Technical Report

UCAM-CL-TR-850
ISSN 1476-2986

Number 850

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Capability Hardware Enhanced RISC Instructions: CHERI Instruction-set architecture

Robert N.M. Watson, Peter G. Neumann, Jonathan Woodruff, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Michael Roe

April 2014

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

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

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 document describes the rapidly maturing design for the Capability Hardware Enhanced RISC Instructions (CHERI) Instruction-Set Architecture (ISA), which is being developed by SRI International and the University of Cambridge. The document is intended to capture our evolving architecture, as it is being refined, tested, and formally analyzed. We have now reached 70% of the time for our research and development cycle.

CHERI is a *hybrid capability-system architecture* that combines new processor primitives with the 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. 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 via a new *capability coprocessor*.

CHERI's hybrid 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, 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, 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.

This report describes the CHERI architecture and design, and provides reference documentation for the CHERI instruction-set architecture (ISA) and potential memory models, along with their requirements. It also documents our current thinking on integration of programming languages and operating systems. Our ongoing research includes two prototype processors employing the CHERI ISA, each implemented as an FPGA soft core specified in the Bluespec hardware description language (HDL), for which we have integrated the application of formal methods to the Bluespec specifications and the hardware-software implementation.

Acknowledgments

The authors of this report thank other members of the CTSRD team, and our past and current research collaborators at SRI and Cambridge:

Ross J. Anderson	Gregory Chadwick	Nirav Dave	Brooks Davis
Khilan Gudka	Jong Hun Han	Alex Horsman	Alexandre Joannou
Asif Khan	Myron King	Wojciech Koszek	Patrick Lincoln
Anil Madhavapeddy	Ilias Marinos	A. Theodore Markettos	Ed Maste
Andrew Moore	Will Morland	Alan Mujumdar	Prashanth Mundkur
Robert Norton	Philip Paeps	Colin Rothwell	John Rushby
Hassen Saidi	Hans Petter Selasky	Muhammad Shahbaz	Stacey Son
Richard Uhler	Philip Withnall	Bjoern Zeeb	

The CHERI team wishes thank 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			

Finally, we are grateful to Howie Shrobe, MIT professor and past DARPA CRASH program manager, who has offered both technical insight and support throughout this work. We are also grateful to Robert Laddaga, who has succeeded Howie in overseeing the CRASH program.

Contents

Chapter 1

Introduction

The Capability Hardware Enhanced RISC Instructions (CHERI) architecture extends the commodity 64-bit MIPS Instruction-Set Architecture (ISA) with new security primitives to allow software to efficiently implement *fine-grained memory protection* and an *object-capability security model*. CHERI's extensions are intended to support incrementally adoptable, high-performance, formally supported, and programmer-friendly underpinnings for robust and scalable *software compartmentalization* motivated by *t*he principle of least privilege. CHERI is a *hybrid capability-system architecture* in that gradual deployment of CHERI features in existing software is possible, offering a more gentle software adoption path. CHERI has four central design goals aimed at dramatically improving the security of contemporary Trusted Computing Bases (TCBs) through processor support for fine-grained memory protection and scalable software compartmentalization, which at times may conflict:

1. Granular 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 widely deployed approaches, 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. Fine-grained protection also provides the foundation for expressing compartmentalization within application instances.

2. Compartmentalization involves the decomposition of software into isolated components to mitigate the effects of security vulnerabilities by applying sound principles of security, such as abstraction, encapsulation, and especially least privilege. 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 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 features should significantly improve security so that vendors of mobile and embedded devices would seek its feature set from CPU companies (such as MIPS and ARM); 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 wish to 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 nonformal 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 additional coprocessor to the 64-bit MIPS ISA, but our approach is intended to easily support other similar ISAs, such as 64-bit ARM. 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 [70] or Google Native Client (NaCl) [85]. All of these future considerations are considerably enhanced by our use of Bluespec, which provides significant opportunities for rapid redesigns through its use of modular abstraction, encapsulation, and hierarchicalization.

1.1 Motivation

The CHERI CPU architecture provides a hardware foundation for principled, secure systems. Its design builds on and extends decades of research into hardware and operating-system secu-

rity.[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 would 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.

- 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.

- 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 FreeBSD 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* 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 hardware 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] Levy's *Capability-Based Computer Systems* provides a detailed history of segment- and capability-based designs through the early 1990s [38]. However, it leaves off just as the transition to microkernel-based capability systems such as Mach [1] and L4 [39], as well as capability-influenced virtual machines such as the Java Virtual Machine [21], begins. Section 1.6 discuss historical influences on this work in greater detail.

In the following sections, we consider the context and motivation for CHERI, a high-level view of the CHERI design, the role of formal methods in the project, and our work-in-progress research prototype.

1.1.1 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: even as the execution substrates improve in their ability to resist attacks such as buffer overflows and integer vulnerabilities, logical errors 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 [56] and Chromium [58], and most recently in our own Capsicum project [76], has been motivated by a critical security need; however it has seen success only at very coarse separation granularity due to the challenges involved.

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 security domains is expensive due to reliance on software and hardware address-space infrastructure, such as a quickly overflowed Translation Look-aside Buffer (TLB) 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 extremely 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.

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 names-

pace. 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 Design

CHERI 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 implications for the RISC-V and ARM ISAs. CHERI adds the following features to a RISC CPU design via a new *capability coprocessor* that supports granular memory protection within address spaces:

- The *capability register file* describes 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. Capability registers supplement the general-purpose register file, allowing capabilities to displace general-purpose registers in describing data and object references. 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.

- A set of *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 data and further capabilities can be loaded and stored via capability registers (i.e., dereferencing); object capabilities can be invoked, via special

[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 [8].

instructions, allowing a transition between protection domains. 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. Most capability instructions are part of the user-mode 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 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 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 implemention of a wide variety of OS, compiler, and application-originated policies on a common protection substrate that optimizes fast paths through hardware support.

The capability coprocessor is a coprocessor in two senses. First, the capability coprocessor occupies a portion of the existing ISA encoding dedicated to extensions (typically referred to as coprocessor instructions). Second, the capability coprocessor supplements the general-purpose register file with its own ISA-managed registers, as well as performing (and transforming) memory access, and delivering exceptions to the main pipeline, requiring hardware resources that interact with the primary processor pipeline. This behavior is comparable in many ways to system, floating-point, vector, or cryptographic coprocessors that will similarly supplement the base ISA and processor features.

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).

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. Reducing capabilities to 128-bit is not unreasonable, and measurements suggest that cache footprint increases from capabilities can be significantly mitigated through such a change. However, this would also introduce tradeoffs in the granularity of memory (e.g., losing access to the full 64-bit space) and flexibility of the object-capability design (e.g., loss of software-defined permission bits) that would need to be reasoned about carefully.

We believe, however, that the higher-level memory protection and security models we have described would relatively easily apply to variations in ISA-level implementation. This should allow reasonable source-level software portability (leaving aside 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 is moderately portable across underlying ISAs.

1.2.1 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. 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 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.

1.3 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 reducing opportunities for the conflation of code and data, as well as catching many common exploitable programmer bugs; compartmentalization constrains successful attacks via 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.

Somewhat to our chagrin, we report that the CHERI design currently includes no features for resisting covert or side-channel attacks: these have proven increasingly relevant in CPU design, but the tools CHERI provides do not improve resilience against these attacks. In some sense, they increase 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.

1.4 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 two 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 developed extensions to the Bluespec compiler to export an HDL description to SRI's PVS and SAL model checker. We have also developed new tools for efficient SMT (Satisfiability Modulo Theories) modeling of designs (using SRI's Yices), and the automated extraction of key properties from larger Bluespec designs. 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.

A detailed description of formal methods efforts relating to CHERI may be found in the to-be-published *CHERI Formal Methods Report*.

1.5 CHERI and CHERI2 Reference Prototypes

As a central part of this research, we are developing reference prototypes of the CHERI ISA via several CHERI processor designs. These protoypes 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 the companion *BERI Hardware Reference*, BERI Software Reference, and *CHERI User's Guide* documents.

Our first prototype, known simply as CHERI1, is based on Cambridge's MAMBA research processor, and is a single-threaded, single-core implementation intended to allow us to explore ISA tradeoffs. This prototype is implemented in the Bluespec 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.

We have ported the commodity open-source FreeBSD operating system, with support for 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. FreeBSD is able to manage the capability coprocessor and maintain additional thread state for capability-aware user applications, although capability features are not yet used within the kernel for its own internal protection. FreeBSD 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 addition, we have developed a number of capability-enhanced applications demonstrating fine-grained memory protection and in-process compartmentalization to explore security, performance, and programmability tradeoffs. We also have a work-in-progress multi-core version of the CHERI prototype.

Using Bluespec, 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 MRC project we have also targeted CHERI at the second-generation NetFPGA 10G research and teaching board, which we hope to use in ongoing research into datacenter network fabrics. That

work includes the development of Blueswitch, a Bluespec 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 Bluespec source code. 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, known as CHERI2, which deploys additional CPU features, such as support for multi-core operation and simultaneous multi-threading (SMT) support. It also employs a more stylized form of Bluespec that is intended to considerably enhance our formal analysis of the hardware architecure.

1.6 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) [12], and continued over the next decade with MIT's Multics project, described many central tenets of computer security [13, 24]. Dennis and Van Horn's 1965 *Programming Semantics for Multiprogrammed Computations* [16] 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 [63].

In 1975, Saltzer and Schroeder surveyed the rapidly expanding vocabulary of computer security in *The Protection of Information in Computer Systems* [64]. 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* [33], 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 [37], 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, reliablity, survivability, human safety, and so on (e.g., [50, 51], among others).

The Security Research community also blossomed outside of MIT: Wulf's Hydra operating system at Carnegie Mellon University (CMU) [83, 11], Needham and Wilkes' CAP Computer at Cambridge [81], SRI's Provably Secure Operating System (PSOS) [19, 51] hardware-software co-design that included strongly typed object capabilities, Rushby's security kernels supported by formal methods at Newcastle [62], and Lampson's work on formal models of se-

curity 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 [34, 35]. 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 [60].

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 [3]. 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 [29]. 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 [7]. 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 [47] 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.

1.6.1 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 [34, 35].

Systems such as the CAP Computer at Cambridge [81] and Ackerman's DEC PDP-1 architecture at MIT [2] attempted to realize this vision through embedding notions of capabilities in the memory management unit of the CPU, an approach described by Dennis and Van Horn [16]. Levy provides a detailed exploration of segment- and capability-oriented computer system design through the mid-1980s in *Capability-Based Computer Systems* [38].

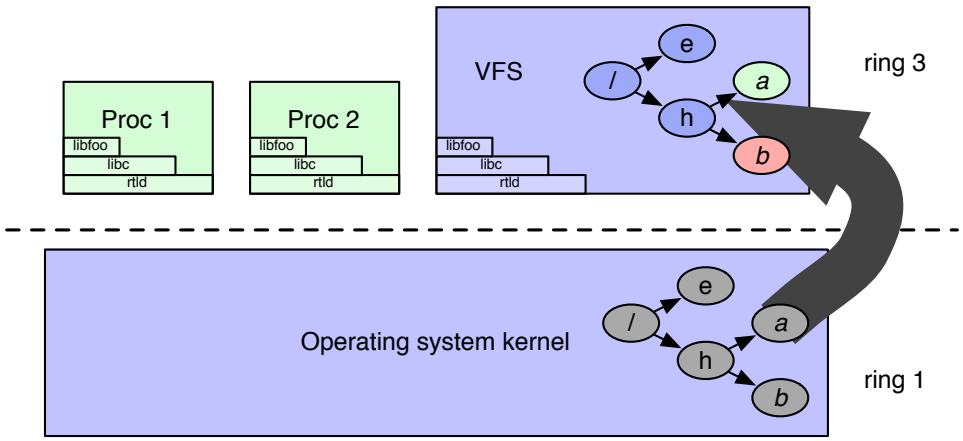


Figure 1.1: The microkernel movement shifted complex OS components, such as file systems, from the kernel to userspace tasks linked by IPC. Microkernels provide a smaller, easier-to-analyze, easier-to-debug, and more robust foundation in the face of dramatic increases in OS complexity.

1.6.2 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 [15]. 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) [55].

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 [11, 84] 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.[3] 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 [27].

Successors to Hydra at CMU include Accent and Mach [57, 1], both microkernel systems intended to explore the decomposition of a large and decidedly un-robust operating system kernel. Figure 1.1 illustrates the principle of microkernel design: 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

[3]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* [45]

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 [35]. This further aligned with proposals by Andrews [4] and Rushby [62] 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 [18].

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 [40].

The notion that the microkernel, rather than the hardware, is responsible for implementing the protection semantics of capabilities also aligned well with the simultaneous research (and successful technology transfer) of RISC designs, which eschewed microcode by shifting complexity to the compiler and operating system. Without microcode, the complex C-list peregrinations of CAP's capability unit, and protection domain transitions found in other capability-based systems, become less feasible in hardware. Simple virtual memory designs based on fixed-size pages and few semantic constraints 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 [65] and Neumann's Provably Secure Operating System [20] design study were ground-up operating system designs based soundly in formal methodology.[4] In contrast, Schroeder's MLS kernel design for Multics [66], the DoD Kernelized Secure Operating System (KSOS) [42], and Bruce Walker's UCLA UNIX Security Kernel [71] 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 [72].

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 [9]. 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 [67]. 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 [69]. A significant focus of the FLASK work was performance: an access vector cache is responsible for caching access control decisions throughout the OS to avoid costly up-calls and message passing (with associated context switches) to the security server. NSA and SCC

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

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 [48] – effectively, a composition of mandatory access control and capabilities in service to application security. This approach is embodied by Efstathopoulos et al.'s Asbestos [17] and Zeldovich et al.'s Histar [87] 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 [25], Shapiro's EROS [68] (now CapROS) continues the investigation of higher-assurance software capability designs, seL4 [31], a formally verified, capability-oriented microkernel, has also continued along this avenue. General-purpose systems also have adopted elements of the microkernel capability design philosophy, such as Apple's Mac OS X [5] (which uses Mach interprocess communication (IPC) objects as capabilities) and Cambridge's Capsicum [76] 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 [23, 22]. 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 [45], Wagner's Joe-E capability-safe subset of Java [44], and Miller's Caja capability-safe subset of JavaScript continue a language-level exploration of capability security [46].

1.6.3 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 [41]. 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 [6], and the DTrace instrumentation tool that, like BPF, uses a narrow virtual instruction set to implement the D language [10].

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 [86]. NaCl is closely related to Software Fault Isolation (SFI) [70], 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 [49]. 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 [26], and type-safe languages such as Ruby [61], 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 [43].

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 [59]; 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 [73]. Web browsers frequently incorporate many of these techniques including Java Virtual Machines and a JavaScript interpreter.

1.6.4 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 [18]), 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 [54, 14]. 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 [51, 52]. 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.

Capsicum Capsicum is a lightweight OS capability and sandbox framework included in FreeBSD 9.x and later [76, 75]. 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 both 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 motivations CHERI's exploration of more scalable approaches.

1.6.5 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 the chapter, 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.

1.7 Publications

As our approach has evolved, and project developed, we have published a number of papers and reports describing aspects of the work. The revisiting of capability-based approaches is described in *Capabilities Revisied: A Holistic Approach to Bottom-to-Top Assurance of Trustworthy Systems*, published at the Layered Assurance Workshop (LAW 2010) [53], 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) [74]. Most recently, we have published *The CHERI capability model: Revisiting RISC in an age of risk* at the International Symposium on Computer Architecture (ISCA 2014) [82].

We have additionally prepared a number of technical reports, including this document, describing our approach and prototypes. To date, this includes the *CHERI Instruction-Set Architecture* [79], the *CHERI User's Guide* [78], the *BERI Hardware Reference* [80], and the *BERI Software Reference* [77]. Further research publications and technical reports on the topics of the CHERI hardware-software security model, compiler approaches, and applications of formal methods will be forthcoming.

1.8 Version History

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 permissions; modified instruction encodings for CGetLen, CMove, and CSetLen; specification of reset state for capability registers; and clarification of the CIncBase instruction.

1.5 This version of the document was produced almost two years into the CTSRD project. It documented a significant revision 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 Bluespec 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 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.

1.9 Document Structure

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

Chapter 1 introduces CHERI: its motivations, goals, context, philosophy, and design.

Chapter 2 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 3 describes the CHERI capability coprocessor, its register file, tagged memory, and other ISA-related semantics.

Chapter 4 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 6 discusses the programming language and operating system implications of CHERI, including its impact on operating-system kernels, language runtimes, and compilers.

Chapter 7 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.

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, may be found in accompanying reports.

Chapter 2

CHERI Architecture

In this chapter we discuss the high-level design for the CHERI instruction-set architecture (ISA) and consider both the semantics and mechanism of CHERI's memory and object capabilities. We discuss CHERI in relative isolation from the general-purpose ISA, as our approach might reasonably apply to a number of RISC ISAs (e.g., including MIPS and ARM), but potentially also to CISC ISAs (such as Intel and AMD 32-bit and 64-bit ISAs).[1] In Chapter 3, we consider in detail an instantiation of the CHERI model in an extension to the 64-bit MIPS ISA.

2.1 Design Goals

As described in Chapter 1, the key observation motivating the CHERI design is that page-oriented virtual memory, nearly universal in commodity CPUs, is neither an efficient nor a programmer-friendly primitive for fine-grained memory protection or scalable hardware-supported compartmentalization. Virtual addressing, implemented by a memory management unit (MMU) and translation look-aside buffer (TLB), clearly plays an important role by dis-associating 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 for hardware isolation actively deter employment of the principle of least privilege. These concerns translate into three high-level security design goals for CHERI:

1. Management of security context must be a "fast path" that avoids expensive operations such as TLB entry invalidation, frequent ring transitions, and cache-busting OS supervisor paths. This is a natural consequence of the integral role security functions (such as creation, refinement, and delegation of memory and object rights) play in fine-grained compartmentalized code.

[1] One idea we have considered is that CHERI-like semantics might be accomplished through extensions to Google's Native Client ISA [85], which uses a strict and statically analyzable subset of Intel and ARM ISAs to ensure memory safety. An exciting possibility is that we might extend the LLVM intermediate representation [36] to capture notions of segmentation and capability protection, in which case either NaCl or CHERI back ends might be targeted as underlying execution substrates. This naturally raises the question, "why new hardware" – one that we have constantly in mind, and believe will be constructively answerable in terms of functionality, performance, and formal assurances.

2. Security domain switches must be inexpensive and efficient, with cost scaling linearly with the number of switches and actual code/data footprint (and hence general-purpose cache performance), rather than scaling as a product of the number of security domains and controlled objects regardless of code and data cache footprints. CHERI is intended to support at least two orders of magnitude more active security domains per CPU than current MMU-based systems (going from tens or hundreds to at least tens of thousands of domains).

3. Security domain switches must allow shared object namespaces that provide a unified view that connotes both efficient and programmer-comprehensible delegation. Compartmentalized applications should able to be programmed and debugged without unnecessary recourse to distributed-system methodology.

These security goals, 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 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.[2] 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.

2.2 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 2.1). 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 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

[2]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.

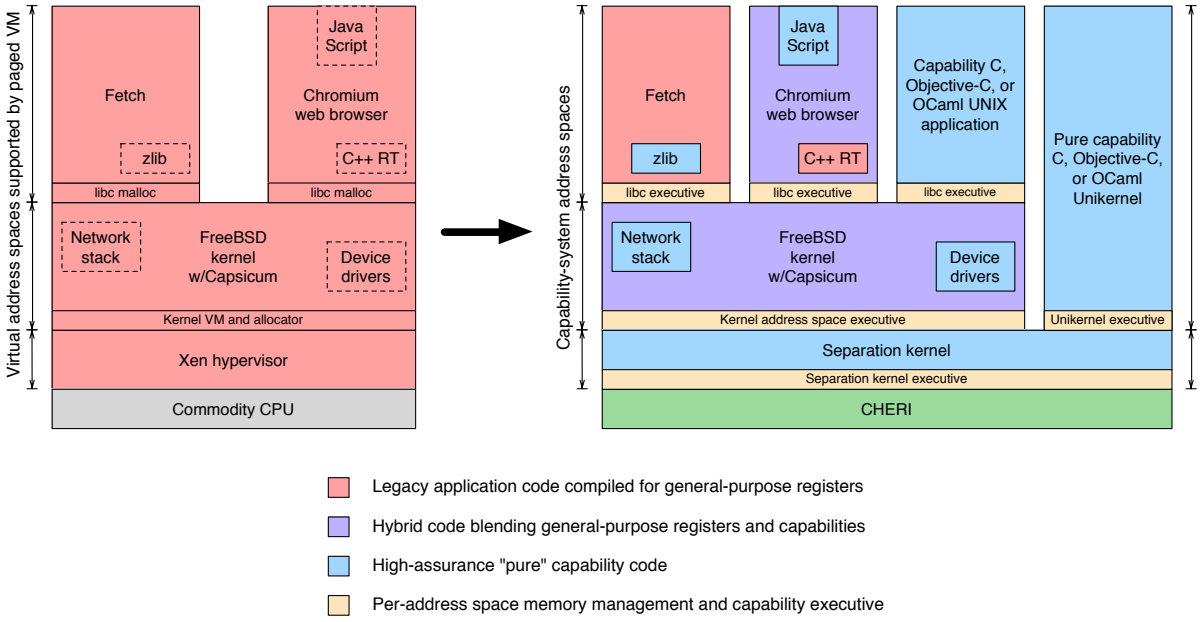


Figure 2.1: CHERI's hybrid capability architecture: initially, legacy software components execute without capability awareness, but security-sensitive TCB elements or particularly risky code bases are converted. In the long term, all packages are converted, implementing least privilege throughout the system.

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 2.1 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.[3] Useful comparison might also be made between our notion of an in-address-space executive and a microkernel, as the executive will

[3]Similar observations about the criticality of the run-time linker for both security and performance in capability systems have been made by Karger [30].

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.

2.3 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 [76] (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 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 [76] 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 pro-

cessing), 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.

2.4 Capability Model

Chapter 3 provides detailed documentation of the registers, capabilities, and new instructions currently defined in CHERI. These concepts are briefly introduced here.

2.4.1 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.

2.4.2 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.

Controlled manipulation is enforced by virtue of the ISA: instructions that manipulate capability register fields (e.g., base, 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, so that system security cannot rely solely on the goodness of compilers.

2.4.3 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.

Each capability register has 256 bits; unlike general-purpose registers, capability registers are structured, and contain a number of fields with defined semantics and constrained values:

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.

Object type / entry address Notionally the *object type*, used to ensure that corresponding code and data capabilities for the object are used together correctly.

Base This is the base address of a memory region.

Length This defines the length of a memory region.

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 sizable capability register file. 32 registers is nicely symmetric with the MIPS ISA, but in practice leads to a substantial overhead; we have considered reducing the number to 16 or even 8 to reduce hardware resource, cache footprint, and context-switch time. We have also pondered schemes to reduce the size of capability registers themselves; simple reduction of the addresses to smaller numbers to 40-bit from 64-bit, reflecting the largest virtual addresses supportable in the MIPS TLB, might allow reduction to 128-bit. Another approach might be to differentiate larger object capabilities, which require types, from pure memory capabilities, which could be represented more compactly, but would require compilers to handle multiple capability sizes. Finally, more complex techniques, such as Low-Fat Pointers [32] might also prove useful. Once again, hardware specifications written in Bluespec 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.

2.4.4 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.

2.4.5 Ephemeral Capabilities and Revocation

To this end, capabilities may be further tagged as *ephemeral*, which allows them to be processed in registers, stored in constrained memory regions, and passed on via invocation of other objects. The goal of capability ephemerality 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, ephemeral 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, *ephemeral 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, ephemeral capabilities can be limited to a particular thread, with bounded delegation time down the (logical) stack.[4]

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[5], so causing the C runtime to not reuse address space is now

[4]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 ephemeral 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.

[5]As is 640K of memory. It has also not escaped our notice that there is a real OS cost to maintaining the

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.

2.4.6 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 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 [28], 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.

2.4.7 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:

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.

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 because incorrect handling could otherwise lead to serious vulnerabilities.

2.4.8 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.

2.4.9 Capability Instructions

Various newly added instructions are documented in detail in Chapter 3. 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.

2.4.10 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.[6] In this view, hardware thread contexts, or register files, are simply caches of thread state to be managed by the processor.

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

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

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

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

These concerns argue for at least three features: a logical stack spanning many stack fragments bound to individual security domains, a fresh source of ephemeral stacks ready for reuse, and some notion of a do-not-transfer facility in order to prevent the further propagation of a capability (perhaps implemented via a revocation mechanism, but other options are readily apparent). PSOS explored similar notions of propagation-limited capabilities with similar motivations.

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.

2.4.11 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 to direct DMA to arbitrary physical memory addresses, although hybrid models – such as allowing DMA 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, how-

ever: 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. Another alternative would be to pursue the notion of smart buses used by peripherals – for example, making them capability aware.

Chapter 3

Capability Coprocessor

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.

NOTE: the instruction-set architecture described here is preliminary; we expect to refine it significantly as a result of ongoing discussion, hardware prototyping, practical experimentation, and user feedback!

3.1 Capability Registers

Table 3.1 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.

Conventions for Capability Register Use

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.

The current convention used by LLVM makes the following reservations for calls within a protection domain:

Register(s)	Description
PCC	Program counter capability (**PCC**); the capability through which **PC** is indirected by the processor when fetching instructions.
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...**C23**	General-purpose capability registers referenced explicitly by capability-aware instructions.
RCC (**C24**)	Return code capability; after a `CJALR` instruction, the previous value of **PCC** is saved in **RCC**.
C25	General-purpose capability register reserved for use in exception handling.
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 3.1: Capability registers defined by the capability coprocessor.

- **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. Capability returns are placed in **C3**.

- **C11**-**C16** are caller-save registers.

- **C17**-**C24** are callee-save registers.

When calling less-trusted code, there is no guarantee that a non-malicious callee will abide by these conventions. Thus, all registers should be regarded as caller-save. Additionally, all capability registers that are not part of the explicit argument set should be invalidated using the CClearTag instruction. This will prevent leakage of rights to untrustworthy callees. Where rights are explicitly passed to a callee, it may be desirable to clear the non-ephemeral bit which will (in a suitably configured runtime) prevent further propagation of the capability. Similar concerns apply to general-purpose registers, which should be preserved by the caller if their correct preservation is important, and cleared if they might leak sensitive data.

Protected Procedure Calls

A protected procedure call, instruction **CCall**, escapes to a handler which 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 protected procedure return, instruction **CReturn**, also escapes to a handler which 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 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

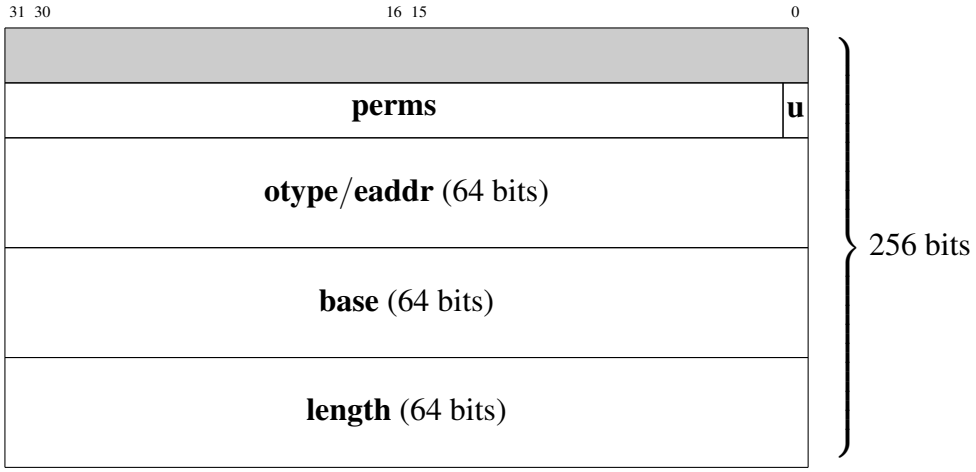


Figure 3.1: Contents of a capability

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.

Capabilities and Exception Handling

KCC and **KDC** hold the code capability and data capability which describe the protection domain of the system exception handler. When an exception occurs, **KCC** is moved to **PCC** and the victim **PCC** is copied to **EPCC** so that the exception may return to the correct address.

When an exception handler returns with **eret**, **EPCC** is moved into **PCC**.

3.2 Capabilities

The CHERI processor is currently always defined to be big-endian, in contrast to traditional MIPS, which allows endianness to be selected by the supervisor. Figure 3.1 illustrates the format of a capability.

Each capability register contains the following fields:

- Tag bit ("**tag**", 1 bit)
- Unsealed flag ("**u**", 1 bit)
- Permissions mask ("**perms**", 31 bits)
- Object type ("**otype**/**eaddr**", 64 bits)
- Base virtual address ("**base**", 64 bits)
- Length in bytes ("**length**", 64 bits)

3.2.1 tag

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.

3.2.2 u

The **u** flag indicates whether a capability is usable for general-purpose capability operations. If this flag is cleared, 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**/**eaddr** fields, both capabilities will have their **u** flag set and will be made available in the next cycle, thus entering a new security domain.

3.2.3 perms

The 31-bit **perms** bit vector governs the permissions of the capability including read, write and execute permissions. The contents of this field are listed in table 3.2. Bits 15–30 may be used by application programs for user-defined permissions; they can be checked using the **CCheckPerm** instruction.

3.2.4 otype/eaddr

This 64-bit field holds the virtual address of the entry point of an executable capability. This field also holds the "type" of a non-executable capability. The **CSetType** instruction sets the **otype**/**eaddr** field to the absolute virtual address of an entry point of an executable capability. The **CSealCode** instruction can then seal the executable capability and treat the entry point as a unique object type. Furthermore, the **CSealData** instruction may seal a non-executable capability with the **otype**/**eaddr** of an unsealed executable capability. Possession of a capability with the *Permit_Set_Type* permission authorizes a domain to call **CSetType** with a type within the capability's range. This arrangement provides for the construction of matching executable and data-only capabilities of the same **otype**/**eaddr** to be used in protected procedure calls.

3.2.5 base

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

3.2.6 length

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

3.2.7 Capability Permissions

Table 3.2 shows constants currently defined for memory permissions; remaining bits are software-defined.

Non_Ephemeral Allow this capability to persist beyond a protected procedure return.

Value	Name
0	Non_Ephemeral
1	Permit_Execute
2	Permit_Load
3	Permit_Store
4	Permit_Load_Capability
5	Permit_Store_Capability
6	Permit_Store_Ephemeral_Capability
7	Permit_Seal
8	Permit_Set_Type
9	Reserved
10	Access_EPCC
11	Access_KDC
12	Access_KCC
13	Access_KR1C
14	Access_KR2C

Table 3.2: Memory permission bits for the **perms** capability field

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_Ephemeral_Capability Allow this capability to be used as a pointer for storing ephemeral capabilities.

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

Permit_Set_Type Allow setting the **otype**/**eaddr** of this capability to any value between **base** and **base+length**-1 if Permit_Execute is also set.

Access_EPCC Allow access to **EPCC** when this capability is in **PCC**.

Access_KR1C Allow access to **KR1C** when this capability is in **PCC**.

Access_KR2C Allow access to **KR2C** when this capability is in **PCC**.

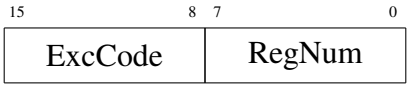


Figure 3.2: Capability Cause Register

Access_KCC Allow access to **KCC** when this capability is in **PCC**.

Access_KDC Allow access to **KDC** when this capability is in **PCC**.

Ephemeral capabilities can be stored only via capabilities that have the Permit_Store_Ephemeral_Capability permission bit set; normally, this permission will be set only on capabilities that, themselves, have the Non-Ephemeral bit cleared.

3.3 Capability Exceptions

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). The *ExcCode* field within the **cause** register of coprocessor 0 will be set to 18 (*C2E*, coprocessor 2 exception) when the cause of the exception is that the attempted operation is prohibited by the capability system. The current **PCC** will be moved to **EPCC** and **KCC** will be moved into **PCC**, which should allow the kernel exception handler to run successfully.

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 3.2. The possible values for the *ExcCode* of **capcause** are shown in table 3.3. If the last instruction to throw an exception did not throw a capability exception, then the *ExcCode* field of **capcause** will be *None*. *ExcCode* values from 128 to 255 are reserved for use by application programs. (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_EPCC* is not set, so the operating system can prevent user space programs from reading **capcause** directly by not granting them *Access_EPCC* permission.

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 3.4. The *RegNum* field of **capcause** is set to the register which caused the highest priority exception.

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
0x10	Non_Ephemeral 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_Ephemeral_Capability Violation
0x17	Permit_Seal Violation
0x18	Permit_Set_Type Violation
0x19	*reserved*
0x1a	Access_EPCC Violation
0x1b	Access_KDC Violation
0x1c	Access_KCC Violation
0x1d	Access_KR1C Violation
0x1e	Access_KR2C Violation
0x1f	*reserved*

Table 3.3: Capability Exception Codes

Priority	Description
1	Access_EPCC Violation
	Access_KDC Violation
	Access_KCC Violation
	Access_KR1C Violation
	Access_KR2C Violation
2	Tag Violation
3	Seal Violation
4	Type Violation
5	Permit_Seal Violation
6	Permit_Set_Type Violation
7	Permit_Execute Violation
8	Permit_Load Violation
	Permit_Store Violation
9	Permit_Load_Capability Violation
	Permit_Store_Capability Violation
10	Permit_Store_Ephemeral_Capability Violation
11	Non_Ephemeral Violation
12	Length Violation
13	User-defined Permission Violation
14	Call Trap
	Return Trap

Table 3.4: Exception Priority

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 which 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.

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*.

Software Emulation 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 in software by a trap handler. This exception uses a different trap handler vector, at 0x100 above the general purpose exception handler. The exception cause will be *C2E* and **capcause** will be *Call Trap* for CCall and *Return Trap* for CReturn.

3.4 CPU Reset

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

- The **tag** bit is set.
- The **u** bit is set.
- **base** = 0
- **length** = $2^{64} - 1$
- **otype**/**eaddr** = 0
- All permissions bits are set.
- All unused bits are set.

The initial values of **PCC** and **EPCC** will allow the system to initially execute code relative to virtual address 0. The initial value of **C0** will allow general-purpose loads and stores to all of virtual memory for the bootstrapping process. The initial value of **IDC** will allow the creation of any further capabilities required to bootstrap the system.

3.5 Changes to Standard MIPS Processing

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

Instruction fetch When the CPU fetches an instruction from **PC**, it indirects the instruction fetch through **PCC**. If the instruction fetch is not permitted due to a bounds check failure or a permission error (*Permit_Execute* not set), coprocessor 2 exception (*C2E*) is thrown.

If an exception occurs during instruction fetch (e.g. AdEL, or a TLB miss) then *BadVAddr* is set equal to **PCC**.base + **PC**.

Load and Store instructions When the CPU performs a standard MIPS load or store instruction, the address to be read from (or written to) is indirected through **C0**. **C0** must have the appropriate permission (*Permit_Store* or *Permit_Load*) set, and the addresses read or written must be between **C0**.**base** and **C0**.**base** + **C0**.**length** − 1. If the load or store is not permitted due to a memory bound check failure or a permission error, a coprocessor 2 exception (*C2E*) is thrown.

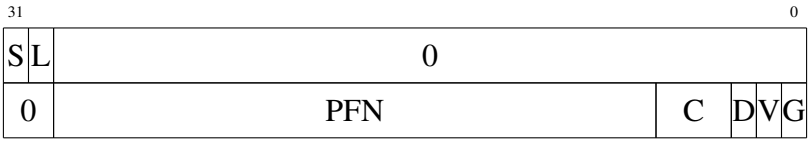


Table 3.5: EntryLo Register

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 **C0**.**base** and checked against the permissions and bounds of **C0**.

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

3.6 Changes to the TLB

CHERI adds two new fields to the EntryLo register, shown as L and S in Table 3.5. If L is set, capability loads are disabled for the page; if a load capability instruction is used on a page with the L bit set, then an exception will be raised, setting CP0.Cause.ExcCode to 16. This exception will be raised even if the corresponding tag bit was not set (i.e., the bytes to be loaded were non-capability data). If S is set, capability stores are disabled for the page; if a store capability instruction is used on a page with the S bit set, then an exception will be raised, setting CP0.Cause.ExcCode to 17. This exception will be raised even if the tag bit was unset in the capability register to be stored (i.e., it contained non-capability data).

3.7 Proposed Extensions to the CHERI ISA

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

- Move MIPS memory-access interposition from **C0** to a control register accessed via special instructions, as is the case for **PCC**.
- Define a `NULL` capability, which could be loaded from **C0**, to make checks easy to implement. A capability `NULL` would have the tag bit set (valid capability), but grant no access rights (cannot be dereferenced, and can be safely delegated – unlike C `NULL` which, when cast to a capability, grants full address-space rights).
- Provide some means of efficiently implementing a capability tag-clearing `memcpy()` that will not throw a fault when a capability is found in source memory. This might be a new load instruction variant that clears tags, or could be a variant on **CClearTag** that preserves (useful) data while clearing the tag. With an explicit zero-capability register, `CClearTag` could just clear the tag, not the data; however, an exception would still result if the source did not have load-capability permissions.

- Allow **CCall/CReturn** instructions to partially implement capability call/return semantics while still throwing an exception to allow software processing. For example, they might implement all necessary checks so that software does not need to do this.

- Implement an explicit *capability cursor*, completing support for fat-pointer style operation. A capability cursor will differentiate the point of access for a load or store operation from the bounds imposed on that access; the cursor would move flexibly but loads and stores would trigger an exception if they occur while the cursor points outside of the permitted bounds. This might be of particular use in supporting C pointer arithmetic, in which some application programs temporarily construct pointers that are invalid – e.g., during packet parsing – that they will not dereference. Early prototyping of such an approach, reusing the entry address/type field as a cursor, suggests that it might prove effective in improving C compatibility for complex buffer management code.

The following changes have been discussed for longer-term consideration in the CHERI ISA:

- Allow **CReturn** to accept code/data capability arguments, which might be ignored for the time being.

- 32-byte capabilities impose measurable overhead; implementing a 16-byte "compressed capability" representation, usable for pure data capabilities, might reduce this overhead. In order for this to be useful, the compiler must be able to statically determine and control use of compressed capabilities. Initial simulations of 128-bit capabilities suggest a substantial reduction in cache footprint.

Chapter 4

Instruction-Set Reference

CHERI 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. Table 4.1 lists available capability coprocesor instructions.

4.1 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
- 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

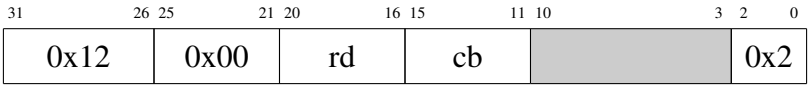
Mnemonic	Description
CGetBase	Move base to a general-purpose register
CGetLen	Move length to a general-purpose register
CGetTag	Move tag bit to a general-purpose register
CGetUnsealed	Move unsealed bit to a general-purpose register
CGetPerm	Move permissions field to a general-purpose register
CGetType	Move object type field to a general-purpose register
CGetPCC	Move the PCC and PC to general-purpose registers
CGetCause	Move capability exception cause register to a general-purpose register
CSetCause	Set the capability exception cause register
CIncBase	Increase Base
CMove	Pseudo-instruction for CIncBase with no change to the base
CSetLen	Set Length
CClearTag	Clear the tag bit
CAndPerm	Restrict Permissions
CSetType	Set the **otype**/**eaddr** of an executable capability
CCheckPerm	Check **perms** field
CCheckType	Check **otype**/**eaddr** field
CFromPtr	Create capability from pointer
CToPtr	Capability to pointer
CBTU	Branch if capability tag is unset
CBTS	Branch if capability tag is set
CSC	Store Capability Register
CLC	Load Capability Register
CL[BHWD][U]	Load Byte, Half-Word, Word or Double Via Capability Register (Unsigned)
CS[BHWD]	Store Byte, Half-Word, Word or Double Via Capability Register
CLLD	Load linked doubleword via capability register
CSCD	Store conditional doubleword via capability pregister
CJR	Jump Capability Register
CJALR	Jump and link Capability Register
CSealCode	Seal an executable capability
CSealData	Seal a non-executable capability with the **otype**/**eaddr** of an executable capability
CUnseal	Unseal a sealed capability
CCall	Protected procedure call into a new security domain
CReturn	Return to the previous security domain

Figure 4.1: Capability coprocessor instruction summary

CGetBase: Move Base to a General-Purpose Register

Format (4)

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()
else
    rd ← cb.base
end if
```

Exceptions

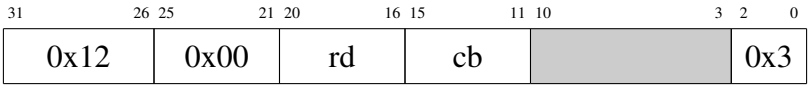
A coprocessor 2 exception is raised if:

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

CGetLen: Move Length to a General-Purpose Register

Format (4)

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()
else
    rd ← cb.length
end if
```

Exceptions

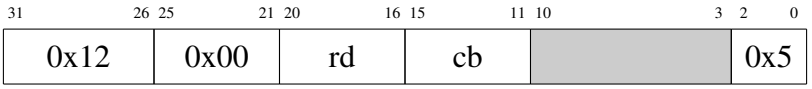
A coprocessor 2 exception is raised if:

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

CGetTag: Move Tag to a General-Purpose Register

Format (4)

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()
else
    rd[0] ← cb.tag
    rd[1:63] ← 0
end if
```

Exceptions

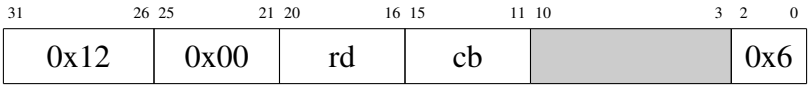
A coprocessor 2 exception is raised if:

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

CGetUnsealed: Move sealed bit to a General-Purpose Register

Format (4)

CGetUnsealed rd, cb



Description

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

Pseudocode

```
if register_inaccessible(cb) then
    raise_c2_exception()
else
    rd[0] ← cb.unsealed
    rd[1:63] ← 0
end if
```

Exceptions

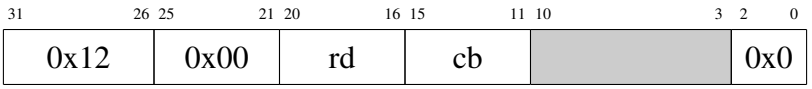
A coprocessor 2 exception is raised if:

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

CGetPerm: Move Memory Permissions Field to a General-Purpose Register

Format (4)

CGetPerm rd, cb



Description

The least significant 15 bits (bits 0 to 14) 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()
else
    rd[0:14] ← cb.perms
    rd[15:63] ← 0
end if
```

Exceptions

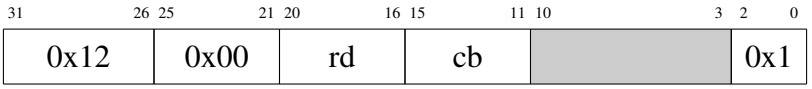
A coprocessor 2 exception is raised if:

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

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

Format (4)

CGetType rd, cb



Description

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

Pseudocode

```
if register_inaccessible(cb) then
    raise_c2_exception()
else
    rd ← cb.otype
end if
```

Exceptions

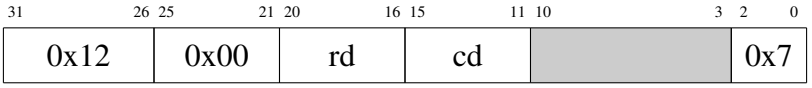
A coprocessor 2 exception is raised if:

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

CGetPCC: Move the PCC and PC to General-Purpose Registers

Format (4)

CGetPCC rd(cd)



Description

General-purpose register *rd* is set equal to the **PC** and the capability register *cd* is set to the **PCC**.

Pseudocode

```
if register_inaccessible(cd) then
    raise_c2_exception()
else
    rd ← PC
    cd ← PCC
end if
```

Exceptions

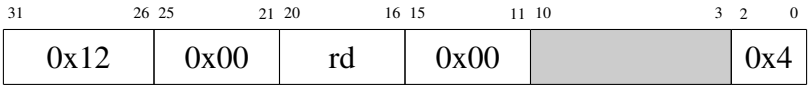
A coprocessor 2 exception is raised if:

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

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

Format (4)

CGetCause rd



Description

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

Pseudocode

```
if not PCC.perms.Access_EPCC then
    raise_c2_exception(exceptionAccessEPCC, 0xff)
else
    rd ← CapCause
end if
```

Exceptions

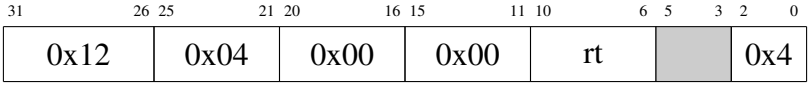
A coprocessor 2 exception is raised if:

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

CSetCause: Set the Capability Exception Cause Register

Format (1)

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_EPCC then
    raise_c2_exception(exceptionAccessEPCC, 0xff)
else
    CapCause ← rt
end if
```

Exceptions

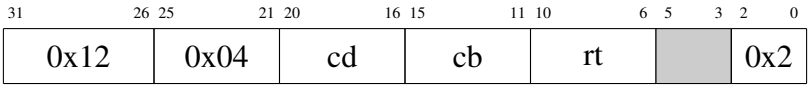
A coprocessor 2 exception is raised if:

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

CIncBase: Increase Base

Format (1)

CIncBase cd, cb, rt
CMove cd, cb



Description

Capability register *cd* is replaced with the contents of capability register *cb* with the **base** field set to the sum of its previous value and the contents of general-purpose register *rt*. The **length** field of capability register *cd* is replaced with *cb.***length** minus the contents of general-purpose register *rt*, ensuring that capability register *cd* points to a subset of the original memory region.

Pseudocode

```
if register_inaccessible(cd) then
    raise_c2_exception()
else if register_inaccessible(cb) then
    raise_c2_exception()
else if not cb.tag and rt ≠ 0 then
    raise_c2_exception(exceptionTag, cb)
else if not cb.unsealed and rt ≠ 0 then
    raise_c2_exception(exceptionSealed, cb)
else if rt > cb.length then
    raise_c2_exception(exceptionLength, cb)
else
    cd ← cb
    cd.base ← cb.base + rt
    cd.length ← cb.length − 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 the corresponding bit in **PCC**.**perms** is not set.
- *cb.***tag** is not set and $rt \neq 0$.
- *cb.***u** is not set and $rt \neq 0$.
- $rt > cb.$**length**

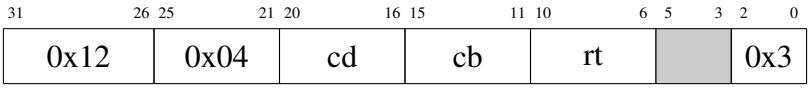
Notes

- **CIncBase** can be used to copy one register to another by setting *rt* equal to zero. If *rt* is zero, the operation will succeed even if *cb*.**u** is not set, allowing it to be used to copy sealed capabilities. **CIncBase** also succeeds if *rt* is zero and *cb*.**tag** is unset, allowing it to be used to copy non-capability data items between capability registers.
- In assembly language, **CMove** *cd*, *cb* is a pseudo-instruction which the assembler converts to **CIncBase** *cd*, *cb*, *$zero*.

CSetLen: Set Length

Format (1)

CSetLen cd, cb, rt



Description

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

Pseudocode

```
if register_inaccessible(cd) then
    raise_c2_exception()
else if register_inaccessible(cb) then
    raise_c2_exception()
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if not cb.unsealed then
    raise_c2_exception(exceptionSealed, cb)
else if rt > cb.length then
    raise_c2_exception(exceptionLength, cb)
else
    cd ← cb
    cd.length ← rt
end if
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.()
- *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.***u** is not set.
- $rt > cb$.**length**

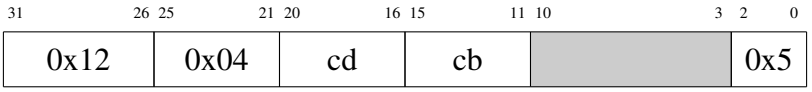
Notes

Unlike **CIncBase**, this operation will always raise an exception if *cb.***tag** or *cb.***u** are unset, even if the length is unchanged.

CClearTag: Clear the tag bit

Format (1)

CClearTag cd, cb



Description

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

Pseudocode

```
if register_inaccessible(cb) then
    raise_c2_exception()
else if register_inaccessible(cd) then
    raise_c2_exception()
else
    cd ← cb
    cd.tag ← false
end if
```

Exceptions

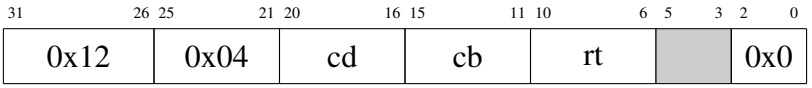
A coprocessor 2 exception is raised if:

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

CAndPerm: Restrict Permissions

Format (1)

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()
else if register_inaccessible(cb) then
    raise_c2_exception()
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if not cb.unsealed then
    raise_c2_exception(exceptionSealed, cb)
else
    cd ← cb
    cd.perms ← cb.perms ∩ rt
end if
```

Exceptions

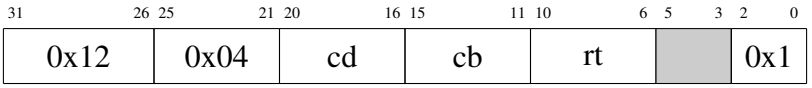
A coprocessor 2 exception is raised if:

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

CSetType: Set the otype of a Capability

Format (1)

CSetType cd, cb, rt



Description

Capability register *cd* is replaced with the contents of *cb*, with the **otype**/**eaddr** field set to *cb*.**base**+*rt* and the *Permit_Seal* bit in the **perms** field set.

Purpose

CSetType is used to set the **otype**/**eaddr** field of a capability. **otype**/**eaddr** has two related purposes:

1. If the capability is subsequently sealed with **CSealCode** and called with **CCall**, then control will be transferred to the address given by its **otype**/**eaddr**. In terms of object oriented programming, the **otype**/**eaddr** is the address of some code that implements the methods of a class.

2. If the capability is subsequently used as the *ct* parameter to **CSealData** or **CUnseal**, **otype**/**eaddr** acts as a unique identifier for a user-defined class: only subsystems that have a capability for that **otype**/**eaddr** value are permitted to seal or unseal capabilities with that **otype**/**eaddr**. In terms of object oriented programming, **otype**/**eaddr** grants permission to create or examine the internal structure of objects of a particular class.

The connection between the two is that it is the methods of the class (purpose 1) that are granted permission to create or examine the internal structure of members of the class (purpose 2). The same field is used for both purposes to save bits within a capability, and because the entry point of the methods serves as a convenient unique identifier for the class.

Pseudocode

```
if register_inaccessible(cd) then
    raise_c2_exception()
else if register_inaccessible(cb) then
    raise_c2_exception()
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if not cb.unsealed then
    raise_c2_exception(exceptionSealed, cb)
else if not cb.perms.Permit_Set_Type then
    raise_c2_exception(exceptionPermitSetType, cb)
else if rt ≥ cb.length then
    raise_c2_exception(exceptionLength, cb)
```

else
cd $\leftarrow$ cb
cd.otype $\leftarrow$ cb.base + rt
cd.perms.Permit_Seal $\leftarrow$ **true**
end if

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb*.**u** is not set.
- *cb*.**perms**.*Permit_Set_Type* is not set.
- *rt* $\geq$ *cb*.**length**.

CCheckPerm: Raise exception if don't have 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* which 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()
else if not cs.tag then
    raise_c2_exception(exceptionTag, cs)
else if cs.perms ∩ rt ≠ rt then
    raise_c2_exception(exceptionUserDefined, cs)
end if
```

Exceptions

A coprocessor 2 exception is raised if:

- *cs* 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.
- There is a bit which is set in *rt* and is not set in *cs*.**perms**.

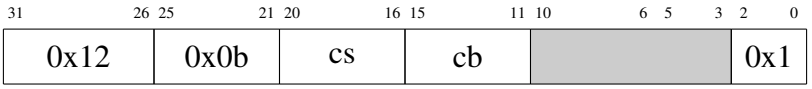
Notes

- 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*.**u**.

CCheckType: Raise exception if otypes don't match

Format (3)

CCheckType cs, cb



Description

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

Pseudocode

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

Exceptions

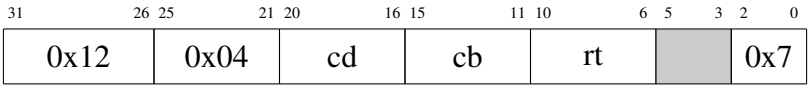
A coprocessor 2 exception is raised if:

- *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.
- *cs*.**tag** is not set.
- *cb*.**tag** is not set.
- *cs*.**u** is set.
- *cs*.**u** is set.
- *cs*.**otype**/**eaddr**$\neq$ *cb*.**otype**/**eaddr**.

CFromPtr: Create capability from pointer

Format (1)

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 set, all other bits unset). If *rt* is non-zero, then *cd* will be a capability whose base is *cb*.**base**+*rt*.

Pseudocode

```
if register_inaccessible(cd) then
    raise_c2_exception()
else if register_inaccessible(cb) then
    raise_c2_exception()
else if rt = 0 then
    cd.tag = true
    cd.base = 0
    cd.length = 0
    cd.perms = ∅
    cd.reserved = 0
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if not cb.unsealed then
    raise_c2_exception(exceptionSealed, cb)
else if rt > cb.length then
    raise_c2_exception(exceptionLength, cb)
else
    cd ← cb
    cd.base ← cb.base + rt
    cd.length ← cb.length − rt
end if
```

Exceptions

A coprocessor 2 exception is raised if:

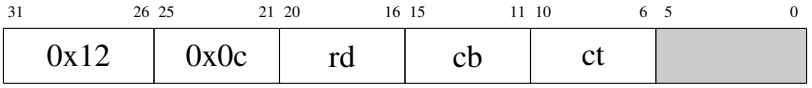
- *cb* or *cd* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and the corresponding bit in **PCC**.**perms** is not set.
- *cb*.**tag** is not set.
- *cb*.**u** is not set.

- $rt > cb$.**length**.

CToPtr: Capability to Pointer

Format

CToPtr rd, cb, ct



Description

If *cb* has its tag bit set but is of zero length (i.e. it is either the NULL capability, or some other capability of zero length), then *rd* is set to zero.

If the memory addresses *cb*.**base** ... *cb*.**base** + *cb*.**length** − 1 are contained within *ct*.**base** ... *ct*.**base** + *ct*.**length** − 1, then *rd* is set to the offset *cb*.**base** − *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()
else if register_inaccessible(ct) then
    raise_c2_exception()
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if not ct.tag then
    raise_c2_exception(exceptionTag, ct)
else if cb.length = 0 then
    rd ← 0
else if cb.base < ct.base then
    raise_c2_exception(exceptionLength, ct)
else if cb.base + cb.length > ct.base + ct.length then
    raise_c2_exception(exceptionLength, ct)
else
    rd ← cb.base − ct.base
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 the corresponding bit in **PCC**.**perms** is not set.
- *cb*.**tag** is not set.

- *ct*.**tag** is not set.
- *cb*.**length** $\neq 0$ **and** *cb*.**base** $<$ *ct*.**base**
- *cb*.**length** $\neq 0$ **and** *cb*.**base** $+$ *cb*.**length** $>$ *ct*.**base** $+$ *ct*.**length**

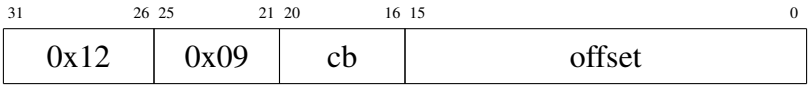
Notes

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

CBTU: Branch if tag is unset

Format (6?)

CBTU cb, offset



Description

Sets the **PC** to **PC+***offset*, 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()
else if not cb.tag then
    if PC + offset + 4 > PCC.length then
        raise_c2_exception(exceptionLength, 0xff)
    else
        execute_delay_slot()
        PC ← PC + offset
    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 the corresponding bit in **PCC**.**perms** is not set.
- **PC+***offset*+4 is greater than **PCC**.**length** and *cb.tag* is not set.

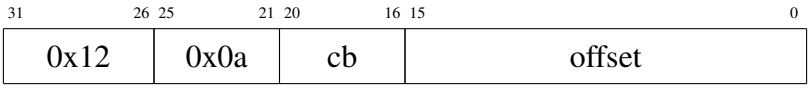
Notes

1. 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.

CBTS: Branch if tag is set

Format (6?)

CBTS cb, offset



Description

Sets the **PC** to **PC**+*offset*, 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()
else if cb.tag then
    if PC + offset + 4 > PCC.length then
        raise_c2_exception(exceptionLength, 0xff)
    else
        execute_delay_slot()
        PC ← PC + offset
    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 the corresponding bit in **PCC**.**perms** is not set.
- **RPC**+*offset*+4 is greater than **PCC**.**length** and *cb.tag* is set.

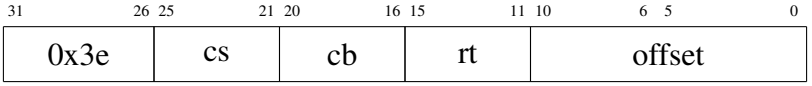
Notes

1. 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.

CSC: Store Capability Register

Format (3)

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** + general-purpose register *rt*, and the bit in the tag memory associated with *cb.***base** + *rt* is set. Capability register *cb* must contain a capability that grants permission to store capabilities. The virtual address *cb.***base** + *rt* must be 32-byte word aligned.

The capability is stored in memory in the format described in Figure 3.1. **base**, **length** and **otype**/**eaddr** 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 **u** bit, the next significant bit is the *Non_Ephemeral* bit, the next is *Permit_Execute* and so on.

Pseudocode

```
if register_inaccessible(cs) then
    raise_c2_exception()
else if register_inaccessible(cb) then
    raise_c2_exception()
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if not cb.unsealed 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_Ephemeral_Capability and not cs.perms.Non_Ephemeral
then
    raise_c2_exception(exceptionPermitStoreEphemeralCapability, cb)
end if
addr ← cb.base + rt + offset
if rt + offset + 32 > cb.length then
    raise_c2_exception(exceptionLength, cb)
else if rt + offset < 0 then
    raise_c2_exception(exceptionLength, cb)
else if align_of(addr) < 32 then
    raise_exception(exceptionAdES)
else
    mem[addr] ← cs
```

tags[toTag(addr)] ← cs.tag
end if

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *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.
- The virtual address *rt* + offset + 32 is greater than *cb*.**length**.
- *cb*.**perms**.*Permit_Store_Capability* is not set.
- *cb*.**u** is not set.
- *cb*.**perms**.*Permit_Store_Ephemeral* is not set and *cs*.**perms**.*Non_Ephemeral* is not set.

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

- The virtual address *cb*.**base** + *rt* + *offset* is not 32-byte word aligned.

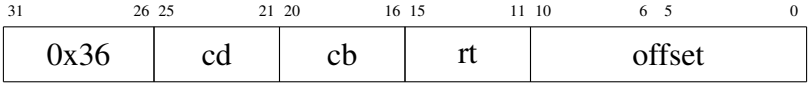
Notes

- If the address alignment check fails and one of the security checks fails, a coprocessor 2 exception (and not an address error exception) is raised. The priority of the exceptions is security-critical, because otherwise a malicious program could use the type of the exception that is raised to test the bottom bits of a register that it is not permitted to access.
- *offset* is interpreted as a signed integer.
- 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.

CLC: Load Capability Register

Format (1)

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** + general-purpose register *rt*. Capability register *cb* must contain a capability that grants permission to load capabilities. The virtual address *cb.***base** + *rt* must be 32-byte word aligned.

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

Pseudocode

```
if register_inaccessible(cd) then
    raise_c2_exception()
else if register_inaccessible(cb) then
    raise_c2_exception()
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if not cb.unsealed then
    raise_c2_exception(exceptionSealed, cb)
else if not cb.perms.Permit_Load_Capability then
    raise_c2_exception(exceptionPermitLoadCapability, cb)
end if
addr ← cb.base + rt + offset
if rt + offset + 32 > cb.length then
    raise_c2_exception(exceptionLength, cb)
else if rt + offset < 0 then
    raise_c2_exception(exceptionLength, cb)
else if align_of(addr) < 32 then
    raise_exception(exceptionAdEL)
else
    cd ← mem[addr]
    cd.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.***perms**.*Permit_Load_Capability* is not set.
- *cb.***u** is not set.
- *rt* + *offset* + 32 is greater than *cb.***length**.
- *rt* + *offset* < 0.

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

1. The virtual address *cb.***base** + *rt* is not 32-byte word aligned.

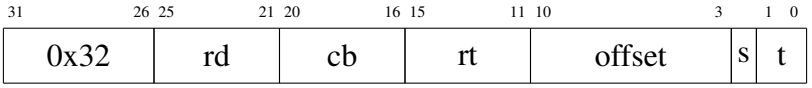
Notes

- 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.

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** + *rt* + *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 both to 0.

Pseudocode

```
if register_inaccessible(cb) then
    raise_c2_exception()
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if not cb.unsealed then
    raise_c2_exception(exceptionSealed, cb)
else if not cb.perms.Permit_Load then
    raise_c2_exception(exceptionPermitLoad, 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
addr ← cb.base + rt + offset
if offset + rt + size > cb.length then
    raise_c2_exception(exceptionLength, cb)
else if offset + rt < 0 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
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and the corresponding bit in **PCC**.**perms** is not set.
- Immediate *offset* + *rt* + *size* is greater than *cb*.**length**. **Check depends on the size of the data loaded.**
- *cb*.**perms**.*Permit_Load* is not set.
- *cb*.**u** is not set.

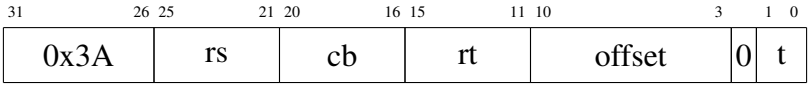
Notes

- This instruction reuses the opcode from the Load Word to Coprocessor 2 (LWC2) instruction in the MIPS Specification.
- *rt* and *offset* are treated as signed integers.
- The result of the addition does not wrap around (i.e., an exception is raised if cb.**base**+*rt*+*offset* is less than zero, or greater than *maxaddr*).

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** + *rt* + *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()
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if not cb.unsealed 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
addr ← cb.base + rt + offset
if rt + offset + size > cb.length then
    raise_c2_exception(exceptionLength, cb)
else if rt + offset < 0 then
    raise_c2_exception(exceptionLength, cb)
else if align_of(addr) < size then
    raise_exception(exceptionAdES)
else
    mem[addr:addr + size − 1] ← rd[0:size − 1]
    tags[toTag(addr)] ← false
end if
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb*.**tag** is not set.
- *cb* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and the corresponding bit in **PCC**.**perms** is not set.
- Immediate *offset* + *size* is greater than *cb*.**length**.
- *cb*.**perms**.*Permit_Store* is not set.
- *cb*.**u** is not set.

Notes

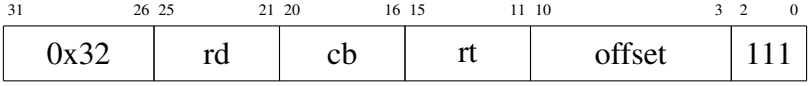
- This instruction reuses the opcode from the Store Word from Coprocessor 2 (SWC2) instruction in the MIPS Specification.
- If *t* is 3 and *e* is 1, then the instruction is CSCD (Store Conditional Doubleword via Capability).
- *rt* and *offset* are treated as signed integers.

- The result of the addition does not wrap around (i.e., an exception is raised if cb.**base**+*rt*+*offset* is less than zero, or greater than *maxaddr*).

CLLD: Load Linked Doubleword via Capability

Format

CLLD rd, rt, offset(cb)
CLLDR rd, rt(cb)
CLLDI rd, offset(cb)



Description

CLLD and CSCD are used to implement safe access to data shared between different threads. The typical usage is that CLLD is followed (an arbitrary number of instructions later) by CSCD to the same address; the CSCD will only succeed if there have been no context switches since the preceding CLLD.

The exact conditions under which CSCD 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.

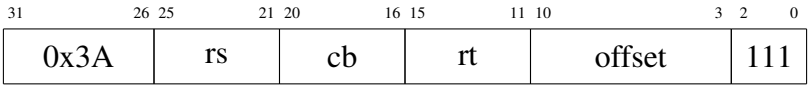
Pseudocode

```
addr ← cb.base + rt + offset
if register_inaccessible(cb) then
    raise_c2_exception()
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if not cb.unsealed then
    raise_c2_exception(exceptionSealed, cb)
else if not cb.perms.Permit_Load then
    raise_c2_exception(exceptionPermitLoad, cb)
else if offset + rt + 8 > cb.length then
    raise_c2_exception(exceptionLength, cb)
else if offset + rt < 0 then
    raise_c2_exception(exceptionLength, cb)
else if align_of(addr) < 8 then
    raise_c2_exception(exceptionAdEL)
else
    rd ← mem[addr:addr+7]
    linkedFlag ← true
end if
```

CSCD: Store Conditional Doubleword via Capability

Format

CSCD rs, rt, offset(cb)
CSCDR rs, rt(cb)
CSCRI rs, offset(cb)



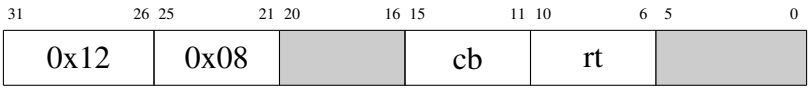
Pseudocode

```
addr ← cb.base + rt + offset
if register_inaccessible(cb) then
    raise_c2_exception()
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if not cb.unsealed then
    raise_c2_exception(exceptionSealed, cb)
else if not cb.perms.Permit_Store then
    raise_c2_exception(exceptionPermitStore, cb)
else if rt + offset + 32 > cb.length then
    raise_c2_exception(exceptionLength, cb)
else if rt + offset < 0 then
    raise_c2_exception(exceptionLength, cb)
else if align_of(addr) < 32 then
    raise_exception(AdES)
else if not linkedFlag then
    rs ← 0
else
    mem[addr:addr+7] ← rs
    tags[toTag(addr)] ← false
    rs ← 1
end if
```

CJR: Jump Capability Register

Format (3)

CJR rt(cb)



Description

PCC is loaded from *cb*, and **PC** is loaded from *rt*. (As the program counter is relative to **PCC**, this instruction will branch to the address *cb*.**base** + *rt*).

Pseudocode

```
if register_inaccessible(cb) then
    raise_c2_exception()
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if not cb.unsealed then
    raise_c2_exception(exceptionSealed, cb)
else if not cb.perms.Permit_Execute then
    raise_c2_exception(exceptionPermitExecute, cb)
else if not cb.perms.Non_Ephemeral then
    raise_c2_exception(exceptionNonEphemeral, cb)
end if
if rt + 4 > cb.length then
    raise_c2_exception(exceptionLength, cb)
else if align_of(cb.base + rt) < 4 then
    raise_exception(exceptionAdEL)
else
    execute_delay_slot()
    PC ← rt
    PCC ← 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 the corresponding bit in **PCC**.**perms** is not set.
- *cb*.**tag** is not set.
- *cb.***u** is not set.
- *cb.***perms**.*Permit_Execute* is not set.

- *cb.***perms**.*Non_Ephemeral* is not set.
- Register *rt* + 4 is greater than *cb.***length**.

An address error exception is raised if:

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

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

CJALR: Jump and Link Capability Register

Format (3)

CJALR rt(cb)

Description

The current **PCC** is saved in the return capability register (capability register number 24) and **PC** is saved in general purpose register *ra* (register number 31). **PCC** is then loaded from capability register *cb*, and **PC** is loaded from *rt*. As **PC** is interpreted relative to **PCC** during instruction fetch, this instruction will jump to the code at address *cb*.**base**+ *rt*.

Pseudocode

```
if register_inaccessible(cb) then
    raise_c2_exception()
else if not cb.tag then
    raise_c2_exception(exceptionTag, cb)
else if not cb.unsealed then
    raise_c2_exception(exceptionSealed, cb)
else if not cb.perms.Permit_Execute then
    raise_c2_exception(exceptionPermitExecute, cb)
else if not cb.perms.Non_Ephemeral then
    raise_c2_exception()
end if
if rt + 4 > cb.length then
    raise_c2_exception(exceptionLength, cb)
else if align_of(cb.base + rt) < 4 then
    raise_exception(exceptionAdEL)
else
    execute_delay_slot()
    R31 ← PC
    RCC ← PCC
    PC ← rt
    PCC ← cb
end if
```

Exceptions

A coprocessor 2 exception will be raised if:

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

- *rt* + 4 is greater than *cb.***length**.
- *cb.***u** is not set.
- *cb.***tag** is not set.
- *cb.***perms**.*Non_Ephemeral* is not set.

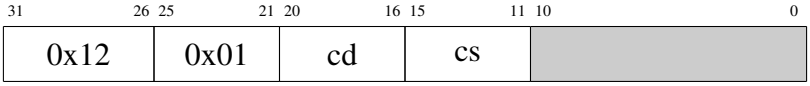
An address error exception will be raised if

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

CSealCode: Seal an Executable Capability

Format (5)

CSealCode cd, cs



Description

If

- capability register *cs* is unsealed;
- *cs*.**perms**.*Permit_Seal* is set;
- and *cs*.**perms**.*Permit_Execute* is set;

then

- *cd*.**u** is cleared.
- the other fields of *cd* (including **otype**/**eaddr**) are copied from *cs*.

Pseudocode

```
if register_inaccessible(cd) then
    raise_c2_exception()
else if register_inaccessible(cs) then
    raise_c2_exception()
else if not cs.tag then
    raise_c2_exception(exceptionTag, cs)
else if not cs.unsealed then
    raise_c2_exception(exceptionSealed, cs)
else if not cs.perms.Permit_Seal then
    raise_c2_exception(exceptionPermitSeal, cs)
else if not cs.perms.Permit_Execute then
    raise_c2_exception(exceptionPermitExecute, cs)
else
    cd ← cs
    cd.u ← false
end if
```

Exceptions

A coprocessor 2 exception is raised if:

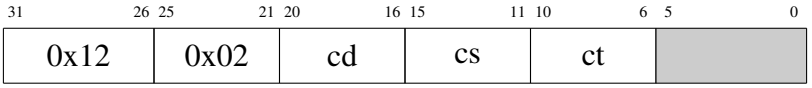
- *cd* or *cs* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and the corresponding bit in **PCC**.**perms** is not set.

- *cs*.**tag** is not set.
- *cs*.**u** is not set.
- *cs*.**perms**.*Permit_Seal* is not set.
- *cs*.**perms**.*Permit_Execute* is not set.

CSealData: Seal a Data Capability

Format (5)

CSealData cd, cs, ct



Description

If

- capability register *ct* contains an unsealed capability;
- *ct*.**perms**.*Permit_Seal* is set;
- capability register *cs* contains an unsealed capability;
- and *cs*.**perms**.*Permit_Execute* is not set

then

- *cd*.**otype**/**eaddr** is set to *ct*.**otype**/**eaddr**;
- *cd*.**u** is cleared;
- and the other fields of *cd* are copied from *cs*.

Pseudocode

```
if register_inaccessible(cd) then
    raise_c2_exception()
else if register_inaccessible(cs) then
    raise_c2_exception()
else if register_inaccessible(ct) then
    raise_c2_exception()
else if not ct.tag then
    raise_c2_exception(exceptionTag, ct)
else if not cs.tag then
    raise_c2_exception(exceptionTag, cs)
else if not ct.unsealed then
    raise_c2_exception(exceptionSealed, ct)
else if not cs.unsealed then
    raise_c2_exception(exceptionSealed, cs)
else if not ct.perms.Permit_Seal then
    raise_c2_exception(exceptionPermitSeal, ct)
else if cs.perms.Permit_Execute then
    raise_c2_exception(exceptionPermitExecute, cs)
else
```

cd ← cs
cd.u ← **false**
cd.otype ← ct.otype
end if

Exceptions

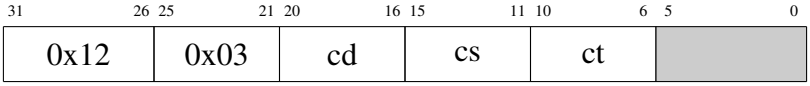
A coprocessor 2 exception is raised if:

- *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.
- *ct*.**tag** is not set.
- *ct*.**u** is not set.
- *ct*.**perms**.*Permit_Seal* is not set.
- *cs*.**tag** is not set.
- *cs*.**u** is not set.
- *cs*.**perms**.*Permit_Execute* is set.

CUnseal: Unseal a sealed capability

Format (5)

CUnseal cd, cs, ct



Description

The sealed capability in cs is unsealed with ct and the result placed in cd. The not-ephemeral bit of cd is the AND of the ephemeral bits of cs and ct. ct must be unsealed, have Permit_Seal permission, and have the same **otype**/**eaddr** as cs.

Pseudocode

if register_inaccessible(cd) **then**
    raise_c2_exception()
else if register_inaccessible(cs) **then**
    raise_c2_exception()
else if register_inaccessible(ct) **then**
    raise_c2_exception()
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.unsealed **then**
    raise_c2_exception(exceptionSealed, cs)
else if not ct.unsealed **then**
    raise_c2_exception(exceptionSealed, ct)
else if ct.otype $\neq$ cs.otype **then**
    raise_c2_exception(exceptionType, ct)
else if not ct.perms.Permit_Seal **then**
    raise_c2_exception(exceptionPermitSeal, ct)
else
    cd $\leftarrow$ cs
    cd.u $\leftarrow$ **true**
    cd.perms.Non_Ephemeral $\leftarrow$ cs.perms.Non_Ephemeral **and** ct.perms.Non_Ephemeral
end if

Exceptions

A coprocessor 2 exception is raised if:

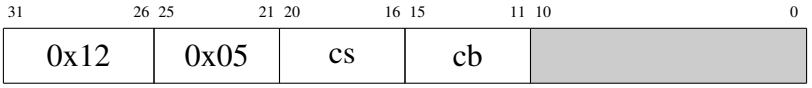
- *cd*, *cs*, or *ct* is one of the reserved registers (**KR1C**, **KR2C**, **KCC**, **KDC** or **EPCC**) and the corresponding bit in **PCC**.**perms** is not set.
- *ct*.**tag** is not set.

- *ct*.**u** is not set.
- *ct*.**perms**.*Permit_Seal* is not set.
- *ct*.**otype**/**eaddr** $\neq$ *cs*.**otype**/**eaddr**.
- *cs*.**tag** is not set.
- *cs*.**u** is set.

CCall: Call into a new security domain

Format (3)

CCall cs, cb



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 which 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.

In the current implementation of CHERI, **CCall** is implemented by the hardware that raises an exception, and the rest of the instruction's behavior is to be implemented in software by the trap 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 **CCall** being implemented in software.

1. The program counter (**PC**) + 4, the stack pointer (**SP**), and 16 bytes of padding are pushed onto the trusted system stack. (The padding exists to keep the trusted system stack aligned on a 32-byte boundary).
2. **PCC** is pushed onto the trusted system stack.
3. **IDC** is pushed onto the trusted system stack.
4. *cs* is unsealed and the result placed in **PCC**.
5. *cb* is unsealed and the result placed in **IDC**.
6. The program counter is set to *cs*.**otype**/**eaddr** − *cs*.**base**. (i.e. control branches to virtual address *cs*.**otype**/**eaddr**, but because the program counter is relative to **PCC**.**base**, this must be subtracted).

Pseudocode (hardware)

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

Pseudocode (software)

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

else if not cs.tag **then**
    raise_c2_exception(exceptionTag, cs)
else if not cb.tag **then**
    raise_c2_exception(exceptionTag, cb)
else if cs.unsealed **then**
    raise_c2_exception(exceptionSealed, cs)
else if cb.unsealed **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_Seal **then**
    raise_c2_exception(exceptionPermitSeal, cs)
else if not cs.perms.Permit_Execute **then**
    raise_c2_exception(exceptionPermitExecute, cs)
else if cs.otype $<$ cs.base **then**
    raise_c2_exception(exceptionLength, cs)
else if cs.otype $>$ cs.base + cs.length $-$ 1 **then**
    raise_c2_exception(exceptionLength, cs)
else
    TSS $\leftarrow$ TSS $-$ 32
    mem[TSS .. TSS + 7] $\leftarrow$ PC + 4
    mem[TSS + 8 .. TSS + 15] $\leftarrow$ SP
    tags[toTag(TSS)] $\leftarrow$ **false**
    TSS $\leftarrow$ TSS $-$ 32
    mem[TSS .. TSS + 31] $\leftarrow$ PCC
    tags[toTag(TSS)] $\leftarrow$ PCC.tag
    TSS $\leftarrow$ TSS $-$ 32
    mem[TSS .. TSS + 31] $\leftarrow$ IDC
    tags[toTag(TSS)] $\leftarrow$ TSS.tag
    PCC $\leftarrow$ cs
    PCC.unsealed $\leftarrow$ **true**
    IDC $\leftarrow$ cb
    IDC.unsealed $\leftarrow$ **true**
    PC $\leftarrow$ cs.otype $-$ cs.base
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 the corresponding bit in **PCC**.**perms** is not set.

- *cs*.**u** is set.
- *cb*.**u** is set.
- *cs*.**otype**/**eaddr** $\neq$ *cb*.**otype**/**eaddr**
- *cs*.**perms**.*Permit_Execute* is not set.
- *cs*.**perms**.*Permit_Seal* is not set.
- *cs*.**otype**/**eaddr** $<$ *cs*.**base**
- *cs*.**otype**/**eaddr** $>$ *cs*.**base** + *cs*.**length** - 1
- 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**).

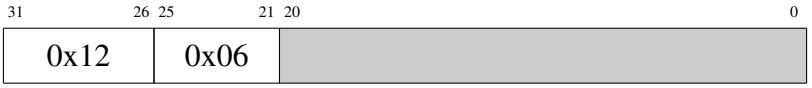
Notes

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**.

CReturn: Return to the previous security domain

Format (3)

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.
3. The program counter (**PC**) and stack pointer (**SP**) are 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 software.

Pseudocode (hardware)

```
raise_c2_exception(exceptionReturn, 0xff)
```

Pseudocode (software)

```
IDC ← mem[TSS .. TSS + 31]
IDC.tag ← tags[toTag(TSS)]
TSS ← TSS + 32
PCC ← mem[TSS .. TSS + 31]
PCC.tag ← tags[toTag(TSS)]
TSS ← TSS + 64
PC ← mem[TSS − 32 .. TSS − 25]
SP ← mem[TSS − 24 .. TSS − 17]
```

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**.

4.2 Assembler Pseudo-Instructions

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

4.2.1 Capability Move

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

```
# The following are equivalent:
  CMove $c1, $c2
  CIncBase  $c1, $c2, $zero
```

4.2.2 Get/Set Default Capability

`CGetDefault` and `CSetDefault` get and set the capability register that is used 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.

```
# The following are equivalent:
  CGetDefault $c1
  CIncBase $c1, $c0, $zero
```

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

4.2.3 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:

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

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

The store operations are similar:

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

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

Chapter 5

Design Rationale

During the design of CHERI, we considered many different capability architectures and design approaches. This chapter describes the design choices that were made, briefly outlines some possible alternatives, and provides a rationale for the choices that were made.

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 are within virtual address spaces, imposing an ordering in which capability protections are evaluated before virtual-memory protections; this in turn had impliations 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, which we used modeled on similar conventional RISC instructions.

- Incremental deployment within programs meant that not all pointers would immediately be converted to capabilities, implying that both might coexist in the same memory; also, there was a strong desire to embed capabilities within data structures, rather than store them in separate segments, requiring fine-granularity tagging.

- Incremental deployment and compatibility with the UNIX model implied retaining the general-purpose memory management unit (MMU) pretty much as currently designed, including support for variable page size, TLB layout, etc. The MIPS ISA describes a software-managed TLB rather than hardware page-table walking as is present in most other ISAs; this is not fundamental to our approach, and either model would work.

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 that management of the two register files will remain largely the same, stack spilling will behave 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 if implementation and compatibility, rather than 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 more tightly coupled specific capability registers 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, making 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 within capabilities revisits both of these ideas.

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 `CSetLen`; setting an upper bound would require a 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. 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) provide a useful middle ground.

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 enountered 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 does reemphasize the importance of careful investment of encoding bits for instructions.

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. (Although, 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 on to 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 other, more direct, ways for a program to discover the contents of a capability it owns, so this is not 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, but 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 `CSetBase` – an important reason not to limit capability field retrieval instructions to tagged values.

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.

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

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.

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 permissions 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).

Capabilities Do Not 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.

In contrast, the base of a CHERI capability can be incremented (via `CIncBase`) but not decremented. When CHERI capabilities are used from C, a pointer with type attribute `__capability` can be incremented but not decremented.

An alternative architecture would have included a "cursor" field within a capability, that could be both incremented and decremented without changing **base**. This would have given `__capability` variables semantics that were closer to ordinary C pointers, at the expense of making capabilities take up more space in memory, with a reduction in performance as a result.

In comparison, the CCured language includes both `FSEQ` and `SEQ` pointers. CHERI capabilities are analogous to CCured's `FSEQ` pointers. Programming languages that need semantics similar to to CCured's `SEQ` can be implemented on CHERI by compiling them as the pair of a CHERI capability and an integer that acts as a cursor into the array.

We are now actively exploring variations on the CHERI ISA that do implement tags in order to improve source-code level compatibility. This turns out to be particularly critical for packet-parsing code that will frequently perform a series of pointer operations and then validate the resulting pointer against a bound (often incorrectly); with CHERI as defined, invalid pointers cannot be constructed and such operations will generate an exception, or require compiler support for coupling pointers with capabilities. As there is substantial space in the CHERI capability – in particular, an unused entry address/type field intended for use with object capabilities – introducing a cursor is relative straightforward, although obviously must be done with care. The model we have prototyped allows the cursor to float freely outside of the bounds specified by the capability base and length, generating an exception only if it is used to load and store from disallowed addresses, tracking an invalid dereference. Separate instructions get and set the cursor, allowing current CHERI code to work unmodified, with `CSetBase` also adjusting the cursor.

NULL Has 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 set and the **base** and **length** fields are zero; effectively, NULL is an array of length zero.

An alternative representation we have could have chosen for NULL would have been with the **tag** bit unset, and zero in the **base** field; effectively, NULL would be the integer zero.

Many of the CHERI instructions are agnostic as to which of these two conventions for NULL is employed, but the `CFtomPtr` and `CToPtr` are aware of the convention.

One advantage of having NULL's **tag** bit unset would have been that it would be possible for code to conditionally branch on a capability being NULL by using the `CBTS` or `CBTU` instruction.

One advantage of the convention we have chosen is that dynamically typed languages can distinguish between a pointer to a valid object, NULL, a non-zero integer and the integer zero. For example, a dynamically typed language could use capabilities with **tag** unset for small integers (including zero), capabilities with **tag** set pointing to an arbitrary precision integer for large integers, and the NULL capability (**tag** unset, **length** zero) for an optional parameter being absent.

Permission Bits Determine the Type of a Capability

In CHERI, a capability's permission bits together with the **u** bit determine what kind of capability it is. A capability for a region of memory has **u** and *Permit_Load* and/or *Permit_Store* set; a capability for an object has **u** unset and *Permit_Execute* unset; a capability to call a protected subsystem (a "call gate") has **u** unset and *Permit_Execute* set; a capability that allows the

owner to create objects whose type identifier (**otype**/**eaddr**) falls within a range has **u** unset and *Permit_Set_Type* 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 **u**). 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.

Object Types are Addresses

In CHERI, the **otype**/**eaddr** field serves both as a unique identifier for an object type and as the address of the executable code that implements the methods on that object type.

An alternative architecture would have been to include separate fields within a capability for the object type id and for the address of the code that implements the object's methods.

The architecture we have chosen allows us to keep the size of capabilities small (which is important for performance) at the cost of some conceptual confusion caused by these multiple uses of the **otype**/**eaddr** field.

Treating the set of object type identifiers as being the same as the set of memory addresses has the additional advantage that it simplifies assigning type identifiers to protected subsystem: 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 *Permit_Set_Type* set for a range of memory addresses, and they are then able to use types within that range. 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 there was a type manager and it 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).

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 (**u** clear) 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**/**eaddr** field of **PCC**.

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.

Chapter 6

CHERI in Programming Languages and Operating Systems

We capture some of our early thoughts on the topic of use of CHERI instructions in programming languages and operating systems. The goal of our software work is to test several fundamental hypotheses underlying the CHERI architecture:

- That a hardware capability model provides superior performance when large numbers of protection domains are required.
- That protection domains within address spaces offer improved programmability and debuggability for compartmentalized TCB components.
- That there are (fairly) natural mappings from higher-level language pointer and reference models into memory-capability semantics.
- That capability adaptation of common TCB components offers dramatically improved robustness and security.
- That a hardware capability model and MMU-based virtual addressing can not only coexist, but also facilitate an adoption of capability approaches – offering both an incremental adoption path with immediate security benefits and a long-term vision for software security improvement.
- That a fully virtualizable per-address-space capability system is feasible and practical, while allowing use of capability models within individual hierarchical rings and address spaces.

To this end, we are developing a significant software stack that will utilize the CHERI feature set, implementing and exercising various aspects of the hybrid capability model.

6.1 Development Plan and Status

At the time of writing, we are roughly three and one-half years into a five-year research project in hardware and software security. We have successfully prototyped a fully pipelined 64-bit CPU implementing the 64-bit MIPS and CHERI ISAs; detailed information on this prototype

can be found in the accompanying *BERI Hardware Reference*. The following sections document our recent accomplishments and continuing strategy for exploring the software implications of the CHERI architecture.

6.2 Open-Source Foundations

During the initial bring-up phase of our prototype CHERI CPU, CHERI's support for incremental adoption has proven invaluable: we will be able to rely on current open source boot loaders, operating systems, programming languages, compilers, debuggers, and applications, to selectively deploy capability features in the most critical software foundations and the most vulnerable services.

6.3 Current Software Implementation

We have extended existing 64-bit MIPS versions of FreeBSD operating system, GNU assembler, and LLVM/Clang compiler suite to support CHERI ISA features.

6.4 CheriBSD

We have extended the existing 64-bit MIPS port of FreeBSD to support a range of Altera/Terasic hardware peripherals, and our CHERI ISA extensions. CheriBSD maintains a coprocessor 2 context for each user thread; it implements CCall/CReturn exception handlers, and recovery paths for when a sandbox triggers a hardware exception (e.g., due to an invalid memory reference). CheriBSD includes a new libcheri which implements a sandbox API and a growing set of system services that may be delegated to sandboxed code.

6.4.1 Extended GNU Assembler (gas)

We have extended the GNU assembler (gas) to support the CHERI ISA, by allowing assembly files, and an inline assembler from C to make use of CHERI instructions. Tools such as **objdump** are also able to interpret CHERI instructions. We have not yet extended the linker to support new CHERI-related linkage types; instead, we rely on hybrid behavior to implement programs for the time being.

6.5 Extended LLVM/Clang

LLVM is a framework for implementing compilers that comprises a well-defined intermediate representation (IR), a set of APIs for generating this representation, optimization passes for transforming it, and back ends for generating native code. We have extended the MIPS back end in LLVM to provide support for capability instructions.

We reserve address space 200[1] for capability pointers. Any pointer to address space 200 is assumed to be a capability. We also add explicit integer to pointer and pointer to integer patterns in the back end. These are required because all existing LLVM back ends regard pointers and integers as interchangeable.

The LLVM back end also provides an alternative assembler. This currently lacks some MIPS instructions, so is not yet a replacement for the GNU assembler, but does provide support for both inline assembly and for stand-alone assembly files.

The modifications to LLVM are intended to make experimentation with programming languages easier. Any language front end that can generate LLVM IR can be modified to support capabilities and we are free to experiment with new languages and modifications to others.

In addition to the support for capabilities as pointers, we also provide a number of intrinsics that map closely to instructions. The example below uses the `llvm.cheri.set.cap.length` intrinsic, which sets the length of a capability. This example shows the LLVM IR for a simple function that wraps the C standard malloc() in one that returns a capability that will enforce the length.

```
1  define i8 addrspace(200)* @cmalloc(i64 %s) nounwind {
2  entry:
3    ; Call malloc()
4    %call = tail call i8* @malloc(i64 %s) nounwind
5    ; Convert the C0-relative pointer to a capability
6    %0 = ptrtoint i8* %call to i64
7    %1 = inttoptr i64 %0 to i8 addrspace(200)*
8    ; CSetLen
9    %2 = tail call i8 addrspace(200)* @llvm.cheri.set.cap.length(i8
       addrspace(200)* %1, i64 %s)
10   ret i8 addrspace(200)* %2
11 }
```

Clang is a front end for LLVM that generates LLVM IR from C-family languages (C, C++, Objective-C, and Objective-C++). Our first work on language extensions involves providing capability support to C. Programmers can annotate pointers as being capabilities, which triggers CHERI rather than MIPS code generation for any resulting memory accesses. C-language types such as `const` now perform dynamic permission refinement. We have an experimental version of the same compiler code that now supports stack access via capability.

Objective-C provides a late-bound object oriented model on top of C that makes it an interesting test ground for experimentation. Combined with the MIT-licensed GNUstep Objective-C runtime, we can experiment with adding language features to Objective-C based on our extensions to C.

6.5.1 Extended CHERI Unit-Test Suite

Using the CHERI assembler, we have extended our existing MIPS ISA test suite to exercize various aspects of the CHERI ISA. The current test suite validates the behavior of memory

[1]The number 200 is subject to change and should not be relied on. Address spaces under 256 are intended to be reserved for architecture-agnostic uses, so we may either move this to a higher number, or retain a low number as a general fat-pointer address space in LLVM IR.

capabilities and capability exceptions, including monotonic decrease in rights using capability manipulation instructions. More detailed implementation status for the CHERI hardware prototype may be found in the *BERI Hardware Reference*.

6.6 Future Plans

Between 2010 and 2014, we completed basic prototyping of the CHERI hardware and software platform. We are now considering potential future directions for further hardware and software experimentation, including bindings to higher-level languages such as Objective-C, OCaml, and Java. We are also exploring how adding capability support to LLDB and the LLVM debugger would allow us to develop capability-aware programs, as well as considering security implications for designing and implementing debuggers.

Chapter 7

Future Directions

The CTSRD project, of which CHERI is just one element, has now been in progress for three and a half years. Our focuses to date have been in several areas:

1. Design the CHERI instruction set architecture based on a hybrid object-capability model. As part of this work, develop a PVS formal model of the ISA, and analyze properties about program expressivity.

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 a second version of the ISA intended to better support automatic compilation, which is now implemented.

3. Prototype, test, and refine CHERI ISA extensions, which are incorporated via a new capability coprocessor, CP2.

4. Port the FreeBSD operating system first to a capability-free version of CHERI, known as BERI. This is known as FreeBSD/BERI.

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.

6. Adapt the Clang/LLVM compiler suite to be able to generate CHERI ISA instructions as directed by C-language annotations.

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 CHERI2 on the NetFGPA 10G board.

9. Develop techniques for translating Bluespec hardware designs into PVS representations so that they can be used for formal analysis purposes. The SRI PVS tool suite has been embedded in the BluespecVerilog compiler chain to enable formal verification, model checking (SAL), and SMT solving (Yices) inline with the compilation.

We have made a strong beginning, but clearly there is much to do. From this vantage point, we see a number of tasks ahead, which we detail in the next few sections.

7.1 An Open-Source Research Processor

One of our goals for the CHERI processor is to produce a reference Bluespec processor implementation, which can then be used as a foundation not only for CHERI, but also for other research projects in 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 a considerable amount of time preparing CHERI for open sourcing, including enhancing our test suite, updating documentation, and preparing a new open-source license intended for hardware-software projects (derived from the Apache software license).

7.2 Formal Methods for Bluespec

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 Bluespec designs 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 is to link formal models of the hardware itself with the ISA specification and software compiled to that ISA. 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.

7.3 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.

7.4 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 adding memory tag support to paging and swapping, enhancing TLB support to include CHERI-related flags, and continuing to adapt userspace OS components, such as the system library and runtime linker, to use CHERI capability features. This work depends heavily 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 2; 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.

7.5 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.

Bibliography

[1] 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.

[2] W. B. Ackerman and W. W. Plummer. An implementation of a multiprocessing computer system. In *SOSP '67: Proceedings of the First ACM Symposium on Operating System Principles*, pages 5.1–5.10, New York, NY, USA, 1967. ACM.

[3] J. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, October 1972. (Two volumes).

[4] G. R. Andrews. Partitions and principles for secure operating systems. Technical report, Cornell University, Ithaca, NY, USA, 1975.

[5] Apple Inc. Mac OS X Snow Leopard. `http://www.apple.com/macosx/`, 2010.

[6] 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.

[7] R. Bisbey II and D. Hollingworth. Protection Analysis: Project final report. Technical report, USC Information Sciences Institute (ISI), Marina Del Rey, California, 1978.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] P. J. Denning. Fault tolerant operating systems. *ACM Computing Surveys*, 8(4):359–389, 1976.

[16] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[17] 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.

[18] 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.

[19] 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`.

[20] 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.

[21] L. Gong. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, Reading, Massachusetts, 1999.

[22] 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.

[23] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[24] R. Graham. Protection in an information processing utility. *Communications of the ACM*, 11(5), May 1968.

[25] N. Hardy. KeyKOS architecture. *SIGOPS Operating Systems Review*, 19(4):8–25, 1985.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] P. A. Karger. Using registers to optimize cross-domain call performance. *SIGARCH Computer Architecture News*, 17(2):194–204, 1989.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] B. W. Lampson. Protection. *SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[36] 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.

[37] 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.

[38] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[39] J. Liedtke. On microkernel construction. In *SOSP'95: Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, Dec. 1995.

[40] 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.

[41] 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.

[42] 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.

[43] M. K. McKusick and G. V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004.

[44] 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.

[45] M. S. Miller. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006.

[46] 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`.

[47] J. H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.

[48] A. C. Myers and B. Liskov. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31:129–142, October 1997.

[49] 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.

[50] P. G. Neumann. Holistic systems. *ACM Software Engineering Notes*, 31(6):4–5, November 2006.

[51] 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.

[52] 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.

[53] P. G. Neumann and R. N. Watson. Capabilities Revisied: A Holistic Approach to Bottom-to-Top Assurance of Trustworthy Systems. In *Proceedings of the Fourth Annual Layered Assurance Workshop*, 2010.

[54] E. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, Massachusetts, 1972.

[55] 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.

[56] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, 2003.

[57] 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).

[58] 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.

[59] 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.

[60] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974.

[61] Ruby Users Group. Ruby Programming Language. `http://www.ruby-lang.org/`, October 2010.

[62] 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)).

[63] J. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[64] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] R. N. 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)*, 2012.

[75] R. N. M. Watson. New Approaches to Operating System Security Extensibility. Technical report, Ph.D. Thesis, University of Cambridge, Cambridge, UK, October 2010.

[76] 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.

[77] 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, phone +44 1223 763500, 2014.

[78] 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, phone +44 1223 763500, 2014.

[79] 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, phone +44 1223 763500, 2014.

[80] 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, phone +44 1223 763500, 2014.

[81] M. Wilkes and R. Needham. *The Cambridge CAP Computer and Its Operating System*. Elsevier North Holland, New York, 1979.

[82] 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)*, 2014.

[83] 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.

[84] W. Wulf, R. Levin, and S. Harbison. *Hydra/C.mmp: An Experimental Computer System*. McGraw-Hill, New York, 1981.

[85] 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.

[86] 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.

[87] 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.