Technical Report

UCAM-CL-TR-941
ISSN 1476-2986

Number 941

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

An Introduction to CHERI

Robert N. M. Watson, Simon W. Moore,
Peter Sewell, Peter G. Neumann

September 2019

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

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

This version of the report incorporates minor changes to the September 2019 original, which were released October 2019.

Approved for public release; distribution is unlimited

This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 ("CTSRD"), with additional support from FA8750-11-C-0249 ("MRC2"), HR0011-18-C-0016 ("ECATS"), and FA8650-18-C-7809 ("CIFV") as part of the DARPA CRASH, MRC, and SSITH research programs.

The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

We also acknowledge the EPSRC REMS Programme Grant (EP/K008528/1), the ERC ELVER Advanced Grant (789108), the Isaac Newton Trust, the UK Higher Education Innovation Fund (HEIF), Thales E-Security, Microsoft Research Cambridge, Arm Limited, Google, Google DeepMind, HP Enterprise, and the Gates Cambridge Trust.

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

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

ISSN 1476-2986

Abstract

CHERI (Capability Hardware Enhanced RISC Instructions) extends conventional processor Instruction-Set Architectures (ISAs) with *architectural capabilities* to enable fine-grained memory protection and highly scalable software compartmentalization. CHERI's hybrid capability-system approach allows architectural capabilities to be integrated cleanly with contemporary RISC architectures and microarchitectures, as well as with MMU-based C/C++-language software stacks.

CHERI's capabilities are unforgeable tokens of authority, which can be used to implement both explicit pointers (those declared in the language) and implied pointers (those used by the runtime and generated code) in C and C++. When used for C/C++ memory protection, CHERI directly mitigates a broad range of known vulnerability types and exploit techniques. Support for more scalable software compartmentalization facilitates software mitigation techniques such as sandboxing, which also defend against future (currently unknown) vulnerability classes and exploit techniques.

We have developed, evaluated, and demonstrated this approach through hardware-software prototypes, including multiple CPU prototypes, and a full software stack. This stack includes an adapted version of the Clang/LLVM compiler suite with support for capability-based C/C++, and a full UNIX-style OS (CheriBSD, based on FreeBSD) implementing spatial, referential, and (currently for userspace) non-stack temporal memory safety. Formal modeling and verification allow us to make strong claims about the security properties of CHERI-enabled architectures.

This report is a high-level introduction to CHERI. The report describes our architectural approach, CHERI's key microarchitectural implications, our approach to formal modeling and proof, the CHERI software model, our software-stack prototypes, further reading, and potential areas of future research.

Contributors

The authors of this report thank members of the CTSRD, MRC2, ECATS, CIFV, and REMS teams, our past and current research collaborators at SRI International and the University of Cambridge, as well as our colleagues at other institutions who have provided invaluable contributions to this work: Hesham Almatary, Jonathan Anderson, John Baldwin, Hadrien Barrel, Thomas Bauereiss, Ruslan Bukin, David Chisnall, James Clarke, Nirav Dave, Brooks Davis, Lawrence Esswood, Nathaniel W. Filardo, Khilan Gudka, Brett Gutstein, Alexandre Joannou, Robert Kovacsics, Ben Laurie, A. Theo Markettos, J. Edward Maste, Marno van der Maas, Alfredo Mazzinghi, Alan Mujumdar, Prashanth Mundkur, Steven J. Murdoch, Edward Napierala, Kyndylan Nienhuis, Robert Norton-Wright, Philip Paeps, Lucian Paul-Trifu, Alex Richardson, Michael Roe, Colin Rothwell, Peter Rugg, Hassen Saidi, Stacey Son, Domagoj Stolfa, Andrew Turner, Munraj Vadera, Jonathan Woodruff, Hongyan Xia, and Bjoern A. Zeeb. We express special appreciation to James Clarke, Brooks Davis, Nathaniel W. Filardo, and Alex Richardson for their detailed feedback on this report.

Acknowledgments

We further acknowledge support, feedback, and suggestions from Sam Ainsworth, Ross J. Anderson, Lee Badger, Graeme Barnes, Stuart Biles, Matthias Boettcher, David Brazdil, Brian Campbell, Gregory Chadwick, Serban Constantinescu, Simon Cooper, Chris Dalton, Dominique Devriese, Jeremy Epstein, Wedson Filho, Anthony Fox, Paul J. Fox, Virgil Gligor, Li Gong, John Goodacre, Mike Gordon, Paul Gotch, Richard Grisenthwaite, Tom Grocutt, Steven Hand, Jong Hun Han, Andy Herbert, Andy Hopper, Alex Horsman, Warren A. Hunt Jr, Timothy Jones, Asif Khan, Myron King, Chris Kitching, Wojciech Koszek, Patrick Lincoln, Anil Madhavapeddy, Ilias Marinos, Tim Marsland, Doug Maughan, Kayvan Memarian, Dejan Milojicic, Andrew W. Moore, Will Morland, Greg Morrisett, Bryan Randell, John Rushby, Hassen Saidi, Kenneth F. Shotting, Hans Petter Selasky, Andrew Scull, Muhammad Shahbaz, Bradley Smith, Lee Smith, Ian Stark, Joe Stoy, Richard Uhler, Tom Van Vleck, Jacques Vidrine, Hugo Vincent, Philip Withnall, and Sam M. Weber.

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

This work would not have been possible without support from our sponsors. We are especially grateful to Howie Shrobe, Robert Laddaga, Stu Wagner, Jonathan Smith, John Launchbury, Dale Waters, Linton Salmon, Daniel Adams, Laurisa Goergen, Marnie Dunsmore, and John Marsh at DARPA for their support over many years in helping us to realize CHERI.

Chapter 1

Introduction

CHERI (Capability Hardware Enhanced RISC Instructions) extends conventional hardware Instruction-Set Architectures (ISAs) with new architectural features to enable fine-grained memory protection and highly scalable software compartmentalization. New memory-protection features allow historically memory-unsafe programming languages such as C and C++ to support strong, compatible, and efficient protection against currently widely exploited vulnerabilities. Scalable compartmentalization enables the fine-grained decomposition of operating-system (OS) and application code to limit the effects of security vulnerabilities to a degree unsupportable by current architectures.

CHERI is a hybrid capability architecture extension in that it is able to blend architectural capabilities with conventional MMU-based architectures and microarchitectures, and with conventional software stacks based on virtual memory and C/C++. This approach allows incremental deployment within existing ecosystems, which we have demonstrated through hardware and software prototyping. We have developed:

- ISA changes to introduce *architectural capabilities*, hardware-supported descriptions of permissions that can be used, in place of integer addresses, to refer to data, code, and objects in protected ways (Chapter 2);
- New microarchitecture demonstrating that capabilities can be implemented efficiently in hardware, including support for efficient *tagged memory* to protect capabilities in memory, and *compressed capabilities* to reduce memory overhead (Chapter 3); and
- Formal models of CHERI-extended ISAs, used as the architecture definition, as readable documentation, for architecture design exploration, for automatic construction of executable ISA-level simulators, for automatic test generation, and for mechanized verification of formal statements and proofs of the architecture's security properties (Chapter 4).

These features enable new capability-based software constructs that are incrementally deployable within existing software ecosystems. Through extensive prototyping and co-design, we have demonstrated and evaluated:

- New software construction models that use capabilities to provide fine-grained memory protection and scalable software compartmentalization (Chapter 5);

- Language and compiler extensions to use capabilities in implementing memory-safe C and C++, and Foreign Function Interfaces (FFIs) for higher-level managed languages (Section 6.1);

- OS extensions to use (and support application use of) fine-grained memory protection (spatial, referential, and (non-stack) temporal memory safety) and abstraction extensions to support scalable software compartmentalization (Section 6.2); and

- Application-level adaptations to operate correctly with CHERI memory protection and software compartmentalization (Section 6.3).

CHERI is a hardware/software/semantics co-design project, combining architecture design, hardware implementation, adaption of mainstream software stacks, and formal semantics and proof. The CHERI ideas have been developed first as a modification to 64-bit MIPS. They have now also been developed for an experimental version of 64-bit ARMv8-A (in collaboration with Arm), and also for 32/64-bit RISC-V. We have prototyped complete software stacks for CHERI by adapting widely used open-source software such as Clang/LLVM, FreeBSD, FreeRTOS, and applications such as WebKit, OpenSSH, and PostgreSQL. We have formally modeled the architecture and constructed a number of proofs about its security, as well as used these models for microarchitectural validation and software bring-up in our prototypes. These models also have the potential to support further activities such as formal proofs about software and microarchitecture.

In this report, we briefly introduce each of these elements, referring the reader to our papers and technical reports for further details on these topics (Chapter 7). We also consider promising areas for future research (Chapter 8).

Chapter 2

The CHERI Architecture

CHERI extends conventional ISAs, which use machine words to represent language-level integers and pointers, with a new type of hardware-supported data, the *architectural capability* [13, 14, 15, 18]. Capabilities can be used to protect (virtual) addresses intended to be used as code or data pointers – those arising from source-language pointers, and also those used in the underlying implementations of language features such as local and global variables, thread-local storage, return addresses, C++ vtable pointers, and inter-library linkage. All memory accesses, including loads, stores, and instruction fetch, must be authorized by a capability. As with existing kinds of hardware-supported data (integers, floats, vectors), capabilities are held in registers and in memory; they are loaded, stored, and manipulated using new *capability-aware instructions*. Compatibility with current software designs has been essential to our approach: CHERI composes well with contemporary RISC architectures, microarchitectures, compiler implementations, operating-system design, and application structure.

The authoritative reference for the CHERI architecture is the CHERI ISA specification [14]. That report describes our overall research approach, architecture-neutral protection model, and mappings into the 64-bit MIPS and 32/64-bit RISC-V architectures. It also provides detailed design rationale for a number of key CHERI design choices.

2.1 A Portable Architectural Protection Model

We have experimented with adding the CHERI protection model to a number of ISAs, beginning with 64-bit MIPS, which was the baseline for our original research. More recently, we have developed an experimental version for 64-bit ARMv8-A (in collaboration with Arm), and also for 32/64-bit RISC-V. We have also sketched a possible integration of CHERI with the 64-bit x86 architecture. Except for architecture-specific compiler backend code, machine-dependent aspects of the OS kernel (such as early boot, context switching, and exception handling) and userspace runtime (such as the run-time linker), CHERI-aware C/C++ code is portable across underlying architectures. We describe the CHERI protection model before considering CHERI integration with specific architectures in Section 2.9.

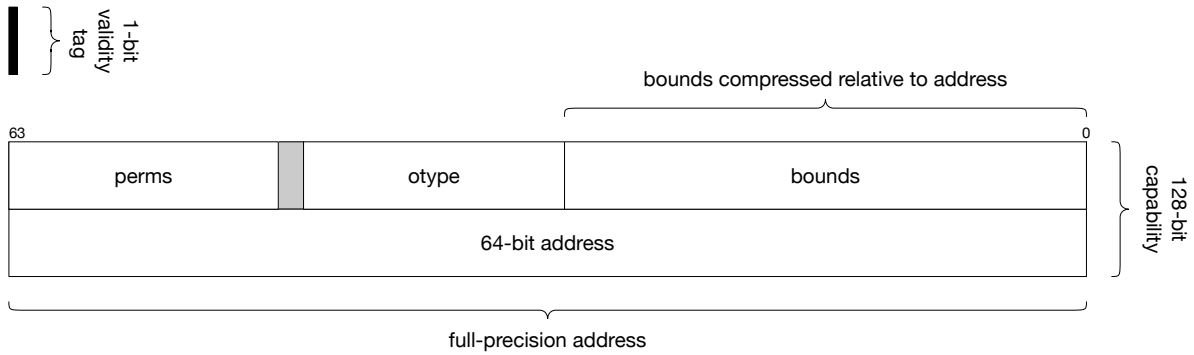

Figure 2.1: 128-bit CHERI Concentrate capability representation: 64-bit address and metadata in addressable memory and 1-bit tag out of band.

2.2 Capabilities

CHERI capabilities are twice the width of the native integer pointer type of the baseline architecture: there are 128-bit capabilities on 64-bit platforms, and 64-bit capabilities on 32-bit platforms. Each capability consists of an integer address of the natural size for the architecture (e.g., 32 or 64 bit), and also additional metadata that is compressed in order to fit in the remaining 32 or 64 bits of the capability (Figure 2.1). In addition, they are associated with a 1-bit validity "tag" whose value is maintained in registers and memory by the architecture. Each element of the capability contributes to the protection model, and is enforced by hardware:

Validity tag The tag tracks the validity of a capability; if invalid, the capability cannot be used for load, store, instruction fetch, or other operations. It is still possible to extract fields from an invalid capability, including its address. Capability-aware instructions maintain the tag (if desired) as capabilities are loaded and stored, and as capability fields are accessed, manipulated, and used – as long as the rules described in Section 2.3 are followed.

Bounds The lower and upper bounds describe the portion of the address space to which the capability authorizes loads, stores, and/or instruction fetches.

Permissions The permissions mask controls how the capability can be used – for example, by restricting loading and storing of data and/or capabilities or by prohibiting instruction fetch.

Object type If this is not equal to -1, the capability is "sealed" (with this object type) and cannot be modified or dereferenced. Sealed capabilities can be used to implement opaque pointer types. This is the foundation on which controlled non-monotonicity can be used to support fine-grained, in-address-space compartmentalization.

When stored in memory, valid capabilities must be naturally aligned – i.e., at 64-bit or 128-bit boundaries, depending on capability size – as that is the granularity at which in-memory tags are maintained (Section 3.1). Partial or complete overwrites with data, rather than a complete

overwrite with a valid capability, lead to the in-memory tag being cleared, preventing corrupted capabilities from later being dereferenced.

Capability bounds compression reduces the memory footprint of capabilities: The full capability, including address, permissions, and bounds fits within 128 bits (plus the 1-bit out-of-band tag) (Section 3.2). Bounds compression takes advantage of redundancy between the address and the bounds, which occurs because a pointer typically falls within (or close to) its associated allocation, and because allocations are typically well aligned. The compression scheme uses a floating-point representation, allowing high-precision bounds for small objects, but requiring stronger alignment and padding for larger allocations.

2.3 Architectural Rules for Capability Use

The architecture enforces several important security properties on changes to capability metadata. First, there are properties about the execution of single instructions:

Provenance validity ensures that valid capabilities can only be constructed by instructions that do so explicitly (not by byte manipulation, for example), from other valid capabilities. This property applies to capabilities in registers and in memory.

Capability monotonicity ensures that, when any instruction constructs a new capability (except in sealed capability manipulation and exception raising), it cannot exceed the permissions and bounds of the capability from which it was derived.

Then there are properties about executions, especially:

Reachable capability monotonicity ensures that, in any execution, of arbitrary code, until execution is yielded to another domain, the set of *reachable capabilities* (those accessible to the current program state via registers, memory, sealing, unsealing, and constructing sub-capabilities) cannot increase.

At boot time, the architecture provides initial capabilities to the firmware, allowing data access and instruction fetch across the full address space. Additionally, all tags are cleared in memory. Further capabilities can then be derived (in accordance with the monotonicity property) as they are passed from firmware to boot loader, from boot loader to hypervisor, from hypervisor to the OS, and from the OS to the application. At each stage in the derivation chain, bounds and permissions may be restricted to further limit access. For example, the OS may assign capabilities for only a limited portion of the address space to the user software, preventing use of other portions of the address space.

Similarly, capabilities carry with them *intentionality*: when a process passes a capability as an argument to a system call, the OS kernel can carefully use only that capability to ensure that it does not access other process memory that was not intended by the user process – even though the kernel may in fact have permission to access the entire address space through other capabilities it holds. This is important as it prevents “confused deputy” problems, in which a more privileged party uses an excess of privilege when acting on behalf of a less privileged party, performing operations that were not intended to be authorized. For example, this prevents

the kernel from overflowing the bounds on a userspace buffer when a pointer to the buffer is passed as a system-call argument.

These architectural properties provide the foundation on which a capability-based OS, compiler, and runtime can implement C/C++-language memory safety and compartmentalization.

2.4 General-Purpose Capability Registers

CHERI capabilities can be held in capability registers – architectural registers extended to hold the capability tag and the full capability data width – as well as in tagged memory. When in registers, capabilities can be used as operands to capability-aware instructions that inspect, manipulate, dereference, and otherwise operate on capabilities (Section 2.6). CHERI can be implemented in two ways with respect to the general-purpose register file:

A split capability register file introduces a new general-purpose capability register file, in the style of a floating-point register file, that complements the existing general-purpose integer register file.

A merged capability register file extends the existing general-purpose integer registers to include the tag and additional width required to hold capabilities, in the style of 32-bit extensions to 16-bit architectures, and 64-bit extensions to 32-bit architectures.

Both approaches allow implementation of the CHERI software protection models described later in this report, but have differing architectural, microarchitectural, memory-footprint, and software-stack tradeoffs. Compiled binary code will be specific to one or the other approach as instantiated in the architecture.

As capability values move between registers and memory, tags track the flow of valid, uncorrupted capabilities through the system, controlling future use of capability values. Tagging capability registers themselves, and not just memory locations, allows the implementation of capability-oblivious code: for example, C-language `memcpy()` utilizes capability load and store instructions so that any valid tags will be maintained across a suitably aligned memory-copy operation, retaining pointer dereferenceability, but is also able to copy untagged data.

2.5 Special Capability Registers

In addition to general-purpose capability registers, some existing special-purpose registers require extension to capability width, and some entirely new capability-width special-purpose registers are required. For example, the program counter (PC) and exception program counter (EPC) of the architecture are extended to full capabilities, allowing capabilities to constrain control flow (PCC), and to be properly saved and restored during exception handling (EPCC). Another special capability register, the default data capability (DDC) automatically indirects and controls all integer-relative loads and stores, allowing non-capability-aware code to be constrained using a capability. Depending on the baseline architecture, other extensions may be necessary to handle, for example, capability-based access to thread-local storage.

2.6 Capability-Aware Instructions

CHERI adds new instructions to the baseline ISA, performing the following classes of functions:

Retrieve capability fields Retrieve integer values for various capability fields, including its tag, address, permissions, and object type. This generally includes conditional move and comparison instructions for certain fields to improve the density of generated code.

Manipulate capability fields Set or modify, subject to monotonicity, various capability fields, including the address, permissions, and object type. This includes capability pointer-arithmetic instructions.

Load or store via capabilities Load integer, capability, or other values via a suitably authorized capability. This may include instructions to access data relative to the program-counter capability.

Control flow Perform a jump or jump-and-link-register to a capability destination.

Special capability registers Retrieve and set the values of special capability registers – e.g., of the exception program-counter capability (EPCC) during exception handling.

Compartmentalization These instructions support fast protection-domain transitions, to be discussed in Section 2.7.

An important aspect of CHERI's intentionality is that instructions expect either a capability or integer operand, and never dynamically select one interpretation or another based on the tag value. If a capability-relative load, store, or other operation attempts to use an untagged capability value for access, a hardware exception will be thrown. This prevents manipulation of capability bits as data, which will clear the tag, from causing this corrupted capability to have a DDC-relative integer-address interpretation rather than triggering an exception. For a split register file, this static distinction is additionally necessary to identify which register file to use.

2.7 Controlled Non-Monotonicity

Capability monotonicity prevents new capability values with greater rights from being derived from prior capability values with fewer rights. This prevents a variety of attack techniques from being successful, and is an essential foundation for software compartmentalization. However, there are legitimate use cases where monotonicity might prevent current design patterns; for example, memory allocators frequently store metadata just outside of the bounds of a memory allocation. A CHERI-enabled memory allocator must either include the metadata within the bounds of a returned allocation (a potential route for attack), or exclude it (which prevents access to allocator metadata via the pointer). In most situations, rederivation is the preferred solution: software should retain a more privileged capability – e.g., to the full memory mapping

from which the allocation is made – and then configure a suitably expanded capability on demand. However, there are three architectural circumstances in which non-monotonic behavior is directly supported. In both cases, control will be reliably transfered to a protected vector, and access may be granted to additional data capabilities not available to the code triggering the control-flow transfer:

Exception handling When an exception is thrown, the existing architectural mechanism performs a ring transition and transfers control to a well-defined (and protected) vector. With CHERI, suitably privileged code also gains access to additional capability registers providing additional rights to the exception handler, which may be distinct from those held by the interrupted code. This will typically be used to grant the exception handler access to data and further kernel capabilities, for example.

CCall to sealed capabilities CHERI also defines a new control-flow instruction, `CCall`, which compares two sealed operand registers to check that they have the same object type, and then unseals the registers, installing the first in PCC. This transfers control to a well-defined (and protected) vector, and also grants access to an additional data capability.

Jump to a sentry capability In addition to `CCall`, CHERI also defines a reserved object type permitting a `CCall`-like control-flow transfer without unsealing an additional data capability.

The former mechanism extends exception handling to support the CHERI protection model. Both mechanisms can also be used for the purpose of implementing software compartmentalization, which require non-monotonicity to allow control to implement domain transition (Section 5.2). The latter mechanism allows domain transition to be implemented without requiring the use of exceptions or ring transitions.

2.8 A Hybrid Capability Architecture

A key design goal for CHERI is to support the continued use of C/C++-language and virtual-memory-based hypervisors, operating systems, and applications. This is possible because CHERI is a hybrid capability architecture designed to integrate a capability model with a conventional MMU-based architecture in non-disruptive and incrementally adoptable manner for software stacks. Several design choices stem from these goals:

Capabilities on MMU-enabled systems describe virtual addresses When a capability authorizes a memory access, its address field (and bounds) are interpreted with respect to the current addressing mode. If translation is disabled, then the capability will have a physical or guest-physical interpretation. If translation is enabled, then the capability will have a virtual-address interpretation. This allows CHERI capabilities to be used to implement code and data pointers in much the same manner as integer addresses are used today. This has a number of implications, including that care must be taken in allowing capabilities derived with respect to one address space in another address space – e.g., passing capabilities between UNIX processes.

ISA	Status	Width(s)	Register file	On fail?	MMU	Cap mode
MIPS	Full	64-, 128-bit	Split	Exception	TLB	No
ARMv8-A	Experimental	128-bit	Merged	Clear tag	PTE	Yes
RISC-V	Draft	64-, 128-bit	Both	Exception	PTE	Yes
x86-64	Sketch	128-bit	Merged	TBD	PTE	Yes

Figure 2.2: We have extended several architectures to implement the CHERI protection model. Architecture-specific design choices include: Whether to use a split or merged register file; Whether to deliver an exception or clear a tag on instruction failure; Whether virtual-memory permissions for capabilities appear in software-managed TLB entries or architectural page-table entries; And whether a capability encoding mode is required to conserve opcode space.

The default data capability (DDC) constrains integer-relative memory accesses When capability-unaware load and store instructions are used, these will be implicitly indirected via, and controlled by, DDC. This allows the data accesses of capability-unaware code to be controlled. In some architectures, the additional implied ADD to relocate data accesses may present a microarchitectural challenge as it can affect a key critical path.

The program-counter capability (PCC) constrains instruction fetches When the processor fetches instructions, it will do so via PCC, which may relocate or constrain what instructions can be executed. This allows capabilities to be used to improve control-flow robustness. It is believed that, unlike DDC, the additional implied add associated with PCC does not present a microarchitectural challenge.

In general, CHERI extensions to an architecture are intended to take on the style and flavor of that architecture. They should generally conform to architectural expectations for the number of read and write ports on register files, available indexing modes for load and store instructions, address-space layout conventions, etc. This allows existing software design choices regarding ABIs, code generation, memory layout, program linkage, and so on, to be retained, as well as improving comparability with the un-CHERI-enabled baseline for performance purposes.

2.9 CHERI in Specific Architectures

When extending an architecture with the CHERI protection model, a number of design choices must be made (Figure 2.2). In general, we have aimed to be sympathetic to existing conventions, stylistic and functional, in that ISA. This reduces friction and effort with respect to hardware implementations, architectural design choices, Application Binary Interfaces (ABIs), compiler optimizations, software structure, and so on. This includes instruction opcode choices (e.g., sizes of immediate fields), MMU behaviors (TLB permission bits for MIPS' software TLB, and PTE permission bits for ARMv8-A's and RISC-V's architectural page-table walkers), and so on.

Other areas can involve considerable subtlety. CHERI mandates monotonicity in most instruction transformations of capability values, but this could be accomplished in multiple ways.

A capability bounds-setting instruction might throw an exception if a non-monotonic transformation is requested, for example, but the monotonicity property would also be maintained if, instead, the tag on the resulting capability value were to be cleared. The former leads to an immediate exception being fired, which may be preferable for software debugging, whereas the latter will trigger an exception only when a load, store, or instruction fetch is requested via the now-invalid capability. In some architectures (e.g., MIPS), many instructions can throw exceptions; in others (e.g., ARMv8), data-dependent exception-throwing is avoided in most instructions, to simplify instruction dependency analysis in out-of-order processors' scheduling. Even in CHERI-MIPS, writing data to memory containing a tagged capability will lead to the silent clearing of a tag, rather than throwing an exception, both for microarchitectural reasons (generating a precise architectural exception at that point in the memory subsystem would be challenging) and also for software reasons: current software frequently overwrites memory (e.g., during stack-frame reuse), and throwing an exception would be undesirable in those circumstances.

A further challenging area lies in opcode allocation. To implement intentionality, CHERI requires that instructions expect any specific operand to either be a capability or an integer address, with an exception thrown if that expectation is violated. This ensures that a value with a tag stripped due to (for example) memory corruption cannot accidentally take on a DDC-relative interpretation. However, in order to offer the full scope of load-store instruction variants, this requires a doubling of opcode space, which can be a constrained (and non-renewable) resource in some architectures. One alternative approach is to introduce a new opcode encoding, *capability mode*, in which the same opcode for an integer-relative instruction is reused for its capability-relative variant based on a processor status bit. As MIPS has a relatively narrow set of load-store operations, we simply allocate additional opcode space for new capability-relative instructions. However, ARMv8-A and RISC-V have richer sets of load-store operations, so we reuse opcode space and depend on a capability encoding-mode bit. This requires further mechanism, which varies by architecture, to support controlled transition between opcode encodings.

Chapter 3

CHERI Microarchitecture

A principal design goal of the CHERI architecture has been to add new architectural primitives with only limited impact on the overall microarchitecture of contemporary processor and memory-subsystem designs. We have explored potential approaches to integrating CHERI into multiple microarchitectures including our locally developed pipelined BERI 64-bit MIPS core, and also the Bluespec Piccolo 32/64-bit RISC-V core. We identified key microarchitectural challenges in the following areas:

Tagged memory Conventional DRAM does not support capability tagging. Architecturally, the CHERI protection model does not require a particular implementation of tagging, just that tags be suitably protected and properly coherent with the data they protect. In early designs, we used a simple look-aside tag table stored in DRAM and maintained by the memory controller along with a cache – however, performance analysis revealed a significant DRAM access-rate overhead to this approach. This led us to design a hierarchical tag table able to benefit from the non-uniform distribution of capabilities in memory: inevitably, some pages are rich in capabilities (e.g., stacks, vtable storage), whereas others are not (e.g., memory mapped files, video and image data) (Section 3.1). Another option would be to reuse additional metadata storage present in more contemporary DDR designs, including bits available for use in ECC, to hold tags.

Capability compression In the absence of capability compression, CHERI capabilities would be $4\times$ rather than $2\times$ the native address size, given the need to store three separate virtual addresses (bottom bound, capability address, and upper bound) as well as metadata. Bounds compression, which exploits redundancy between these three addresses, is therefore essential to reducing the dynamic memory footprint of pointer-intensive applications (such as language runtimes). Developing a compression scheme that balanced software requirements for precision with microarchitectural efficiency was a significant challenge (Section 3.2).

CHERI also affects some other aspects of microarchitecture; for example:

- In some microarchitectures, bus widths and/or data-path widths may need to be increased to the capability width from the natural integer width – especially those without vector instruction support.

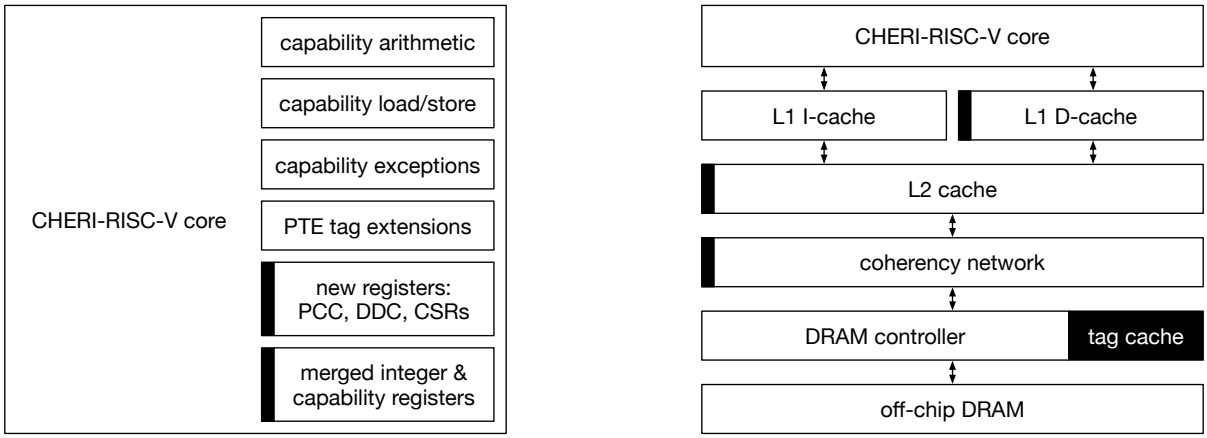

Figure 3.1: CHERI-RISC-V extensions to a processor core and memory hierarchy.

- DDC transformation of capability-unaware loads and stores implicitly introduces a further addition in effective address calculation, which may impact the critical path.
- Some aspects of CHERI benefit from data-dependent exceptions or other behavior – for example, in implementing a page-table capability dirty bit, or exceptions on an attempt to store a tagged capability to a page not supporting capabilities – which may impact some current microarchitectural design choices.

Overall, however, CHERI has been designed to avoid changing fundamental design choices in current architectures and microarchitectures: essential elements such as pipeline structure, memory subsystem designs including caches, MMUs, and so on, retain their current structure.

3.1 Tag Controllers and Tag Caches

Today, widely used DRAM and memory subsystems do not support additional out-of-band metadata storage. While extending the pipeline, register files, and caches to include tags is straightforward with CHERI, as we are careful to adopt the same cache consistency properties for tags as for the data they protect, DRAM tag storage has an impact across the System-on-Chip.

In early prototyping, we utilized a straightforward *tag table*: memory was partitioned, allowing a small reserved (and protected) portion to be set aside to hold tags for the remainder. A *tag controller*, affine to the memory controller, is responsible for presenting a tagged-memory abstraction to the remainder of the memory subsystem, along with a *tag cache* to exploit spatial locality to improve performance. Analysis showed that while this performed well, it introduced substantial additional DRAM traffic due to notably different locality properties for tags and the lines of data they protect.

We therefore proposed a *hierarchical tag table* able to exploit the variable density of asserted tags across pages of DRAM [7]. In this model, higher levels in the table now indicate the presence of tags in contiguous regions of memory represented by lower levels of the table. If no bit is present in a higher level of the table, then any corresponding regions in lower levels

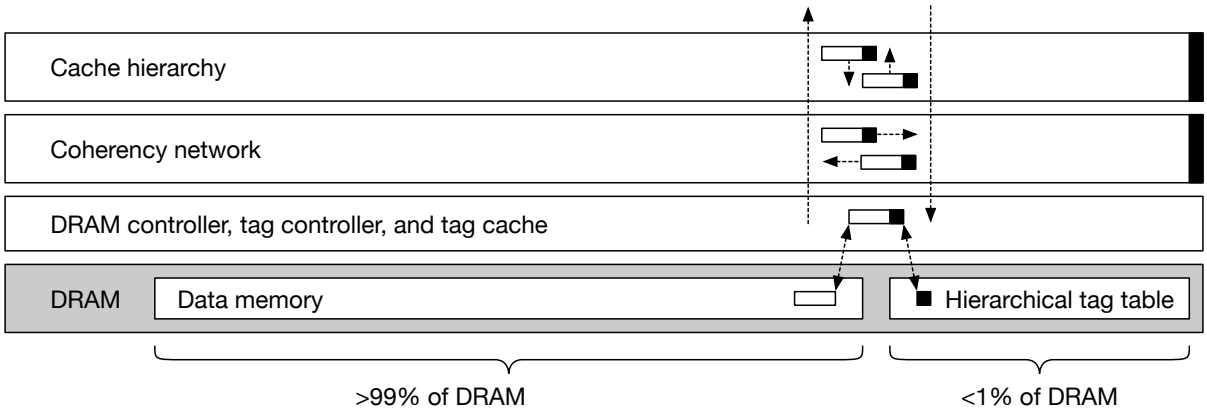

Figure 3.2: A DRAM-affine tag controller and tag cache maintain a non-addressable in-memory hierarchical tag table. Memory moving through the ordinary memory hierarchy, including cache coherency network, L3, L2, and L1 caches hold tags with their corresponding cache lines.

of the table contain no valid tags. In this model, little expense is paid if no tags are used, as all results are returned from the highest level of the table. DRAM traffic gradually increases as the proportion of valid tags increases – i.e., as pointer density increases. Practical experimentation shows that this technique generates minimal DRAM overhead for most common workloads.

3.2 Capability Compression

Architecturally, capabilities consist of a series of fields of natural integer register size for the architecture. For 64-bit systems, this includes a 64-bit address, lower bound, and upper bound, as well as additional bits dedicated to storing permissions, the object type, and so on. In order to reduce the memory footprint of a capability from $4\times$ the natural integer size to $2\times$ the natural integer size, capability bounds compression is utilized, as well as shrinking the sizes of the permissions and object-type fields. We have explored a number of such schemes, with the most recent being CHERI Concentrate [17].

Bounds compression is possible because in-bounds pointers have substantial redundancy with the lower- and upper-bound addresses (Figure 3.3). In addition, stronger alignment requirements on bounds allow safe use of less precise bounds, at the cost of internal allocator fragmentation. The CHERI Concentrate representation utilizes a floating-point-style representation for bounds, offering high precision for smaller memory allocations, and then gradually increasing alignment requirements for lower and upper bounds as memory allocations grow. Key design concerns included:

- Permitting a range of microarchitectural implementations, including partial or full decompression at the time that capabilities are loaded, and re-compression on store.
- Ensuring that there is no additional load-to-use delay when capabilities are loaded.

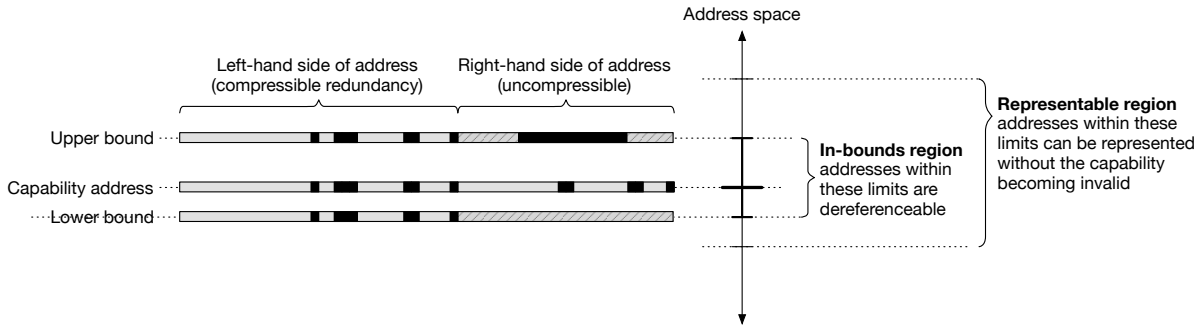

Figure 3.3: CHERI Concentrate exploits redundancy between a capability's address and bounds to reduce in-memory overhead. A key design requirement is supporting the construction of (but not access via) out-of-bounds pointers, which are routinely found in executing real-world C code.

- Allowing fast parallel bounds checks to take place alongside memory accesses without introducing additional delay or potential side channels.
- Allowing fast concurrent representability checks to take place during capability manipulation so that failure detection does not introduce additional delay.
- Ensuring that where precision is reduced, this does not lead to substantial additional memory fragmentation, incompatibility, or insecurity for real-world software workloads.

Chapter 4

Formal Modeling and Verification

Formal modeling and verification are essential parts of the engineering process, and of the resulting artifacts of the CHERI project. Our CHERI extensions to both the 64-bit MIPS and 32/64-bit RISC-V ISAs are defined in the Sail modeling language [1] (and previously in L3 [6]). Sail is a clean engineer-friendly first-order imperative language for ISA specification, with lightweight dependent types (type-checked using SMT) for static type-checking of bitvector lengths; it has also been used to give complete sequential ISA definitions for ARMv8-A (automatically translated from the Arm-internal ASL definition [11]) and for RISC-V, and for ISA semantics integrated with architectural concurrency models. The Sail CHERI architecture models are available as open source:

- CHERI-MIPS: `https://github.com/CTSRD-CHERI/sail-cheri-mips`
- CHERI-RISCV: `https://github.com/CTSRD-CHERI/sail-cheri-riscv`

For example, Figure 4.1 shows the Sail definition of the various CHERI-MIPS *Load Integer via Capability Register* instructions, e.g. `CLB rd, rt, offset(cb)`, which loads a byte from the virtual address specified by the capability in register `cb` plus a register and literal offset. The Sail definition includes a clause of the `ast` abstract instruction type (Line 1), a clause of the `execute` function that defines how these instructions behave (Lines 3–27), and clauses of the `decode` function that maps machine-code words to abstract instructions (Lines 29–42). The `execute` clause makes explicit the checks that the capability is valid, is not sealed and permits loading (Lines 6–11), and that the computed virtual address is within the bounds and suitably aligned (Lines 17-22) along with the exceptions raised if each fails.

Sail and L3 are used to generate, from these primary models:

- Reference documentation, automatically incorporated into the CHERI ISA specification [14];
- Executable ISA-level simulators, in C and OCaml, used as oracles to test hardware against and for software bring-up;
- Hardware instruction test cases [2], used for hardware testing;
- SMT-LIB definitions of the architecture, used for SMT checking of intended properties; and

```
union clause ast = CLoad : (regno, regno, regno, bits(8), bool, WordType)

function clause execute (CLoad(rd, cb, rt, offset, signext, width)) = {
  checkCP2usable();
  let cb_val = readCapRegDDC(cb);
  if not (cb_val.tag) then
    raise_c2_exception(CapEx_TagViolation, cb)
  else if cb_val.sealed then
    raise_c2_exception(CapEx_SealViolation, cb)
  else if not (cb_val.permit_load) then
    raise_c2_exception(CapEx_PermitLoadViolation, cb)
  else {
    let 'size   = wordWidthBytes(width);
    let cursor  = getCapCursor(cb_val);
    let vAddr   = (cursor + unsigned(rGPR(rt)) + size*signed(offset)) %
      pow2(64);
    let vAddr64 = to_bits(64, vAddr);
    if (vAddr + size) > getCapTop(cb_val) then
      raise_c2_exception(CapEx_LengthViolation, cb)
    else if vAddr < getCapBase(cb_val) then
      raise_c2_exception(CapEx_LengthViolation, cb)
    else if not (isAddressAligned(vAddr64, width)) then
      SignalExceptionBadAddr(AdEL, vAddr64)
    else {
      let pAddr = TLBTranslate(vAddr64, LoadData);
      memResult : bits(64) = extendLoad(MEMr_wrapper(pAddr, size), signext);
      wGPR(rd) = memResult;
    } } }

function clause decode (0b110010 @ rd:regno @ cb:regno @ rt:regno @
  offset:bits(8) @ 0b000) = Some(CLoad(rd,cb,rt,offset,false,B)) /*CLBU*/
function clause decode (0b110010 @ rd:regno @ cb:regno @ rt:regno @
  offset:bits(8) @ 0b100) = Some(CLoad(rd,cb,rt,offset,true, B)) /*CLB*/
function clause decode (0b110010 @ rd:regno @ cb:regno @ rt:regno @
  offset:bits(8) @ 0b001) = Some(CLoad(rd,cb,rt,offset,false,H)) /*CLHU*/
function clause decode (0b110010 @ rd:regno @ cb:regno @ rt:regno @
  offset:bits(8) @ 0b101) = Some(CLoad(rd,cb,rt,offset,true, H)) /*CLH*/
function clause decode (0b110010 @ rd:regno @ cb:regno @ rt:regno @
  offset:bits(8) @ 0b010) = Some(CLoad(rd,cb,rt,offset,false,W)) /*CLWU*/
function clause decode (0b110010 @ rd:regno @ cb:regno @ rt:regno @
  offset:bits(8) @ 0b110) = Some(CLoad(rd,cb,rt,offset,true, W)) /*CLW*/
function clause decode (0b110010 @ rd:regno @ cb:regno @ rt:regno @
  offset:bits(8) @ 0b011) = Some(CLoad(rd,cb,rt,offset,false,D)) /*CLD*/
```

Figure 4.1: Sail definition of the CHERI-MIPS *Load Integer via Capability Register* instructions

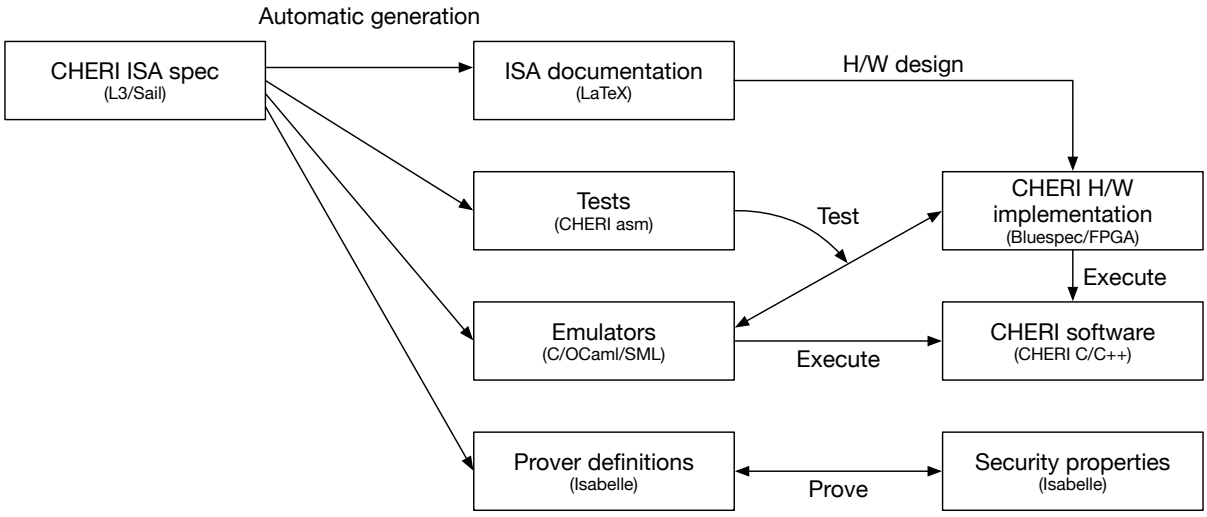

Figure 4.2: The main artifacts of the CHERI engineering process. Those in the central column are all automatically generated from the Sail (and previously L3) formal ISA specifications. The CHERI hardware design is tested against the generated emulators, using both auto-generated and (not shown) manually written tests. The CHERI software stack, including adaptions of Clang and FreeBSD, is developed by running above the generated emulators, the hardware, and (not shown) a QEMU emulator. The security properties are stated and proved in terms of the automatically generated Isabelle version of the ISA specification.

- Theorem-prover definitions of the architecture, in Coq, Isabelle/HOL, and HOL4, used to state and prove security properties.

Figure 4.2 illustrates these and how they are used in the CHERI engineering process. Importantly, developing these Sail and L3 definitions did not require expertise in semantics or theorem proving, so the researchers and engineers who would otherwise write a prose/pseudocode architecture document could write and own them. As is familiar from other uses of formal specification, this low-cost activity already brought several benefits, even before any proof work was undertaken. The security properties we define for CHERI-MIPS:

1. Capture the memory access intentions of the instructions;

2. Capture the reachable-capability monotonicity property, that arbitrary code, if given some initial permissions, cannot increase those during its execution (up to the point of any domain transition);

3. Capture the property that arbitrary code, if not initially given permission to access particular system registers and memory regions, leaves them invariant as that code is executed; and

4. Capture the guarantees one has (and the required assumptions) when executing an untrusted subprogram within a controlled isolation boundary.

Finally, we mathematically prove these security properties, with machine-checked Isabelle proof, in a way that is scalable to the entire ISA and that can be integrated in the ISA design process. This gives a level of confidence that is not achievable with testing alone. More details of all this are in the Technical Report [10].

Chapter 5

CHERI Software Model

CHERI capabilities are an architectural primitive that can be used for a variety of software purposes up and down the software stack, with potential uses in firmware and boot loaders, operating systems, language runtimes, CHERI-specific compartmentalization libraries, and compiler-generated code for the C and C++ application programs [3, 4, 5, 9, 15, 16]. In our research, we have pursued two central use cases of CHERI capabilities within current C/C++-language software stacks:

Fine-grained memory protection By utilizing capabilities instead of integers to implement C/C++-language pointers, and through modest extensions to the operating system and language runtime, we have implemented strong and efficient spatial, referential, and temporal memory safety for these traditionally memory-unsafe languages (Section 5.1).

Scalable software compartmentalization Capabilities provide an alternative means to construct the software isolation and controlled communication required to implement compartmentalized software designs. Unlike MMU-based compartmentalization (i.e., implemented using virtual memory), capability-based techniques allow for more granular and scalable data sharing, as well as a single-address-space programming model (Section 5.2).

In the remainder of this section, we consider how architectural capabilities can support these goals in software design. We describe our specific software prototypes in greater detail in Chapter 6.

5.1 Fine-Grained Memory Protection

The underlying principle in CHERI C and C++ memory protection is to implement pointers (both explicit in the language and implied in the runtime environment) using capabilities. We define two new compilation modes, with corresponding C-language interpretations, calling conventions, Application Binary Interfaces (ABIs), and so on:

Pure-capability code implements all C/C++ pointer types, as well as all implied pointers (e.g., return addresses, the stack pointer, and so on) using capabilities. This is an ABI-disruptive change, as pointer size has increased, changing the in-memory layout of data

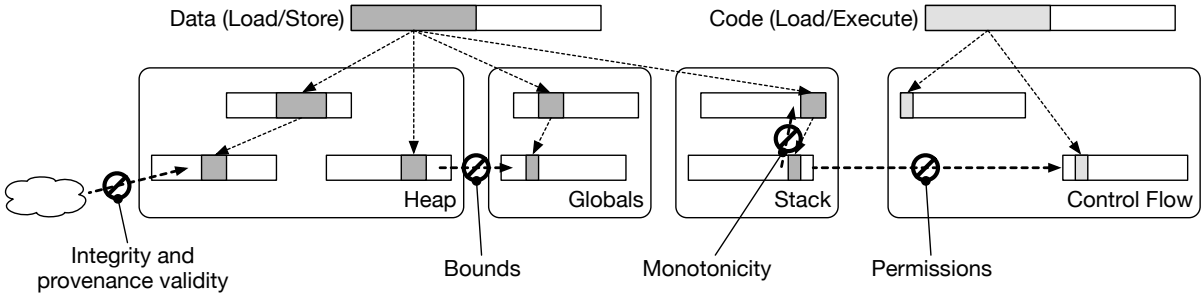

Figure 5.1: CHERI capabilities provide integrity and provenance validity, bounds, monotonicity, and permissions-related protections for a variety of pointer and memory types in the C/C++ runtime environment

structures, etc., just as 64-bit code has a different ABI from 32-bit code. Because pointers and integers are implemented using different types, additional care must be used so that pointer values retain tags where intended; for example, the C type `uintptr_t` (implemented using the hardware capability type) must be used to hold values that could be integers or pointers, as **`long`** (implemented using the hardware integer type) has room only for the address portion of a pointer, not its metadata and tag. Overall, however, relatively little code experiences disruption with the introduction of strong memory safety.

Hybrid-capability code implements pointer types using integers by default, interpreted with respect to a global default data capability (DDC) able to address code, globals, heap, and stack(s). When special language-level annotations are added to program source, specific pointers or pointer types are instead implemented using capabilities, offering additional protection. Hybrid-capability code is primarily for use in specialized environments: the boot loader, early kernel startup, and in code bridging communications between legacy code and pure-capability code (such as ABI compatibility layers).

5.1.1 Pure-Capability C/C++: Spatial Memory Safety

For the purposes of this document, we focus primarily on pure-capability code, which has an easier deployment path through recompilation and – for selected pieces of software only – minor source-code adaptations. In pure-capability code, all C/C++ pointers experience the spatial protection properties associated with capabilities (Figure 5.1):

Integrity and provenance validity The architecture enforces that all valid code or data pointers be derived from other valid pointers through valid transformations. This prevents the injection of pointer values over the network and detects the in-memory corruption of code or data pointers.

Bounds When suitably narrowed, bounds prevent erroneous manipulation and use of pointers from accessing an unintended object. Bounds must be narrowed by the software stack, capturing the intent of the program. Typically, bounds narrowing will be automatically inserted by the compiler (for stack allocations or upon taking references to sub-objects)

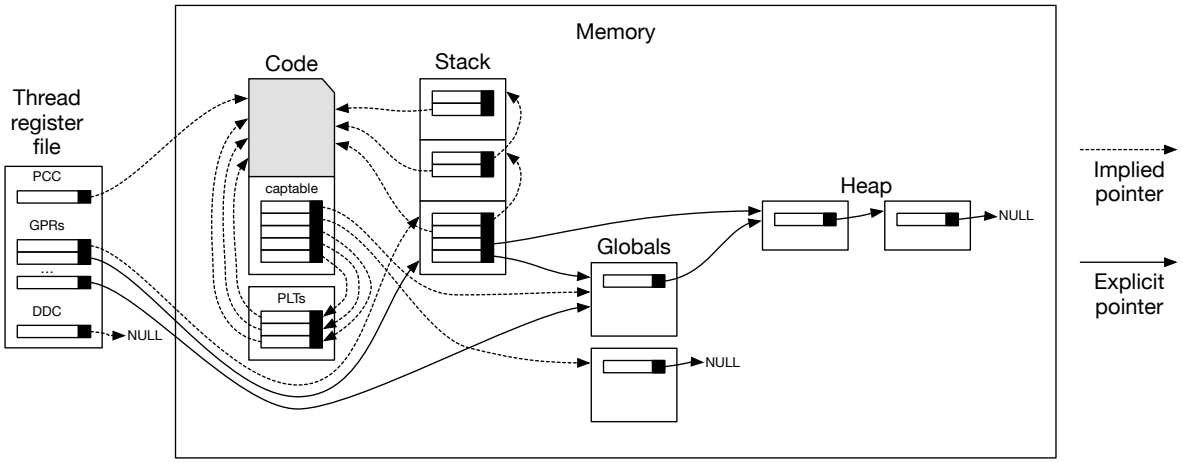

Figure 5.2: For any memory in the address space to be reachable, executing code must have some capability path from its register file – whether an implied pointer maintained by the C runtime or compiler-generated code, or explicitly via a language-level pointer.

or the runtime library (for heap allocations) or kernel system-call interface (for `mmap`); few source changes are required.

Permissions When suitably masked, permissions prevent pointers from being used for architectural purposes other than for what they were intended. For example, an improperly manipulated data pointer cannot be used for code execution. Masking must also be done by the software stack, but, again, typically such masking is carried out in only a few places (notably, the runtime loader and kernel).

Monotonicity Monotonicity prevents the broadening of bounds or increasing of permissions on pointers. This ensures that a pointer to a memory allocation or function cannot be improperly changed to include other objects or code by virtue of a software bug.

Figure 5.2 illustrates an abstract program execution environment – e.g., over a bare-metal runtime, or in a pure-capability UNIX process – and how capabilities can be configured to grant access to key program components and data structures – but nothing more. The underlying software runtime – typically some combination of kernel user-process setup and userspace runtime linker – will configure initial memory mappings, installing capabilities suitable to access a variety of memory types. By minimizing bounds and permissions, individual pointer privileges can be minimized, reducing the impact of potential improper uses, and causing many failure modes (such as pointer corruption and buffer overflows) to trigger hardware exceptions.

The pure-capability execution environment will normally have the default data capability (DDC) set to a NULL pointer, requiring that all pointer dereferences occur explicitly via capabilities: attempts at integer pointer dereferences will trigger an exception. The program-counter capability (PCC) will also be limited to constrain potential control flow, with only valid control-flow pointers having the execute permission, and bounds limited to prevent use of inappropriate memory for code execution.

These protections are not limited to explicit language-level pointers such as memory mappings, stack allocations, heap allocations, and globals. They can also be used by the C/C++-language runtime to protect its internal data structures, limiting the effects of compiler code-generation bugs as well attacks on those internal structures. Figure 5.3 illustrates a broad variety of capability use cases in the pure-capability run-time environment. In each case, capabilities enforce strong integrity, provenance validity, bounds checks, permission checks, and monotonicity on internal pointer types. This includes return addresses and C++ vtable pointers, which are frequently subject to memory-corruption attacks in current exploit chains.

5.1.2 Pure-Capability C/C++: Temporal Memory Safety

In addition to imposing spatial protection (bounds and permission checks) and referential protection (integrity and provenance validity enforcement), CHERI can also be used to implement strong C/C++-language temporal memory safety. Capability metadata and protection properties directly support reliable temporal memory safety:

Capability tags Tags provide an architectural mechanism by which to certainly, precisely sweep for pointers in system registers and memory.

Integrity and provenance validity These properties ensure that pointers cannot be improperly introduced (or re-introduced) to unallocated or reallocated memory.

Bounds Bounds ensure that pointers point to a specific memory allocation or sub-allocation, thereby removing a traditional source of ambiguity when analyzing heap arenas.

Permissions A software-defined permission can be used to distinguish different notions of pointer ownership, including ownership by the memory allocator.

Monotonicity Monotonicity prevents pointers from having their bounds and permissions modified to include other memory allocations, or to improperly mark them as owned by the memory allocator.

These properties enable a variety of temporal-safety techniques including (non-moving) garbage collection and *sweeping revocation*. Revocation-based temporal safety has programs continue to utilize explicit `free()` calls to release memory but delays reuse of address space (e.g., by subsequent `malloc()` calls) until any outstanding capabilities to it have been revoked (replaced with non-dereferenceable values) in registers and memory. (The allocator itself may retain access; capability permissions are used to preventing revocation of capabilities used by the allocator itself.) Revocation sweeps are infrequent, as freed address space (but not necessarily any associated physical pages) is held in *quarantine* until enough has accumulated to merit. A variety of architectural techniques can be used to accelerate the searches, including tracking capability spread through memory using capability dirty bits in the page table (which are set whenever valid capabilities are written to the page), and efficient scanning of memory for valid tags using explicit tag-load instructions.

Reference	Type	Origin of bounds	Bounds granularity
execve-time mappings	Implied	Kernel	Mapping or structures
ELF auxiliary args	Explicit	Kernel	Array and individual args
ARGV	Explicit	Kernel	Array and individual args
environ	Explicit	Kernel	Array and individual args
mmap mappings	Explicit	Kernel	Mapping
Program counter	Implied	Run-time linker	Mapping or function
Return addresses	Implied	Run-time linker	Mapping or function
PLT pointers	Implied	Run-time linker	PLT entries
Captable (GOT) pointer	Implied	Run-time linker	Captable mapping
TLS segment pointers	Implied	Run-time linker	TLS segment
C++ vtable pointers	Implied	Run-time linker	Vtable
Stack pointers	Implied	Kernel or thread library	Stack segment
Function pointers	Explicit	Run-time linker	Mapping or function
Stack allocation pointers	Explicit	Compiler-generated code	Stack allocation
Heap allocation pointers	Explicit	Heap allocator	Memory allocation
Global variable pointers	Explicit	Run-time linker	Global variable
TLS variable pointers	Explicit	Run-time linker	TLS segment[1]
Sub-object pointers	Explicit	Compiler-generated code	C/C++ struct field size
Varargs pointers	Explicit	Compiler-generated code	Varargs array

Figure 5.3: CHERI capabilities can be substituted for integers in implied and explicit pointers throughout the C/C++-language and its runtime.

5.2 Scalable Software Compartmentalization

Conventional MMU-based software compartmentalization decomposes larger software applications into components that run in isolated processes, linked only by controlled communication implemented using Inter-Process Communication (IPC). This widely deployed technique, found in applications ranging from Google's Chromium web browser to most Apple iOS applications, limits the impact of software compromise by reducing rights and further attack surfaces available to attackers. This technique is especially important because it provides resilience in the presence of not only exploits for unknown vulnerabilities in known classes (such as buffer overflows), but also protects against future as-yet undiscovered classes of vulnerability and exploit techniques. However, MMU-based compartmentalization designs impose substantial scalability limits due to utilizing multiple address spaces and page-granularity sharing: the number of compartments and their communication is severely limited, with performance significantly impacted as the number of compartments and their communication grow.

CHERI capabilities permit the construct of isolation and controlled sharing within address spaces, offering potentially greater compartmentalization scalability. Compartments are con-

[1]More ideally, pointers to thread-local variables would have bounds set to those of the variable rather than of the TLS segment. However, we have not yet experimented with adding this implied GOT-like level of indirection.

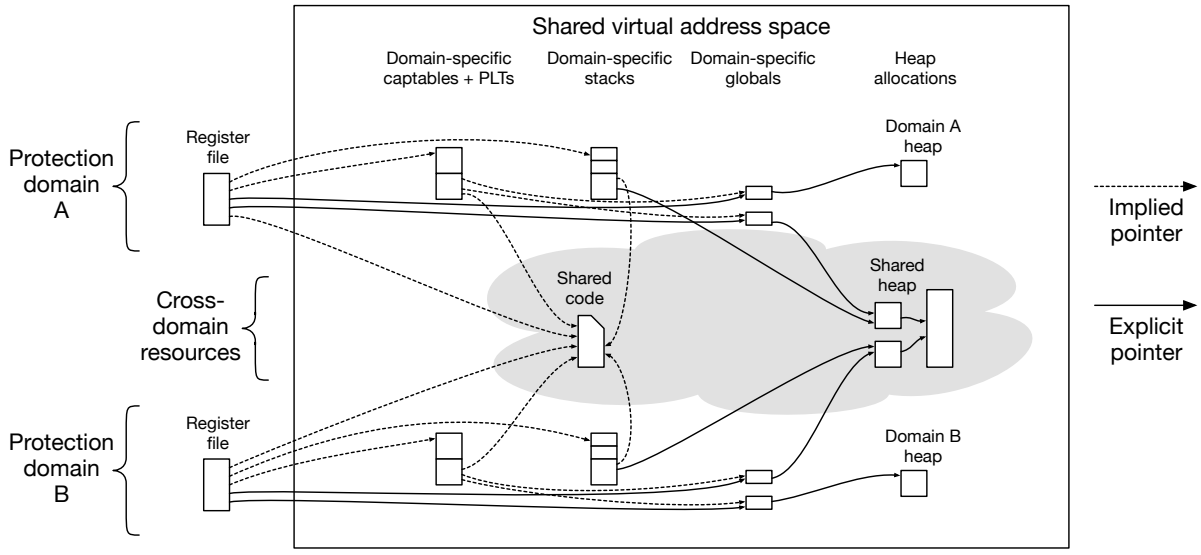

Figure 5.4: Compartments are constructed from closed subgraphs of the global capability graph within a shared virtual address space. Cross-compartment sharing of code and data can be achieved by delegating capabilities for shared resources across multiple compartments.

structed utilizing closed graphs of capabilities, in which an executing compartment has no access to the resources of other compartments, nor broader system memory (Figure 5.4). Bounds and permissions ensure that capabilities assigned to compartments grant access only to intended resources; monotonicity ensures that these rights cannot be modified to include other resources. Temporal safety is important to ensure that data and capabilities do not improperly leak between compartments when memory is freed and reused.

Switching between compartments can be implemented using one of two architectural mechanisms for controlled non-monotonicity (Section 2.7): exception handling, which gives access to additional exception-handling capabilities; or a special jump instruction (`CCall`) that atomically unseals and jumps to a pair of code and data capabilities whose object types match. Both architectural mechanisms provide a means by which available capabilities can be widened, but only when executing previously determined code paths. The jump-based mechanism avoids the cost of exceptions (which is microarchitecturally significant) and avoids the need for a privileged ring transition. It also allows the possibility of more scalable software designs: multiple implementations of domain transition can co-exist within a single address space.

The semantics of protection domains and domain transitions are flexible, as these architectural primitives support a variety of potential software uses. These include synchronous function-call-like semantics for domain transition, as well as asynchronous message passing. Compartmentalization models could more resemble libraries in the former case, and processes in the latter, depending on the implementation.

Chapter 6

CHERI Software-Stack Prototypes

We have developed a reference software stack for the CHERI architecture exploring several key software design dimensions opened up by capabilities. This work has had a number of aims, including playing an essential part in our hardware-software co-design effort to develop CHERI, to allow evaluation and demonstration of the CHERI approach at scale, and to act as templates for use. The reference stack includes:

CHERI Clang/LLVM/LLD An extended version of the Clang/LLVM compiler suite and LLD linker that are able to compile and link hybrid-capability and pure-capability code for multiple CHERI-enabled architectures.

CHERI GDB An extended version of the GDB debugger that is able to debug hybrid-capability and pure-capability code on multiple CHERI-enabled architectures.

CheriFreeRTOS An extended version of the FreeRTOS embedded operating system that is able to be compiled as pure-capability code, offering spatial and referential memory safety.

CheriOS microkernel An experimental CHERI-specific nanokernel and microkernel illustrating a potential set of design choices available when CHERI is an essential part of an OS design, and its use is maximized. This is a single-address-space, asynchronous message-passing OS intended to support extremely granular compartmentalization side-by-side with strong memory safety.

CHERI Hafnium An experimental extended version of Google's ARMv8-A Hafnium hypervisor that is compiled as pure-capability code, offering spatial and referential memory safety for the hypervisor itself. It is also able to host CHERI-aware virtual machines.

CheriBSD kernel An extended version of the FreeBSD operating system whose kernel can be compiled either as hybrid-capability or pure-capability code, offering different degrees of kernel memory protection. The CheriBSD kernel is also able to host legacy, hybrid-capability, and pure-capability userspace environments. The pure-capability process environment is known as CheriABI, and is a new OS ABI based on ubiquitous userspace use of architectural capabilities. CheriBSD is also able to offer optional temporal memory

safety for (non-stack) allocations in pure-capability userspace applications. In addition, CheriBSD provides a `libcheri` intra-process compartmentalization framework, which allows library-like components to operate in compartments within a larger application process.

CheriBSD hybrid userspace An extended version of the FreeBSD userspace that is minimally modified to support hybrid-capability code execution, including modest additions to the C runtime (CRT) and system libraries (including `libc`).

CheriBSD CheriABI userspace An extended version of the FreeBSD userspace that supports pure-capability execution, with modest further extensions.

CheriBSD applications A set of extended applications able to operate in the CheriABI process environment, including integrated FreeBSD programs such as OpenSSH, and also third-party applications such as Apple's WebKit and the PostgreSQL database. These all operate with full spatial, referential, and temporal memory safety.

This software stack demonstrates a number of points in the design space. It explores varying degrees of incorporation and adoption of CHERI protection illustrating CHERI's incremental adoptability properties. It also illustrates architectural neutrality for both the CHERI protection model and software designed for it. In the following sections, we further elaborate certain key design choices to illustrate the impact of CHERI on off-the-shelf software design.

6.1 Compiler and Toolchain

We have extended the Clang, LLVM, and LLD compiler and toolchain to support the CHERI protection model, targeting both hybrid-capability and pure-capability compilation modes. This required a number of changes:

- Better differentiate integer and pointer types in Clang code generation, in its use of the LLVM Intermediate Representation (IR).
- Adapt the compiler to no longer make various assumptions about pointers, such as conflating pointer size and addressable range.
- Instruct the compiler to use capabilities for various types of implied pointers (Figure 5.3). This includes stack pointers and return addresses, but also C++-specific constructs such as pointers-to-members, pointers to vtables and the entries within these vtables.
- Introduce bounds setting at various points in the compilation process, especially as relates to stack allocations and sub-objects. In the case of stack allocation, avoid generating capabilities except where pointers may escape local scope leading to potential overflow.
- Add minor C-language extensions in the form of new qualifiers (e.g., to qualify a pointer type as implemented with capabilities in the hybrid mode, or to disable sub-object bounds on a type), as well as new built-ins allowing capability properties to be queried and modified (e.g., to limit bounds and permissions).

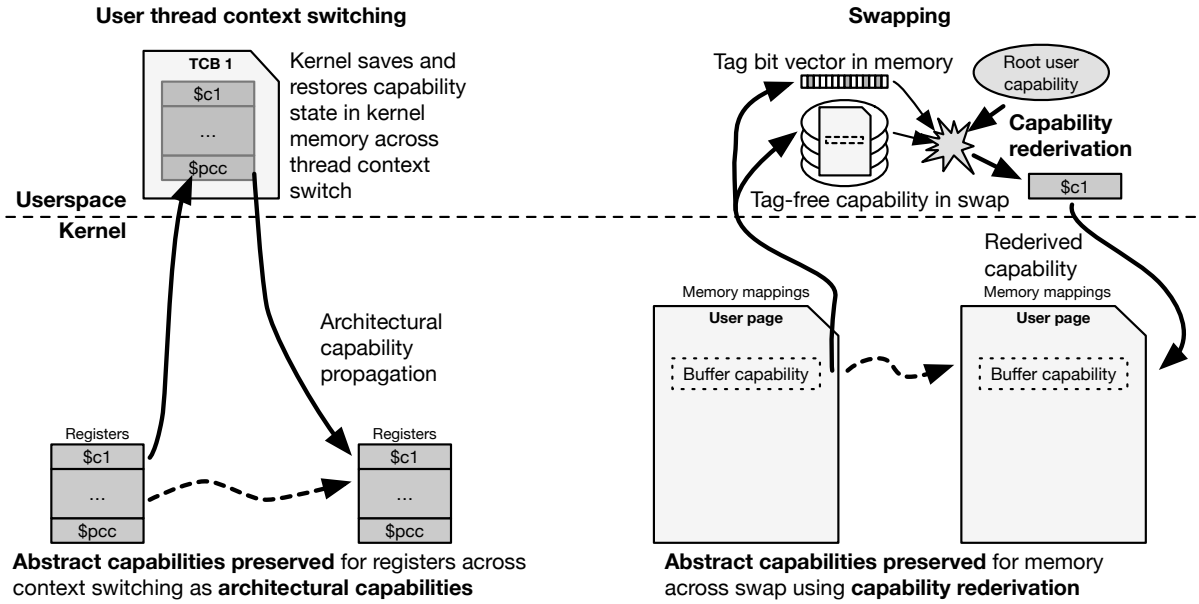

Figure 6.1: The kernel implements the notion of an *abstract capability*: despite the mechanics of virtual addressing and the UNIX process model, programmers work against a uniform capability abstraction within virtual address spaces, which largely aligns with their expectations for current integer pointer behavior.

- Introduce new code generation in the backend to implement capability types, as well as new ABIs.
- Update optimization passes to be aware of capabilities – especially in later phases where existing DAG pattern recognition may be disrupted, or where optimizations may cause tags to be improperly preserved or lost.
- Add new linkage metadata and modified code generation to support tightly bounded globals and restricted PCC bounds, especially with dynamic linkage.
- Add compiler diagnostics to detect idioms that are not compatible with hybrid-capability or pure-capability C/C++.

6.2 Operating System

We have similarly extended the FreeBSD operating system to support the CHERI protection model. This fork of FreeBSD is known as CheriBSD, and is maintained for CHERI adaptations of multiple architectures. We had a number of design goals:

- Support strong spatial, referential, and (non-stack) temporal memory safety throughout UNIX, including both kernel and userspace.

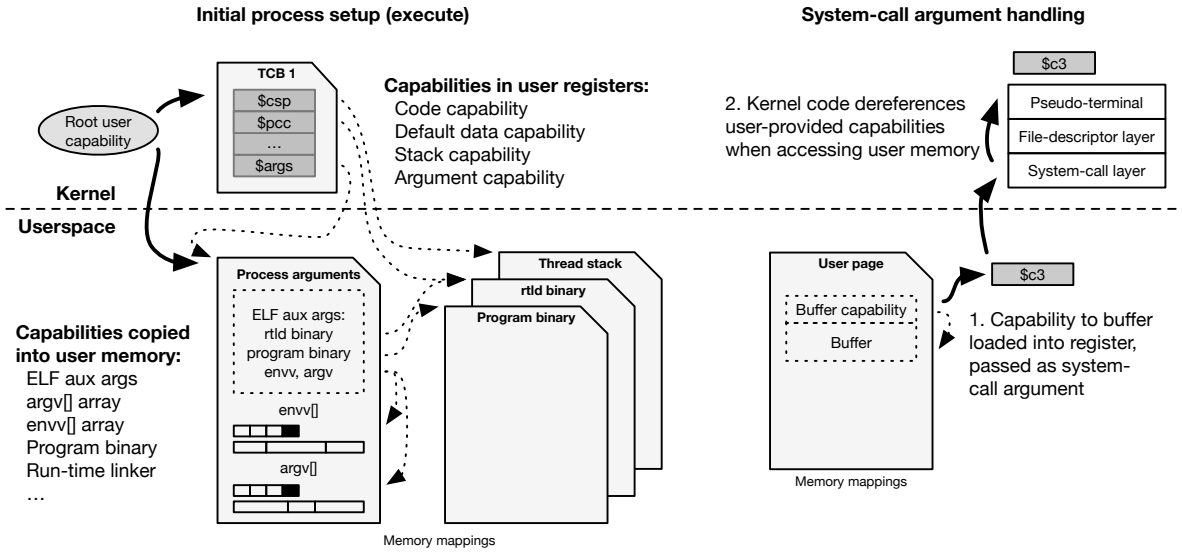

Figure 6.2: During pure-capability process startup, initial capability rights for memory mappings are granted to user code; these and further memory mappings are the only capabilities to which that user code will have access. When system-call arguments are copied in, user-provided capabilities are utilized for all memory accesses, to ensure that confused-deputy attacks (in which the kernel accesses user memory despite the user process not intentionally requesting that use) are prevented.

- Implement the concept of an *abstract capability* throughout kernel and userspace: despite interruptions in architectural provenance (e.g., because a capability is swapped to disk and back again), provide the programmer-facing appearance of continuity of provenance validity. This is similar to the notion of an abstract user address space, in which valid virtual addresses appear to always be dereferenceable, even though in practice the OS may not yet have demand paged contents from a disk file, zero filled the memory, or performed a pending copy-on-write. Only the regions of address space for which capabilities are explicitly assigned to the user process will be reachable – e.g., as a result of the initial process memory mappings, or later system calls (Figure 6.1).

- Utilize capability intentionality to limit confused-deputy attacks. For example, although the kernel may have the right to access the entire user address space, always employ user-provided capability arguments to system calls when accessing user memory on its behalf (Figure 6.2). This prevents the kernel from accidentally bypassing spatial protections configured by user memory allocators.

- Illustrate the potential adoption benefits of CHERI's hybrid capability model by showing how differing degrees of CHERI integration can co-exist within a single system. Examples include: (1) supporting hybrid-capability and pure-capability kernels, both hosting hybrid-capability and pure-capability user processes, as well as conventional CHERI-unaware processes; and (2) Maintaining current MMU-based OS abstractions and services such as processes and demand paging expected by off-the-shelf UNIX applications.

- Utilize the intra-address-space protection properties of CHERI to support single-address-space software compartmentalization, improving performance and programming model for fine-grained software compartmentalization.
- ... While simultaneously avoiding substantial disruption to key OS design choices, Application Programming Interfaces (APIs), file formats, and management models.

A key goal has been to utilize CHERI's hybrid model by continuing to support MMU-based software structures, such as the conventional UNIX supervisor protection and process models, while enabling varying degrees of capability use in the operating system and applications. Baseline CHERI support required the following general classes of kernel changes:

- Initialize CHERI support in the early architecture-dependent boot.
- Implement capability-aware context switching for kernel and user threads, saving and restoring general-purpose and selected special-purpose capability registers.
- Maintain tags in user and kernel virtual memory, ensuring that when pages undergo copy-on-write, are swapped to disk, and so on, tags are retained.
- Ensure that tags are not retained where to do so would negatively impact security: for memory mappings of files, when copying packet data, and so on.
- Implement both hybrid-capability and pure-capability versions of the kernel, supporting varying degrees of kernel memory safety and C-language models.
- In the pure-capability kernel: Retain tags across in-kernel memory copies, so that kernel pointers remain tagged (e.g., during structure data copies). Better differentiate address and pointer types in the virtual-memory subsystem. Refine bounds and permissions on kernel pointers during kernel run-time linking and kernel memory allocation.
- With a hybrid-capability kernel: Annotate system-call argument pointer types as capabilities to maintain intentionality.
- Extend the kernel debugger to support debugging capability-related state.
- Extend signal delivery to report CHERI-related exceptions, and also provide CHERI-related diagnostics including a capability-extended register frame.
- Implement userspace debugger extensions to support debugging capability-based applications, including extensions to `ptrace(2)`, `truss(1)`, and also to the core-dump format (e.g., to save capability register values and memory tags).
- Implement CheriABI, a pure-capability userspace process environment in which all pointers (explicit or implied) are implemented using capabilities, including all system-call arguments.
- Implement support for (non-stack) temporal memory safety using sweeping revocation for user memory and user pointers stashed in the kernel on behalf of user processes.

- Correct kernel memory-safety bugs discovered as a result of pure-capability testing.

The following further changes were made to the userspace code base:

- Build two userspace library and application environments, one for the hybrid-capability ABI, and the other for the pure-capability ABI.
- Implement hybrid-capability and pure-capability C startup code to suitably set up the userspace execution environment following `execve(2)`.
- Extend the run-time linker to support execution of pure-capability binaries. This includes parsing the metadata emitted by the compiler/linker and using it to initialize bounded data and code pointers, as well as to provide isolation between shared objects. Add support for features such as lazy resolution of function symbols.
- Modify `libc` to preserve tags across memory copies, as well as memory-copy-like operations such as `qsort(3)`.
- For the pure-capability ABI: Better differentiate address and pointer types, especially with respect to use of `uintptr_t`. In `libc` and `libpthread`, refine bounds and permissions for memory allocations .
- Implement `libcheri`, a userspace compartmentalization runtime similar to a run-time linker.
- Correct userspace memory-safety bugs discovered as a result of pure-capability testing.

6.3 Applications

In order to develop, demonstrate, and evaluate the CHERI approach, we have ported a large number of userspace C and C++ libraries, utilities, and applications to CheriBSD. This includes the complete FreeBSD base operating system (system libraries, common programs such as shells, OpenSSH, and so on), as well as third-party applications such as the PostgreSQL database and WebKit web-rendering framework. These are all compiled using pure-capability C/C++ running over in the CheriABI process environment to provide strong spatial, referential, and (non-stack) temporal memory safety.

In most cases, this requires few code changes except as identified above: depending on code style and vintage, improvements in type use (e.g. adoption of `uintptr_t`) may be required where there is substantive confusion regarding integer and pointer types. In some cases, applications ship with their own memory allocators or other C runtime functions, such as custom memory copying and sorting functions. These require additional adaptation to introduce support for bounds in new allocators, and to ensure that tag bits are suitably preserved as intended for memory copying. More contemporary coding styles tend to require fewer (if any) code changes, as they will have been developed against 32-bit and 64-bit architectures, utilize recent C coding styles that incorporate proper function-prototype support, and so on.

We have also experimented with introducing CHERI support in the language runtime of the Java programming language to implement safer Foreign Function Interfaces (FFIs) [3]. In this prototype, Java Native Interface (JNI) components were compiled as pure-capability code. This allowed the Java Virtual Machine (JVM) to impose aspects of the Java memory-safety and sandboxing models on JNI code, including spatial and temporal safety in C/C++-language references to the Java heap. While a limited case study, this approach opens the door to a variety of new techniques to allow managed languages to more safely interact with other managed languages and also native code within a single virtual address space.

While OS code experiences little performance overhead utilizing CHERI, some application types are more sensitive to pointer-size growth – in particular, language runtimes. Here additional performance analysis and optimization may be required – for example, by choosing to utilize integer offsets to allocations rather than pointer types in some cases. This technique is already used by the Android Runtime (ART) and others to mitigate the expense of 64-bit pointers by utilizing 32-bit integer types instead. Some care is required to ensure that sacrificing the potential benefits of CHERI pointer protection in selected cases – for example, in target-language heap references – does not introduce undesired security vulnerabilities. For example, utilizing pure-capability code for all native C/C++ pointers and allocations in the language runtime will provide strong protection against many vulnerabilities and exploit techniques, even if target-language references utilize integer offsets within one or several capability-bounded target-language heaps.

Chapter 7

Further reading

General information on the CHERI project, including our papers, technical reports, hardware prototypes, software prototypes, and formal proofs, may be found on the CHERI website:

https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/

7.1 Architecture

The primary reference for the CHERI Instruction-Set Architecture (ISA) is the ISA specification; at the time of writing, the most recent version is CHERI ISAv7 [14]:

https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf

We published an article describing our research and development process for the CHERI architecture in the MIT Press book, *New Solutions for Cybersecurity* in 2017 [12]:

https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2017mit-cybersecurity-cheri-web.pdf

7.2 Formal Modeling

The CHERI formal specification, lightweight rigorous engineering, and formal statements and proofs of architectural security properties, are detailed in the following technical report [10]:

https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-940.pdf

7.3 Microarchitecture

We published a paper on efficient tagged memory at ICCD 2017:

https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201711-iccd2017-efficient-tags.pdf

We published a paper on the CHERI Concentrate compressed capability model in IEEE Transactions on Computers 2019 [17]:

https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2019tc-cheri-concentrate.pdf

7.4 Software

We published a paper on fine-grained C-language memory protection using the CHERI architecture at ASPLOS 2015 [4]:

https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201503-asplos2015-cheri-cmachine.pdf

We published a paper on fine-grained software compartmentalization using the CHERI architecture at IEEE SSP ("Oakland") 2015 [16]:

https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201505-ssp2015-cheri-compartment.pdf

We published a paper on CheriABI and the adaptation of a complete OS userspace and application suite to a pure-capability process environment at ASPLOS 2019 [5]:

https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201904-asplos-cheriabi.pdf

We published a paper on C-language pointer provenance, and the implications for software design, at POPL 2019 [8]; CHERI C was a case study in the practical enforcement of capability provenance-validity enforcement:

https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201901-popl-cerberus.pdf

We published a paper on the feasibility of temporal memory safety using CHERI capabilities at MICRO 2019 [19]:

https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201910micro-cheri-temporal-safety.pdf

Chapter 8

Future Research

Over nine years of hardware-software-model co-design, we have developed the CHERI protection model, its instantiation in multiple instruction-set architectures and processor microarchitectures, as well as new software models and fully elaborated prototypes of those models. However, many questions remain to be answered, leaving fertile ground for future research relating to CHERI and its application:

- What architectural extensions and microarchitectural techniques exist to further reduce overheads for spatial, referential, and temporal memory safety?
- How should CHERI interact with other emerging architectural and microarchitectural security models, such as enclave technologies?
- How can we evaluate the impact of CHERI on security, and also validate that concrete software artifacts are effective in using CHERI to improve security?
- How can CHERI be utilized to support memory safety and compartmentalization for higher-level managed languages, their Just-In-Time (JIT) compilers, and their Foreign Function Interfaces (FFIs)?
- What are the most appropriate new software structures to support fine-grained software compartmentalization within a single address space?
- Can CHERI support accurate and compatible C-language garbage collection for temporal safety and greater programmability, not just sweeping-based revocation?
- How should CHERI be used by, and more generally interact with, virtualization environments such as those found in contemporary cloud environments – including operational services such as virtual-machine migration?
- If hybrid compatibility with existing software stacks were no longer an issue, how would a software stack designed specifically with CHERI in mind be structured, implemented, and optimized?
- What architectural, microarchitectural, and software techniques can be used to reduce CHERI overhead for pointer-dense applications such as language runtimes?

- How can and should architectural CHERI support interact with microarchitectural side channels: can it assist with mitigating some by supporting additional constraints on speculative execution; and does it suffer from the same attacks due to the introduction of additional types of protection domain between which leakage must be controlled?

- Can CHERI be used to improve the interface with emerging non-volatile memory-mapped flash by providing more appropriate protection facilities – e.g., that protect pointers in persistent memory, or in scaling better than MMU-based techniques to anticipated large non-volatile memory sizes?

- How can the architectural formal models that we have developed (and are continuing to develop) be used to support verification of both supporting microarchitecture and the software stacks built over CHERI?

Bibliography

[1] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In ***POPL 2019****: Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, Jan. 2019. Proc. ACM Program. Lang. 3, POPL, Article 71.

[2] B. Campbell and I. Stark. Randomised testing of a microprocessor model using SMT-solver state generation. *Sci. Comput. Program.*, 118:60–76, 2016.

[3] D. Chisnall, B. Davis, K. Gudka, D. Brazdil, A. Joannou, J. Woodruff, A. T. Markettos, J. E. Maste, R. Norton, S. Son, M. Roe, S. W. Moore, P. G. Neumann, B. Laurie, and R. N. M. Watson. CHERI-JNI: Sinking the Java security model into the C. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2017)*, April 2017.

[4] D. Chisnall, C. Rothwell, B. Davis, R. Watson, J. Woodruff, S. Moore, P. G. Neumann, and M. Roe. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XX, New York, NY, USA, 2014. ACM.

[5] B. Davis, R. N. M. Watson, A. Richardson, P. Neumann, S. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka, A. Joannou, B. Laurie, A. T. Markettos, E. Maste, A. Mazzinghi, E. T. Napierala, R. Norton, M. Roe, P. Sewell, S. Son, and J. Woodruff. CheriABI: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment. In *Proceedings of the 24nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019)*, April 2019.

[6] A. C. Fox. Directions in ISA specification. In *ITP*, pages 338–344, 2012.

[7] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. M. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. Son, and A. T. Markettos. Efficient tagged memory. In *Proceedings of the 2017 IEEE 35th International Conference on Computer Design (ICCD)*, November 2017.

[8] K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell. Exploring C semantics and pointer provenance. In ***POPL 2019****: Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, Jan. 2019. Proc. ACM Program. Lang. 3, POPL, Article 67.

[9] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. Watson, and P. Sewell. Into the depths of C: elaborating the de facto standards. In *Proceedings of PLDI 2016*, June 2016.

[10] K. Nienhuis, A. Joannou, A. Fox, M. Roe, T. Bauereiss, B. Campbell, M. Naylor, R. M. Norton, S. W. Moore, P. G. Neumann, I. Stark, R. N. M. Watson, and P. Sewell. Rigorous engineering for hardware security: formal modelling and proof in the CHERI design and implementation process. Technical Report UCAM-CL-TR-940, University of Cambridge, Computer Laboratory, Sept. 2019.

[11] A. Reid. Trustworthy Specifications of Arm v8-A and v8-M system Level Architecture. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD 2016)*, pages 161–168, October 2016.

[12] R. N. M. Watson, P. G. Neumann, and S. W. Moore. Balancing Disruption and Deployability in the CHERI Instruction-Set Architecture (ISA). In H. Shrobe, D. L. Shrier, and A. Pentland, editors, *New Solutions for Cybersecurity*, chapter 5. MIT Press/Connection Science, 2018.

[13] R. N. M. Watson, P. G. Neumann, J. Woodruff, J. Anderson, R. Anderson, N. Dave, B. Laurie, S. W. Moore, S. J. Murdoch, P. Paeps, M. Roe, and H. Saidi. CHERI: a research platform deconflating hardware virtualization and protection. In *Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012)*, March 2012.

[14] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, D. Chisnall, B. Davis, N. W. Filardo, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7). Technical Report UCAM-CL-TR-927, University of Cambridge, Computer Laboratory, June 2019.

[15] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera. Fast Protection-Domain Crossing in the CHERI Capability-System Architecture. *IEEE Micro*, 36(5):38–49, September 2016.

[16] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, May 2015.

[17] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. Norton, T. Bauereiss, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, A. T. Markettos, M. Roe, P. G. Neumann, R. N. M. Watson, and S. W. Moore. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Transactions on Computers*, 2019.

[18] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA 2014)*, June 2014.

[19] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. M. Watson, and T. M. Jones. CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (IEEE MICRO 2019)*, MICRO-52 '17, October 2019.