CHERI
Capability Hardware Enhanced RISC Instructions Architecture Document

Version 1.7

This interim document is not released for public consumption

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

SRI International and the University of Cambridge[1] [2] [3]

March 16, 2013

[1]Sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

[2]Portions of this work were sponsored by the RCUK's Horizon Digital Economy Research Hub grant, EP/G065802/1.

[3]Portions of this work were sponsored by Google, Inc.

Contents

Abstract

This document describes a preliminary design for SRI International and the University of Cambridge's Capability Hardware Enhanced RISC Instructions (CHERI) instruction set architecture (ISA). The document is intended to capture our thoughts early in the research and development cycle.

CHERI is a *hybrid capability architecture* that combines new hardware primitives with the commodity MIPS 64-bit ISA in order to allow software to efficiently implement an *object-capability security model*. These extensions will support incrementally adoptable, high-performance, formally supported, and programmer-friendly underpinnings for fine-grained software decomposition and compartmentalisation motivated by the principle of least privilege. The CHERI approach addresses known performance and robustness gaps in commodity ISAs that prevent the adoption of more secure programming models centred around the principle of least privilege. To this end, CHERI blends traditional paged virtual memory with a per-address space capability model by adding capability registers, capability instructions, and tagged memory to the MIPS ISA via a new *capability coprocessor*.

CHERI's hybrid approach, adapted from the Capsicum security model, allows incremental adoption of capability-oriented software design: more robust software implementations can be deployed where they are most needed, while leaving less critical software largely unmodified. For example, we anticipate 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 in particularly high-risk software libraries, such as data compression, image processing, and video processing, 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 design, provides reference documentation for the CHERI instruction set architecture (ISA), potential memory models and their requirements, as well as documenting our current thinking on programming language and operating system integration. Our ongoing research includes two prototype CPUs employing the CHERI ISA, implemented as an FPGA soft core using the Bluespec hardware description language (HDL), for which we have developed a direct link between the implementation and formal methods applications. The prototypes and our formal methods work are described in separate documents, the *CHERI Platform Reference*, *CHERI User's Guide*, and *CHERI Formal Methods Report*.

Acknowledgments

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

Ross J. Anderson	Gregory Chadwick	Nirav Dave
Brooks Davis	Khilan Gudka	Wojciech Koszek
Steven Hand	Asif Khan	Myron King
Patrick Lincoln	Anil Madhavapeddy	Ilias Marinos
Andrew Moore	Will Morland	Alan Mujumdar
Robert Norton	Philip Paeps	John Rushby
Hassen Saidi	Muhammad Shahbaz	Stacey Son
Richard Uhler	Philip Withnall	

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	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, DARPA CRASH programme manager, who has offered both technical insight and support throughout this work.

Chapter 1

Introduction

The Capability Hardware Enhanced RISC Instructions (CHERI) architecure extends the commodity 64-bit MIPS instruction set architecture (ISA) with new security primitives in order to allow software to efficiently implement an *object-capability security model*. CHERI's extensions are intended to support incrementally adoptable, high-performance, formally supported, and programmer-friendly underpinnings for fine-grained *software compartmentalisation* motivated by the principle of least privilege. CHERI is a *hybrid capability architecture* in that gradual deployment of CHERI features in existing software is possible, offering a more gentle technology transfer path. CHERI has three central design goals, which at times come into tension:

1. We seek to dramatically improve security for the software Trusted Computing Bases (TCBs) by improving hardware support for *software compartmentalisation*. Compartmentalisation is the decomposition of software into isolated components in order to mitigate the effects of security vulnerabilities by applying the principle of least privilege. We believe that adoption of compartmentalisation 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 virtualisation from separation in order 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.

2. Simultaneously, we require a realistic technology transfer path 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 must sufficiently improve security that vendors of mobile and embedded devices demand its feature set from CPU companies (such as MIPS and ARM); they must also conform to vendor expectations for performance, power use, and compatibility in order to compete with less secure alternatives.

3. Finally, we wish to draw on formal methodology wherever feasible to improve our confi-

dence 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. This goal requires us to not only perform research into CPU and software design, but also to develop new formal methodology.

Our selection of RISC as a foundation lies in 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 perceived as having 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 designed CHERI as an additional coprocessor to the 64-bit MIPS ISA, but our approach is intended to easily support other similar ISAs, such as 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. It is also worth considering 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 such as found in software fault isolation techniques [66] or Google Native Client (NaCl) [75].

1.1 Motivation

The CHERI CPU architecture is a hardware foundation for principled, secure systems; its design builds on and extends decades of research into hardware and operating system security[1] However, some of the historic approaches CHERI incorporates (especially capability designs) 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 not? Several factors have motivated our decision to begin 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.

[1] Levy's *Capability-Based Computer Systems* provides a detailed history of segment- and capability-based designs through the early 1990s [35]. However, it leaves off just as the transition to microkernel-based capability systems such as Mach [1] and L4 [36], as well as capability-centric virtual machines such as the Java Virtual Machine [21], begins. Section 1.6 discuss historical influences on this work in greater detail.

- 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 in order to more easily express and implement various policies.
- Reaching the tail end of a "compatibility at all costs" trend in CPU design brought out by reaching 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 been seen, 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 rewritten applications conforming to new security models, programming languages, hardware architectures, and user input modalities.
- 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 ahead of their times.

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 the 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.

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.

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. These problems are compounded by the observation that the end-user systems are also an epicentre for multi-party security composition, where a single web browser or office suite manages state, user interface, and code execution for countless different security domains, and must simultaneously provide strong isolation and appropriate sharing. The results are not only a significant risk of compromise leading to financial loss or disruption of critical infrastructure, but frequent occurrence of such events.

Software vulnerabilities appear inevitable: even as the execution substrates improve, resisting attacks such as buffer overflows and integer vulnerabilities, logical errors will necessarily persist. Past research has shown that compartmentalising 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 vulnerabilities to accomplish their goals.

1.1.2 The Compartmentalisation Problem

The *compartmentalisation 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, compartmentalisation of TCB components (such as operating system kernels and central system services) has caused significant difficulty for software developers, limiting its applicability for large-scale, real-world applications, and leading to the abandonment of promising research such as 1990s *microkernel* projects. A recent resurgence of compartmentalisation, applied in userspace to applications such as OpenSSH [51] and Chromium [53], and most recently in our own Capsicum project [71], has been motivated by a critical security need, but seen success only at very coarse separation granularity due to the challenges involved.

On current 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 compartmentalised applications are difficult to design, write, debug, maintain, and extend, raising 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 it easy to isolate complete applications (or even multiple operating systems), but hard to efficiently and easily express 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 the quickly overflowed Translation Look-aside Buffer (TLB) lead to massive performance degradation. Above an extremely low threshold, performance overhead from context switching between security domains goes from extremely expensive to intolerable: each TLB entry is an access control list, with each object (page) requiring multiple TLB entries, one for each authorised security domain.

High-end server CPUs have in the low hundreds of TLB entries, and even recent network embedded devices reach only the low thousands; the TLB footprint of fine-grained, compartmentalised 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 namespace. Moving 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.

[2]Through extreme discipline, a programming model can be constructed that keeps mappings of multiple address spaces synchronised, 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 couple UNIX processes via shared memory associated with data types – a promising separation approach constrained by the realities of current CPU design [8].

1.2 The CHERI Design

The fundamental technical goal of the CHERI approach is to address the compartmentalisation problem for elements of the TCBs of security-critical workstations, servers, mobile devices, and embedded devices, while providing a reasonable assurance of correctness and a realistic technology transition path from existing hardware and software platforms.

To this end, we have designed CHERI as an instruction set architecture (ISA) extension to the widely used 64-bit MIPS ISA; we are also simultaneously considering the implications for the ARM ISA. CHERI adds the following features to the RISC CPU design via a new *capability coprocessor* that supports granular memory protection within address spaces:

- Capability registers, which can describe either a memory segment with limited rights, or an object capability that can be used with secure method invocation primitives to separate security domains.
- Capability instructions for creating, managing, constraining, loading and storing data, and invocation.
- Tagged memory, allowing capabilities to be safely loaded and stored in system memory without loss of integrity.

Wherever possible, CHERI systems make use of existing hardware designs: cache memory, system busses, commodity DRAM, and commodity peripheral devices such as NICs and display cards. We currently plan to focus on enforcement of CHERI security properties on applications running on a host processor; however, in future work, we hope to consider the effects of implementing CHERI in peripheral processors, such as those found in NICs or graphics cards.

1.2.1 A Hybrid Capability Architecture

Unlike in 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, decomposition can be applied selectively throughout existing software stacks, providing 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, and network applications such as web browsers and web servers.

CHERI addresses current limitations on compartmentalisation by extending virtual memory-based separation with hardware-enforced protection within address spaces. This approach restores a single address-space programming model for compartmentalised software, facilitating efficient, programmable, and robust separation through a 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, providing 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 multiply instantiate unmodified libraries, efficiently and easily gating 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 might represent the interests 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, but also protection against denial of service.

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: 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, and while not itself a distributed system, could form a foundation for one.

It is somewhat to our chagrin that 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 unauthorised communication channels. As such, we hope side channel attacks are a topic we will be able to explore in future work.

1.4 Formal Methodology

Throughout this project, we are applying formal methodology in order to 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, as 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 features in a manner not feasible today. 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.

2. We have developed extensions to the Bluespec compiler in order to export an HDL description to PVS. This will allow us to verify low-level properties of the hardware design, and also, ideally, link ISA-level descriptions to the CPU implementation.

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

1.5 CHERI and CHERI2 Reference Prototypes

As a central part of this research, we are developing reference prototype implementations of the CHERI CPU, whose functions are to allow us to explore, validate, evaluate, and demonstrate the CHERI approach. A detailed description of the current prototypes, both from architectural and practical use perspectives, may be found in the companion *CHERI Platform Reference* and *CHERI User's Guide* documents.

Our first prototype, known simply as CHERI, is based on Cambridge's Tiger MIPS research CPU, 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.

At the time of writing, CHERI1 is a pipelined baseline 64-bit MIPS CPU, and 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 DE-4 FPGA development boards; we use these boards in both mobile tablet-style and network configurations. FreeBSD is able to manage the capability coprocessor, maintaining additional thread state for capability-aware user applications, although capability features are not yet used within the kernel for its own internal protection. We have also adapted the Clang and LLVM compiler suite to allow capability use to be directed by language-level annotations in C. 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, as well as synthesise 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. In the future, should it become desirable, we will be able to construct an ASIC design from the same Bluespec source code. We hope to release the CHERI soft core as *open-source hardware*, making it available for more widespread use in research (and potentially in industry).

We have also developed a second prototype, known as CHERI2, which both deploys additional CPU features, such simultaneous multi-threading (SMT) support, but also employs a more stylised form of Bluespec intended to support formal verification. CHERI2 is now also able to boot the FreeBSD operating system, although other aspects of its design are continuing to mature.

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, and would describe 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 [59].

In 1975, Saltzer and Schroeder surveyed the rapidly expanding vocabulary of computer security in *The Protection of Information in Computer Systems* [60]. They enumerated design principles such as the *principle of least privilege*, which demands that computations run with only the privileges they require, as well as 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* [31], which considered how hardware memory protection addressed both types of failure.

The security research community also blossomed outside of MIT: Wulf's Hydra operating system at Carnegie Mellon University (CMU) [73, 11], Needham and Wilkes' CAP Computer at Cambridge [72], SRI's Provably Secure Operating System design (PSOS) [19, 47], Rushby's security kernels supported by formal methods at Newcastle [58], and Lampson's work on formal models of security protection at the Berkeley Computer Corporation all explored the structure of operating system access control, and especially the application of capabilities to the protection problem [32, 33]. Another critical offshoot from the Multics project was Ritchie and Thompson's UNIX operating system at Bell Labs, which simplified concepts from Multics, becoming the basis for countless directly and indirectly derived products such as today's Solaris, FreeBSD, Mac OS X, and Linux operating systems [56].

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 analysed 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 bypassing hardware and OS protection [28]. 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.

Forty-five 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 net-

working and cryptography technologies that transform the context of operating system security. However, the themes identified in that period 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 and the hardware and software protection techniques explored in operating systems [44]. 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 labelling 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, it was envisioned that high-assurance systems would employ a capability-oriented design mapping program structure and security policy into hardware enforcement; for example, Lampson's BCC design exploited this linkage to approximate least privilege [32, 33].

Systems such as the CAP Computer at Cambridge [72] and Ackerman's DEC PDP-1 architecture at MIT [2] attempted to realise 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* [35].

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 demise of hardware capability systems is 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) [50].

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, 74] embodied this approach, in which microkernel message passing between separate tasks stood in for hardware-assisted security domain crossings at capability invocation. Hydra develops a number of ideas, including the relationship between capabilities and object references, refining the *object-capability* paradigm, as well as pursuing the separation of

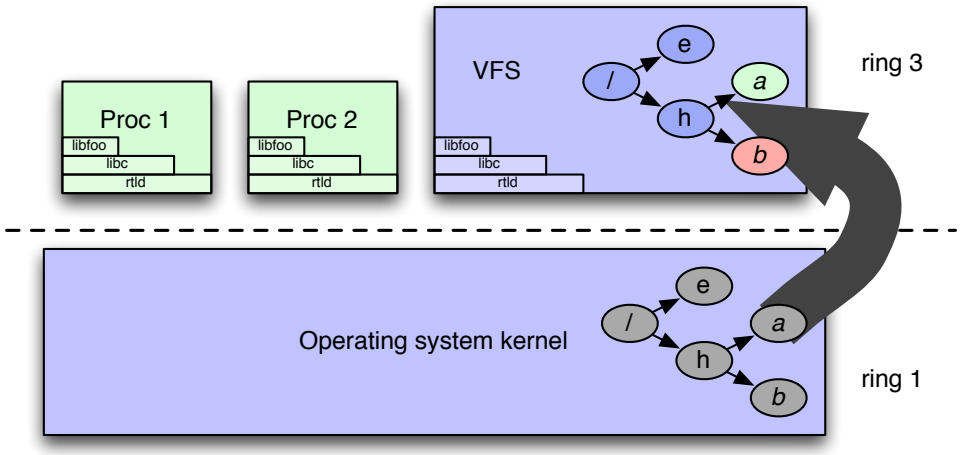


Figure 1.1: The microkernel project shifts complex OS components, such as file systems, from the kernel to userspace tasks linked by IPC. Microkernels provide a smaller, easier-to-analyse, easier-to-debug, and more robust foundation in the face of dramatic increases in OS complexity.

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 [52, 1], both microkernel systems intended to explore a 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 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 [33]. This further aligned with proposals by Andrews [4] and Rushby [58] 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: that by offering a single namespace shared by all protection domains, the distributed system programming problem had been 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,

[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* [42]

supported by hardware, software, and formal methods [37].

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 standardised 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 [61] and Neumann's Provably Secure Operating System [20] design study were ground-up operating system designs grounded in formal methodology[4]. In contrast, Schroeder's MLS kernel design for Multics [62], the DoD Kernelized Secure Operating System (KSOS) [39], and Bruce Walker's UCLA UNIX Security Kernel [67] 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 [68].

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, accomplishing sufficient assurance to target 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, building on the TMach approach while adding new flexibility [63]. 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 [65]. A significant focus of the FLASK work was performance: an access vector cache is responsible for caching access control decisions throughout the OS in order to avoid costly up-calls and message passing (with associated context switches) to the security server. NSA and SCC eventually migrated FLASK to the FLUX microkernel developed by the University of Utah in the search for improved performance. This flurry of operating system security research, invigorated by the rise of microkernels and their congruence with security kernels, 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 [45] – effectively, a composition of mandatory access control and capabilities in service to application security. This approach is embodied by

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

Efstathopoulos et al.'s Abestos [17] and Zeldovich et al.'s Histar [77] research operating systems.

Despite the decline of both hardware-oriented and microkernel capability system design, capability models continue to interest both research and industry. Shapiro's EROS (now CapROS) continues the investigation of higher-assurance software capability designs [64], and was inspired by the proprietary KEYKOS system [25]. seL4 [30], a formally verified, capability-oriented microkernel, has also continued along this avenue. General-purpose systems also adopt elements of the microkernel capability design philosophy, such as Apple's Mac OS X using Mach interprocess communication (IPC) objects as capabilities [5], and Cambridge's Capsicum [71] 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 [42], Wagner's Joe-E capability-safe subset of Java [41], and Miller's Caja capability-safe subset of JavaScript continue a language-level exploration of capability security [43].

1.6.3 Language and runtime approaches

Direct reliance on hardware for enforcement is central to both historic and current systems, but is not the only approach to enforcing isolation. The notion that limits on expressibility in a programming language can be used to enforce security properties is frequently deployed in contemporary systems in order 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 typing 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, in order to avoid data copying and context switching when sniffing network packet data [38]. 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 which, 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, allowing confinement properties to be validated [76]. NaCl relates closely to Software Fault Isolation (SFI) [66], 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 [46]. As mentioned in the previous section, the Java Virtual Machine (JVM) model is similar, combining 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, ensuring that only safe accesses have been expressed. C subsets, such as Cyclone [26], and type-safe languages such as Ruby [57], 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: some (BPF, D) limit expressibility that may prevent potentially useful constructs from being used, such as loops bounded by invariants rather than instruction limits, as well as imposing a 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 [40].

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 [54], and 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 them [69]. 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 hard address space separation, as used in Mach, EROS, and UNIX, comes at significant cost: attempts to compartmentalise system software and applications sacrifice the programmability benefits of a language-based capability design (a point made convincingly by Fabry [18]), as well as introducing significant performance overhead to cross security domain boundaries. However, running these existing software designs is critical in order 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, restoring programmability and performance so as to facilitate fine-grained compartmentalisation 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 [49, 14]. The Multics hardware provided independent virtual memory segments, pag-

ing, 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 inner rings enabled Multics to support multi-level 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, with encapsulated modular abstraction, interlayer state mappings, and abstract programs relating each layer to those on which it depended [47, 48]. The hardware design provided tagged, typed, unforgeable capabilities that were required for every operation. 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 access controls. The design allowed application layers to execute instructions efficiently, with object-oriented capability-based addressing directly to the hardware.

Capsicum Capsicum is a lightweight OS capability and sandbox framework planned for inclusion in FreeBSD 9 [71, 70]. Capsicum extends, rather than replaces, UNIX APIs, providing new kernel primitives (sandboxed capability mode and capabilities) and a userspace sandbox API. These tools support compartmentalisation of monolithic UNIX applications into logical applications, an increasingly common goal supported poorly by discretionary and mandatory access control. This approach was demonstrated by adapting core FreeBSD utilities and Google's Chromium web browser to use Capsicum primitives, showing significant complexity and robustness benefits to Capsicum over other confinement techniques.

1.7 Version history

This is the eighth version of the *CHERI Architecture Document*.

1.0 This first version of the CHERI architecture document was prepared for a six-month deliverable to DARPA. It includes 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 follows a week-long meeting in Cambridge, UK, in which many aspects of the CHERI architecture are now being formalised, including details of the capability instruction set.

1.2 The third version of the architecture document comes as the first annual reports from the CTSRD project are being prepared, 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 have significantly refined the CHERI ISA with respect to object capabilities, as well as maturing notions such as a trusted stack and the role of an operating system supervisor. The formal methods portions of the document have been dramatically expanded: we now have proofs of correctness for many basic security properties. Satisfyingly, many "future work" items in earlier versions of the report are becoming completed work in this version!

1.3 The fourth version of the architecture document has been released while the first functional CHERI prototype is in testing, and takes into account initial experiences adapting an microkernel to exploit CHERI capability features. This has 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 updates and clarifies 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; clarification of the CIncBase instruction.

1.5 This version of the document has been produced almost two years into the CTSRD project, and documents a significant revision to the CHERI ISA motivated by our efforts to introduce C-language extensions and compiler support for CHERI, as well as 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 are significantly updated.

1.6 This version brings incremental refinements to version 2 of the CHERI ISA, and also introduces early discussion of the CHERI2 prototype.

1.7 Roughly two and a half years into the project, this version clarifies and extends 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, is added to allow querying information on the most recent CP2 exception (e.g., bounds-check vs type-check violations); priorities are provided, and also clarified with respect to coprocessor exceptions vs. other MIPS ISA exceptions (e.g., unaligned access). This is the first version of the *CHERI Architecture Document* released to early adopters.

1.8 Document Structure

This document is an introduction to, and reference manual for, the CHERI CPU architecture:

Chapter 1 introduces CHERI: its motivations, goals, 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 is an instruction set architecture (ISA) reference for CHERI, providing detailed information on new instructions and their semantics.

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

Chapter 5 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 expand our consideration of the CHERI 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

We discuss our high-level design choices for the CHERI ISA, considering 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, 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 will consider in detail an instantiation of the CHERI model in an instruction set architecture extension to MIPS.

2.1 Design goals

As described in Chapter 1, the key observation motivating the CHERI design is that page-oriented virtual memory, near-universal in commodity CPUs, is neither an efficient nor programmer-friendly primitive for fine-grained, hardware-supported compartmentalisation. Virtual addressing, implemented by a memory management unit (MMU) and translation look-aside buffer (TLB), clearly plays an important role by disassociating physical memory allocation and address space management, facilitating strong separation, OS virtualisation, and VM features 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, refine-

[1] One idea we have considered is that CHERI-like semantics might be accomplished through extensions to Google's Native Client ISA [75], which uses a strict and statically analysable subset of Intel and ARM ISAs to ensure memory safety. An exciting possibility is that we might extend the LLVM intermediate representation [34] 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 answerable in terms of both performance and formal methodology.

ment, and delegation of memory and object rights) play in fine-grained compartmentalised code.

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, providing a unified view connoting both efficient and programmer-comprehensible delegation. It should be possible to write and debug compartmentalised applications without unnecessary recourse to distributed system methodology.

These security goals, combined with observations about TLB performance and a desire to compartmentalise 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.

Since we wish to consider delegation of memory and object references within an address space 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 is an explicit capability system, in which common capability manipulation operations are unprivileged instructions, avoiding transfer of control to a supervisor during regular operations. In historic capability systems, microcode was used to implement complex capability operations, and also privileged capability operations. In contemporary RISC CPU designs, the moral and functional equivalent is an exception triggering the supervisor – however, entering a supervisor 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 runtime linker, minimising hardware knowledge of higher-level language constructs.

CHERI capabilities may refer to regions of memory, implementing bounded memory access (*segments*), but also to *objects* that can be *invoked*, allowing the implementation of *protected subsystems* – services that execute in a different security domain than the caller. At the moment of object invocation, object capabilities are *unwrapped* to allow the object implementation access to private resources, and 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]. The semantics of capabilities are discussed in greater detail later in this and the following chapter.

2.2 A Hybrid Capability Architecture

Despite our complaints about the implications of virtual addressing for compartmentalisation, we feel that virtual memory is a valuable hardware facility: it provides a strong separation model, makes implementing facilities such as swapping and paging easier, and by virtue of its virtual layout, can significantly improve software maintenance and system performance. CHERI therefore adopts a *hybrid capability 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.

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

- *Page-oriented virtual memory systems* create a process abstraction via an executive (often the operating system kernel), which configures the MMU to create a *process* abstraction. 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, in which type information and protection constraints map from the programming language into instruction selection. Code at any given moment in execution exists in a protection domain consisting of a dynamic set of rights whose delegation is controlled by the flow of code – these *instantaneous rights* have sometimes been referred to as *spheres of protection* in the operating system and security literature. Such a design is generally recognised to offer greater assurance, as the *principle of least privilege* is applied at a fine 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 hybridised, 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.

[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.

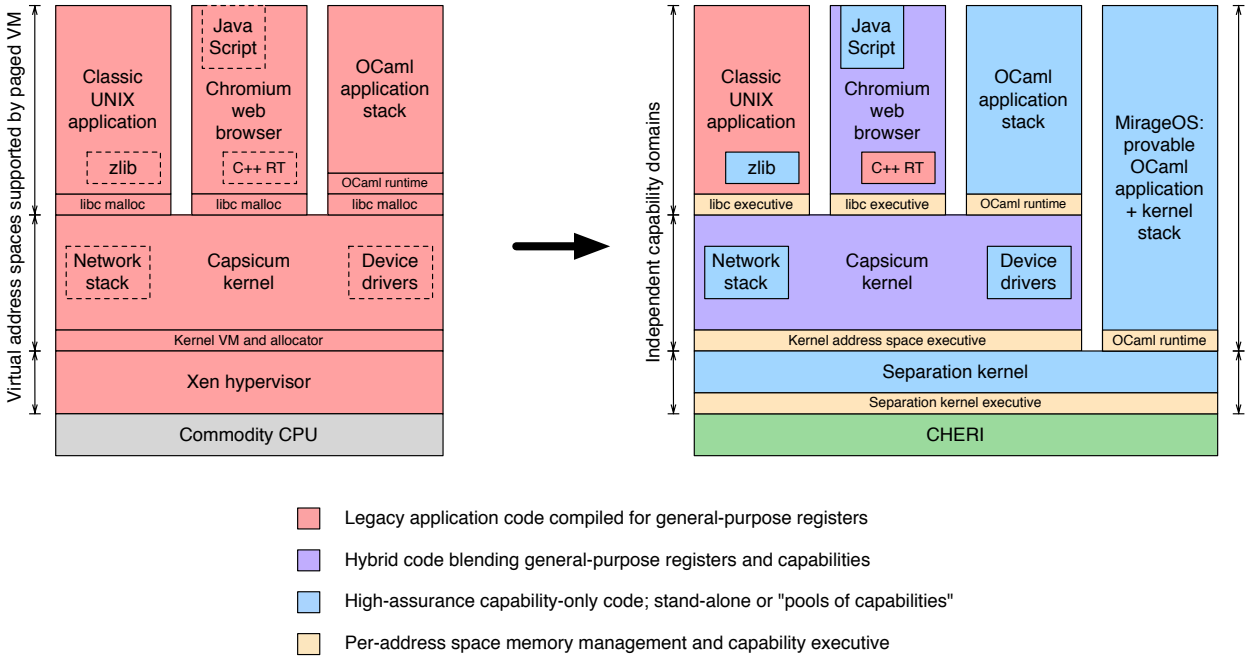


Figure 2.1: CHERI's hybrid capability architecture: initially, legacy software components execute without capability awareness, but security-sensitive TCB elements or particularly risky code bases are converted. In the long term, all packages are converted, implementing least privilege throughout the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

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 as fully capability-aware, utilising capability features for robustness and security throughout their structure. The notion of a capability-aware *executive*, likely some blend of the runtime linker and low-level system libraries, such as `libc`, also becomes valuable: the executive will set up safe linkage between mutually untrusting components (potentially with differing degrees of capability support, and hence differing ABIs), as well as ensuring that memory is managed safely to prevent memory reuse bugs escalating to security vulnerabilities[3].

2.3 The CHERI Software Stack

The notion of hybrid design is key to our adoption argument: CHERI systems will be 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

[3]Similar observations about the criticality of the runtime linker for both security and performance in capability systems have been made by Karger [29].

and security of individual system components, which may fluidly interact with other un-enhanced components. This notion of hybrid design first arose in Cambridge's Capsicum [71], 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 side by side with capability-mode processes, allowing sandboxes to be constructed, and using a capability model, rights delegated to sandboxes by applications that embody complex security policies – one such example from our USENIX Security Capsicum paper [71] is the Chromium web browser, which must map the distributed World Wide Web security model into local OS containment primitives.

CTSRD's software stack will employ hybrid design principles from the bottom up: a new capability-enhanced Separation Kernel and Hypervisor (SKAH) will implement an internal, hardware-supported capability model used to ensure its robustness. SKAH will then provide an execution substrate on which both commodity systems built on traditional RISC instruction models, such as Capsicum, can run side by side with a pure capability-oriented software stack, such as capability-adapted language runtimes. Further, Capsicum and its applications will be able to employ CHERI features in their own implementations. We will extend 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 CPU design allows software contexts to operate in one of two modes: pure capability mode, in which only accesses to the virtual address space authorised by delegated capabilities will be permitted, and hybrid mode, in which the use of capability instructions implies capability constraints, but traditional load and store instructions allow direct access to the virtual address space. For example, in this model Capsicum might employ capability-oriented instructions in the implementation of risky data manipulations (such as network packet processing), while still relying on traditionally written and compiled code for the remainder of the kernel. Similarly, within the Chromium web browser, the JavaScript interpreter might be implemented in terms of capability-oriented instructions to offer greater robustness, while the remainder of Chromium uses traditional instructions.

One particularly interesting property of our hardware design is that it is possible for capabilities to take on different semantics within different address spaces, with each address space's executive integrating memory management and capability generation. In the Capsicum kernel, for example, virtual addressing and capability use can be blended, with the compiler and kernel memory allocator using 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.

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 Reference

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

2.4.1 Capabilities Are for Compilers

Throughout, it is important to distinguish the notion of the hardware security model from 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 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 thought of as the protection domain of a thread.

There are also several implicit capability registers associated with each hardware thread, including a memory capability corresponding 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 is 256-bit; unlike general-purpose registers, capability registers are structured, containing a number of fields with defined semantics, and whose values are constrained:

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

Permissions A permissions mask that 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 The base address of a memory region.

Length The length of a memory region.

Object invocation is a central operation in the CHERI ISA, as it implements protected subsystem domain transition, atomically updating the set of rights (capabilities) held by a hardware thread, and providing a trustworthy return path for later use. When an object capability is invoked, its data and code capabilities are *unsealed*, allowing 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. Object type checking, a notion first introduced via PSOS's *type enforcement,* serves functions both at the ISA level – providing object atomicity despite the use of two independent capabilities to describe an object, and in supporting language-level typing features. As indicated earlier, the hardware capability type may be used to support language-level types, but should not be confused with language-level types.

2.4.3 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 authorise) 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, but in coordination with the software stack, can provide for hardware-supported bounds on delegation periods, and software-supported generalised 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.

2.4.4 Ephemeral capabilities and Revocation

To this end, capabilities may be further tagged as *ephemeral*, allowing 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* 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 to ephemeral objects to capability registers and limited portions of memory. Through appropriate memory management by the address space executive, the desired protection property that ephemeral capabilities be limited to a particular thread, and have bounded delegation time down the (logical) stack, can be enforced.[4]

Generalised 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 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.

2.4.5 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. 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, authorising certain operations based on what ring a processor is executing in, 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 capability register state required within the execution of each ring, authorising appropriate access for the trap handler.

[4] It has been recommended that we substitute a generalised 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, but 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.

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 authorise operations, although violation of a security property (i.e., an attempt to broaden a memory capability) will lead to a trap, allowing a software supervisor in a lower ring to provide alternative semantics.

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 (i.e., full access to the userspace address space, rather than just narrow access), 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 authorisation policy on application components by wrapping system calls safely from userspace.

2.4.6 Traps, interrupts, and exception handling

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

New exceptions New exception opportunities are introduced for both existing and new instructions, which may trap if insufficient rights are held, or an invalid operation is requested. For example, attempts to read a capability from memory using a capability without the read capability permission will trigger a trap.

Reserved capability registers for exception handling New exception handling functionality is required to ensure that exception handlers can, themselves, execute. We expect that this will imply a need to reserve two capability registers for use by exception handlers, as is the case with today's MIPS ABI, which reserves two general-purpose registers for use by exception handlers (essentially meaning that they may be overwritten at any point in execution so cannot be used by applications). However, as capabilities have security implications, it may be that we need to prevent any use of reserved exception capabilities from outside of exception handlers. The exact mechanism for this remains under discussion: use of the registers might simply check whether an exception handler is in flight (and otherwise trigger an exception); another possibility is that capabilities might specify the ring(s) from which they are available.

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 $ECPC$ 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, writes to TLB control registers in CP0 occur 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 required no-ops. This can be handled through the squashing of in-flight instructions, which may add complexity to pipeline processing – incorrect handling could lead to serious vulnerabilities.

We anticipate that further complications will come to light as we implement capability support for exception handling, and it may prove one of the more complex aspects of the CHERI ISA design.

2.4.7 Tagged Memory

As with general-purpose registers, it is desirable to store capability register values in memory – 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 authorised 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.

Additional bits are present in TLB entries to indicate whether a given memory page is configured to have regular or ephemeral 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 implementing paging of capability memory pages. Use of these instructions is conditioned on notions of ring and capability context privilege.

2.4.8 Capability Instructions

Various new instructions have been introduced, which 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, adopting similar calling conventions, and so on. The number of instructions has also been minimised to the extent possible.

2.4.9 Object Capabilities

The CHERI design calls for two forms of capabilities: capabilities describing regions of memory, offering bounded buffer "segment" semantics, and object capabilities, permitting the implementation of protected subsystems. In our model, object capabilities are represented by a pair of sealed code and data capabilities, providing the necessary information to implement a protected subsystem domain transition. Object capabilities are "invoked" using the JALCR instruction, which is responsible for unsealing the capabilities, performing a safe security domain transition, passing arguments, and returning return values, as required by the semantics of the method being invoked.

In traditional capability designs, invocation of an object capability triggered microcode responsible for state management. In CHERI, we initially plan to prototype object capability invocation through a trap to the supervisor, the moral equivalent to microcode invocation in a RISC system. However, in the long term, we will investigate the congruence of object capability invocation with message passing primitives between 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.

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 it seems that this might be a more efficient structure 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 behaviour? How can callers safely delegate arguments by reference for the duration of the call, bounding the period of retention of a capability by a callee – 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 by reusing 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).

Establishing a concrete model for capability invocation will be a critical task over the coming months. However, it is our goal to be able to support many different semantics as required by different programming languages, from an enhanced C language to Java. By adopting a RISC-like

[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, using the explicit message passing primitive to cleanly isolate the two contexts, while still allowing shared arguments to to be passed using pointers to common storage, or delegated using explicit capabilities. This would allow application developers additional flexibility for optimization.

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. Our initial prototyping approach, in which invoke instructions simply always trap, will allow us to experiment, and then optimise performance through hardware enhancements as our experience grows.

2.4.10 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 moral 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, it would be desirable to be able to reuse current peripheral networking and video devices with CHERI CPUs while maintaining desired security properties.

For the time being, we anticipate that device drivers will 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 virtualisation in commodity CPUs, where guest OSes require DMA memory performance but cannot be allowed arbitrary access to physical memory. Exploring I/O MMU and SR/IOV-like models for capabilities is high on our todo list once the processor itself is up and running.

In the longer term, one quite interesting idea is embedding CHERI support in peripheral devices themselves, requiring the device to implement a CHERI-aware TCB that would synchronise protection information with the host OS. This type of model appeals to ideas from heterogenous computing, and is one we hope to explore in greater detail in the future.

Chapter 3

CHERI Instruction Set Architecture (ISA) Reference

This chapter describes the CHERI extensions to the MIPS ISA. New instructions are implemented as a MIPS co-processor, co-processor 2. Where possible, we have conformed to existing conventions for MIPS instruction formats, but did introduce one additional opcode format. In addition to adding new instructions, the behaviour of some standard MIPS instructions and exception handling has been modified in CHERI. 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 co-processor. 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. The top 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.

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.

Conventions for Capability Register Use

We anticipate developing a set of ABI conventions regarding use of the other general-purpose capability registers similar to those for non-capability 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:

- **C16**-**C23** are callee-save registers.
- **C1**-**C4** are used to pass arguments and are treated as clobbered by function calls.
- **C1** is used to return capability values from functions.

All other capability registers are treated as caller-save. When calling across a privilege boundary, all registers must be explicitly cleared to prevent information leakage, and so all are treated as caller-save.

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** and the previous **IDC** 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. The callee should invalidate all registers that are not intended to be passed to the caller before the return.

It is the responsibility of the caller to ensure that its protection domain is entirely embodied in the capability in **IDC** so that it can restore its state upon return.

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**.

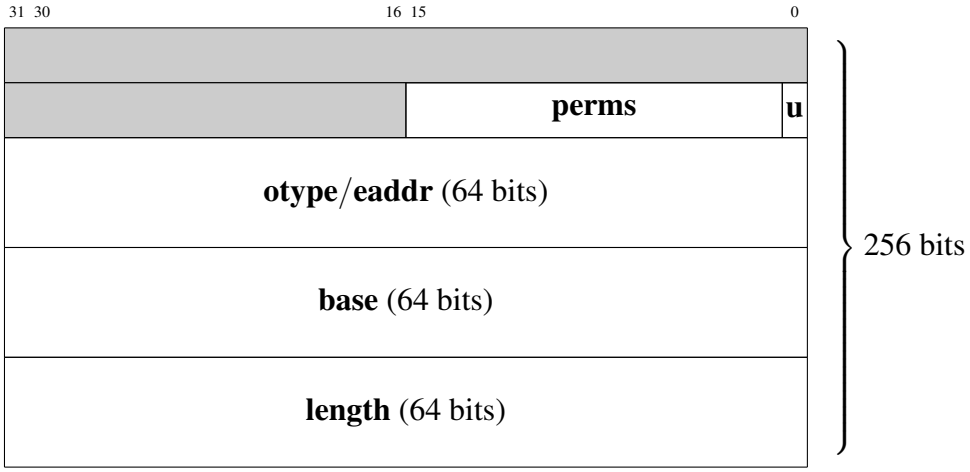


Figure 3.1: Contents of a capability

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**", 15 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 15-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.

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, treating 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 authorises 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.

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

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.

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_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**.

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

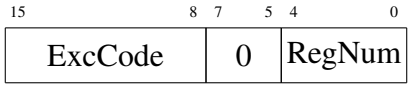


Figure 3.2: Capability Cause Register

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 co-processor 0 will be set to 18 (*C2E*, co-processor 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.

The capability co-processor 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*.

The *RegNum* field of **capcause** holds the number of the capability register whose permission was violated in the last exception. If this register was **PCC**, then *RegNum* holds 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.

All capability exceptions (C2E) are of higher priority than address error exceptions (AdEL, AdEL).

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.

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
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
0x1a	Access_EPCC Violation
0x1b	Access_KDC Violation
0x1c	Access_KCC Violation
0x1d	Access_KR1C Violation
0x1e	Access_KR2C Violation

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	Call Trap
	Return Trap

Table 3.4: Exception Priority

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, AdEL and AdES to prevent a process 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 the number of the capability register, not 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 initialised 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 behaviour of standard MIPS instructions when a capability co-processor 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 capability type problem, memory bound check failure, or permission error, co-processor 2 exception (*C2E*) is thrown.

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**, which must have appropriate permissions of *Permit_Store* or *Permit_Load* set. If the load or store is not permitted due to a memory bound check failure or a permission error, a co-processor 2 exception (*C2E*) is thrown.

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

Branch on Coprocessor 2 Instructions The MIPS ISA describes four instructions, BC2T, BC2F, BC2TL and BC2FL for branch on condition code in coprocessor 2 if true or false, with the L-suffixed variants indicating that the condition is considered likely. These instructions all use the same opcode, with 3 bits for identifying the condition code and two for distinguishing the true/false and hinted/unhinted variants.

The CHERI architecture provides a single instruction using this opcode and reusing the five bits of the condition code and flags to identify a capability register. This branches if the capability contains an invalid capability.

3.6 Capability Instruction Set Overview

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 3.3 lists available capability coprocesor instructions.

3.7 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

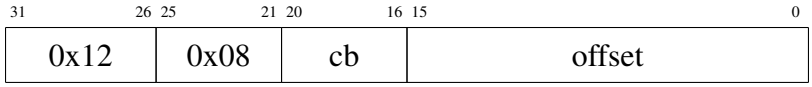
Mnemonic	Description
CGetPerm	Move permissions field to a general-purpose register
CGetBase	Move base to a general-purpose register
CGetLen	Move length to a general-purpose register
CGetType	Move object type field to a general-purpose register
CGetUnsealed	Move unsealed bit to a general-purpose register
CGetPCC	Move the PCC and PC to general-purpose registers
CGetTag	Move tag bit to a general-purpose register
CGetCause	Move capability exception cause register to a general-purpose register
CAndPerm	Restrict Permissions
CIncBase	Increase Base
CSetLen	Set Length
CSetType	Set the **otype**/**eaddr** of an executable capability
CClearTag	Invalidate a capability register
BC2F	Branch if capability is invalid
CSC	Store Capability Register
CLC	Load Capability Register
CL[BHWD][U]	Load Byte, Half-Word, Word or Double Via Capability Register (Unsigned)
CS[BHWD][H]	Store Byte, Half-Word, Word or Double Via Capability Register (High)
CLLD	Load linked doubleword via capability register
CSCD	Store conditional doubleword via capability register
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 3.3: Capability coprocessor instruction summary

BC2F: Branch if capability is invalid

Format (6?)

BC2F cb, offset



Description

Sets the **RPC** to **RPC**+*offset*, where *offset* is sign extended, if *cb* contains an invalid capability.

Pseudocode

```
if cb.tag then
    if PC + offset + 4 > PCC.length then
        raise_c2_exception()
    else
        PC ← PC + offset
    end if
end if
```

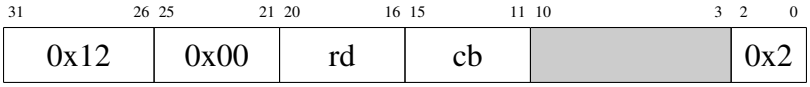
Exceptions

A coprocessor 2 exception is raised if **RPC**+*offset*+4 is greater than **PCC**.**length**.

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

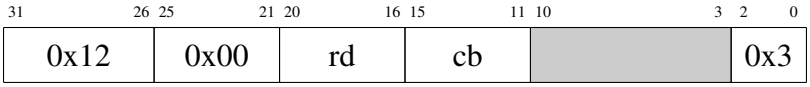
A coprocessor 2 exception is raised if:

- *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

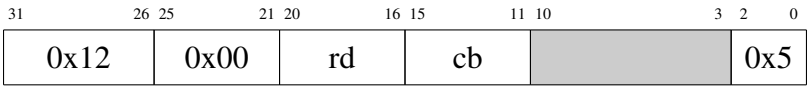
A coprocessor 2 exception is raised if:

- *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

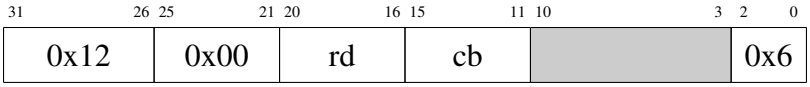
A coprocessor 2 exception is raised if:

- *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

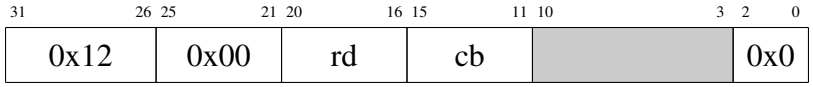
A coprocessor 2 exception is raised if:

- *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

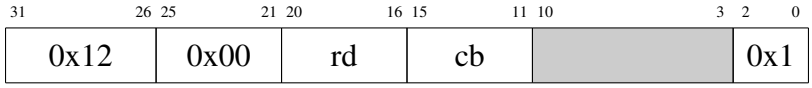
A coprocessor 2 exception is raised if:

- *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

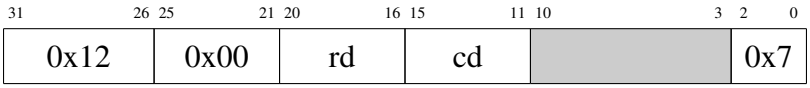
A coprocessor 2 exception is raised if:

- *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

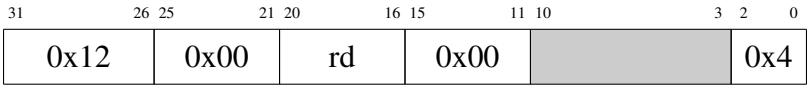
A coprocessor 2 exception is raised if:

- *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

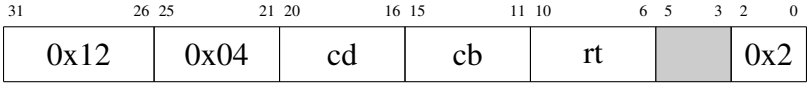
A coprocessor 2 exception is raised if:

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

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 $\neq 0$ **then**
    raise_c2_exception(exceptionTag, cb)
else if not cb.unsealed **and** rt $\neq 0$ **then**
    raise_c2_exception(exceptionSealed, cb)
else if rt $>$ cb.length **then**
    raise_c2_exception(exceptionLength, cb)
else
    cd $\leftarrow$ cb
    cd.base $\leftarrow$ cb.base $+$ rt
    cd.length $\leftarrow$ 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

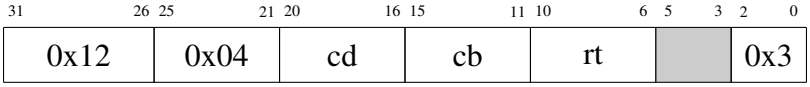
- **CIncBase** can be used to copy one register to another, by setting *rt* equal to zero. If *rt* is zero, the operation will succeed even if *cb.***u** is not set, allowing it to be used to copy sealed capabilities. **CIncBase** also succeeds if *rt* is zero and *cb.***tag** is unset, allowing it to 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**

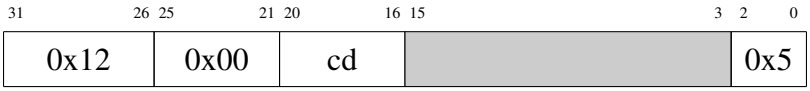
Notes

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

CClearTag: Invalidates a Capability

Format (1)

CClearTag cd



Description

Capability register *cd* is cleared, as is its associated tag.

Pseudocode

if register_inaccessible(cd) **then**
    raise_c2_exception()
else
    cd.tag ← **false**
    cd.unsealed ← **false**
    cd.perms ← ∅
    cd.type ← 0
    cd.base ← 0
    cd.length ← 0
    cd.reserved ← 0
end if

Exceptions

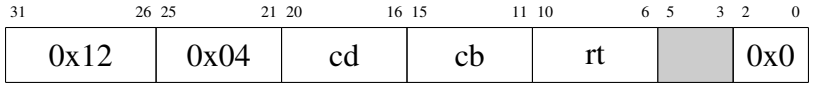
A coprocessor 2 exception is raised if:

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

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

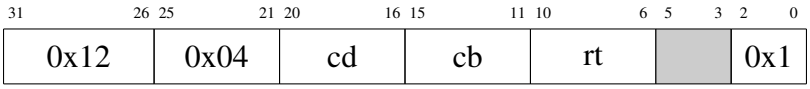
A coprocessor 2 exception is raised if:

- *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



- *cd*.**otype**/**eaddr** is set equal to *cb*.**base**+*rt*
- *cd*.**perms**.*Permit_Seal* is set
- All the other fields of *cd* are unchanged.

Pseudocode

if register_inaccessible(cd) **then**
    raise_c2_exception()
else if register_inaccessible(cb) **then**
    raise_c2_exception()
else if not cd.tag **then**
    raise_c2_exception(exceptionTag, cd)
else if not cb.tag **then**
    raise_c2_exception(exceptionTag, cb)
else if not cd.unsealed **then**
    raise_c2_exception(exceptionSealed, cd)
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 not cd.perms.Permit_Execute **then**
    raise_c2_exception(exceptionPermitExecute, cd)
else if rt $\geq$ cb.length **then**
    raise_c2_exception(exceptionLength, cb)
else if cb.base + rt $<$ cd.base **then**
    raise_c2_exception()
else if cb.base + rt + 1 $>$ cd.base + cd.length **then**
    raise_c2_exception()
else if cb.base + rt $> 2^{64} - 1$ **then**
    raise_c2_exception()
else
    cd.otype $\leftarrow$ cb.base + rt
    cd.perms.Permit_Seal $\leftarrow$ **true**
end if

Exceptions

A coprocessor 2 exception is raised if:

- *cd*.**tag** is not set.
- *cd*.**u** is not set.
- *cd*.**perms**.*Permit_Execute* is not set.
- *cb*.**tag** is not set.
- *cb*.**u** is not set
- *cb*.**perms**.*Permit_Set_Type* is not set.
- $rt \geq cb$.**length**
- *cb.***base** + $rt < cd.$**base**.
- *cb.***base** + $rt + 1 > cd.$**base** + *cd.***length**
- *cb.***base** + *rt* overflows a 64-bit unsigned integer.

Notes

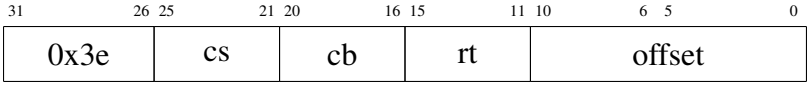
This instruction can be used only to create an executable capability whose **otype**/**eaddr** is equal to a virtual address between **base** and **base+length**-1.

In a typical use of this instruction, *cb* and *cd* will be the same register, and so the two checks of *rt* against *cb* and *cd* are equivalent. To allow the instruction to be used in a more flexible way by applications, *cd* and *cb* can be different registers, with the execute permission being derived from *cd* and the seal permission being derived from *cb*. In this case, *rt* is checked against both capabilities. The check against *cb* is necessary for security. The check against *cd* is not needed for security, but if it fails it is highly suggestive of an erroneous program (in particular, if the operation were allowed to succeed it would result in a capability that raises an exception when it is used as an argument to `CCall`.)

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.

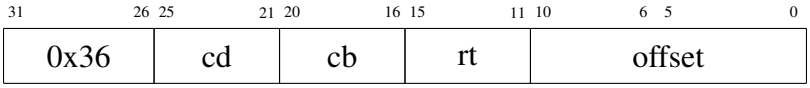
Notes

- 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.
- 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
    cs ← mem[addr]
    cs.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.

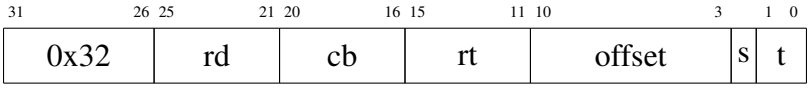
Notes

- This instruction reuses the opcode from the Load Doubleword to Coprocessor 2 (LDC2) instruction in the MIPS Specification.
- 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, extending 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 capability that grants permission to load data and be a valid capability.

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 behaviour 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.

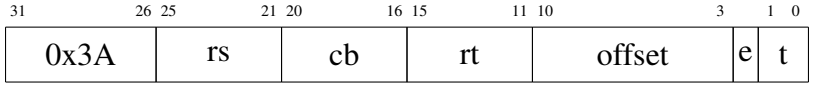
Notes

- 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)
CSBH rs, rt, offset(cb)
CSHH rs, rt, offset(cb)
CSWH rs, rt, offset(cb)
CSBR rs, rt(cb)
CSHR rs, rt(cb)
CSWR rs, rt(cb)
CSDR rs, rt(cb)
CSBHR rs, rt(cb)
CSHHR rs, rt(cb)
CSWHR rs, rt(cb)
CSBI rs, offset(cb)
CSHI rs, offset(cb)
CSWI rs, offset(cb)
CSDI rs, offset(cb)
CSBHI rs, offset(cb)
CSHHI rs, offset(cb)
CSWHI rs, offset(cb)



Purpose

Stores the some or all of a register in an memory.

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 part of the of the register to store depends on the value of the *t* and *e* fields. The *e* field specifies the end of the word to store, with a value of 0 indicating the low end and a value of 1 indicating the high end. The values of *t* define the size:

0 byte (8 bits)

1 halfword (16 bits)

2 word (32 bits)

3 doubleword (64 bits)

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 if s = 0 then
    mem[addr:addr + size − 1] ← rd[0:size − 1]
    tags[toTag(addr)] ← false
else
    mem[addr:addr + size − 1] ← rd[8-size:7]
    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.

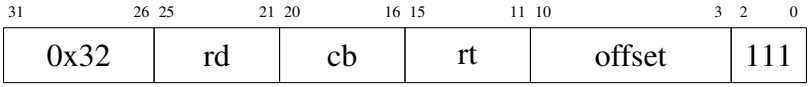
Notes

- 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, offset, rt(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 preceeding 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.

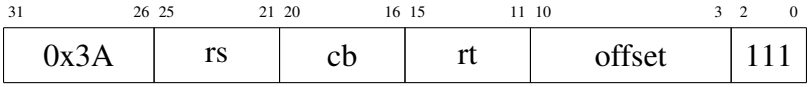
Pseudocode

```
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
addr ← cb.base + rt + offset
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, offset, rt(cb)
CSCDR rs, rt(cb)
CSCRI rs, offset(cb)



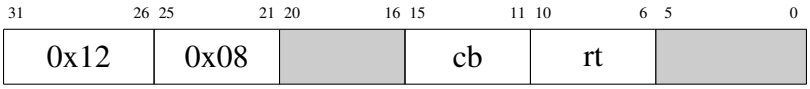
Pseudocode

```
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_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(cb.base + rt + offset) < 32 then
    raise_exception(AdES)
else if not linkedFlag then
    rs ← 0
else
    mem[addr:addr+7] ← rs
    tags[toTag(cb.base + rt + offset)] ← false
    rs ← 1
end if
```

CJR: Jump Capability Register

Format (3)

CJR rt(cb)



Description

PCC is loaded from *cb*, and **PC** starts at the address *cb.***base** + general-purpose register *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
addr ← cb.base + rt
if rt + 4 < cb.length then
    raise_c2_exception(exceptionLength, cb)
else if align_of(addr) < 4 then
    raise_exception(exceptionAdEL)
else
    execute_delay_slot()
    PC ← addr
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 less 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

Creates a code capability for the address the subroutine will return to, and stores this capability in the return capability register (capability register number 24). The bounds and permissions fields of this capability will be taken from the current program capability. **PC** is stored in general-purpose register 31. **PCC** is then loaded from capability register *cb*, and the **PC** jumps to address *cb.***base** + general-purpose register *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
addr ← cb.base + rt
if rt + 4 < cb.length then
    raise_c2_exception(exceptionLength, cb)
else if align_of(addr) < 4 then
    raise_exception(exceptionAdEL)
else
    execute_delay_slot()
    R31 ← PC
    PC ← addr
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 less than *cb.***length**.
- *cb.***u** is not set.
- *cb.***tag** is not set.
- *cb.***perms**.*Non_Ephemeral* is not set.

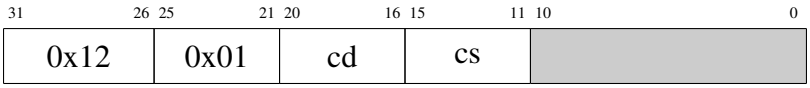
An address error exception will be raised if

- *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

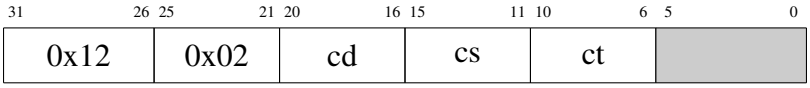
A coprocessor 2 exception is raised if:

- *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

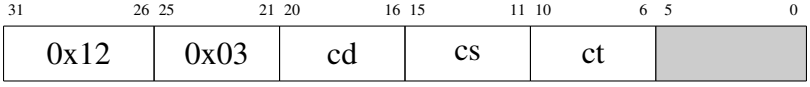
A coprocessor 2 exception is raised if:

- *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 ≠ cs.otype then
    raise_c2_exception(exceptionType, ct)
else if not ct.perms.Permit_Seal then
    raise_c2_exception(exceptionPermitSeal, ct)
else
    cd ← cs
    cd.u ← true
    cd.perms.Non_Ephemeral ← cs.perms.Non_Ephemeral and ct.perms.Non_Ephemeral
end if
```

Exceptions

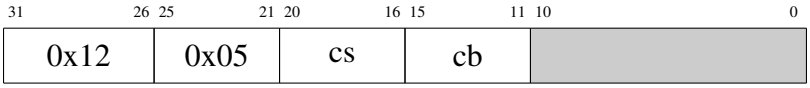
A coprocessor 2 exception is raised if:

- *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 can be unsealed 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 implemenation of CHERI, **CCall** is implemented by the hardware raising an exception, and the rest of the behavior of the instruction 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. **PCC** is pushed onto the trusted system stack.
2. **IDC** is pushed onto the trusted system stack.
3. *cs* is unsealed and the result placed in **PCC**.
4. *cb* is unsealed and the result placed in **IDC**.
5. The program branches to the address *cs*.**otype**/**eaddr**.

Pseudocode (hardware)

```
raise_c2_exception(exceptionCall, 0xff)
```

Pseudocode (software)

```
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 ≠ 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
    mem[TSS .. TSS + 31] ← PCC
    tags[toTag(TSS)] ← PCC.tag
    TSS ← TSS − 32
    mem[TSS .. TSS + 31] ← IDC
    tags[toTag(TSS) ← TSS.tag
    TSS ← TSS − 32
    PCC ← cs
    PCC.unsealed ← true
    IDC ← cb
    IDC.unsealed ← true
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*.**u** is set.
- *cb*.**u** is set.
- *cs*.**otype**/**eaddr** ≠ *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**.

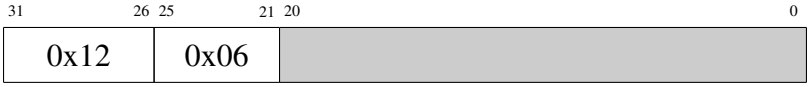
Notes

From the point of view of security, CCall needs to be an atomic operation (i.e. the caller cannot decide to just do some it, because this 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 CPU branches to the address given in **IDC.otype/eaddr**.

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

The capability exception code will be 0x06 and the handler vector will be 0x100 above the general purpose exception handler.

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**.

Chapter 4

CHERI in Programming Languages and Operating Systems

We capture some of our early thoughts on the topic of programming language and operating system use of CHERI instructions. 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 compartmentalised TCB components.
- 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 virtualisable per-address-space capability system is feasible and practical, allowing use of capability models within individual hierarchical rings and address spaces.

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

4.1 Development plan and status

At the time of writing, we are roughly fifteen months into a four-year research project in hardware and software security. We have now successfully prototyped a fully pipelined 64-bit MIPS CPU

Package	License	Description
Das U-boot	GPL	"Universal" boot loader
FreeBSD	BSD	Tightly integrated UNIX operating system; mature, software capability-enabled via Capsicum, POSIX-compliant, multithreaded
DTrace	CDDL	Software tracing and profiling facility
gcc/ld/gdb	GPL	FSF gcc compiler, linker, toolchain, debugger
clang/LLVM	BSD	Next generation compiler and toolchain; extremely extensible with powerful intermediate representation (IR); MIPS 64 support currently weak
Apache	Apache	Web server
Chromium	BSD	Web browser based on WebKit framework
X.org	MIT	X windows server
KDE/Gnome	GPL	Desktop environments and application suites

Table 4.1: Open source foundation for CHERI software research; we will prefer BSD, MIT, and Apache licenses in order to encourage technology transfer.

implementing a significant portion of the CHERI ISA; detailed information on this current prototype may be found in the accompanying *CHERI Preliminary User's Guide*. Because we remain relatively early in the project, it is hard to predict either the nature of our results or the schedule under which they will be accomplished. The following sections document our recent accomplishments and continuing strategy for exploring the software implications of the CHERI architecture.

4.2 Open source foundations

During the initial bring-up phase of our prototype CHERI CPU, CHERI's support for incremental adoption should prove invaluable: we will be able to rely on current open source boot loaders, operating systems, programming languages, compilers, and applications (a partial list appears in Table 4.1), selectively deploying capability features in the most critical software foundations, as well as the most vulnerable services.

4.3 Current software implementation

With a preliminary hardware prototype, we have begun to adapt existing MIPS toolchain components for CHERI, as well as begun the process of developing software that uses CHERI's ISA features.

4.3.1 Extended GNU assember (gas)

We have extended the GNU assembler (gas) to support the CHERI ISA, allowing assembly files, as well as inline assembler from C, to make use of CHERI instructions.

4.4 Extended LLVM/Clang

LLVM is a framework for implementing compilers, comprising a well-defined intermediate representation (IR), a set of APIs for generating this representation, optimisation 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 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 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.

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.

4.4.1 Extended CHERI unit test suite

Using the CHERI assembler, we have extended our existing MIPS ISA test suite to exercise various aspects of the CHERI ISA. The current test suite validates the behaviour of memory capabilities and capability exceptions, including monotonic decrease in rights using capability manipulation instructions. Currently, capability call (CCall) and return (CReturn) are not tested. More detailed implementation status for the CHERI hardware prototype may be found in the *CHERI Preliminary User's Guide*.

4.4.2 Deimos demonstration microkernel

Motivated by a demonstration of CHERI for the November, 2011 DARPA CRASH PI meeting, we have implemented Deimos, a demonstration microkernel that employs hardware capabilities, rather than the memory management unit (MMU) to enforce memory protections. Deimos blends conventional 64-bit MIPS machine code with a small amount of hand-crafted MIPS and CHERI assembly to sandbox applications, delegating rights to a VGA frame buffer and touch screen using capabilities. To support a basic operating system and applications, we have begun to develop a prototype ABI, including capability-aware calling conventions.

The resulting demonstration, implemented on the Terasic tPad touch screen FPGA board, illustrates hardware-enforced confinement using the CHERI hybrid capability model, as well as a trusted path to the user using I/O delegation. Detailed information on the Deimos microkernel may be found in the *CHERI Preliminary User's Guide*. Deimos has provided an initial validation of our hybrid capability model by transparently sandboxing unmodified MIPS code, supporting hybrid operation using capabilities, even within the MIPS kernel ring.

4.5 Future plans

In 2012, we plan to significantly improve the maturity of our hardware and software research platforms. Our short-term goal is to bring up the unmodified FreeBSD operating system on the CHERI prototype. We can then begin a programme of experimentation through modifications to various aspects of the commodity software stack: boot loaders, hypervisors, operating system kernels, language runtimes, and mainstream applications.

4.5.1 Short term: ABI and linker exploration

Once we have established an initial reasonable set (or perhaps multiple sets) of semantics, we will push initial capability support into the C compiler, runtime linker, and C library. Our goal at that point will be to demonstrate the use of memory capabilities in a lightly modified C runtime environment, as well as to investigate the application binary interface (ABI) requirements of a capability processor. We anticipate developing several new ABIs, as well as *bridging code* to link software components using those different ABIs.

Initially, we expect a new ABI for function invocation where both sides of the invocation are capability-aware, introducing capability register conventions for caller-save, callee-save, function arguments and return values, temporary value, the return address, and stack/global pointers. We will also need to define stack conventions, alignment requirements, and a language runtime exception model, which we anticipate modeling on existing MIPS conventions to the greatest extent possible.

We anticipate that the role of the runtime linker and low-level application libraries, such as the C runtime and `libc`, will be critical to this effort. Our process model is premised in a runtime linker beginning execution with the privilege to derive any required capabilities to operate its address space – not dissimilar to the boot-time environment in which the initial content of **C0** is an all-powerful capability. The memory allocator will play a key role in enforcing address space non-reuse, our first cut approach for ensuring that capabilities delegating access to the heap do not allow unintended access to later objects occupying the same portion of the address space. The exact role and use of ephemeral capabilities must be explored in much greater detail, as they must be used correctly in order to ensure safe reuse of stack space; an alternative approach we have discussed in only the most theoretical of terms of a log-structured stack, which would avoid any reuse.

These explorations will lead in turn to consideration of the role of the linking process, both static linking and runtime linking. We will need to tag compiled objects with their ABI model so that legacy (capability-unaware) code can be linked with capability-aware code, which might require the runtime linker to construct shims mapping between the ABIs. We anticipate encountering numerous unanticipated obstacles in this area, as linkers are complex and subtle beasts!

A further ABI will be required around protected subsystem invocation, including defining the security semantics of information flow between caller and callee, as well as interactions with the runtime linker (which will ultimately be responsible for gluing together all components in a robust and secure manner). Our initial implementation of object capability invocation will be via a trap to the OS kernel, allowing easy software exploration of the semantics space.

4.5.2 Longer term: language extensions for capabilities

With initial OS and language runtime support for capabilities in a workable state, we will then move on to language extensions to support capabilities.

Our initial target will be modest modifications to the C language to make using memory capabilities in applications straightforward; one possibility is the introduction of a `__capability` qualifier that instructs the compiler to make use of capability registers, memory, and instructions in handling certain data types; relatively transparent support for converting between capabilities and pointers would be required, and certain operations (such as arbitrary pointer arithmetic) would be unsupportable for capability types.

We also hope to consider the impact of a capability-enabled execution substrate for compiled, interpreted, and just-in-time (JIT) compiled type-safe languages. We have early interests in Ada, Java and OCaml, which incorporate capability notions in their language structure and runtime. Reusing aspects of Cambridge's research on MirageOS, an OCaml-based operating system may offer early dividends.

In this phase, experimentation with real-world software will be key to our approach: we hope to deploy early capability support in the FreeBSD kernel, including in sensitive and memory-intensive components such as device drivers and the network stack. We see early promising investigations in converting the kernels Berkeley Packet Filter (BPF) just-in-time compiler to use capability instructions, as well as modifying the kernel memory allocator to support returning memory capabilities instead of pointers for certain kernel memory types (such as `mbuf` storage used in manipulating packet data) using new kernel programming interfaces (KPIs). Similarly, in userspace, we hope to demonstrate improvements in security early through the deployment of memory capabilities in sensitive and historically vulnerable libraries for string, image, video, and audio processing.

We will also begin our investigation of the integration of object capability support into current programming languages; as the C language has no integrated notion of an object, we anticipate explicit new language primitives that allow C components to provide object services. We are interested in the idea of mapping C++ classes and objects into object capabilities, but have done little exploration of the potential implications of that idea.

Many of our core security objectives for CHERI rely on relatively easy use of object capabilities in the programming language, so finding a convenient mapping will be critical. Object capability support is also central to our incrementalism argument, as that mechanism is required to confine legacy code within a capability-aware application; we anticipate that the runtime linker will play a key role in setting up "legacy" sandboxes, as well as providing the stubs required to pass arguments in and out of sandboxes.

4.6 Programming languages

In the following section, we consider potential uses of capabilities in the C and Ada programming languages. This discussion is not intended to be rigorous – rather, we hope to stimulate thought on potential linkages between general-purpose registers, capability registers, memory, and programming language semantics and mechanism.

4.6.1 The function of registers

An important initial observation is that general-purpose registers serve two functions: first, as the operands to CPU instructions, and second, as a compiler-managed cache of main memory. We anticipate that capability registers will play a very similar role for capability-aware software, acting as operands (sometimes implicit) to instructions, and acting as a compiler-managed cache of capabilities in main memory.

Capability registers come at a significantly higher silicon cost than general-purpose registers, largely because of their greater size (256 bits vs. 64 bits). The number of registers, 32, is an arbitrary number selected to mirror the 32 general-purpose registers in MIPS. However, it could be that following further investigation, we reduce the number of registers to 16; the sole impact of that modification (other than minor compiler modifications and some assembly changes) should be to reduce the size of the compiler-managed cache, leading to performance changes that can then be evaluated.

4.6.2 Pointers/access types

We envision three general strategies for integrating CHERI capability support into programming languages:

1. Existing language semantics are mapped into new hardware primitives, shifting enforcement responsibility from hardware to software; for example, in mapping Ada or Java programs into CHERI instructions.

2. New language elements are added in order to allow access to new hardware primitives, exposing new protection functionality to the programming language environment; this will be of particular benefit in low-level languages such as C and C++.

3. At hybrid points between programming languages, such as at the juncture between C routines in the Java runtime environment and byte-code interpretation, or between CHERI-aware C and non-CHERI-aware C, employ CHERI instructions to enforce controlled communication properties meeting the invariants of both sides.

From these general strategies, a number of specific strategies could arise. For example, an Ada access type might be represented directly using memory or object capabilities, perhaps linked to a general purpose register carrying language-level state about its use. Ada array slicing (e.g., passing a subrange of an array to a subprogram) can be implemented by just modifying the address in the general purpose register. If there is a requirement to prevent the called subprogram from accessing parts of the array outside the slice, this can be achieved by subsetting the memory capability, shifting its base and length. Alternatively, subsetting could be accomplished for an array implemented as an object capability using interposition.

For C, a new `__capability` qualifier might be introduced to trigger use of memory capabilities rather than simple pointers to refer to memory regions; the language might support converting from a pointer to a memory capability, but not vice versa, providing integrated support for bounded buffers. Violation of capability properties would generate a runtime exception, which for C would be delivered via a signal handler; for C++, it would be desirable to map capability-generated exceptions into runtime exceptions.

Object capability integration will be significantly more tricky for C, which lacks direct language features for object orientation. A related concern is how C++ exceptions should be propagated across protection domain boundaries; this is an issue that will require careful examination.

4.6.3 Function calls/subprogram calls

We anticipate that most function invocations or subprogram calls will remain within protection domains, allowing the current, or a new capability-aware, ABI to be employed, but not requiring an object capability invocation. If a protect boundary is not crossed, then the caller will need to ensure that **C31** and **R31** provide useful access to a stack. When jumping to a code capability, not only will a new value of **PC** need to be specified, but also a new value for **PCC** that the target **PC** is relative to.

More generally, the handling of stacks around protection boundary crossings is an issue of significant concern; ephemeral capabilities provide a foundation for low-level stack properties (such as limiting the flow of capabilities referring to the stack), but higher-level semantics, such as fresh stack instances being available for use on boundary crossings will require further investigation. Certainly, the traditional behaviour of sharing a single, contiguous stack along the entire call stack of a thread seems unviable.

A C function pointer (or Ada access to subprogram type) can be represented as a pair of a code capability and the address of the subprogram. The subprogram call can be made using a *jalcr* instruction.

4.6.4 Calling a protected subsystem

Object capability invocation is effectively code capability invocation through a call gate, providing well-defined protection semantics to both the caller and the callee. While we have not defined the precise syntax and semantics of this instruction yet, we anticipate that it will accept an object capability argument, and then a small number of explicit register arguments to be passed to and from the invoked protected subsystem. As such, it will be implemented using a different ABI than a normal function invocation, which assumes mutual access to registers, stacks, and so on, as well as mutually trusting behaviour regarding leaked data in registers across the invocation boundary.

Among other concerns, it seems clear to us that the stack should not be shared between caller and callee unless access is specifically delegated; implementing call-by-reference to stack values is critical to C language applications that frequently rely on the stack to hold temporary values to which pointers can be taken. In the CHERI model, ephemeral capabilities can be used to prevent the unbounded sharing of such capabilities outside of the current thread of execution, acting effectively as a form of taint.

An object capability invocation might reasonably be thought of as a message-passing event between threads with different register files (and may be implemented that way in hardware at some future point). Messages will be constructed from general-purpose and capability registers, capturing function arguments, return values, and so on.

The representation of these calls in a high-level language is also for further study. One possible option (for Ada programs) is to use the Ada Distributed Systems Annex, and to treat each protected subsystem as an Ada "partition" of the program. There seems to be no comparable equivalent in the C language environment other than inter-process communication, which seems a poor model; in both C and C++, we may find that nontrivial language extensions are required.

4.6.5 Dynamic linking

It seems reasonable to anticipate that each code object, such as a library, exists in a contiguous segment, to which execution access can be granted using a memory capability. Individual instruction addresses will necessarily be relative to the code capability describing the library; likewise, jump tables and in-place modifications to the instruction stream to set jump targets may also need code capabilities to be inserted. New CHERI instructions *jalcr* and *jcr* allow calling and returning using code capabilities without an implied change of protection domain.

4.7 The operating system

Here, we consider changes required to the operating system kernel; as the kernel, itself, is a large C-based application, we anticipate that it will make much the same use of memory and object capabilities to compartmentalise its own components. However, certain changes are required only for the operating system, due to its roles in booting, virtual memory management, and exception handling.

4.7.1 Booting

We anticipate several types of "booting" in CHERI-based systems:

1. The boot loader beginning execution on a reset CPU
2. A stand-alone operating system kernel loaded by a boot loader; this might be a separation kernel, hypervisor, or general-purpose operating system loaded directly
3. A general-purpose or specialised legacy or hybrid capability operating system loaded on top of a separation kernel, contained within an MMU-enforced protection domain
4. A general-purpose or specialised capability-based operating system loaded on top of a separation kernel, contained by the capability enforcement mechanism

In the first case, the CPUs initial state is set by hardware; in all other cases, initial state will by definition be a convention, with certain rights passed into the booting OS by either MMU memory delegation or capability delegation, depending on the runtime model. It is worth observing that several of these cases resemble the initial state found by a runtime linker loading in a traditional OS process.

4.7.2 Virtual memory management

While it is conceivable that an entirely capability-based environment might simply not enable the MMU, we anticipate using the MMU in most of the above scenarios, if only to provide swapping facilities. In the case of a legacy or hybrid operating system, MMU management is a key OS feature, used to provide page-oriented virtual memory facilities to applications, semantics such as POSIX `fork`, and so on. It is fair to say that we do not yet fully understand the interactions between the capability model and MMUs, but certain changes will be necessary:

- Additional TLB bits will be present to enable loading and storing of ephemeral and nonephemeral capabilities in pages.
- Swapping functions must preserve tag bits for memory.
- Some general strategy for when and how tag bits can be stored in an external medium will need to be supported – for example, will it be possible to store capabilities in memory backed to a memory mapped file?

- The operating system kernel itself might reasonably use memory capabilities to safely reference user memory, eliminating broad classes of current attacks.

4.7.3 Exceptions

In MIPS, "exceptions" are event triggering entry of an exception handler; they include traps explicitly triggered by software, such as system calls, but also page faults and interrupts. As described in Chapters 2 and 3, exception handlers require capabilities to execute, and we have reserved **KCC** and **KDC** for this purpose.

Most work saving and restoring interrupted contexts on MIPS is performed in software, with the exception handler running with privilege and juggling a small number of registers. This work will become no simpler with in CHERI: capability registers must be held and manipulated to operate at all; further, additional capability state, including **C0**...**C31** and **EPCC** must be reserved and restored in order to properly manage the interrupted context. We anticipate a small but nontrivial body of assembly code being added to existing MIPS context switch support in FreeBSD to handle this behaviour.

Chapter 5

Future Directions

The CTSRD project, of which CHERI is just one element, has now been in progress for two 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 prove 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 compartmentalising applications using Capsicum and CHERI features. This is a sub-project 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 DE-4 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 verification purposes.

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.

5.1 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. Capability processor extensions to MIPS would, then, not only be a core research result from this project, but also the first example of research conducted on the reference processor.

Prototyping a CPU for internal use and preparing a CPU for broad public inspection are markedly different activities: we will need to continue to refine our implementation, perform more extensive testing, but also collaborate with both direct participants in the CTSRD project and outside, such as at Bluespec, to ensure that the implementation can be useful in this form. Among other considerations, we are investigating the applicability of current open source licenses to open hardware projects.

5.2 Formal Methods for Bluespec

After a week-long meeting with Joe Stoy from Bluespec earlier in the project, and in light of existing research [55], we believe that a strong link can be made between the Bluespec programming language and formal methods tools developed at SRI, such as PVS. However, it is also the case that the Bluespec toolchain as of the beginning of our project provided little access to intermediate state during the compilation process, potentially requiring formal tools to reproduce elements of that compilation process. To compensate for this limitation, we are collaborating with Bluespec to enhance its compiler suite to export additional information, which can then be fed into verification tools such as PVS. This has led to additional collaboration between SRI and Bluespec in incorporating Yices into the Bluespec compiler and has resulted in significant improvements to Bluespec compilation itself. This is leading us directly into a significant programme of exploration into properties to be proven, ideally linking our prototype CPU implementation directly to ISA properties.

5.3 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), as

well as extend the C programming language to allow protection properties to be specified by the programmer. We have extended the GNU assembler, as well as 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.

5.4 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 starting 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.

It is also desirable for the kernel to 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.

5.5 Evaluating Performance and Programmability

This report describes a fundamental premise: that through an in-address space capability model, performance and programmability for compartmentalised 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 compartmentalisation 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 compartmentalisation, such as that developed in the Capsicum project. An early element of this work will certainly include testing security context switch speed as the number of security domains increases, in order to confirm our hypothesis regarding TLB size and highly compartmentalised software, but also that capability context switching can be made orders of magnitude fast 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 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.

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

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

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

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

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

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

[43] 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`.

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

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

[46] 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.

[47] 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.

[48] 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.

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

[50] 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.

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

[52] 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).

[53] 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.

[54] 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.

[55] D. Richards and D. Lester. A monadic approach to automated reasoning for Bluespec SystemVerilog. *Innovations in Systems and Software Engineering*, pages 1–11, 2011. 10.1007/s11334-011-0149-0.

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

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

[58] 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)).

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

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

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

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

[71] 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.

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

[73] 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.

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

[75] 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.

[76] 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.

[77] 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.