pseudocode (256-bit capabilities)

```
cursor ← (cb.base + cb.offset) mod 2^64
if register_inaccessible(cd) then
    raise_c2_exception(exceptionAccessSystem, cd)
else if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if cb.sealed then
    raise_c2_exception(exceptionSealed, cb)
else if cursor < cb.base then
    raise_c2_exception(exceptionLength, cb)
else if cursor + rt > cb.base + cb.length then
    raise_c2_exception(exceptionLength, cb)
else
    cd ← cb with base ← cursor, length ← rt, offset ← 0
end if
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb* or *cd* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cb*.**tag** is not set.
- *cb*.**s** is set.
- *cursor* $<$ *cb*.**base**
- *cursor* $+$ *rt* $>$ *cb*.**base** $+$ cb.**length**

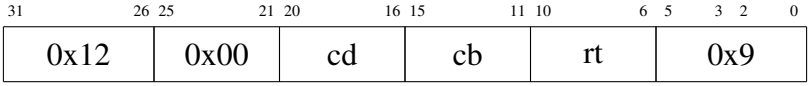
Notes

- In the above pseudocode, arithmetic is over the mathematical integers and *rt* is unsigned, so a large value of *rt* cannot cause *cursor* $+$ *rt* to wrap around and be less than *cb*.**base**. Implementations (that, for example, will probably use a fixed number of bits to store values) must handle this overflow case correctly.

CSetBoundsExact: Set Bounds Exactly

Format

CSetBoundsExact cd, cb, rt



Description

Capability register *cd* is replaced with a capability with **base** *cb*.**base**+ *cb*.**offset**, **length** *rt*, and **offset** zero. When capability compression is in use, an exception is thrown if the requested bounds cannot be represented exactly.

Pseudocode

```
cursor ← (cb.base + cb.offset) mod 2^64
if register_inaccessible(cd) then
    raise_c2_exception(exceptionAccessSystem, cd)
else if register_inaccessible(cb) then
    raise_c2_exception(exceptionAccessSystem, cb)
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if cb.sealed then
    raise_c2_exception(exceptionSealed, cb)
else if cursor < cb.base then
    raise_c2_exception(exceptionLength, cb)
else if cursor + rt > cb.base + cb.length then
    raise_c2_exception(exceptionLength, cb)
else if not representable(cb.sealed, cb.base + cb.offset, rt, 0) then
    raise_c2_exception(exceptionInexact, cb)
else
    cd ← cb with base ← cursor, length ← rt, offset ← 0
end if
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb* or *cd* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cb*.**tag** is not set.
- *cb*.**s** is set.
- *cursor* $<$ *cb*.**base**
- *cursor* + *rt* $>$ *cb*.**base** + cb.**length**

- The requested bounds cannot be represented exactly.

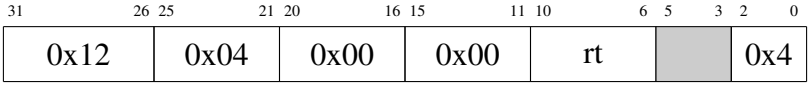
Notes

- In the above pseudocode, arithmetic is over the mathematical integers and *rt* is unsigned, so a large value of *rt* cannot cause *cursor* + *rt* to wrap around and be less than *cb*.**base**. Implementations (that, for example, will probably use a fixed number of bits to store values) must handle this overflow case correctly.

CSetCause: Set the Capability Exception Cause Register

Format

CSetCause rt



Description

The capability cause register value is set to the low 16 bits of general-purpose register *rt*.

Pseudocode

```
if not PCC.perms.Access_System_Registers then
    raise_c2_exception_noreg(exceptionAccessSystem)
else
    CapCause ← rt[0 .. 15]
end if
```

Exceptions

A coprocessor 2 exception is raised if:

- **PCC**.**perms**.*Access_System_Registers* is not set.

Notes

- `CSetCause` does not cause an exception to be raised (unless the permission check for *Access_System_Registers* fails). `CSetCause` will typically be used in an exception handler, where the exception handler wants to change the cause code set by the hardware before doing further exception handling. (e.g., when the original cause code was `CCall`, the `CCall` handler detects that `CCall` should fail, and it sets *CapCause* to the reason it failed). In cases like this, it is important that **EPC** (etc.) are not overwritten by `CSetCause`.

CSetLen: *Instruction Removed*

Format

Instruction removed

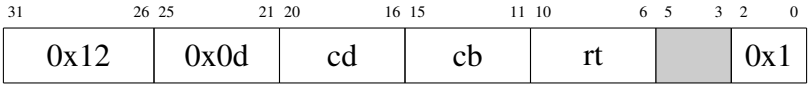
Description

This instruction has been removed from the CHERI ISA in favor of `CSetBounds`. The opcode remains reserved.

CSetOffset: Set Cursor to an Offset from Base

Format

CSetOffset cd, cb, rt



Description

Capability register *cd* is replaced with the contents of capability register *cb* with the **offset** field set to the contents of general purpose register *rt*.

If capability compression is in use, and the requested **base**, **length** and **offset** cannot be represented exactly, then *cd*.**tag** is cleared, *cd*.**base** and *cd*.**length** are set to zero, *cd*.**perms** is cleared and *cd*.**offset** is set equal to *cb*.**base**+ *rt*.

Pseudocode

if register_inaccessible(cd) **then**
    raise_c2_exception(exceptionAccessSystem, cd)
else if register_inaccessible(cb) **then**
    raise_c2_exception(exceptionAccessSystem, cb)
else if cb.tag **and** cb.sealed **then**
    raise_c2_exception(exceptionSealed, cb)
else if not representable(cb.sealed, cb.base, cb.length, rt) **then**
    cd ← int_to_cap((cb.base + rt) $\mod 2^{64}$)
else
    cd ← cb **with** offset ← rt
end if

Exceptions

A coprocessor 2 exception is raised if:

- *cb* or *cd* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *cb.***tag** is set and *cb.***s** is set.

Notes

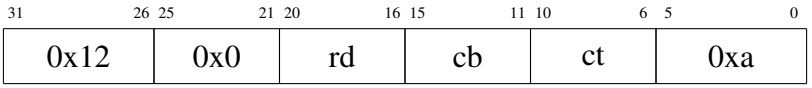
- `CSetOffset` can be used on a capability register whose tag bit is not set. This can be used to store an integer value in a capability register, and is useful when implementing a variable that is a union of a capability and an integer (`intcap_t` in C). The in-memory representation that will be used if the capability register is stored to memory might be surprising to some users (with the 256-bit representation of capabilities, **base** + **offset** is stored in the **cursor** field in memory) and may change if the memory representation of capabilities changes, so compilers should not rely on it.

- When capability compression is in use, and the requested offset is not representable, the result preserves the requested **base** + **offset** (i.e., the cursor) rather than the architectural field **offset**. This field is mainly useful for debugging what went wrong (the capability cannot be dereferenced, as **tag** has been cleared), and for debugging we considered it more useful to know what the requested capability would have referred to rather than its **offset** relative to a **base** that is no longer available. This has the disadvantage that it exposes the value of **base** to a program, but **base** is not a secret and can be accessed by other means. The main reason for not exposing **base** to programs is so that a garbage collector can stop the program, move memory, modify the capabilities and restart the program. A capability with **tag** cleared cannot be dereferenced, and so is not of interest to a garbage collector, and so it doesn't matter if it exposes **base**.

CSub: Subtract Capabilities

Format

CSub rd, cb, ct



Description

Register *rd* is set equal to (*cb*.**base** + *cb*.**offset** − *ct*.**base** − *ct*.**offset**) $\mod 2^{64}$.

Pseudocode

if register_inaccessible(cb) **then**
    raise_c2_exception(exceptionAccessSystem, cb)
else if register_inaccessible(ct) **then**
    raise_c2_exception(exceptionAccessSystem, ct)
else
    rd ← (cb.base + cb.offset − ct.base − ct.offset) $\mod 2^{64}$
end if

Exceptions

A coprocessor 2 exception is raised if:

- *cb* or *ct* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.

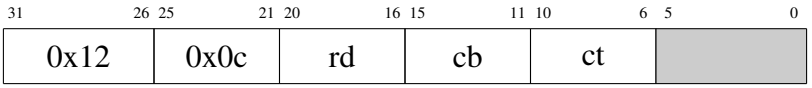
Notes

- `CSub` can be used to implement C-language pointer subtraction, or subtraction of `intcap_t`.
- Like `CIncOffset`, `CSub` can be used on either valid capabilities (**tag** set) or on integer values stored in capability registers (**tag** not set).
- If a copying garbage collector is in use, pointer subtraction must be implemented with an atomic operation (such as `CSub`). Implementing pointer subtraction with a non-atomic sequence of operations such as `CGetOffset` has the risk that the garbage collector will relocate an object part way through, giving incorrect results for the pointer difference. If *cb* and *ct* are both pointers into the same object, then a copying garbage collector will either relocate both of them or neither of them, leaving the difference the same. If *cb* and *ct* are pointers into different objects, the result of the subtraction is not defined by the ANSI C standard, so it doesn't matter if this difference changes as the garbage collector moves objects.

CToPtr: Capability to Pointer

Format

CToPtr rd, cb, ct



Description

If *cb* has its tag bit unset (i.e. it is either the NULL capability, or contains some other non-capability data), then *rd* is set to zero. Otherwise, *rd* is set to *cb*.**base** + *cb*.**offset** - *ct*.**base**

This instruction can be used to convert a capability into a pointer that uses the C language convention that a zero value represents the NULL pointer. Note that *rd* will also be zero if *cb* and *ct* have the same base; this is similar to the C language not being able to distinguish a NULL pointer from a pointer to a structure at address 0.

Pseudocode

if register_inaccessible(cb) **then**
    raise_c2_exception(exceptionAccessSystem, cb)
else if register_inaccessible(ct) **then**
    raise_c2_exception(exceptionAccessSystem, ct)
else if not ct.tag **then**
    raise_c2_exception(exceptionTag, ct)
else if not cb.tag **then**
    rd $\leftarrow$ 0
else
    rd $\leftarrow$ (cb.base + cb.offset $-$ ct.base) $\bmod 2^{64}$
end if

Exceptions

A coprocessor 2 exception will be raised if:

- *cb* or *ct* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and **PCC**.**perms**.*Access_System_Registers* is not set.
- *ct*.**tag** is not set.

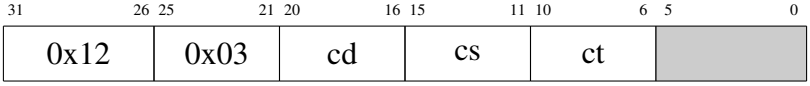
Notes

- *cb* or *ct* being sealed will not cause an exception to be raised. This is for further study.

CUnseal: Unseal a Sealed Capability

Format

CUnseal cd, cs, ct



Description

The sealed capability in cs is unsealed with ct and the result placed in cd. The global bit of cd is the AND of the global bits of cs and ct. ct must be unsealed, have *Permit_Seal* permission, and ct.**base** + ct.**offset** must equal cs.**otype**.

Pseudocode

if register_inaccessible(cd) **then**
    raise_c2_exception(exceptionAccessSystem, cd)
else if register_inaccessible(cs) **then**
    raise_c2_exception(exceptionAccessSystem, cs)
else if register_inaccessible(ct) **then**
    raise_c2_exception(exceptionAccessSystem, ct)
else if not cs.tag **then**
    raise_c2_exception(exceptionTag, cs)
else if not ct.tag **then**
    raise_c2_exception(exceptionTag, ct)
else if not cs.sealed **then**
    raise_c2_exception(exceptionSealed, cs)
else if ct.sealed **then**
    raise_c2_exception(exceptionSealed, ct)
else if ct.base + ct.offset $\neq$ cs.otype **then**
    raise_c2_exception(exceptionType, ct)
else if not ct.perms.Permit_Seal **then**
    raise_c2_exception(exceptionPermitSeal, ct)
else if ct.offset $\geq$ ct.length **then**
    raise_c2_exception(exceptionLength, ct)
else
    cd $\leftarrow$ cs **with** sealed $\leftarrow$ **false**, otype $\leftarrow$ 0
    cd.perms.Global $\leftarrow$ cs.perms.Global **and** ct.perms.Global
end if

Exceptions

A coprocessor 2 exception is raised if:

- *cd*, *cs*, or *ct* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and the corresponding bit in **PCC**.**perms** is not set.

- *cs*.**tag** is not set.
- *ct*.**tag** is not set.
- *cs*.**s** is not set.
- *ct*.**s** is set.
- *ct*.**offset** $\geq$ *ct*.**length**
- *ct*.**perms**.*Permit_Seal* is not set.
- *ct*.**base** + *ct*.**offset** $\neq$ *cs*.**otype**.

Notes

- There is no need to check if *ct*.**base** + *ct*.**offset** $>$ *max_otype*, because this can't happen: *ct*.**base** + *ct*.**offset** must equal *cs*.**otype** for the **otype** check to have suceeded, and there is no way *cs*.**otype** could have been set to a value that is out of range.

6.7 Assembler Pseudo-Instructions

For convenience, several pseudo-instructions are accepted by the assembler. These expand to either single instructions or short sequences of instructions.

6.7.1 CMove

Capability Move

`CMove` is a pseudo operation that moves a capability from one register to another. It expands to a `CIncOffset` instruction, with `$zero` as the increment operand.

```
1 # The following are equivalent:
2   CMove $c1, $c2
3   CIncOffset $c1, $c2, $zero
```

6.7.2 CGetDefault, CSetDefault

Get/Set Default Capability

`CGetDefault` and `CSetDefault` get and set the capability register that is implicitly employed by the legacy MIPS load and store instructions. In the current version of the ISA, this register is **C0**. These pseudo-operations are provided for the benefit of the LLVM compiler: the compiler can more easily detect that a write to **C0** affects the meaning of subsequent legacy MIPS instructions if these are separate pseudo-operations.

```
1 # The following are equivalent:
2   CGetDDC $c1
3   CGetDefault $c1
4   CIncOffset $c1, $c0, $zero
```

```
# The following are equivalent:
  CSetDDC $c1
  CSetDefault $c1
  CIncOffset $c0, $c1, $zero
```

6.7.3 CGetEPCC, CSetEPCC

Get/Set Exception Program Counter Capability

Pseudo-operations are provided for getting and setting **EPCC**. In the current ISA, EPCC is a numbered register and so can be accessed with `CMove`, but in future revisions of the ISA it might be moved to a special register (similar to **PC** not being a numbered register in the MIPS ISA).

```
# The following are equivalent:
  CGetEPCC $c1
  CIncOffset $c1, $epcc, $zero
```

```
# The following are equivalent:
  CSetEPCC $c1
  CIncOffset $epcc, $c1, $zero
```

6.7.4 GGetKCC, CSetKCC

Get/Set Kernel Code Capability

```
# The following are equivalent:
  CGetKCC $c1
  CIncOffset $c1, $kcc, $zero
```

```
# The following are equivalent:
  CSetKCC $c1
  CIncOffset $kcc, $c1, $zero
```

6.7.5 CGetKDC, CSetKDC

Get/Set Kernel Data Capability

```
# The following are equivalent:
  CGetKDC $c1
  CIncOffset $c1, $kdc, $zero
```

```
# The following are equivalent:
  CSetKDC $c1
  CIncOffset $kdc, $c1, $zero
```

6.7.6 Capability Loads and Stores of Floating-Point Values

The current revision of the CHERI ISA does not have instructions for loading floating point values directly via capabilities. MIPS does provide instructions for moving values between

integer and floating point registers, so a load or store of a floating point value via a capability can be implemented in two instructions.

Four pseudo-instructions are defined to implement these patterns. These are `clwc1` and `cldc1` for loading 32-bit and 64-bit floating point values, and `cswc1` and `csdc1` as the equivalent store operations. The load operations expand as follows:

```
1  cldc1    $f7, $zero, 0($c2)
2  # Expands to:
3  cld    $1, $zero, 0($c2)
4  dmtc1    $1, $f7
```

Note that integer register `$1` (`$at`) is used; this pseudo-op is unavailable if the `noat` directive is used. The 32-bit variant (`clwc1`) has a similar expansion, using `clwu` and `mtc1`.

The store operations are similar:

```
1  csdc1    $f7, $zero, 0($c2)
2  # Expands to:
3  dmfc1    $1, $f7
4  csd    $1, $zero, 0($c2)
```

The specified floating point value is moved from the floating point register to `$at` and then stored using the correct-sized capability instruction.

Chapter 7

Decomposition of CHERI Features

The CHERI ISA extension extends a RISC ISA to support capability pointers, that is, unforgeable references to memory. These pointers require a new hardware-defined register format, similar to floating point. To protect these pointers and make them unforgeable, CHERI distinguishes capability pointers from integers using tags in both registers and memory. Furthermore, to make performance of capability pointers competitive with unprotected pointers, CHERI proposes a full set of registers that support capability pointers. This chapter decomposes these features of the CHERI capability model with some discussion of the cost and benefit of each. The features discussed in this chapter are a subset of those in Section 3.2. These were selected as features that are independently useful, but which compose to form the full CHERI capability model. This suggests a logical sequence of adoption that could be consistent and useful in increments, if a full implementation is considered too expensive.

7.1 CHERI Feature Decomposition

We may decompose CHERI support into several independently useful features:

- Virtual memory segmentation
  - Global data segment offsets
  - Multiple segment registers
- Pointer permissions
- Tags
  - Sealed pointers
- Bounded pointers

Although they are carefully designed to be composable without adverse interactions, these features are individually very useful – and in total present a complete capability computing platform. We discuss each of these features and their respective costs in Section 7.1. We then discuss how each applies to common vulnerability mitigation techniques in Section 7.2.

7.1.1 Data and Code Segmentation

The standard virtual-memory model provides a flat virtual address space to each process with a set of valid pages. Applications often have very complex inner structure that is not sufficiently expressed in this scheme. The simple per-process page-set model limits the protection that the hardware can provide, as all instructions in the process have equal access to the page set.

CHERI provides a segment register that constrains all user-space memory accesses to a contiguous region of address space. CHERI provides one segment register for data, and another for instruction fetch. This simple virtual memory segmentation mechanism would greatly enhance certain security models. *Data and code segmentation* enables both limiting special execution contexts and protecting special memory regions by limiting the general execution context. Both of these techniques could find wide application in security software mechanisms.

Data and Code Segmentation Cost The hardware cost of *data and code segmentation* is very low. If the design uses absolute pointers inside of a segment, then the base and bound registers are simply two 64-bit values that are checked against every address translation. Thought should be devoted to software use cases, as the majority of the cost would be in software adoption.

A *data and code segmentation* mechanism without capability protection should allow access to the segment register only from supervisor mode. This would increase software cost of crossing between segments, but would be necessary for effective compartmentalization.

Global Data-Capability Offsets

If the design uses absolute pointers inside of a segment, then the base and bound registers are simply two 64-bit values that are checked against every address translation. However the design might use segment-relative addressing, implementing *global data capability offsets*, to enable convenient relocation within the address space. In this case the base of the segment register would be added to the address of any memory reference to compute the virtual address.

Costs Supporting this requires an extra add on the load/store path. To improve timing, a simpler transformation might be used at the expense of segment granularity. For example, the segment may be defined by an address and a bit mask to define power-of-two sized regions, or even a small number of bits for the top of the virtual address if segment sizes are fixed.

Multiple Segment Registers

While it is possible to implement simple segmentation systems with only one segment register, efficient sharing between segments is enabled by adding a very small number of segment registers. Two segment registers could describe the current compartment and a shared segment. If more segments are needed, a small set of segment registers could be installed by the supervisor but selected by the user, similar to IA32, which would optimize use of frequently used *shadow spaces* and protected structures that should not be writable from common data accesses.

Costs Adding more segment registers adds to the hardware complexity of memory addressing and to the cost of context switches. In the limit, segment registers would simply be a register file with full forwarding, as implemented in the CHERI prototype, to enable segment use as *fat-pointers*.

7.1.2 Pointer Permissions

CHERI includes permission bits on every capability pointer. These include read, write, and execute, with other permissions such as "unsealed" – which forbids dereference to enforce safe handling of opaque pointers. The upper bits of a 64-bit pointer may be used to hold these memory access permissions, which can help enforce programmer intent – for example, distinguishing between executable pointers and data pointers.

This feature would be of limited strength by itself as the permission bits could be easily forged. However, in combination with *tags*, permissions might be protected by hardware.

Costs Interpreting the top bits of pointers as permissions has a very low cost in hardware. There would be a software cost to ensure that the upper bits of a pointer are not employed for other uses when the feature is enabled.

7.1.3 Tags

Tags are required by CHERI to make capability pointers unforgeable, but tags are also useful as a standalone feature. Tagged memory has been studied extensively on its own (e.g., [21, 30]). A tagging system allows a program to attach a small amount of metadata to any word in memory that can be preserved across copies through registers. While CHERI enforces a hardware interpretation of these tags to guarantee pointer behavior in the face of untrustworthy programs, tags can also be very powerful when controlled purely by software.

Costs *Tags* require some additions to the memory subsystem to keep tags alongside the data for each line. At the bottom of the tagged cache hierarchy should be some controller that makes sure that tags are found for each memory request and are stored when a line is evicted. The simplest design of such a system would require error-correcting codes memory and use the ECC bits (typically 4 per 64-bit word) to store a tag. An alternative would be a table in a region of DRAM that is no system accessible.

7.1.4 Sealing

Pointer *sealing* allows a pointer to become immutable and un-dereferenceable until it has been unsealed. Sealed pointers have a type associated with them, which identifies the type that must be used to unseal them.

Costs Supporting *sealing* costs some bits in the pointer for identifying the type (currently 20 are proposed for the 128-bit prototype) and two permissions for identifying sealed pointers and those with the permission to seal. Pointer *sealing* requires *tags*, though cryptography might be used with a loss in encoding efficiency.

7.1.5 Bounds Checking

CHERI capability pointers include base and bound fields in addition to the basic pointer. This structure has been called a *fat pointer* in compiler literature. A *fat-pointer* structure can be supported natively in hardware much like the floating point formats are today, performing bounds

checks automatically when it is dereferenced to enforce spatial memory safety. Unlike software schemes, a hardware implementation can use an efficient compressed format, saving memory and data movement instructions, which are the greatest performance cost of *fat-pointer* schemes.

Costs *Fat-pointer* support is likely to be significant if a design is to achieve performance parity with standard pointers. A *fat pointer* should be larger than an existing pointer if it supports the same virtual address space as the original pointer as well as holding base and bounds, so *fat-pointer* support is likely to introduce larger registers or a new register file. While doubling the width of the general-purpose register file or adding a new register file is a notable cost, bounds-checking memory accesses should not add to the critical path as the bounds check can happen in parallel with memory translation.

7.2 Vulnerability Mitigation Strategies

The components of CHERI protection noted in Section 7.1 have applications with respect to at least three vulnerability mitigation strategies.

- Compartmentalization
- Control-Flow Integrity
- Memory Safety

7.2.1 Compartmentalization

Compartmentalization is a vulnerability mitigation strategy that isolates program modules from one another such that any untoward behavior in one module cannot affect any other module. Software fault isolation (SFI) attempts to achieve compartmentalization within an address space without dedicated hardware. Implementations of SFI include Google Native Client for Internet distribution of native executables [118] and Microsoft's BGI for kernel module protection [92]. These involve static verification of machine code and stringent machine-code style requirements.

Hardware *data and code segmentation* would make software fault isolation (SFI) systems trivial, allowing the current execution to be constrained to an island in the address space. Many SFI implementations first used the simple segmentation mechanism in IA32 but were forced to use more creative solutions for AMD64.

There are two sources of inefficiency in current SFI techniques. The machine-code verifiers reduce the total set of CPU features that isolated code may use (for example, preventing modification to one or more registers), or require explicit bounds checks before every memory access, either of which could reduce performance. This would not apply with non-bypassable hardware enforcement. The second overhead concerns communication with the outside world. It is not possible to delegate access to arbitrary buffers to isolated code, because the analysis techniques enforce a relatively simple check: that all memory accesses are in a single contiguous range in the address space. If two compartments must communicate, all data must be copied from one space to the other.

Providing *multiple segment registers* would permit sharing at a finer granularity, statically delegating specific regions to a compartment. This would still require some copying, but for some uses would allow data to be generated or consumed in shared regions.

Sharing using *multiple segment registers* is similar to multiple processes with shared memory regions, and does not provide a programmer model that is particularly convenient. Adding *bounded pointers* protected by *tags* allows delegating objects at a granularity that is more natural for programmers. This is a significant benefit when adding compartmentalization to existing software – which would pass data between functions or modules by pointer passing.

Capability sealing extends this flexibility, allowing pointers to be delegated between different compartments, yet not directly accessed. This is analogous in high-level programming languages to providing objects with no public fields, or in low-level environments to kernel-controlled resources accessed by file descriptors. Sealing also allows cross-domain procedure calls to be implemented between compartments, with a sealed pair of a code and data capability providing an entry point.

Similarly, *pointer permissions* reduce the need for defensive copying. For example, if an input buffer that will be reused is delegated to an untrusted component then passing a read-only capability ensures that the contents of the buffer will not be modified. This ensures that the caller can continue to trust the contents (at least, to the same level that it was trustworthy before the call). To gain the same assurance without pointer permissions, the caller would need to copy the contents into a temporary buffer, turning a constant time operation into a linear time operation.

Treating pointers inside compartments as *global data capability offsets* provides several efficiency gains. First, it allows compartments to be relocated: if *bounded pointers* are protected by *tags*, they can be accurately identified as pointers and updated, as any interior pointers are simply relative integers. Second, it allows better sharing between compartments. When running multiple instances of the same library, the code and constant data (and initial copies of globals) can all be shared, even if they contain pointers.

7.2.2 Memory and Type Safety

Memory safety enforces object boundaries as intended by the programmer. Memory safety might be violated spatially (by accessing beyond the bounds of an object) or temporally (by accessing an object when it no longer exists in the program).

Enforcing memory safety both protects integrity and confidentiality of data in the program, that is, it prevents tampering with data or leaking data that was not intended to be observed.

Memory safety is a weaker property than type safety. Memory safety guarantees that an access is to a valid object. Type safety guarantees that objects may be accessed only in a way that respects the properties associated with the objects of that type. While memory safety is a prerequisite for type safety, type safety goes further to enforce programmer intent than memory safety alone.

Data and code segmentation can be seen as providing very coarse-grained memory safety – as it works on program compartments rather than individual objects.

Tags in CHERI prevent type confusion between pointers and data, a simple but crucial aspect of type safety. This is a stronger property than memory safety for these types, ensuring that the object is not only valid but that it is being used appropriately as a pointer or merely as data. Of course, *tags* alone would prevent pointers from being forged, but could not bounds-

check valid pointers. As a result, tagged pointers to be transformed to point anywhere else in the address space.

When *tags* are used to protect *bounded pointers*, a system can enforce strong spatial memory safety. Every access to memory must be via a valid pointer, which will check that it is in bounds. A number of research projects have explored the use of bounded pointers in software to enforce spatial memory safety, even in C – which does not expect bounds enforcement (e.g., [39, 66]). It might be noted that systems such as Mondrian Memory Protection permit arbitrary pointer arithmetic and check only that the resulting address is a valid object, not necessarily *the correct valid object*. *Bounded pointers* are able to distinguish between individual objects, not allowing a pointer to one object to be transformed into a pointer into foreign object.

Temporal memory safety is not natively accelerated by bounded pointers, but requires garbage collection or other conventional techniques. *Tags* make it possible to accurately differentiate between pointer and non-pointer data in the system and therefore make it possible to implement accurate garbage collection in software, even for languages such as C for which this is traditionally impossible.

Other aspects of type safety can be enforced in hardware with *pointer permissions* – for example, ensuring that immutable objects can not be accidentally modified. More importantly, *pointer permissions* make it possible to prevent confusion between code and data pointers. JIT environments must be able to both modify and execute the same memory pages. A common approach to avoid leaking pointers that are writable and executable is to map the same physical page in two locations, one writable and the other executable. With bounded pointers embedding permissions, it becomes possible to have a single memory mapping (reducing TLB pressure), yet still ensure that code with access to the executable pointer may not modify the memory.

Sealing is useful primarily for passing references between untrusted components, but can also be used to enforce type safety by protecting against opaque pointer modification. For example, pointers to C++ objects could be sealed on creation and unsealed when invoking methods on them or accessing public fields (possibly accompanied by a software type check). This would ensure that nothing that was not created as a C++ object could be accidentally interpreted as a C++ object and, similarly, that C++ objects were not accidentally modified by code unaware of their structure.

7.2.3 Control-Flow Integrity (CFI)

Control-Flow Integrity (CFI) [1] attempts to constrain the execution of a program to its intended control-flow graph to avoid control-flow hijacking – such as ROP and JOP attacks. Several CHERI components can enhance CFI schemes.

Data and code segmentation simplifies classic control-flow integrity. The classic CFI mechanism proposed to use IA32 segmentation to protect a shadow stack such that return addresses were not stored in the same stack with temporary data and needed to be accessed explicitly only by the call and return routines. *Data and code segmentation* would provide such a mechanism by allowing the executable to swap in a new segment for only the few instructions that need shadow stack access, preventing access to this memory from any other code in the application. *Data and code segmentation* would also enable convenient protected shadow spaces for other metadata that might be used by CFI schemes, simplifying schemes like cryptographically enforced control-flow integrity (CCFI) [56].

Pointer permissions allow some forms of pointers to be differentiated. With read/write/ex-

ecute permissions, it is possible to differentiate code and data pointers: any pointer that does not have the execute permission is a data pointer.

Control-Flow Integrity can also be enforced by *tags* [46, 56]. Tags might be used to protect control-flow pointers, to ensure that they are not overwritten by data. This would allow control-flow integrity that (for example) uses a shared stack – as all writes involving pointers other than those for control flow would clear the tag bits.

For stronger CFI, *tags* should at least be able to differentiate among the following data and pointers:

- Non-pointer data
- Data pointers
- Function pointers
- Return addresses

This requires at least two tag bits per 64-bit word, giving a 3.125% overhead. More complex CFI schemes require being able to differentiate between normal data pointers and pointers to C++ vtables or equivalent.

Sealing of pointers protected by *tags* could use the type field to encode a large number of pointer types. This would essentially use a single tag bit to protect a type field in spare bits of the pointer. For example, the compiler could seal the return pointer with a specific type before spilling it to the stack, preventing its use by anything that did not unseal it with the correct type. The same operation could be performed with C++ vtables, or any other type of pointer that must be explicitly differentiated.

Ultimately, CFI can not be decoupled from memory safety. Control-flow exploits traditionally depended on memory corruption; in contrast, CFI attempts to enforce the control-flow graph in the face of arbitrary memory safety violations. However, recent work at MIT has shown that CFI cannot be enforced fully in the absence of memory safety[23], although partial implementations can certainly increase the difficulty for attackers. We note that *bounded pointers* (with their automatic bounds checking) compose synergistically with CFI to reduce the attack surface of a program, and that the benefits in this section can be fully realized only in conjunction with those in Section 7.2.2.

Chapter 8

Design Rationale

During the design of CHERI, we considered many different capability architectures and design approaches. This chapter describes the various design choices; it briefly outlines some possible alternatives, and provides rationales for the selected choices.

8.1 High-Level Design Approach: Capabilities as Pointers

Our goals of providing fine-grained memory protection and compartmentalization led to an early design choice to approach capabilities as a form of pointer. This rapidly led to a number of conclusions:

- Capabilities exist within virtual address spaces, imposing an ordering in which capability protections are evaluated before virtual-memory protections; this in turn had implications for the hardware composition of the capability coprocessor and conventional MMU interact.

- Capabilities are treated by the compiler in much the same way as pointers, meaning that they will be loaded, manipulated, dereferenced, and stored via registers and to/from general-purpose memory 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 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 currently designed, including support for variable page sizes, TLB layout, and so on. The MIPS ISA describes a software-managed TLB rather than hardware page-table walking – as is present in most other ISAs. However, this is not fundamental to our approach, and either model would work.

8.2 Capability-Register File

The decision to separate the capability-register file from the general-purpose register file is somewhat arbitrary from a software-facing perspective: we envision capabilities gradually displacing general-purpose registers as pointers, but where management of the two register files will remain largely the same, with stack spilling behaving the same way, and so on. We selected the separate representation for a few pragmatic reasons:

- Coprocessor interfaces frequently make the assumption of additional register files (a la floating-point registers).
- Capability registers are quite large, and by giving the capability coprocessor its own pipeline for manipulations, we could avoid enforcing a 256-wide path through the main pipeline.
- It is more obvious, given a coprocessor-based interface, how to provide compatibility support in which the capability coprocessor is "disabled," the default configuration in order to support unmodified MIPS compilers and operating systems.

However, it is entirely possible to imagine a variation on the CHERI design in which (more similar to the manner in which the 32-bit x86 ISA was extended to support 64-bit registers) the two files were conflated and able to hold both general-purpose and capability registers. Early in our design cycle, capability registers were able to hold only true capabilities (i.e., with tags); later, we weakened this requirement by adding an explicit tag bit to each register, in order to improve support for capability-oblivious code such as memory-copy routines able to copy data structures consisting of both capabilities and ordinary data. This shifts our approach somewhat more towards a conflated approach; our view is that efficiency of implementation and compatibility (rather than maintaining a negligible effect on the software model) would be the primary reasons to select one approach or another for a particular starting-point ISA.

Another design variation might have specific capability registers more tightly coupled with general-purpose 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, and which makes associative techniques less suitable. Further, we felt that the working-set properties of the two register files might be quite different, and effectively pinning the two to one another would reduce the efficiency of both.

It is worth considering, however, that our recent interest in cursors (fat pointers) within capabilities revisits both of these ideas.

8.3 Representation of Memory Segments

CHERI capabilities represent a region of memory by its base address and length; memory accesses are relative to the base address. An alternative representation would have been for capabilities to contain an upper and lower bound on addresses within the memory region, with memory accesses being given in terms of absolute addresses but checked against the upper and lower bound.

The base and length representation was chosen because it is more convenient for arrays and structures in the C language. Given a capability for an array and an index into the array, the array element can be read with (for example) `CLB` without the need for an addition in software. (In C, all arrays are zero based. This is not the case in other languages, e.g., Ada.) The length of a structure is usually known at compile time, and the length of a capability can be set to the length of a structure with `CSetBounds`; setting an upper bound would require an additional addition instruction to compute it.

Although CHERI does not attempt to keep the base address of a capability secret, the use of base-relative (rather than absolute) addresses for memory accesses reduces the need to keep the absolute base address of a capability in a general-purpose register, and possibly might facilitate code migration to a stricter version of the architecture in which absolute addresses are secret.

The disadvantages of the base and length representation are that:

- There is no way to grant access to the very last byte of the virtual address space (a base of 0 and a length of $2^{64} - 1$ grants access to addresses 0 to $2^{64} - 2$).
- Base-relative addressing is cumbersome for code capabilities. If a program wants to call a subroutine, and to grant the subroutine execute access only to its own instructions and not to the entire program text, then the subroutine needs to be linked differently from the calling program, because branches within the subroutine will be relative to a different base.

A key concern with the current representation is its substantial size – simulation suggests that cache footprint is a dominant factor in performance, although optimization techniques such as CCured would reduce this effect. We believe that a reduction to 128-bit capability registers would come at an observable cost to both protection scalability (e.g., limiting the number of bits in a pointer to 40-48 bits rather than the full 64) as well as compartmentalization functionality (e.g., having fewer software-defined permission bits). However, in practice this may prove necessary to support widespread adoption. In that case, some care must be taken to retain current software flexibility, especially regarding very fine-grained regions of memory, which are highly desirable to support critical protection properties for C – e.g., granular stack protection and arbitrary subdivision of character-based strings into separate bounded regions. It could be that pointer compression techniques eliding specific middle bits in the address space, or possibly trading off size and granularity (e.g., bits might be invested either in describing very small objects at arbitrary alignment, or very large objects at more coarse alignment) would provide a useful middle ground.

8.4 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 but the immediate offset is treated as a signed integer.

Register offsets are treated as unsigned so that given a capability to the entire address space (except for the very last byte, as explained above), a register offset can be used to access any byte within it. Signed register offsets would have the disadvantage that negative offsets would fail the capability bounds check, and memory at offsets within the capability greater than 2^{63} would not be accessible.

Immediate offsets, on the other hand, are signed, because the C compiler often refers to items on the stack using the stack pointer as register offset plus a negative immediate offset. We have already encountered observable difficulty due to a reduced number of bits available for immediate offsets in capability-relative memory operations when dealing with larger stack-frame sizes; it is unclear what real performance cost this might have (if any), but it does reemphasize the importance of careful investment of how instruction bits are encoded.

8.5 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 arithemetic 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.

There is, however, 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 optimizations that can occur inside compilers. The following are equivalent in C:

- a[x + y
- *(a + x + y)
- (a + x)[y]
- (a + y)[x]

They would not be equivalent if they had different behavior on overflow, and the C compiler would not be able to perform optimizations that relied on this kind of reordering.

8.6 Overwriting Capabilities

In CHERI, if a capability in memory is partly overwritten with non-capability data, then the memory contents afterwards will be the capability converted to a byte representation and then overwritten.

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 to have a read-modify-write cycle to the memory, rather than just a write; in general, the MIPS architecture carefully avoids the need for a read-modify-write cycle within a single instruction. (However, once the memory system needs to deal with cache coherence, a write is not that much simpler than a read-modify-write.)

The CHERI behavior also has the advantage that programs can write to a memory location (e.g., when spilling a register onto the stack) without needing to worry about whether that location previously contained a capability or non-capability data.

A potential disadvantage is that the contents of capabilities cannot be kept secret from a program that uses them. A program can always discover the contents of a capability by overwriting part of it, then reading the result as non-capability data. In CHERI, there are intentionally other, more direct, ways for a program to discover the contents of a capability it owns, and this does not present a security vulnerability.

However, there are ABI concerns: we have tried to design the ISA in such a way that software does not need to be aware of the in-memory layout of capabilities. As it is necessarily exposed, there is a risk that software might become dependent on a specific layout. One case of particular note is in the operating-system paging code, which must save and restore capabilities and their tags separately; this can be accomplished by using instructions such as `CGetBase` on untagged values loaded from disk and then refining an in-hand capability using `CSetBounds` – an important reason not to limit capability field retrieval instructions to tagged values. We have proposed a new instruction, `CSetTag`, which would add a tag to an untagged value in a capability-register operand, authorized by a second operand holding a suitably authorized capability, to avoid software awareness of the in-memory layout.

8.7 Reading Capabilities as Bytes

In CHERI, if a data load instruction such as `CLB` is used on a memory location containing a capability, the internal representation of the capability is read. An alternative architecture would have such loads return zero, or raise an exception.

As noted above, because the contents of capabilities are not secret, allowing them to be read as raw data is not a security vulnerability.

8.8 OTypes Are Not Secret

Another consequence of the decision not to make the contents of capabilities secret is that the **otype** field is not secret. It is possible to determine the **otype** of a capability by reading it with `CGetType`, or by reading the capability as bytes. If a program has two pairs of code and data capabilities, (c_1, d_1) and (c_2, d_2) it can check if c_1 and c_2 have the same **otype** by using `CCheckType` on (c_1, d_2), or by invoking `CCall` on (c_1, d_2).

As a result, a program can tell whether it has been passed an object of **otype** O or an interposing object of **otype** I that forwards the `CCall` on to an object of **otype** O (e.g. after

having performed some additional access control checks or auditing first).

8.9 Capability Registers are Dynamically Tagged

In CHERI, capability registers and memory locations have a tag bit that indicates whether they hold a capability or non-capability data. (An alternative architecture would give memory locations a tag bit, where capability registers could contain only capabilities – with an exception raised if an attempt were made to load non-capability data into a capability register with `CLC`.)

Giving capability registers and memory locations a tag bit simplifies the implementation of `cmemcpy()`. `cmemcpy()` is a variant of `memcpy()` that copies the tag bit as well as the data, and so can be used to copy structures containing capabilities. As capability registers are dynamically tagged, `cmemcpy()` can copy a structure by loading it into a capability register and storing it to memory, without needing to know at compile time whether it is copying a capability or non-capability data.

Tag bits on capability registers may also be useful for dynamically typed languages in which a parameter to a function can be (at run time) either a capability or an integer. `cmemcpy()` can be regarded as a function whose parameter (technically a `void *`) is dynamically typed.

8.10 Separate Permissions for Storing Capabilities and Data

CHERI has separate permission bits for storing a capability versus storing non-capability data (and similarly, for loading a capability versus loading non-capability data).

(An alternative design would be just one Permit_Load and just one Permit_Store permission that were used for both capabilities and non-capability data.)

The advantage of separate permission bits for capabilities is that that there can be two protected subsystems that communicate via a memory buffer to which they have Permit_Load and Permit_Store permissions, but do not have Permit_Load_Capability or Permit_Store_Capability. Such communicating subsystems cannot pass capabilities via the shared buffer, even if they collude. (We realized that this was potentially a requirement when trying to formally model the security guarantees provided by CHERI.)

8.11 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. A range check is performed when the capability is dereferenced, so buffer overflows are prevented; thus, the offset can take on intermediate out-of-range values as long as it is not dereferenced.

An alternative architecture would have not included an offset within the capability. This could have been supported by two different capability types in C, one that could not be decremented (but was represented by just a capability) and one that supported decrementing (but was represented by a pair of a capability and a separate integer for the offset). Programming languages that did not have pointer arithmetic could have their pointers compiled as just a capability.

The disadvantage of including offsets within capabilities is that it wastes 64 bits in each capability in cases where offsets are not needed (e.g., when compiling languages that don't have pointer arithmetic, or when compiling C pointers that are statically known to never be decremented).

The alternative (no offset) architecture could have used those 64 bits of the capability for other purposes, and stored an extra offset outside the capability when it was known to be needed. The disadvantage of the no-offset architecture is that C pointers become either unable to support decrementing or enlarging: because capabilities need to be aligned, a pair of a capability and an integer will usually end up being padded to the size of two capabilities, doubling the size of a C pointer, and this is a serious performance consideration.

Another disadvantage of the no-offset alternative is that it makes the seal/unseal mechanism considerably more complicated and hard to explain. A program that has a capability for a range of types has to somehow select which type within its permitted range of types it wishes to use when sealing a particular data capability. The CHERI architecture uses the offset for this purpose; not having an offset field leads to more complex encodings when creating sealed capabilities.

By comparison, the CCured language includes both `FSEQ` and `SEQ` pointers. CHERI capabilities are analogous to CCured's `SEQ` pointers. The alternative (no offset) architecture would have capabilities that acted like CCured's FSEQ, and used an extra offset when implementing SEQ semantics.

8.12 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.

Many CHERI instructions are agnostic as to which of these two conventions for NULL is employed, but the `CFromPtr`, `CToPtr` and `CPtrCmp` operations are aware of the convention.

The advantages of NULL's **tag** bit being unset are:

- Initializing a region of memory by writing zero bytes to it will initialize all capability variables within the region to the NULL capability. Initializing memory by writing zeros is, for example, done by the C `calloc()` function, and by some operating systems.
- It is possible for code to conditionally branch on a capability being NULL by using the `CBTS` or `CBTU` instruction.

8.13 Permission Bits Determine the Type of a Capability

In CHERI, a capability's permission bits together with the **s** bit determine what kind of capability it is. A capability for a region of memory has **s** unset and *Permit_Load* and/or *Permit_Store* set; a capability for an object has **s** set and *Permit_Execute* unset; a capability to call a protected subsystem (a "call gate") has **s** set and *Permit_Execute* set; a capability that allows the owner to create objects whose type identifier (**otype**) falls within a range has **s** set and *Permit_Seal* set.

An alternative architecture would have included a separate *capability type* field, as well as the **perms** field, within each capability; the meaning of the rest of the bits in the capability would have been dependent on the value of the *capability type* field.

A potential disadvantage of not having a *capability type* field is that different kinds of capability cannot use the remaining bits of the capability in different ways.

A consequence of the architecture we have chosen is that it is possible to create many different kinds of capability (2 to the power of the number of permission bits plus **s**). Some of the kinds of capability that it is possible to create do not have a clear use case; they just exist as a consequence of the representation chosen for capabilities.

8.14 Object Types are not Addresses

In CHERI, we make a distinction between the unique identifier for an object type (the **otype** field) and the address of the executable code that implements a method on the type (the **base** + **offset** fields in a sealed executable capability).

An alternative architecture would have been to use the same fields for both, and take the entry address of an object's methods as a convenient unique identifier for the type itself.

The architecture we have chosen is conceptually simpler and easier to explain. It has the disadvantage that the type field is only 24 bits, as there is insufficient space inside the capability for more.

The alternative of treating the set of object type identifiers as being the same as the set of memory addresses enables the saving of some bits within a capability by using the same field for both. It also simplifies assigning type identifiers to protected subsystems: each subsystem can use its start address as the unique identifier for the type it implements. Subsystems that need to implement multiple types, or create new types dynamically can be given a capability with the permission *Permit_Set_Type* set for a range of memory addresses, and they are then able to use types within that range. (The current CHERI ISA does not include the *Permit_Set_Type*

permission; it would only be needed for this alternative approach). This avoids the need for some sort of privileged type manager that creates new type identifiers; such a type manager is potentially a source of covert channels. (Suppose that the type manager and allocated type identifiers in numerically ascending order. A subsystem that asks the type manager twice for a new type id and gets back n and $n + 1$ knows that no other subsystem has asked for a new type id in between the two calls; this could in principle be used for covert communication between two subsystems that were supposed to be kept isolated by the capability mechanism.)

8.15 Unseal is an Explicit Operation

In CHERI, converting a pointer to an opaque object into a pointer that allows the object's contents to be inspected or modified directly is an explicit operation. It can be done directly with the `CUnseal` operation, or by using `CCall` to run the result of unsealing the first argument on the result of unsealing the second argument.

An alternative architecture would have been one with "implicit" unsealing, where a sealed capability (**s** set) 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 [37]: the attacker calls a protected subsystem, passing a sealed capability in a parameter that the called subsystem expects to be unsealed. If unsealing is implicit, the protected subsystem can be tricked by the attacker into using its privileges to read or write to memory to which the attacker does not have access.

The disadvantage of the architecture we have chosen is that protected subsystems need to be careful not to leak capabilities that they have unsealed, for example by leaving them on the stack when they return to their caller. In an architecture with "implicit unseal", protected subsystems would just need to delete their ambient authority for the type before returning, and would not need to explicitly clean up all the unsealed capabilities that they had created.

8.16 EPCC is a Numbered Register

The exception program counter (**EPCC**) is a numbered register. An alternative architecture would have been to make **EPCC** like **PCC**, and only accessible via a special instruction. This alternative architecture would have had several advantages:

- If **EPCC** is set to a bad value (e.g., tag bit unset) and then an exception occurs, it is not obvious what the CPU should do. The behavior in this situation is explicitly undefined in the current CHERI ISA. Raising an exception is problematic because **EPCC** is already invalid at that point. If **EPCC** could only be set using a special instruction, then that instruction could check that the proposed new value of **EPCC** was valid, and raise an exception (using the old, valid, **EPCC**) if it wasn't.

- For compatibility with legacy operating systems that are unaware of capabilities, it is desirable that an exception handler that just sets CP0.**EPC** (and not **EPCC**.**offset**) should

work. But that then raises the question of what happens if operating system code changes both CP0.**EPC** and **EPCC.offset**. Ideally, these should behave as if they are the same register. But this can be complex to implement in hardware: **EPCC** and **EPC** are in different register files, and **EPCC** can be modified by many different capability instructions, because it is a numbered register; it can be complex to make all of these update CP0.**EPC** as well. If **EPCC** was not a numbered register, then only the special instruction for changing **EPCC** should need to be aware that CP0.**EPC** needs to be changed whenever **EPCC** changes.

8.17 CMove is Implemented as CIncOffset

`CMove` is an assembler pseudo-operation that expands to `CIncOffset` with an offset of zero. The `CIncOffset` instruction treats a zero offset as a special case, allowing it to be used to move sealed capabilities and values with the tag bit unset.

A separate opcode for `CMove` would have had the disadvantage that it would have used up one more opcode. The advantage of distinguishing `CMove` (of a capability that might be sealed, or invalid) from `CIncOffset` (of a capability that the programmer assumes to be unsealed, and valid, with an offset that happens to be zero) is that `CIncOffset` could raise an exception if its argument was sealed or invalid. That we don't do this isn't a security problem; but performing the check would catch some programmer errors earlier.

`CIncOffset`, unlike `CIncBase`, can be used on invalid capabilities (so that it can be used to implement increment of a `uintcap_t`). This makes it a more natural expansion for the `CMove` instruction. For security reasons, `CIncOffset` must raise an exception if the offset is non-zero, so `CIncOffset` would need to treat increment the offset of a sealed capability by zero as a special case.

We have concluded that our choice was likely an error: while not particularly harmful, in retrospect a dedicated instruction for `CMove` would be better, and we will make this change in a future ISA revision.

8.18 Instruction Set Randomization

CHERI does not include features for instruction set randomization[44]; the unforgeability of capabilities in CHERI can be used as an alternative method of providing control flow integrity.

However, instruction set randomization would be easy to add, as long as there are enough spare bits available inside a capability (the 128 bit representation of capabilities does not have many spare bits). Code capabilities could contain a key to be used for instruction set randomization, and capability branches such as `CJR` could change the current ISR key to the value given in the capability that is branched to.

8.19 ErrorEPC does not have a capability equivalent

The capability register **EPCC** corresponds to the CP0 register **EPC** in the MIPS ISA; there is no capability equivalent of the MIPS **ErrorEPC** register. The BERI1 implementation does not support `ERET` to **ErrorEPC** as the circumstances under which this might be needed (e.g.,

return from an ECC or parity error) do not occur with BERI1. As we don't support this part of the MIPS ISA in BERI, there is no need to have an equivalent capability register in CHERI (and making it a numbered register would consume scarce numbered registers).

An future extension of the CHERI ISA that both supported `ERET` to **ErrorEPC** and that had special capability registers as unnumbered registers might add an additional special capability register corresponding to **ErrorEPC**.

8.20 KCC, KDC, KR1C, and KR2C are Numbered Registers

The MIPS ISA reserves two general-purpose registers, `$k0` and `$k1` for use in exception handlers, such that the context switch from userspace to kernel can be entirely software-defined. We mirrored this design choice in the reservation of a number of capability registers for use by kernel exception – **KCC** and **KDC**, but also **KR1C** and **KR2C**. For the reserved MIPS registers, there is an opportunity for a small information leak from kernel to userspace, and can be managed by having the software supervisor clear the registers before an exception handler returns. For capability registers such as **KCC** and **KDC**, which are intended to retain privileged contents during normal usermode execution, we rely on **PCC** permission bits (combined with ring-based protections) to control access.

This design avoided the need to introduce further instructions to access reserved registers. In retrospect, however, we are not clear that this was the cleanest design choice, and have begun a migration towards explicitly loading (and storing) **KCC** and **KDC** from (and to) special registers via special instructions: `CGetKCC`, `CSetKCC`, `CGetKDC`, and `CSetKDC`. This is especially important if one were to contemplate a merged general-purpose and capability register file, in which case avoiding further reservations in that file will limit ABI disruption. Finally: these registers are only used very infrequently, and as such take up valuable space that could be available to the compiler, meaning that using up encoding and register-file space is a less good use of micro-architectural resources.

8.21 A Single Privileged Permission Bit

In the current version of the CHERI, one of the capability permission bits is reserved to authorize 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. We employ a single permission bit to conserve space (especially in 128-bit capabilities), but also because it offers a coherent view on architectural privilege: many of the privileged architectural instructions allow bypass of in-address-space memory protection in different ways, and using subsets of those operations safely would be quite difficult. In earlier versions of the CHERI ISA, we employed multiple privileged bits, but did not find the differentiation useful in practical software design. The current single bit authorizes access to privileged CP2 state as relates to the capability model, but we plan also to use it to control access to other privileged features over time, such as instructions that manipulate the MMU, return from an exception handler, etc.

8.22 Interrupts and CCall use the same KCC/KDC

MIPS executes all exception handlers within the same privileged ring, and we have inherited that design choice for our `CCall` exception handler, both with respect to classical ring-based security and also the decision to use a single set of **KCC/KDC** special registers. Given that domain transition without user address spaces does not actually require supervisor privilege, it would make substantial sense to shift the software-defined `CCall`/`CReturn` mechanism to a userspace exception handler. These are not supported by MIPS, and substantial prototyping would be required to evaluate this approach. If that were to be implemented, then it would be necessary to differentiate the code and data capabilities for the domain-transition implementation from the kernel's own code and data capabilities – possibly via additional special registers configured and switched by the kernel on behalf of the userspace language runtime.

8.23 Compressed Capabilities

256-bit capabilities provide for byte-granularity protection, allowing arbitrary subsets of the address space to be described, as well as 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 at substantial space savings.

8.23.1 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, and 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 registers. The only visible semantic changes between 256-bit and 128-bit operation should be: 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 represen-

tation bounds for small memory allocations ($< \frac{3}{4}MiB$) – e.g., as occur on the stack or when performing string or image processing.

8.23.2 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 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: this is most frequently heap and stack allocation, but 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: at least word or pointer alignment so that allocated data structures can store at natural alignment, but also (for larger allocations) 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 ($< 1MiB$), but for larger subsets may experience precision effects. These are accepted in our programmer model, and could permit buffer overflows between subsets that, in the 256-bit model, would be prevented. Arbitrary subsetting, unless specifically annotated to require full precision, will utilize `CSetBounds`, which can return monotonically non-increasing but potentially imprecise bounds.

Two further cases required careful consideration: object capabilities, and the default data capability, for quite different reasons. Object capabilities require additional capability fields (software-defined permission bits and the fairly wide object type field). The default data capability is an ordinary 128-bit capability, but has the property that use of a full cursor (base

plus offset) introduces a further arithmetic addition in a critical path of MIPS loads and stores. In both cases, we have turned to reduced precision (i.e., increased alignment requirements) to eliminate these problems, looking to minimum page-granularity alignment of bounds while retaining fully precise pointers. By requiring strong alignment for default data capabilities, the extra addition becomes a logical or while 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 provision comes at a slight cost to precision: i.e., we accept slightly stronger alignment requirements in return for the ability to allow pointers to be somewhat out of bounds.

8.23.3 Candidate Designs for Compressed Capabilities

Compressed capabilities in the CHERI specification are, in fact, the third candidate scheme that we considered. We document the earlier two schemes here for comparison.

We defined two initial 128-bit formats based on floating-point techniques, one that encodes bounds with differences from the pointer, and a second based on the "low-fat pointer" design [47]. The third candidate synthesizes the advantages of both and is our recommended approach.

CHERI-128 candidate 1

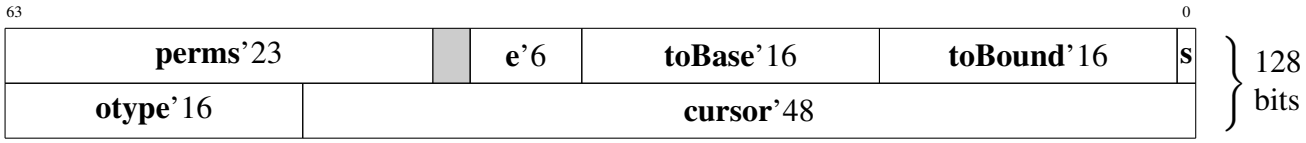


Figure 8.1: CHERI-128 c1 memory representation of a capability

s The **s** flag corresponds directly to the architectural **s** bit, which indicates that a capability is sealed.

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.

$$\mathbf{mask} = -1 << \mathbf{e}$$

$$\mathbf{base} = (\mathbf{toBase} << \mathbf{e}) + \mathbf{cursor}\&\mathbf{mask}$$

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 user-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 only defined when the capability is sealed. If **s** is cleared, the **otype** is zero, and these bits are 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.

$$alignment\_bits = \lceil log_2(X) \rceil - 14$$

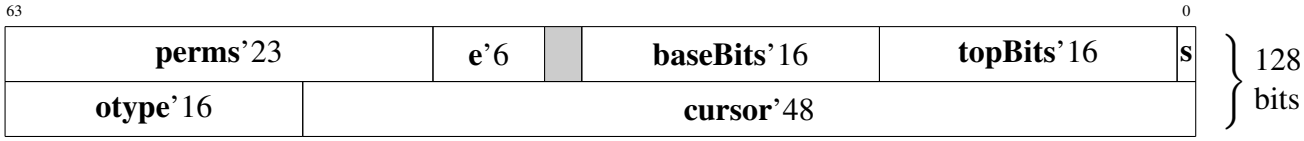


Figure 8.2: CHERI-128 c2 memory representation of a capability

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.

$$\mathbf{base} = \{\mathbf{cursor}[63 : \mathbf{e} + 16] + \mathbf{correction}, \mathbf{baseBits}\} \ll \mathbf{e}$$

The bits above $(\mathbf{e} + 16)$ in **cursor** may differ from **base** by at most 1, i.e.

$$\mathbf{correction} = f(\mathbf{baseBits}, \mathbf{topBits}, \mathbf{cursor}[\mathbf{e} + 15 : \mathbf{e}]) = (1, 0, or - 1)$$

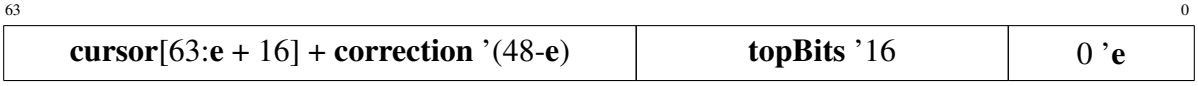


Figure 8.3: CHERI-128 c2 base construction

topBits This 16-bit field gives bits to be inserted into the bits of **cursor** at **e** to produce the representable top of the capability equal to (top - 1024). To compute the top, a circuit must insert **topBits** at **e**, set the lower bits to 0, subtract 1024, and add a potential carry bit. The carry bit is implied if **topBits** is less than **baseBits**, as the top will never be less than the bottom of an object.

$$\mathbf{top} = \{\mathbf{cursor}[63 : \mathbf{e} + 16] + \mathbf{correction}, \mathbf{topBits}, 0\}$$

The bits above $(\mathbf{e} + 16)$ in **cursor** may differ from **top** by at most 1:

$$\mathbf{correction} = f(\mathbf{baseBits}, \mathbf{topBits}, \mathbf{cursor}[\mathbf{e} + 15 : \mathbf{e}]) = (1, 0, or - 1)$$

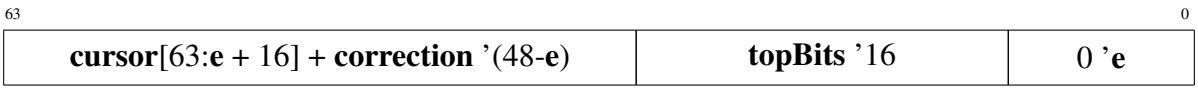


Figure 8.4: CHERI-128 c2 top bound construction

Candidate 2 Notes Candidate 2 is inspired by "Low-fat pointers" [47] which inserts 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. We choose the edges of this range to be $(\mathbf{base} + \mathbf{top})/2 \pm 2^{(\mathbf{e}+15)}$, or the furthest representable point from the centre of the region.

CHERI-128 candidate 3

After substantial exploration, we adopted a third compression model (see Section 5.9), 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.

Alternative exponents The CHERI-128 scheme presented in Chapter 5.9 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 but 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.

Chapter 9

CHERI in High-Assurance Systems

We intend to produce a chapter of the ongoing CHERI formal methods document [74]) (or possibly a separate technical report), describing how we used formal methods when developing CHERI. In the present chapter, we give a 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.

9.1 Unpredictable Behavior

In the pseudocode for the CHERI instructions in Chapter 6, 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 the memory nor the capability registers are modified.

The test suite for our CHERI1 FPGA implementation checks that the CPU follows known CHERI1-specific behavior in the "unpredictable" cases.

9.2 Bypassing the Capability Mechanism Using the TLB

If a program can modify the TLB (the status register has CU0 set, KSU not equal to 2, EXL set or IRL set), then it can bypass the capability mechanism by modifying the TLB. Although composition with the Memory Management Unit and virtual-addressing mechanism in this manner is a critical and intentional part of our design, it is worth considering the implications from the perspective of high-assurance design. The "attack" is as follows: Consider a location in memory whose virtual address is not accessible using the capability mechanism; take its physical address and change the TLB so that its new virtual address is one to which you have a capability, and then access the data through the new virtual address. There are several ways to prevent this attack:

- In CheriBSD, user-space programs are unable to modify the TLB (except through system calls such as `mmap`), and thus cannot carry out this attack. This security argument makes it explicit that the security of the capability mechanism depends on the correctness of the underlying operating system. However, this may not be adequate for high-assurance systems.

- Similarly, a high-assurance microkernel could run untrusted code in user space, with KSU=2, CU0 false, EXL false, and IRL false. A security proof for the combined hardware-software system could verify that untrusted code cannot cause this condition to become false except by reentering the microkernel via a system call or exception.

- A single-address-space microkernel that has no need for the TLB could run on a CHERI-enabled CPU without a TLB. Our CHERI1 FPGA prototype can be synthesized in a version without a TLB, and our formal model in the L3 specification language includes a TLB-less variant. Removing the TLB for applications that don't need it saves chip area, and removes the risk that the TLB could be used as part of an attack.

- We are considering future extensions to CHERI that would allow the capability mechanism to be used for sandboxing in kernel mode; these would allow more control over access to the TLB when in kernel mode. As well as enabling sandboxing of device drivers in monolithic kernels such as that of CheriBSD, the same mechanism could also be used by microkernels.

9.3 Malformed Capabilities

The encoding formats for capabilities can represent values that can never be created using the capability instructions while taking the initial contents of the capability registers as a starting point. For example, in the 256-bit representation, there are bit patterns corresponding to **base** + **length** $> 2^{64}$. The capability registers are cleared on reset, so there will never be malformed capabilities in the initial register contents, and a CHERI instruction will never create malformed capabilities from well-formed ones. However, DRAM is not cleared on system reset, so that it is possible that the initial memory might contain malformed capabilities with the tag bit set.

Operating systems or microkernels are expected to initialize memory before passing references to it to untrusted code. (If you give untrusted code a capability that has the *Load_Capability* permission and refers to uninitialized memory, you don't know what rights you are delegating

to it.) This means that untrusted code should not be in a position to make use of malformed capabilities.

There are (at least) two implementation choices. An implementation of the CHERI instructions could perform access-control checks in a way that would work on both well-formed and malformed capabilities. Alternatively, the hardware could be slightly simplified by performing the checks in a way that might behave unexpectedly on malformed capabilities, and then rely on the capability mechanism (plus the operating system initializing memory) to guarantee that they will never become available to untrusted code.

If the hardware is designed to guard against malformed capabilities, this presents special difficulties in testing. No program whose behavior is defined by the ISA specification will ever trigger the case of encountering a malformed capability. (Programs whose behavior is "unpredictable", because they access uninitialized memory, may encounter them). However, some approaches to automatic test generation may have difficulty constructing such tests.

More generally, however, uninitialized memory might also contain highly privileged and yet entirely well-formed capabilities, and hence references to that memory should be given to less trustworthy code only after suitable clearing. This requirement is present today for current hardware, as uncleared memory on boot might contain sensitive data from prior boots, but this requirement is reinforced in a capability-oriented environment.

9.4 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")[67] 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[38] 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[32] 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.
- 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 rentered 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.

Chapter 10

Future Directions

The CTSRD project, of which CHERI is just one element, has now been in progress for six years. Our focuses to date have been in several areas:

1. Design the CHERI Instruction-Set Architecture based on hybrid memory-capability and object-capability models. As part of this work, develop PVS/SAL formal models of the ISA, and analyze properties about program expressivity; subsequently, develop a more complete L3 model of both MIPS and CHERI ISAs that would allow automated test generation, trace validation, and eventually, software verification.

2. Flesh out the ISA feature set in CHERI to support a real-world operating system – primarily, this has consisted of adding support for the system management coprocessor, CP0, which includes the MMU and exception model, but also features such as a programmable interrupt controller (PIC). We have also spent considerable time refining successive versions of the ISA intended to better support high levels of C-language compatibility, as well as automatic use by compilers, which are now implemented. This work has incorporated ideas from, but also gone substantially beyond, the C-language fat-pointer research literature.

3. Prototype, test, and refine CHERI ISA extensions, which are incorporated via a new capability coprocessor, CP2. We have open sourced the BERI and CHERI processor designs in order to allow reproducible experimentation with our approach, as well as to act as an open-source platform for other future hardware-software research projects.

4. Port the FreeBSD operating system first to a capability-free version of CHERI, known as BERI. This is known as FreeBSD/BERI, and this support has been upstreamed such that new releases of FreeBSD support the BERI processor and its peripheral devices.

5. Adapt FreeBSD to make use of CHERI features – first by adapting the kernel to maintain new state and provide object invocation, and then low-level system runtime elements, such as the system library and runtime linker. This is known as CheriBSD. We have also developed a pure-capability system-call ABI and process environment known as CheriABI, which pushes to an extreme point the use of capabilities to represent all pointers in user code generation and in interaction with a conventional kernel. While open sourced, these changes remain outside of the upstream FreeBSD repository due to their experimental nature.

6. Adapt the Clang/LLVM compiler suite to be able to generate CHERI ISA instructions as directed by C-language annotations, exploring a variety of language models, code-generation models, and ABIs. We have 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.

7. Begin to develop semi-automated techniques to assist software developers in compartmentalizing applications using Capsicum and CHERI features. This is a subproject known as Security-Oriented Analysis of Application Programs (SOAAP), and performed in collaboration with Google.

8. Develop FPGA-based demonstration platforms, including an early prototype on the Terasic tPad, and more mature server-style and tablet-style prototypes based on the Terasic DE4 board. We have also made use of CHERI on the NetFGPA 10G board.

9. Develop a Qemu-based fast ISA simulator, making the CHERI model more accessible to those without easy or affordable access to hardware emulation platforms.

10. Develop techniques for translating BSV language hardware specifications into PVS representations so that they can be used for formal analysis purposes. The SRI PVS tool suite has been embedded in the Bluespec SystemVerilog compiler chain to enable formal verification, model checking (SAL), and SMT solving (Yices) inline with the compilation.

11. In addition, we are working on a new project task to consider extensions of the CHERI hardware-software architecture that would enable trustworthy systems to be developed despite the presence of numerous potentially untrustworthy microprocessors and I/O devices, some which might otherwise have overly permissive unmonitored direct access to processor memory.

We have made a strong beginning, but clearly there is still much to do in our remaining CTSRD efforts. From this vantage point, we see a number of tasks ahead, which we detail in the next few sections.

In addition, our companion project for DARPA's Mission-oriented Resilient Cloud program has been extending some of the CHERI concepts, notably pursuing multicore and multithreaded CHERI in anticipation of its use in network controllers and cloud datacenters. We have also been experimenting with applications of the CHERI hardware architecture for trustworthy switching and other purposes, and considering efficient operating systems (e.g., DIOS and MirageOS) that might be well adapted to CHERI.

10.1 An Open-Source Research Processor

One of our goals for the CHERI processor is to produce a reference BSV processor implementation, which can then be used as a foundation not only for CHERI, but also for other research projects relating to the hardware-software interface. Capability processor extensions to the MIPS ISA would then be a core research result from this project, but also the first example of research conducted on the reference processor.

We have spent significant effort in open-sourcing CHERI, including enhancing our test suite, updating documentation, and creating a new open-source license intended for hardware-software projects (derived from the Apache software license).

10.2 Formal Methods for BSV

We have created prototype descriptions of the CHERI ISA in PVS and SAL, and are collaborating with the REMS project at Cambridge to develop an L3 model of the MIPS ISA, with the intent of also applying it to CHERI. We have used our formal models to automatically generate test suites, and to prove higher-level properties about what the ISA can represent. We have created new tools to automatically process BSV hardware specifications for use in theorem proving and model checking, and developed new tools to improve SMT performance and to extract higher-level properties from hardware designs. Our longer-term goal has been to link formal models of the hardware itself with the ISA specification and software compiled to that ISA. To that end, we have developed an effective front-end tool (Smten) that embeds the SRI formal analysis tools (PVS, SAL, Yices) into the BSV build chain, considerably simplifying the analysis effort. We hope that, by the completion of the CTSRD project, we will also be able to prove a number of basic but interesting properties about the hardware design, such as correctness of pipelining and the capability coprocessor. Perhaps we may even be able to extend the formal analysis into the lower-layer system software – such as properties relating to capability-based protection in sandboxing and compiling. See the companion draft document on CHERI Formal-Methods [74].

10.3 ABI and Compiler Development

We have targeted our CHERI ISA extensions at compiler writers, rather than for direct use by application authors. This has required us to design new Application Binary Interfaces (ABIs), and to extend the C programming language to allow specification of protection properties by programmers. We have extended the GNU assembler and the Clang/LLVM compiler suite to generate CHERI instructions, and begun to experiment with modifications to applications. We anticipate significant future work in this area to validate our current approach, but also to extend these ideas both in C and other programming languages, such as Objective C. We are also interested in CHERI instructions as a target for just-in-time compilation by systems such as Dalvik.

10.4 Hardware Capability Support for FreeBSD

With a capability processor prototype complete, and a FreeBSD/BERI port up and running, we have begun an investigation into adding CHERI capability support to the operating system. Currently, the CheriBSD kernel is able to maintain additional per-thread CHERI state for user processes via minor extensions to the process and thread structures, as well as exception-handling code. We have also prototyped object-capability invocation, which we are in the process of integrating with the operating system. A number of further tasks remain, including continuing to explore userspace memory models and their relationship with the userspace

object-capability model, and also continued development of the userspace address-space executive that manages memory, code linking, and so on. This work depends heavily on continuing work on Clang/LLVM support for capabilities.

We need to explore security semantics for the kernel to limit access to kernel services (especially system calls) from sandboxed userspace code. This will require developing our notions of privilege described in Chapter 4; the userspace runtime and kernel must agree on which services (if any) are available without passing through a trusted protected subsystem, such as the runtime linker.

Ideally, the kernel should make use of capabilities, initially for bounded memory buffers (offering protection against kernel buffer overflows, for example), but later protected subsystems. An iterative refinement of hardware and software privilege models will be required: for example, a sandboxed kernel subsystem should not be able to modify the TLB without going through a kernel protected subsystem, meaning that simple ring-based notions of privilege for MMU access are insufficient.

10.5 Evaluating Performance and Programmability

This report describes a fundamental premise: that through an in-address space capability model, performance and programmability for compartmentalized applications can be dramatically improved. Once the capability coprocessor and initial programming language, toolchain, and operating system support come together, validating this claim will be critical. We anticipate making early efforts to apply compartmentalization to base system components: elements of the operating system kernel, critical userspace libraries, and critical userspace applications.

Our hybrid capability architecture will ease this experimentation, making it possible to apply, for example, capabilities within `zlib` without modifying an application as a whole. Similarly, capability-aware applications should be able to invoke existing library services, even filtering their access to OS services – a similarly desirable hypothesis to test.

We are concerned not only with whether we can express the desired security properties, but also compare their performance with MMU-based compartmentalization, such as that developed in the Capsicum project. An early element of this work will certainly include testing of security context-switch speed as the number of security domains increases, in order to confirm our hypothesis regarding TLB size and highly compartmentalized software, but also that capability context switching can be made orders of magnitude faster as software size scales.

Appendix A

CHERI ISA Version History

This appendix contains a detailed version history of the CHERI Instruction-Set Architecture. This report was previously made available as the *CHERI Architecture Document*, but is now the *CHERI Instruction-Set Architecture*.

1.0 This first version of the CHERI architecture document was prepared for a six-month deliverable to DARPA. It included a high-level architectural description of CHERI, motivations for our design choices, and an early version of the capability instruction set.

1.1 The second version was prepared in preparation for a meeting of the CTSRD External Oversight Group (EOG) in Cambridge during May 2011. The update followed a week-long meeting in Cambridge, UK, in which many aspects of the CHERI architecture were formalized, including details of the capability instruction set.

1.2 The third version of the architecture document came as the first annual reports from the CTSRD project were in preparation, including a decision to break out formal-methods appendices into their own *CHERI Formal Methods Report* for the first time. With an in-progress prototype of the CHERI capability unit, we significantly refined the CHERI ISA with respect to object capabilities, and matured notions such as a trusted stack and the role of an operating system supervisor. The formal methods portions of the document was dramatically expanded, with proofs of correctness for many basic security properties. Satisfyingly, many 'future work' items in earlier versions of the report were becoming completed work in this version!

1.3 The fourth version of the architecture document was released while the first functional CHERI prototype was in testing. It reflects on initial experiences adapting a microkernel to exploit CHERI capability features. This led to minor architectural refinements, such as improvements to instruction opcode layout, some additional instructions (such as allowing `CGetPerms` 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 per-

missions; 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.

`GetPCC` now returns **PC** as the **offset** field of the returned **PCC** rather than storing it to a general-purpose register. `CJR` and `CJALR` now accept target **PC** values via the offsets of their jump-target capability arguments rather than via explicit general-purpose 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 register for frequent arithmetic operations on pointers.

Interactions between **EPC** and **EPCC** are now better specified, including that use of untagged capabilities has undefined behavior. CBTS and CBTU are now defined to use branch-delay slots, matching other MIPS-ISA branch instructions. CJALR is defined as suitably incrementing the returned program counter, along with branch-delay slot semantics. Additional software-path pseudocode is present for CCall and CReturn.

CAndPerm and CGetPerm use of argument-register or return-register permission bits has been clarified. Exception priorities and cause-code register values have been defined, clarified, or corrected for CClearTag, CGetPCC, CSC, and CSeal. Sign or zero extension for immediates and offsets are now defined CL, CS, and other instructions.

Exceptions caused due to TLB bits controlling loading and storing of capabilities are now CP2 rather than TLB exceptions, reducing code-path changes for MIPS exception handlers. These TLB bits now have modified semantics: **LC** now discards tag bits on the underlying line rather than throwing an exception; **SC** will throw an exception only if a tagged store would result, rather than whenever a write occurs from a capability register. These affect CLC and CSC.

Pseudocode definitions now appear earlier in the chapter, and have now been extended to describe **EPCC** behavior. The ISA reference has been sorted alphabetically by instruction name.

1.12 This is an interim release as we begin to grapple with 128-bit capabilities. This requires us to better document architectural assumptions, but also start to propose changes to the instruction set to reflect differing semantics (e.g., exposing more information to potential capability compression). A new CSetBounds instruction is proposed, which allows both the base and length of a capability to be set in a single instruction, which may allow the micro-architecture to reduce potential loss of precision. Pseudocode is now provided for both the pure-exception version of the CCall instruction, and also hardware-accelerated permission checking.

1.13 This is an interim release as our 128-bit capability format (and general awareness of imprecision) evolves; this release also makes early infrastructural changes to support an optional converging of capability and general-purpose register files.

Named constants, rather than specific sizes (e.g., 256-bit vs. 128-bit) are now used throughout the specification. Reset state for permissions is now relative to available permissions. Two variations on 128-bit capabilities are defined, employing two variations on capability compression. Throughout the specification, the notion of "representable" is now explicitly defined, and non-representable values must now be handled.

The definitions of `CIncOffset`, `CSetOffset`, and `CSeal` have been modified to reflect the potential for imprecision. In the event of a loss of precision, the capability base, rather than offset, will be preserved, allowing the underlying memory object to continue to be accurately represented.

Saturating behavior is now defined when a compressed capability's length could represent a value greater than the maximum value for a 64-bit MIPS integer register.

EPCC behavior is now defined when a jump or branch target might push the offset of PCC outside of the representable range for EPCC.

`CIncBase` and `CSetLen` are deprecated in favor of `CSetBounds`, which presents changes to base and bounds to the hardware atomically. The `CMove` pseudo-operation is now implemented using `CIncOffset` rather than `CIncBase`. `CFromPtr` has been modified to behave more like `CSetOffset`: only the offset, not the base, is modified. Bug fixes have been applied to the definitions of `CSetBounds` and `CUnseal`.

Several bugs in the specification of `CLC`, `CLLD`, `CSC`, and `CSD`, relating to omissions during the update to capability offsets, have been fixed. `CLC`'s description has been updated to properly reflect its immediate argument.

New instructions `CClearHi` and `CClearLow` 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 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 registers, supplementing `CClearHi` and `CClearLo`. All four instructions are described together under `CClearReg`.

A new `CSetBoundsExact` instruction is defined, allowing an exception to be thrown if an attempt to narrow bounds cannot occur precisely. This is intended for use in memory allocators where it is a software invariant that bounds are always exact. A new exception code is defined for this case.

A full range of data widths are now support for capability-relative load-linked, store conditional: CLLB, CLLH, CLLW, CLLD, CSCB, CSCH, CSCW, and CSCD (as well as unsigned load-linked variations). Previously, only a doubleword variation was defined, but cannot be used to emulate the narrower widths as fine-grained bounds around a narrow type would throw a bounds-check exception. Existing load-linked, store-conditional variations for capabilities (CLLC, CSCC) have been updated, including with respect to opcode assignments.

A new 'candidate three' variation on compressed capabilities has been defined, which differentiates sealed and unsealed formats. The unsealed variation invests greater numbers of bits in bounds accuracy, and has a full 64-bit cursor, but does not contain a broader set of software-defined permissions or an object-type field. The sealed variation also has a full 64-bit cursor, but has reduced bounds accuracy in return for a 20-bit object-type field and a set of software-defined permissions.

'Candidate two' of compressed capabilities has been updated to reflect changes in the hardware prototype by reducing toBase and toBound precision by one bit each.

Explicit equations have been added explaining how bounds are calculated from each of the 128-bit compressed capability candidates, as well as their alignment requirements.

Exception priorities have been documented (or clarified) for a number of instructions including CJALR, CLC, CLLD, CSC, CSCC, CSetLen, CSeal, CUnSeal, and CSetBounds.

The behavior of CPtrCmp is now defined when an undefined comparison type is used.

It is clarified that capability store failures due to TLB-enforced limitations on capability stores trigger a TLB, rather than a CP2, exception.

A new capability comparison instruction, CEXEQ, checks whether all fields in the capability are equal; the previous CEQ instruction checked only that their offsets pointed at the same location.

A new capability instruction, CSUB, allows the implementation of C-language pointer subtraction semantics with the atomicity properties required for garbage collection.

The list of BERI- and CHERI-related publications, including peer-reviewed conference publications and technical reports, has been updated.

1.15 - UCAM-CL-TR-876 This version of the CHERI ISA, *CHERI ISAv4*, has been prepared for publication as a University of Cambridge technical report.

The instructions CIncBase and CSetLen (deprecated in version 1.13 of the CHERI ISA) have now been removed in favor of CSetBounds (added in version 1.12 of the CHERI ISA). The new instruction was introduced in order to atomically expose changes to both upper and lower bounds of a capability, rather than requiring them to be updated separately, required to implement compressed capabilities.

The design rationale has been updated to better describe our ongoing exploration of whether special registers (such as **KCC**) should be in the capability register file, and the potential implications of shifting to a userspace exception handler for CCall/CReturn.

1.16 This is an interim update of the instruction-set specification in which aspects of the 128-bit capability model are clarified and extended.

The "candidate 3" unsealed 128-bit compressed capability representation has been to increase the exponent field (**e**) to 6 bits from 4, and the **baseBits** and **topBits** fields have been reduced to 20 bits each from the 22 bits. **perms** has been increased from 11 to 15 to allow for a larger set of software-defined permissions. The sealed representation has also been updated similarly, with a total of 10 bits for **otype** (split over **otypeLow** and **otypeHigh**), 10 bits each for **baseBits** and **topBits**, and a 6-bit exponent. The algorithm for decompressing a compressed capability has been changed to better utilize the encoding space, and to more clearly differentiate representable from in-bounds values. A variety of improvements and clarifications have been made to the compression model and its description.

Differences between, and representations of, permissions for 128-bit and 256-bit capability are now better described.

Capability unrepresentable exceptions will now be thrown in various situations where the result of a capability manipulation or operation cannot be represented. For manipulations such as `CSeal` and `CFromPtr`, an exception will be thrown. For operations such as `CBTU` and `CBTS`, the exception will be thrown on the first instruction fetch following a branch to an unrepresentable target, rather than on the branch instruction itself. CHERI1 and CHERI2 no longer differ on how out-of-bounds exceptions are thrown for capability branches: it uniformly occurs on fetching the target instruction.

The ISA specification makes it more clear that `CEQ`, `CNE`, `CL[TE]U`, and `CEXEQ` are forms of the `CPtrCmp` instruction.

The ISA todo list has been updated to recommend a capability conditional-move (`CCMove`) instruction.

There is now more explicit discussion of the MIPS n64 ABI, Hybrid ABI, and Pure-Capability ABI. Conventions for capability-register have been updated and clarified – for example, register assignments for the stack capability, jump register, and link register. The definition that **RCC**, the return code capability, is register **C24** has been updated to reflect our use of **C17** in actual code generation.

Erroneous references to an undefined instruction `CSetBase`, introduced during removal of the `CIncBase` instruction, have been corrected to refer to `CSetBounds`.

1.17 This is an interim update of the instruction-set architecture enhancing (and specifying in more detail) the CHERI-128 "compressed" 128-bit capability format, better aligning the 128-bit and 256-bit models, and adding capability-related instructions required for more efficient code generation. This is a draft release of what will be considered *CHERI ISAv5*.

The chapter on ISA design now includes a section describing "deep" versus "surface" aspects of the CHERI model as mapped into the ISA. For example, use of tagged capabilities is a core aspect of the model, but the particular choice to have a separate capability register file, rather than extending general-purpose 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 5.9.

Alignment requirements for software allocators (such as stack and heap allocators) in the presence of capability compression are now more concisely described.

The immediate operands to various load and store instructions, including `CLC`, `CSC`, `CL[BHWD][U]`, and `CS[BHWD]` are now "scaled" by the width of the data being stored (with the exception of capability stores, where scaling is by 16 bytes regardless of in-memory capability size). This extends the range of capability-relative loads and stores, permitting a far greater proportion of stack spills to be expressed without additional stack-pointer modification. This is a binary-incompatible change to the ISA.

The textual description of the `CSeal` instruction has been updated to match the pseudocode in using $>=$ rather than $>$ in selecting an exception code.

A redundant check has been removed in the definition of the `CUnseal` instruction, and an explanation added.

Opcodes have now been specified for the `CSetBoundsExact` and `CSub` instructions.

To improve code generation when constructing a **PCC**-relative capability as a jump target, a new `CGetPCCSetOffset` instruction has been added. This instruction has the combined effects of performing sequential `CGetPCC` and `CSetOffset` operations.

A broader set of opcode rationalizations and cleanups have been applied across the ISA, to facilitate efficient decoding and future use of the opcode space. This includes changes to `CGetPCC`.

C25 is no longer reserved for exception-handler use, as **C27** and **C28** are already reserved for this purpose. It is therefore available for ABI use.

The 256-bit architectural capability model has been updated to use a single system permission, Access_System_Registers, to control access to exception-handling and privileged ISA state, rather than splitting it over multiple permissions. This brings the permission models in 128-bit and 256-bit representations back into full alignment from a software perspective. This also simplifies permission checking for instructions such as `CClearReg`. 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 de-emphasized, 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 interation into the processor pipeline and memory subsystem.

A reference has been added to Robert Norton’s PhD dissertation on optimized CHERI domain switching.

A reference has been added to our PLDI 2016 paper on C-language semantics and their interaction with the CHERI model.

The object-type field in both 128-bit and 256-bit capabilities is now 24 bits, with Top and Bottom fields reduced to 8 bits for sealed capabilities. This reflects a survey of current object-oriented software systems, suggesting that 24 bits is a more reasonable upper bound than 20 bits.

The assembly arguments to `CJALR` have been swapped for greater consistency with jump-and-link register instructions in the MIPS ISA.

We have reduced the number of privileged permissions in the 256-bit capability model to a single privileged permission, Access_System_Registers, to match 128-bit CHERI. This is a binary-incompatible change.

We have improved the description of the CHERI-128 model in a number of ways, including a new section on the CHERI-128 representable bounds check.

The architecture chapter contains a more detailed discussion of potential ways to reduce the overhead of CHERI by reducing the number of capability registers, converging the general-purpose 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.

Appendix B

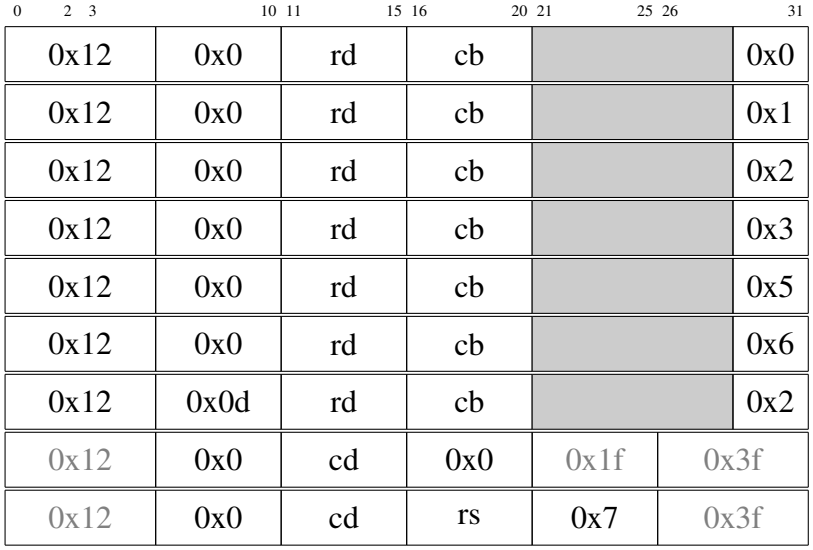
CHERI ISA Quick Reference

This appendix provides a quick reference for CHERI instruction encodings.

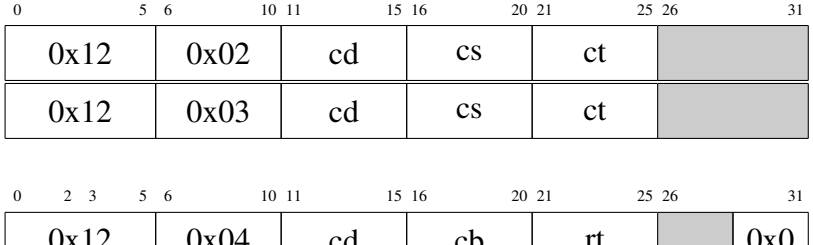
B.1 Existing Encodings

The following encodings are correct for implementations that exist at the time of this document's publication.

B.1.1 Capability Inspection Instructions



B.1.2 Capability Modification Instructions



0x12	0x04	cd	cb		0x5	CClearTag cd, cb
0x12	0x0d	cd	cb	rt	0x0	CIncOffset cd, cb, rt
0x12	0x0d	cd	cb	rt	0x1	CSetOffset cd, cb, rt
0x12	0x01	cd	cb	rt		CSetBounds cd, cb, rt
0x12	0x0	cd	cb	rt	0x9	CSetBoundsExact cd, cb, rt

B.1.3 Pointer Arithmetic Instructions

0 5	6 10	11 15	16 20	21 25	26 31	
0x12	0x0c	rd	cb	ct		CToPtr rd, cb, rt
0x12	0x04	cd	cb	rt	0x7	CFromPtr cd, cb, rt
0x12	0x0	rt	cb	ct	0xa	CSub rt, cb, ct

B.1.4 Pointer Comparison Instructions

0 2	3 5	6 10	11 15	16 20	21 25	26 31	
0x12		0x0e	rd	cb	ct	0	CEQ rd, cb, ct
0x12		0x0e	rd	cb	ct	1	CNE rd, cb, ct
0x12		0x0e	rd	cb	ct	2	CLT rd, cb, ct
0x12		0x0e	rd	cb	ct	3	CLE rd, cb, ct
0x12		0x0e	rd	cb	ct	4	CLTU rd, cb, ct
0x12		0x0e	rd	cb	ct	5	CLEU rd, cb, ct
0x12		0x0e	rd	cb	ct	6	CEXEQ rd, cb, ct

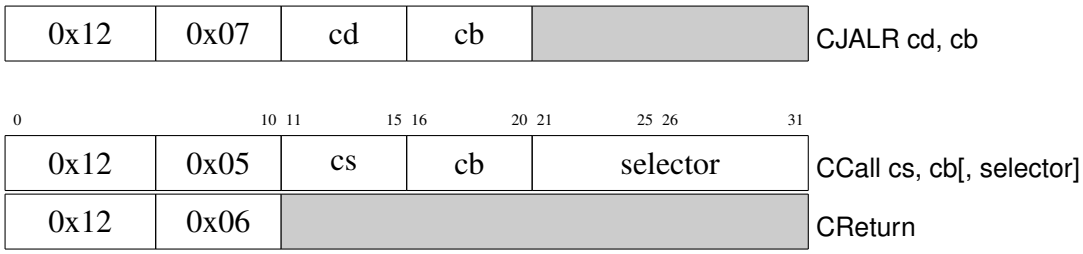
B.1.5 Exception Handling Instructions

0 2	3 5	6 10	11 15	16 20	21 25	26 31	
0x12		0x0	rd	0x0		0x4	CGetCause rd
0x12		0x04	0x0	0x0	rt	0x4	CSetCause rd

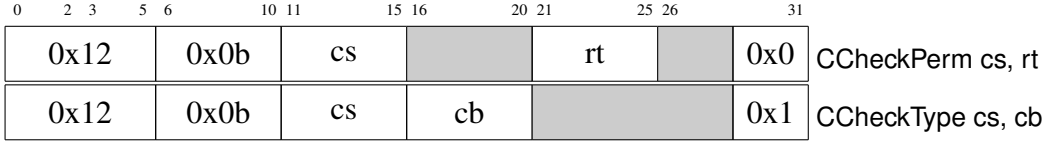
B.1.6 Control Flow Instructions

0 15	16 20	21 25	26 31	
0x12	0x09	cd	offset	CBTU cd, offset
0x12	0x0a	cd	offset	CBTS cd, offset

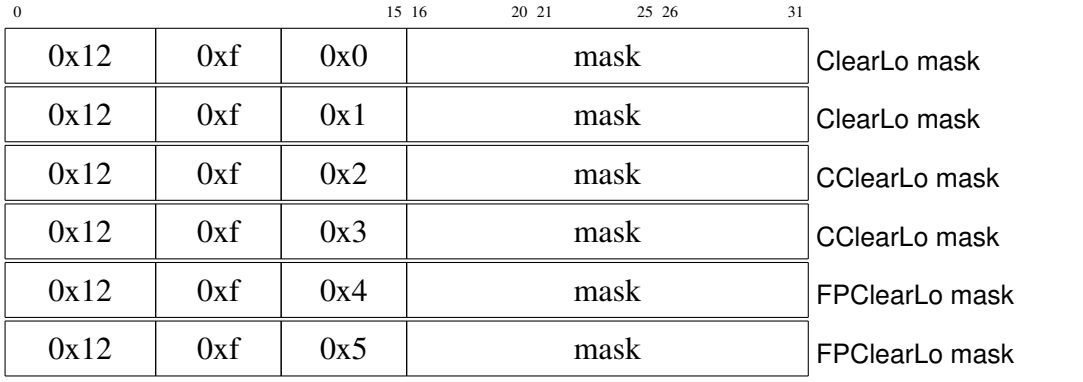
0 5	6 10	11 15	16 20	21 25	26 31	
0x12	0x08		cb			CJR cb



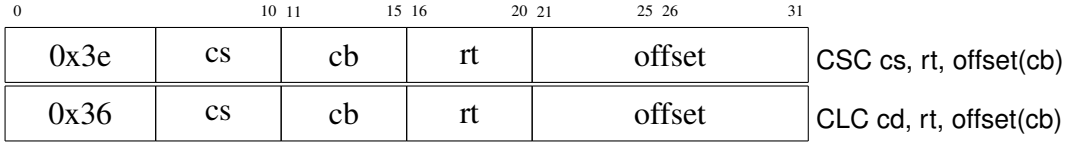
B.1.7 Assertion instructions Instructions

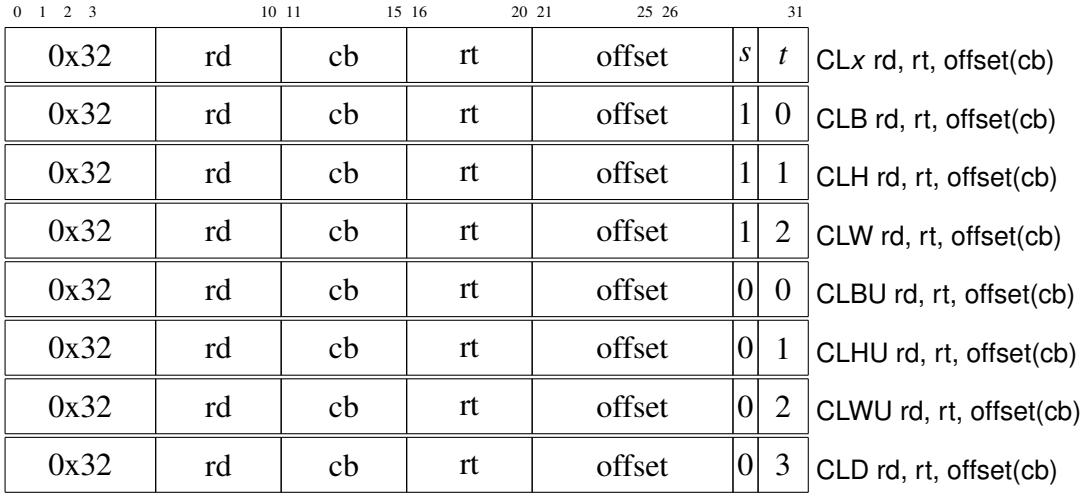


B.1.8 Fast Register Clearing Instructions



B.1.9 Memory Access Instructions





0x3a	rs	cb	rt	offset	0	*t*	CS*x* rs, rt, offset(cb)
0x3a	rs	cb	rt	offset	0	0	CSB rs, rt, offset(cb)
0x3a	rs	cb	rt	offset	0	1	CSH rs, rt, offset(cb)
0x3a	rs	cb	rt	offset	0	2	CSW rs, rt, offset(cb)
0x3a	rs	cb	rt	offset	0	3	CSD rs, rt, offset(cb)

B.1.10 Atomic Memory Access Instructions

0 1 2 3	6 10	11 15	16 20	21 25	26 31	
0x12	0x10	cd	cb		0xf	CLLC cd, cb
0x12	0x10	cs	cb	rd	0x7	CSCC rd, cs, cb

0x12	0x10	rd	cb		1	*s*	*t*	CLL*x* rd, cb
0x12	0x10	rd	cb		1	1	0	CLLB rd, cb
0x12	0x10	rd	cb		1	1	1	CLLH rd, cb
0x12	0x10	rd	cb		1	1	2	CLLW rd, cb
0x12	0x10	rd	cb		1	0	0	CLLBU rd, cb
0x12	0x10	rd	cb		1	0	1	CLLHU rd, cb
0x12	0x10	rd	cb		1	0	2	CLLWU rd, cb
0x12	0x10	rd	cb		1	0	3	CLLD rd, cb
0x12	0x10	rs	cb	rd	0		*t*	CSC*x* rd, cb
0x12	0x10	rs	cb	rd	0		0	CSCB rd, cb
0x12	0x10	rs	cb	rd	0		1	CSCH rd, cb
0x12	0x10	rs	cb	rd	0		2	CSCW rd, cb
0x12	0x10	rs	cb	rd	0		3	CSCD rd, cb

B.1.11 Deprecated and Removed instructions Instructions

0 5	6 10	11 15	16 20	21 25	26 31	
0x12	0x04	cd	cb	rt	0x3	CSetLen cd, cb, rt
0x12	0x04	cd	cb	rt	0x2	CIncBase cd, cb, rt

0 15	16 20	21 25	26 31		
0x32	rd	cb	rt	offset	1	3	CLLD rd, rt, offset(cb)
0x3a	rs	cb	rt	offset	1	3	CSCD rs, rt, offset(cb)

B.2 Proposed New Encodings

The encodings described in this section are part of an ongoing project to rationalize the use of opcode space by the CHERI prototype. These encodings are subject to change, but may be supported in future implementations. Some of the instructions listed in this section are not yet fully specified and their behavior is also subject to change.

B.2.1 Capability Inspection Instructions

0 5	6 10	11 15	16 20	21 25	26 31	
0x12	0x0	rd	cb	0x0	0x3f	CGetPerm rd, cb
0x12	0x0	rd	cb	0x1	0x3f	CGetType rd, cb
0x12	0x0	rd	cb	0x2	0x3f	CGetBase rd, cb
0x12	0x0	rd	cb	0x3	0x3f	CGetLen rd, cb
0x12	0x0	rd	cb	0x4	0x3f	CGetTag rd, cb
0x12	0x0	rd	cb	0x5	0x3f	CGetSealed rd, cb
0x12	0x0	rd	cb	0x6	0x3f	CGetOffset rd, cb
0x12	0x0	cd	0x0	0x1f	0x3f	CGetPCC cd
0x12	0x0	cd	rs	0x7	0x3f	CGetPCCSetOffset cd, rs

B.2.2 Capability Modification Instructions

0 5	6 10	11 15	16 20	21 25	26 31	
0x12	0x0	cd	cs	ct	0xb	CSeal cd, cs, ct
0x12	0x0	cd	cb	ct	0xc	CUnseal cd, cs, ct
0x12	0x0	cd	cs	rt	0xd	CAndPerm cd, cs, rt
0x12	0x0	cd	cs	rt	0xf	CSetOffset cd, cs, rt
0x12	0x0	cd	cs	rt	0x10	CSetBounds cd, cs, rt
0x12	0x0	cd	cs	rt	0x9	CSetBoundsExact cd, cs, rt
0x12	0x0	cd	cb	0xb	0x3f	CClearTag cd, cb
0x12	0x0	cd	cb	rt	0x11	CIncOffset cd, cb, rt

B.2.3 Pointer Arithmetic Instructions

0 5	6 10	11 15	16 20	21 25	26 31	
0x12	0x0	rd	cb	cs	0x12	CToPtr rd, cb, cs
0x12	0x0	cd	cb	rs	0x13	CFromPtr cd, cb, rs

0x12	0x0	rt	cb	cs	0xa	CSub rt, cb, cs
0x12	0x0	cd	cs	0xa	0x3f	CMove cd, cs
0x12	0x0	cd	cs	rs	0x1b	CCMovZ cd, cs, rs
0x12	0x0	cd	cs	rs	0x1c	CCMovN cd, cs, rs

B.2.4 Pointer Comparison Instructions

0x12	0x0	rd	cb	cs	0x14	CEQ rd, cb, cs
0x12	0x0	rd	cb	cs	0x15	CNE rd, cb, cs
0x12	0x0	rd	cb	cs	0x16	CLT rd, cb, cs
0x12	0x0	rd	cb	cs	0x17	CLE rd, cb, cs
0x12	0x0	rd	cb	cs	0x18	CLTU rd, cb, cs
0x12	0x0	rd	cb	cs	0x19	CLEU rd, cb, cs
0x12	0x0	rd	cb	cs	0x1a	CEXEQ rd, cb, cs

B.2.5 Exception Handling Instructions

0 5	6 10	11 15	16 20	21 25	26 31	
0x12	0x0	rd	0x1	0x1f	0x3f	CGetCause rd
0x12	0x0	rs	0x2	0x1f	0x3f	CSetCause rs

B.2.6 Control Flow Instructions

0 15	16 20	21 25	26 31		
0x12	0x9	cd	offset		CBTU cd, offset
0x12	0xa	cd	offset		CBTS cd, offset
0x12	0x11	cd	offset		CBEZ cd, offset
0x12	0x12	cd	offset		CBNZ cd, offset

0 5	6 10	11 15	16 20	21 25	26 31	
0x12	0x0	cb	0x3	0x1f	0x3f	CJR cb
0x12	0x0	cd	cb	0xc	0x3f	CJALR cd, cb

0 10	11 15	16 20	21 25	26 31	
0x12	0x05	cs	cb	selector	CCall cs, cb[, selector]
0x12	0x05	0x0	0x0	0x1	CReturn ; pseudo

B.2.7 Assertion instructions Instructions

0x12	0x0	cs	rt	0x8	0x3f	CCheckPerm cs, rt
0x12	0x0	cs	rt	0x9	0x3f	CCheckType cs, cb

B.2.8 Fast Register Clearing Instructions

0			15 16	20 21	25 26	31	
0x12	0xf	0x0	mask				ClearLo mask
0x12	0xf	0x1	mask				ClearLo mask
0x12	0xf	0x2	mask				CClearLo mask
0x12	0xf	0x3	mask				CClearLo mask
0x12	0xf	0x4	mask				FPClearLo mask
0x12	0xf	0x5	mask				FPClearLo mask

B.2.9 Encoding Summary

All three-register-operand CHERI instructions use the following encoding:

0 5	6 10	11 15	16 20	21 25	26 31
0x12	0x0	r1	r2	r3	func

	000	001	010	011	100	101	110	111
000	CGetPerm*	CGetType*	CGetBase*	CGetLen*	CGetCause*	CGetTag*	CGetSealed*	CGetPCC*
001	CSetBounds	CSetBoundsExact	CSub	CSeal	CUnseal	CAndPerm	UNUSED	CSetOffset
010	UNUSED	CIncOffset	CToPtr	CFromPtr	CEQ	CNE	CLT	CLE
011	CLTU	CLEU	CEXEQ	CCMovN	CCMovZ	UNUSED	UNUSED	UNUSED
100	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
101	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
110	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
011	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	Two Op†

* Deprecated encoding for instruction

† This value is used for two-operand instructions.

This frees several minor opcodes free and allows us to allocate 35 more three-operand instructions immediately, and eight more once the deprecated encodings are removed, without having to allocate a new minor opcode.
All two-operand instructions are of the following form:

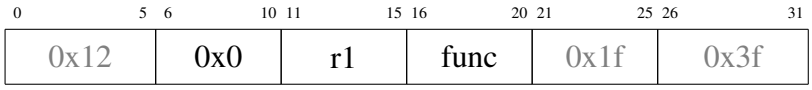
0 5	6 10	11 15	16 20	21 25	26 31
0x12	0x0	r1	r2	func	0x3f

	000	001	010	011	100	101	110	111
00	CGetPerm	CGetType	CGetBase	CGetLen	CGetTag	CGetSealed	CGetOffset	CGetPCCSetOffset
01	CCheckPerm	CCheckType	CMove	CClearTag	CJALR	UNUSED	UNUSED	UNUSED
10	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
11	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	One Op†

† This value is used for two-operand instructions.

This allows us to allocate 21 new two-operand instructions without consuming a minor opcode.

All one-operand instructions are of the following form:



	000	001	010	011	100	101	110	111
00	CGetPCC	CGetCause	CSetCause	CJR	UNUSED	UNUSED	UNUSED	UNUSED
01	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
10	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED
11	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED	UNUSED

Glossary

capability A capability contains a virtual address, capability bounds describing a range of bytes within which the virtual address 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 sealed bit indicating whether it is a sealed capability or unsealed capability. If the capability is sealed, then it also contains a capability object type.

In CHERI, capabilities are used to provide pointers with additional protections in aid of fine-grained memory protection, control-flow integrity (CFI), 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 virtual of its hybrid capability model.

Architecturally, a capability can be viewed as a virtual address, calculated as the sum of the capability base and capability offset. Dereferencing a capability is done relative to that virtual address. The implementation may choose to store the pre-computed virtual address combining the base and offset, to avoid an implied addition on each memory access, and to similarly store the base and length as pre-computed virtual addresses. This also facilitates implementing a compressed capability mechanism such as the CHERI-128 model.

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, and instruction fetch via the program-counter capability (PCC). A capability is either sealed or unsealed, controlling whether it has software-defined or instruction-set-defined behavior, and whether or not its fields are immutable.

Capabilities may be held in a capability register in the capability register file, or stored in suitably aligned tagged memory.

capability base The lower of the two capability bounds, from which the virtual 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 virtual addresses 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 virtual address. The virtual 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. Monotonicity is preserved by capability-based instructions subject to their appropriate use (e.g., not requesting an increase in the length of a capability).

capability object type In addition to fat-pointer metadata such as capability bounds and capability permissions, a sealed capability also contains an integer object type. This type can be set only subject to suitable permissions, and allows multiple capabilities to be marked as linked to one another – for example, to ensure that in implementing a software object model, a code capability for a class is used only with a suitable data capability for an object of the same class, and not data for an object of another class.

capability offset The distance between capability base and the virtual address described by a capability when 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, instrution fetch, load capability, and store capability. Permissions may be manipulated subject to capability monotonicity using capability-based instructions.

capability provenance The property that, following manipulation, a capability remains valid for use only if it is derived from another valid capability using a valid capability operation. Provenance is implemented using a capability tag combined with capability monotonicity, and will be preserved whether a capability is held in a capability register or memory, subject to suitable use of capability-based instructions.

capability register A capability register is an ISA register able to hold a capability including its capability tag, virtual address, and other fat-pointer metadata such as its capability bounds and capability permissions. Capability registers are stored in the capability register file. It might be a special-purpose register intended primarily for capability-related operations (e.g., the capability registers described in the CHERI extensions to the MIPS ISA), or a general-purpose 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 register file The capability registers, including general-purpose capability registers and those that have specific interpretations in the instruction set. The latter include the program-counter capability (PCC), the default data capability, the exception program-counter capability (EPCC), the kernel code capability (KCC), the kernel data capability

(KDC), and the kernel reserved capabilities. Some general-purpose capability registers have well-known conventions for their use in software, including the return capability and the stack capability.

capability tag A capability tag is a 1-bit integrity tag associated with each capability register, and also each capability-sized, capability-aligned location in memory. If the tag is present, 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 subject to operations conforming to capability provenance and capability monotonicity rules – for example, that an attempted modification of capability bounds leads to non-increasing writes, or that in-memory capabilities are written only via capability stores, and not data stores. Subject to these constraints, tags will be preserved by capability-based instructions.

capability validity A capability is valid if its capability tag is set, which permits use of the capability subject to its capability bounds, capability permissions, and so on. Attempts to dereference a capability without a tag set will lead to a hardware exception.

capability-based instructions These instructions accept capabilities as operands, allowing capabilities to be loaded from and stored memory, manipulated subject to capability provenance and capability monotonicity rules, and used for a variety of operations such as loading and storing data and capabilities, as branch targets, and to retrieve and manipulate capability fields – subject to capability permissions.

CCall A trapping instruction, similar to a system call, intended to support invoking objects expressed as a pair of sealed capabilities, representing a code capability and a data capability. The exception code generated depends on whether or not the two operand capabilities have valid capability tags, suitable capability permissions, are both sealed, have matching capability object types, and other requirements associated with joint invocation. The software exception handler is expected to implement software-defined aspects of the object model, including any necessary unsealing of the operand capabilities, storing of any return information (e.g., via a trusted stack), and handle any exceptions reporting failures of ISA-implemented checks. To facilitate optimized software implementations, a separate CCall/CReturn exception vector is used.

CHERI-128 CHERI-128 is a specific compressed capability format that represents a 64-bit virtual address with full precision, and capability bounds relative to that address with reduced precision. This compression model places stronger alignment requirements on the bounds, as well as introducing the idea of representable capabilities (whose capability offset falls within, or close to within, the bounds) and unrepresentable capabilities, whose offset falls too far outside of the bounds to represent. Stronger alignment constraints are placed on sealed capabilities in order to recover further bits for the capability object type field, which is not required for unsealed capabilities. The practical impact of this model is to half the size of a 256-bit capability, at modest cost in memory fragmentation.

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 integrity (CFI) by constraining the available branch and jump targets.

compressed capability A capability whose capability bounds are compressed with respect to its virtual address, allowing its in-memory footprint to be reduced – e.g., to 128 bits, rather than the architectural 256 bits visible to the instruction set when a capability is loaded into a register file. Certain architecturally valid out-of-bounds virtual 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 is a specific compressed capability model that selects a particular point in the tradeoff space around in-memory capability size, bounds alignment requirements, and representability.

control-flow integrity (CFI) 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 last privilege.

CReturn A trapping instruction, similar to a system call, intended to support returning from an object invoked via the CCall instruction. Unlike CCall, in-ISA checks are not performed, leaving any required functionality to software – for example, popping an entry off of a trusted stack. To facilitate optimized software implementations, a separate CCall/CReturn exception vector is used.

data capability A capability whose capability permissions have been configured to permit data load and store, but not instruction fetch (i.e., execute) rights; in contrast to a code capability.

default data capability 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 virtual 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, but also subject to appropriate 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.

exception program-counter capability (EPCC) A reserved capability register into which the running program-counter capability (PCC) will be moved into on an exception, and whose value will be moved back into the program-counter capability on exception return.

fat pointer A pointer (virtual 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 integrity (CFI), 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 last 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, and strong compatibility with the C-language pointer model, all constitute hybrid aspects of its design, in comparison to a more pure capability-system model that might elide those behaviors at a cost to compatibility and adoptability.

invoked data capability (IDC) A capability register reserved by convention to hold the unsealed data capability on the callee side of CCall, and to be saved from the caller context on CCall, to be restored by CReturn. Typically, for the caller side, this will point at a frame on the caller stack sufficient to safely restore any caller state. On the callee side, the invoked data capability will be a data capability describing the objects internal state.

kernel code capability (KCC) A capability register reserved to hold a privileged code capability for use by the kernel during exception handling. This value will be installed in the program-counter capability (PCC) on exception entry, with the previous value of the program-counter capability stored in the exception program-counter capability (EPCC).

kernel data capability (KDC) A capability register reserved to hold a privileged data capability for use by the kernel during exception handling. Typically, this will refer either to the data segment for a microkernel intended to field exceptions, or for the full kernel. Kernels compiled to primarily use legacy instructions might install this in the default data capability for the duration of kernel execution. Use of this register is controlled by capability permissions on the currently executing program-counter capability (PCC).

kernel reserved capabilities These capabilitys, modeled on the MIPS kernel reserved registers, are set aside for use by the operating-system kernel in exception handling – in particular, in allowing userspace registers to be saved so that the kernel context can be installed. As with the MIPS registers, the userspace ABI is not able to use capability

registers set aside for kernel use; unlike the MIPS registers, the kernel reserved capabilities are available for use in the ISA only with a suitably authorized program-counter capability (PCC) installed.

legacy instructions Legacy instructions are those that accept virtual addresses, rather than capabilities, as their operands, requiring use of the default data capability for loads and stores, or that explicitly set the program counter to a virtual 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 integrity (CFI), or more efficient software compartmentalization. While still constrained, these instructions do not conform to the principle of intentional use.

out of bounds When a capability's capability offset falls outside of its capability bounds, it is out of bounds, and cannot be dereferenced. Even if a capability's offset is in bounds, the width of a data access may cause a load, store, or instruction fetch to fall out of bounds, or the further offset introduced via a register index or immediate operand to an instruction. With 256-bit capabilities, all out-of-bounds pointers are representable capabilities. With compressed capabilities, if an instruction shifts the offset too far out of bounds, this may result in an unrepresentable capability, leading to the capability tag being cleared, or an exception being thrown.

pointer A pointer is a language-level reference to a memory object. In conventional ISAs, a pointer is typically represented as a virtual address. In CHERI, pointers can be represented either as a virtual address indirected via the default data capability 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 integrity (CFI). 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 last 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) An extension of the existing program counter to include capability metadata such as a capability tag, capability bounds, and capability permissions. The program-counter capability ensures that instruction fetch occurs only subject

to capability protections. When an exception fires, the value of the program-counter capability will be moved to the exception program-counter capability (EPCC), and the value of the kernel data capability (KDC) moved into the program-counter capability. On exception return, the value of the exception program-counter capability will be moved into the program-counter capability.

representable capability A compressed capability whose capability bounds and capability offset can be represented with full precision; this does not imply that the offset is "within bounds", but does require that it be within some broader window around the bounds.

return capability A capability designated as the destination for the return address when using a capability jump-and-link instruction. A degree of control-flow integrity (CFI) 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 bit A bit in the capability format that indicates whether the capability is a sealed capability or an unsealed capability.

sealed capability A sealed capability is one whose sealed bit is set. A sealed capability's virtual address, capability bounds, capability permissions, and other fields are immutable – i.e., cannot be modified using capability-based instructions. Sealed capabilities also have a capability object type. CHERI's sealing feature allows capabilities to be used to describe software-defined objects, permitting implementation of encapsulation. A sealed capability cannot be directly dereferenced using the instruction set. Unsealing will often be performed in a CCall exception as part of object invocation in the software model – but can be used more flexibly for non-object-oriented models, and also in callees to unseal a sealed argument capability other than the invoked object.

software compartmentalization The configuration of code capabilities and data capabilities available via the capability register file and memory such that software components can be isolated from one another, enabling vulnerability mitigation via the application of the principle of last privilege at the application layer. One approach to implementing software compartmentalization on CHERI is to use sealed capabilities to represent security domains, which can be safely invoked using a suitably crafted CCall exception handler, providing mutual distrust.

stack capability A capability referring to the current stack, whose capability bounds are suitably configured to allow access only to the remaining stack available to allocate at a given point in execution.

tagged memory Tagged memory associates a 1-bit capability tag with each capability-aligned, capability-sized word in memory. Capability-based instructions that load and store capabilities maintain the tag as the capability transits between memory and the capability register file, tracking capability provenance. When data stores (i.e., stores of non-capabilities), the tag on the memory location will be atomically cleared, ensuring the integrity of in-memory capabilities.

Trusted Computing Base (TCB) The subset of hardware and software that is critical to the security of a system; in secure system designs, there is often a goal to minimize the size of the TCB in order to minimize the opportunity for exploitable software vulnerabilities.

trusted stack Some software-defined object-capability models offer strong call-return semantics – i.e., that if a return is issued by an invoked object, or an uncaught exception is generated, then the appropriate caller will be returned to – exactly once. This can be implemented via a trusted stack, maintained by the Trusted Computing Base (TCB) via CCall and CReturn exception handlers. A trusted stack for an object-oriented model will likely maintain at least the caller's program-counter capability (PCC) and invoked data capability (IDC) to be restored on return.

unrepresentable capability A compressed capability whose capability offset is sufficiently outside of its capability bounds that the combined pointer value and bounds cannot be represented with precision; 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 sealed bit is unset. 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 translated by the Memory Management Unit (MMU) into a physical address for the purposes of load, store, and instruction fetch. Capabilitys embed a virtual address, represented in the instruction set as the sum of the capability base and capability offset, as well as capability bounds relative to the address. The integer addresses passed to legacy load and store instructions that would previously have been interpreted as virtual addresses are, with CHERI, transformed (and checked) using the default data capability. Similarly, the integer addresses passed to legacy branch and jump instructions are transformed (and checked) using the program-counter capability (PCC). This in effect introduces a further relocation of legacy addresses prior to virtual address translation.

vulnerability mitigation A set of techniques limiting the effectiveness of the attacker to exploit a software vulnerability, typically achieved through use of the principle of last privilege to constrain injection of arbitrary code, control of the program-counter capability (PCC) via control-flow integrity (CFI) using code capabilities, minimization of data rights granted via available data capabilities, and higher-level software compartmentalization.

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[2] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. Technical report, Computer Science Department, Carnegie Mellon University, August 1986.

[3] W. B. Ackerman and W. W. Plummer. An implementation of a multiprocessing computer system. In *SOSP '67: Proceedings of the First ACM Symposium on Operating System Principles*, pages 5.1–5.10, New York, NY, USA, 1967. ACM.

[4] J. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, October 1972. (Two volumes).

[5] G. R. Andrews. Partitions and principles for secure operating systems. Technical report, Cornell University, Ithaca, NY, USA, 1975.

[6] Apple Inc. Mac OS X Snow Leopard. `http://www.apple.com/macosx/`, 2010.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–283, New York, NY, USA, 1995. ACM.

[8] R. Bisbey II and D. Hollingworth. Protection Analysis: Project final report. Technical report, USC Information Sciences Institute (ISI), Marina Del Rey, California, 1978.

[9] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 309–322. USENIX Association, 2008.

[10] Bluespec, Inc., Waltham, MA. *Bluespec SystemVerilog Version 3.8 Reference Guide*, November 2004.

[11] M. Branstad and J. Landauer. Assurance for the Trusted Mach operating system. In *Proceedings of the Fourth Annual Conference on Computer Assurance COMPASS '89*, pages 9–13. IEEE, June 1989.

[12] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *ATEC '04: Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2004. USENIX Association.

[13] D. Chisnall, C. Rothwell, B. Davis, R. Watson, J. Woodruff, S. Moore, P. G. Neumann, and M. Roe. Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XX, New York, NY, USA, 2014. ACM.

[14] E. Cohen and D. Jefferson. Protection of the Hydra operating system. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, pages 141–160, 1975.

[15] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley. An experimental time-sharing system. In *AIEE-IRE '62 (Spring): Proceedings of the May 1–3, 1962, Spring Joint Computer Conference*, pages 335–344, New York, NY, USA, 1962. ACM.

[16] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the Multics system. In *AFIPS '65 (Fall, part I): Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, part I*, pages 185–196, New York, NY, USA, 1965. ACM.

[17] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. In *AFIPS Conference Proceedings, Fall Joint Computer Conference*, pages 213–229. Spartan Books, November 1965.

[18] P. J. Denning. Fault tolerant operating systems. *ACM Computing Surveys*, 8(4):359–389, 1976.

[19] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[20] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. *SIGPLAN Not.*, 43(3):103–114, Mar. 2008.

[21] U. Dhawan, N. Vasilakis, R. Rubin, S. Chiricescu, J. M. Smith, T. F. Knight Jr, B. C. Pierce, and A. DeHon. Pump: a programmable unit for metadata processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, page 8. ACM, 2014.

[22] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. *SIGOPS Oper. Syst. Rev.*, 39:17–30, October 2005.

[23] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2015.

[24] R. S. Fabry. The case for capability based computers (extended abstract). In *SOSP '73: Proceedings of the Fourth ACM Symposium on Operating System Principles*, page 120, New York, NY, USA, 1973. ACM.

[25] R. J. Feiertag and P. G. Neumann. The foundations of a Provably Secure Operating System (PSOS). In *Proceedings of the National Computer Conference*, pages 329–334. AFIPS Press, 1979. `http://www.csl.sri.com/neumann/psos.pdf`.

[26] A. Fox. *Improved Tool Support for Machine-Code Decompilation in HOL4*, pages 187–202. Springer International Publishing, Cham, 2015.

[27] P. G.Neumann, R. Boyer, R. Feiertag, K. Levitt, and L. Robinson. A Provably Secure Operating System: The system, its applications, and proofs. Technical Report CSL-116, Second edition, Computer Science Laboratory, SRI International, Menlo Park, California, May 1980.

[28] L. Gong. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, Reading, Massachusetts, 1999.

[29] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[30] J. A. González. *Taxi: Defeating Code Reuse Attacks with Tagged Memory*. PhD thesis, Massachusetts Institute of Technology, 2015.

[31] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[32] S. Govindavajhala and A. Appel. Using memory errors to attack a virtual machine. In *Proceedings of the 2003 Symposium on Security and Privacy*, pages 154–165, Oakland, California, May 2003. IEEE Computer Society.

[33] R. Graham. Protection in an information processing utility. *Communications of the ACM*, 11(5), May 1968.

[34] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, S. J. Murdoch, P. G. Neumann, and A. Richardson. Clean application compartmentalization with SOAAP (extended version). Technical Report UCAM-CL-TR-873, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, Dec. 2015.

[35] K. Gudka, R. N. M. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015)*, October 2015.

[36] N. Hardy. KeyKOS architecture. *SIGOPS Operating Systems Review*, 19(4):8–25, 1985.

[37] N. Hardy. The confused deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4), October 1988.

[38] International Standards Organization. *Common Criteria for Information Technology Security Evaluation – Part 3: Security assurance components*, September 2012. Version 3.1, revision 4.

[39] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *ATEC '02: Proceedings of the USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[40] A. Jones and W. Wulf. Towards the design of secure systems. In *Protection in Operating Systems, Proceedings of the International Workshop on Protection in Operating Systems*, pages 121–135, Rocquencourt, Le Chesnay, France, 13–14 August 1974. Institut de Recherche d'Informatique.

[41] P. Karger. Limiting the damage potential of discretionary Trojan horses. In *Proceedings of the 1987 Symposium on Security and Privacy*, pages 32–37, Oakland, California, April 1987. IEEE Computer Society.

[42] P. Karger and R. Schell. Multics security evaluation: Vulnerability analysis. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC), Classic Papers section*, Las Vegas, Nevada, December 2002. Originally available as U.S. Air Force report ESD-TR-74-193, Vol. II, Hanscomb Air Force Base, Massachusetts.

[43] P. A. Karger. Using registers to optimize cross-domain call performance. *SIGARCH Computer Architecture News*, 17(2):194–204, 1989.

[44] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the Tenth ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[45] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an operating-system kernel. *Communications of the ACM*, 53:107–115, June 2009.

[46] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[47] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *20th ACM Conference on Computer and Communications Security*, November 2013.

[48] B. Lampson. Redundancy and robustness in memory protection. In *Information Processing 74 (Proceedings of the IFIP Congress 1974)*, volume Hardware II, pages 128–132. North-Holland, Amsterdam, 1974.

[49] B. W. Lampson. Dynamic protection structures. In *AFIPS '69 (Fall): Proceedings of the November 18-20, 1969, Fall Joint Computer Conference*, pages 27–38, New York, NY, USA, 1969. ACM.

[50] B. W. Lampson. Protection. *SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[51] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the international symposium on code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[52] N. G. Leveson and W. Young. An integrated approach to safety and security based on system theory. *Communications of the ACM*, 57(2):31–35, February 2014. url-http://www.csl.sri.com/neumann/insiderisks.html.

[53] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[54] J. Liedtke. On microkernel construction. In *SOSP'95: Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, Dec. 1995.

[55] S. B. Lipner, W. A. Wulf, R. R. Schell, G. J. Popek, P. G. Neumann, C. Weissman, and T. A. Linden. Security kernels. In *AFIPS '74: Proceedings of the May 6-10, 1974, National Computer Conference and Exposition*, pages 973–980, New York, NY, USA, 1974. ACM.

[56] A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh. Cryptographically enforced control flow integrity. *arXiv preprint arXiv:1408.1451*, 2014.

[57] S. McCanne and V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *USENIX'93: Proceedings of the USENIX Winter 1993 Conference*, Berkeley, CA, USA, 1993. USENIX Association.

[58] E. McCauley and P. Drongowski. KSOS: The design of a secure operating system. In *National Computer Conference*, pages 345–353. AFIPS Conference Proceedings, 1979. Vol. 48.

[59] 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.

[60] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. Watson, and P. Sewell. Into the depths of C: elaborating the de facto standards. In *Proceedings of PLDI 2016*, June 2016.

[61] A. Mettler and D. Wagner. Class properties for security review in an object-capability subset of Java. In *PLAS '10: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 1–7, New York, NY, USA, 2010. ACM.

[62] M. S. Miller. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006.

[63] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript, May 2008. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`.

[64] J. H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.

[65] A. C. Myers and B. Liskov. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31:129–142, October 1997.

[66] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[67] NCSC. *Department of Defense Trusted Computer System Evaluation Criteria (TCSEC)*. National Computer Security Center, December 1985. DOD-5200.28-STD, Orange Book.

[68] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI '96: Proceedings of the Second USENIX symposium on Operating Systems Design and Implementation*, pages 229–243, New York, NY, USA, 1996. ACM.

[69] G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 128–139, New York, NY, USA, 2002. ACM.

[70] P. G. Neumann. Holistic systems. *ACM Software Engineering Notes*, 31(6):4–5, November 2006.

[71] P. G. Neumann, R. Boyer, R. Feiertag, K. Levitt, and L. Robinson. A Provably Secure Operating System: The system, its applications, and proofs. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, May 1980. 2nd edition, Report CSL-116.

[72] P. G. Neumann and R. J. Feiertag. PSOS revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), Classic Papers section*, pages 208–216, Las Vegas, Nevada, December 2003. IEEE Computer Society. http://www.acsac.org/ and http://www.csl.sri.com/neumann/psos03.pdf.

[73] P. G. Neumann and R. N. M. Watson. Capabilities revisited: A holistic approach to bottom-to-top assurance of trustworthy systems. In *Fourth Layered Assurance Workshop*, Austin, Texas, December 2010. U.S. Air Force Cryptographic Modernization Office and AFRL. http://www.csl.sri.com/neumann/law10.pdf.

[74] P. G. Neumann, R. N. M. Watson, N. Dave, A. Joannou, M. Naylor, M. Roe, A. Fox, and J. Woodruff. CHERI Formal Methods, interim version 3.0. Technical report, SRI International and the University of Cambridge, 2015.

[75] R. M. Norton. Hardware support for compartmentalisation. Technical Report UCAM-CL-TR-887, University of Cambridge, Computer Laboratory, May 2016.

[76] E. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, Massachusetts, 1972.

[77] D. A. Patterson and C. H. Sequin. RISC I: A Reduced Instruction Set VLSI Computer. In *ISCA '81: Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 443–457, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[78] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, 2003.

[79] R. Rashid and G. Robertson. Accent: A communications oriented network operating system kernel. In *Proceedings of the Eighth ACM Symposium on Operating System Principles*, pages 64–75, Asilomar, California, December 1981. (ACM Operating Systems Review, Vol. 15, No. 5).

[80] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems*, pages 219–232, New York, NY, USA, 2009. ACM.

[81] D. Rémy and J. Vouillon. Objective ML: a simple object-oriented extension of ML. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 40–53, New York, NY, USA, 1997. ACM.

[82] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974.

[83] Ruby Users Group. Ruby Programming Language. `http://www.ruby-lang.org/`, October 2010.

[84] J. Rushby. The design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating System Principles*, pages 12–21, Asilomar, California, December 1981. (ACM Operating Systems Review, 15(5)).

[85] J. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[86] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[87] W. Schiller. The design and specification of a security kernel for the PDP-11/45. Technical Report MTR-2934, Mitre Corporation, Bedford, Massachusetts, March 1975.

[88] M. D. Schroeder. Engineering a security kernel for Multics. In *SOSP '75: Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, pages 25–32, New York, NY, USA, 1975. ACM.

[89] E. J. Sebes. Overview of the architecture of Distributed Trusted Mach. In *Proceedings of the USENIX Mach Symposium*, pages 20–22. USENIX Association, November 1991.

[90] 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.

[91] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proceedings of the 8th USENIX Security Symposium*, pages 123–139, Washington, D.C., USA, Aug. 1999. USENIX Association.

[92] R. Wahbe, S. Lucco, T. E. Anderson, and S. u. L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, New York, NY, USA, 1993. ACM.

[93] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.

[94] S. T. Walker. The advent of trusted computer operating systems. In *AFIPS '80: Proceedings of the May 19-22, 1980, national computer conference*, pages 655–665, New York, NY, USA, 1980. ACM.

[95] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the 18th USENIX Security Symposium*, pages 417–432, Berkeley, CA, USA, 2009. USENIX Association.

[96] R. N. M. Watson. New Approaches to Operating System Security Extensibility. Technical report, Ph.D. Thesis, University of Cambridge, Cambridge, UK, October 2010.

[97] R. N. M. Watson. New approaches to operating system security extensibility. Technical Report UCAM-CL-TR-818, University of Cambridge, Computer Laboratory, Apr. 2012.

[98] R. N. M. Watson. A decade of OS access-control extensibility. *Commun. ACM*, 56(2), Feb. 2013.

[99] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for Unix. In *Proceedings of the 19th USENIX Security Symposium*. USENIX, August 2010.

[100] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, P. G. Neumann, and J. Woodruff. Bluespec Extensible RISC Implementation (BERI): Software Reference. Technical Report UCAM-CL-TR-853, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, June 2014.

[101] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, P. G. Neumann, and J. Woodruff. Capability Hardware Enhanced RISC Instructions (CHERI): User's guide. Technical Report UCAM-CL-TR-851, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, June 2014.

[102] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, P. G. Neumann, and J. Woodruff. Bluespec Extensible RISC Implementation: BERI Software reference. Technical Report UCAM-CL-TR-869, University of Cambridge, Computer Laboratory, Apr. 2015.

[103] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, P. G. Neumann, and J. Woodruff. Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide. Technical Report UCAM-CL-TR-877, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, Nov. 2015.

[104] R. N. M. Watson, P. G. Neumann, J. Woodruff, J. Anderson, D. Chisnall, B. Davis, B. Laurie, S. W. Moore, S. J. Murdoch, and M. Roe. Capability Hardware Enhanced RISC Instructions (CHERI): Instruction-Set Architecture. Technical Report UCAM-CL-TR-850, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, June 2014.

[105] R. N. M. Watson, P. G. Neumann, J. Woodruff, J. Anderson, D. Chisnall, B. Davis, B. Laurie, S. W. Moore, S. J. Murdoch, and M. Roe. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture. Technical Report UCAM-CL-TR-864, University of Cambridge, Computer Laboratory, Dec. 2014.

[106] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S. W. Moore, S. J. Murdoch, R. Norton, and S. Son. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture. Technical Report UCAM-CL-TR-876, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, Nov. 2015.

[107] 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. Technical Report UCAM-CL-TR-891, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, June 2016.

[108] R. N. M. Watson, J. Woodruff, D. Chisnall, B. Davis, W. Koszek, A. T. Markettos, S. W. Moore, S. J. Murdoch, P. G. Neumann, R. Norton, and M. Roe. Bluespec Extensible RISC Implementation (BERI): Hardware Reference. Technical Report UCAM-CL-TR-852, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, June 2014.

[109] R. N. M. Watson, J. Woodruff, D. Chisnall, B. Davis, W. Koszek, A. T. Markettos, S. W. Moore, S. J. Murdoch, P. G. Neumann, R. Norton, and M. Roe. Bluespec Extensible RISC Implementation: BERI Hardware reference. Technical Report UCAM-CL-TR-868, University of Cambridge, Computer Laboratory, Apr. 2015.

[110] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. s Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, May 2015.

[111] R. N. M. Watson, P. G. N. J. Woodruff, J. Anderson, R. Anderson, N. Dave, B. Laurie, S. W. Moore, S. J. Murdoch, P. Paeps, M. Roe, and H. Saidi. CHERI: a research platform

deconflating hardware virtualization and protection. In *Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012)*, March 2012.

[112] M. Wilkes and R. Needham. *The Cambridge CAP Computer and Its Operating System*. Elsevier North Holland, New York, 1979.

[113] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA 2014)*, June 2014.

[114] J. D. Woodruff. CHERI: A RISC capability machine for practical memory safety. Technical Report UCAM-CL-TR-858, University of Cambridge, Computer Laboratory, July 2014.

[115] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, 1974.

[116] W. Wulf, R. Levin, and S. Harbison. *Hydra/C.mmp: An Experimental Computer System*. McGraw-Hill, New York, 1981.

[117] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society.

[118] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society.

[119] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.