Technical Report

UCAM-CL-TR-877
ISSN 1476-2986

Number 877

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide

Robert N. M. Watson, David Chisnall,
Brooks Davis, Wojciech Koszek,
Simon W. Moore, Steven J. Murdoch,
Peter G. Neumann, Jonathan Woodruff

September 2015

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

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

Approved for public release; distribution is unlimited.
Sponsored by the Defense Advanced Research Projects
Agency (DARPA) and the Air Force Research Laboratory
(AFRL), under contracts FA8750-10-C-0237 ("CTSRD") as
part of the DARPA CRASH research program. The views,
opinions, and/or findings contained in this report are those of
the authors and should not be interpreted as representing the
official views or policies, either expressed or implied, of the
Department of Defense or the U.S. Government.

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

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

ISSN 1476-2986

Abstract

The *CHERI Programmer's Guide* documents the software environment for the Capability Hardware Enhanced RISC Instructions (CHERI) prototype developed by SRI International and the University of Cambridge. The Guide is targeted at hardware and software developers working with capability-enhanced software. It describes how to use the CHERI Clang/LLVM compiler suite and CheriBSD operating system – versions of the off-the-shelf LLVM suite and FreeBSD operating system adapted to use CHERI's protection features – as well as implementation details of both.

Acknowledgments

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

Ross J. Anderson	Jonathan Anderson	Ruslan Bukin	Gregory Chadwick
Nirav Dave	Lawrence Esswood	Khilan Gudka	Jong Hun Han
Alex Horsman	Alexandre Joannou	Asif Khan	Myron King
Chris Kitching	Ben Laurie	Patrick Lincoln	Anil Madhavapeddy
Ilias Marinos	A. Theodore Markettos	Ed Maste	Andrew W. Moore
Will Morland	Alan Mujumdar	Prashanth Mundkur	Robert Norton
Philip Paeps	Alex Richardson	Michael Roe	Colin Rothwell
John Rushby	Hassen Saidi	Hans Petter Selasky	Muhammad Shahbaz
Stacey Son	Andrew Turner	Richard Uhler	Munraj Vadera
Philip Withnall	Hongyan Xia	Bjoern A. Zeeb	

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

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

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

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

Contents

Chapter 1

Introduction

This is the *Programmer's Guide* for the Capability Hardware Enhanced RISC Instructions (CHERI) hardware-software system prototype. The Guide complements the *CHERI Instruction-Set Architecture* [13] (which specifies the CHERI ISA), the *BERI Hardware Reference* [15] (which describes the hardware prototype), and the *BERI Software Reference* [9] (which describes the BERI software development environment).

The *CHERI Programmer's Guide* describes the CHERI-extended Clang/LLVM toolchain [4] and CheriBSD, a CHERI-extended version of the FreeBSD operating system [5]. The Guide is intended to address the needs of hardware and software developers who are prototyping new hardware features or bringing up operating systems, language runtimes, and compilers on CHERI. Future iterations will continue to flesh out operational aspects of the CHERI processor and its use, as well as further details of software design philosophies and implementation approaches to employing CHERI.

1.1 Background

Capability Hardware Enhanced RISC Instructions (CHERI), developed by SRI International and the University of Cambridge, are security extensions for the 64-bit MIPS Instruction Set Architecture (ISA). The CHERI ISA provides direct processor support for fine-grained memory protection and scalable compartmentalization of (and within) system software and application software.

Whereas traditional CPU designs impose heavy performance and programmability penalties for employing fine-grained memory protection and compartmentalization, CHERI's ISA features support fast and easy memory safety and application compartmentalization of C-language systems software. These improvements are made possible by integrated processor support for continuous memory protection and enforcement using memory capabilities and the object-capability model. Contemporary software trusted computing bases (TCBs) such as operating-system kernels and language runtimes are particularly interesting targets, as CHERI will allow us to improve their security and reliability – and, therefore, the security and reliability of applications built on top of those services.

Detailed information on the CHERI ISA and possible uses, as well as related work, may be found in the *CHERI Instruction-Set Architecture* [13], including new coprocessor registers and instructions. The CHERI prototype is a reference implementation of the CHERI ISA, intended to help validate the approach through a complete system implementation. The CHERI

processor is based on the Bluespec Extensible RISC Implementation (BERI) FPGA soft core, and is implemented as an additional coprocessor. The distinction between BERI and CHERI is evolving; however, our hope is that BERI will be a reusable platform across multiple research projects relating to the hardware-software interface. We have open sourced both our hardware prototypes and software support for them in the form of FreeBSD device drivers, extensions to the Clang/LLVM compiler suite, and CheriBSD, a version of FreeBSD enhanced to utilize the CHERI ISA for fine-grained memory protection and compartmentalization.

1.2 Getting CHERI

Our CHERI hardware and software prototypes are distributed as open source via the BERI website:

```
http://www.beri-cpu.org/
```

1.3 Licensing

The BERI hardware design, simulated peripherals, and software tools are available under the BERI Hardware-Software License, a lightly modified version of the Apache Software License that takes into account hardware requirements.

We have released our extensions to the FreeBSD operating system to support BERI under a BSD license; initial support for BERI was included in FreeBSD 10.0, with further features appearing in FreeBSD 10.1. We have also released versions of FreeBSD and Clang/LLVM that support the CHERI ISA under a BSD license; these are distributed via GitHub – details appear in later chapters on installing and using these pieces of software.

We welcome contributions to the BERI project; however, we are only able to accept non-trivial changes when an individual or corporate contribution agreement has been signed. The BERI hardware-software license and contribution agreement may be found at:

```
http://www.beri-open-systems.org/
```

1.4 Publications

As our approach has evolved, and project developed, we have published a number of papers and reports describing aspects of the work. We published several workshop papers laying out early aspects of our approach:

- Our philosophy in revisiting of capability-based approaches is described in *Capabilities Revisited: A Holistic Approach to Bottom-to-Top Assurance of Trustworthy Systems*, published at the Layered Assurance Workshop (LAW 2010) [6], shortly after the inception of the project.

- Mid-way through creation of both the BERI prototyping platform, and CHERI ISA model, we published *CHERI: A Research Platform Deconflating Hardware Virtualization and Protection* at the Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012) [17].

- Jonathan Woodruff, whose PhD dissertation describes our initial CHERI prototype, published a workshop paper on this work at the CEUR Workshop's Doctoral Symposium on Engineering

Secure Software and Systems (ESSoS 2013): *Memory Segmentation to Support Secure Applications* [6].

We have also published a series of conference papers describing our hardware and software approaches in greater detail, along with evaluations of micro-architectural impact, software performance, compatibility, and security:

- In the International Symposium on Computer Architecture (ISCA 2014), we published *The CHERI Capability Model: Revisiting RISC in an Age of Risk* [18]. This paper describes our architectural and micro-architectural approaches with respect to capability registers and tagged memory, hybridization with a conventional Memory Management Unit (MMU), and our high-level software compatibility strategy with respect to operating systems.

- In the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015), we published *Beyond the PDP-11: Architectural support for a memory-safe C abstract machine* [1], which extends our architectural approach to better support convergence of pointers and capabilities, as well as to further explore the C-language compatibility and performance impacts of CHERI in larger software corpora.

- In the IEEE Symposium on Security and Privacy (IEEE S&P 2015), we published *CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization* [16], which describes a hardware-software architecture for mapping compartmentalized software into the CHERI capability model, as well as extends our explanation of hybrid operating-system support for CHERI.

- In the ACM Conference on Computer and Communications Security (CCS 2015), we published *Clean Application Compartmentalization with SOAAP* [3], which describes our higher-level design approach to software compartmentalization as a a form of vulnerability mitigation, including static and dynamic analysis techniques to validate the performance and effectiveness of compartmentalization.

We have additionally released several technical reports, including this document, describing our approach and prototypes. Each has had multiple versions reflecting evolution of our approach:

- The *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture* [11, 12, 13] describes the CHERI instruction set, both as a high-level, software-facing model and the specific mapping into the 64-bit MIPS instruction set. Successive versions have introduced improved C-language support, support for scalable compartmentalization, and compressed capabilities.

- This report, the *Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide* [10], describes in greater detail our mapping of software into instruction-set primitives in both the compiler and operating system; earlier versions of the document were released as the *Capability Hardware Enhanced RISC Instructions: CHERI User's Guide* [8].

- The *Bluespec Extensible RISC Implementation: BERI Hardware Reference* [14, 15] describes hardware aspects of our prototyping platform, including physical platform and practical user concerns.

- The *Bluespec Extensible RISC Implementation: BERI Software Reference* [7, 9] describes non-CHERI-specific software aspects of our prototyping platform, including software build and practical user concerns.

- The technical report, *Clean application compartmentalization with SOAAP (extended version)* [2], provides a more detailed accounting of the impact of software compartmentalization on software structure and security using conventional designs, with potential applicability to CHERI-based designs as well.

Further research publications and technical reports will be forthcoming.

1.5 Version History

1.0 An initial version of the *CHERI User's Guide* documented the implementation status of the CHERI prototype, including the CHERI ISA and processor implementation, as well as user information on how to build, simulate, debug, test, and synthesize the prototype.

1.1 Minor refinements were made to the text and presentation of the document, with incremental updates to its descriptions of the SRI/Cambridge development and testing environments.

1.2 This version of the *CHERI User's Guide* followed an initial demonstration of CHERI synthesized for the Terasic tPad FPGA platform. The Guide contained significant updates on the usability of CHERI features, the build process, and debugging features such as CHERI's debug unit. A chapter was added on Deimos, a demonstration microkernel for the CHERI architecture.

1.3 The document was restructured into hardware prototype and software reference material. Information on the status of MIPS ISA implementation was updated and expanded, especially with respect to the MMU. Build dependencies were updated, as was information on the CHERI simulation environment. The distinction between BERI and CHERI was discussed in detail. The Altera development environment was described in its own chapter. A new chapter was added that detailed bus and device configuration and use of the Terasic tPad and DE4 boards, including the Terasic/Cambridge MTL touch screen display. New chapters were added on building and using CheriBSD, as well as a chapter on FreeBSD device drivers on BERI/CHERI. A new chapter was added on cross-building and using the CHERI-modified Clang/LLVM suite, including C-language extensions for capabilities.

1.4 This version introduced improved Altera build and Bluespec simulation instructions. A number of additional C-language extensions that can be mapped into capability protections were introduced. FreeBSD build instructions were updated for changes to the FreeBSD cross-build system. Information on the CHERI2 prototype was added.

1.5 In this version of the *CHERI User's Guide*, several chapters describe the CHERI hardware prototype have been moved into a separate document, the *CHERI Platform Reference Manual*, leaving the User's Guide focused on software-facing activities.

1.6 This version updated the *CHERI User's Guide* for changes in the CheriBSD build including support for the CFI driver, incorporation of Subversion into the FreeBSD base tree, and non-root cross builds. It also added information on the `quartus_pgm` command, and made a number of minor clarifications and corrections throughout the document.

1.7 In this version, information on building and using FreeBSD/BERI was moved to the *BERI Software Reference*. A short chapter describing CheriBSD was retained, and updated to reflect a migration from Perforce to Github. Information on the CHERI Clang/LLVM compiler was updated to include new C-language extensions. CheriBSD build instructions were extended to support the the CHERI Clang/LLVM compiler. Information on CheriBSD extensions to FreeBSD was expanded.

1.8 - UCAM-CL-TR-851 This version of the *CHERI User's Guide* was made available as a University of Cambridge Technical Report. Information on CheriBSD was updated to reflect enhanced support for sandboxing, signal handling, and testing. A variety of changes were made to reflect open sourcing of the BERI/CHERI processors and associated software stacks. New CHERI Clang built-in functions for clearing capability-pointer tags and querying the capability-program counter are now documented.

1.9 This version of the *CHERI User's Guide* is timed for delivery at the end of the fourth year of the CTSRD Project. It provides significantly more detailed information on the CheriBSD kernel, including information on context switching, kernel support for userspace object capabilities, fault handling, and interfaces used by the userspace TCB.

1.10 This version of the document has been prepared as a draft delivery to DARPA.

1.11 The *CHERI User's Guide* has been renamed to the *CHERI Programmer's Guide*. This version of the document has been prepared for delivery to DARPA, and satisfies the 2015 annual deliverable A010 for what was formerly called the *CHERI User's Guide*. It accompanies and is consistent with the *CHERI Instruction-Set Architecture* Version 1.14.

The chapter on the Deimos demonstration operating system has been removed from the *CHERI User's Guide* to a separately distributed short report, *Deimos - CHERI Demo Operating System from Mars*.

The document has been broken into two parts: the first on CHERI Clang/LLVM, and the second on CheriBSD. New chapters on C-language support and compiler internals have been added on the abstract model, C compiler support, the LLVM and MIPS backend implementation, and CHERI's ABIs. The CheriBSD chapter has been broken up into three chapters on building and using the operating system, on kernel internals, and on userspace.

CHERI Clang/LLVM and CheriBSD build instructions have been updated to describe how to build for both 256-bit and 128-bit versions of the CHERI ISA.

`const` is no longer enforced in hardware. A new qualifier, `__input`, is introduced as a hardware-enforced non-disclaimable version of `const`.

CheriBSD's support for tracing CCall/CReturn using `ktrace` is now described.

CheriBSD's pure-capability CheriABI is now described: this is a process ABI in which all pointers passed between userspace and the kernel are as CHERI capabilities, rather than MIPS pointers. This allows support for pure-capability ABI binaries.

1.12 - UCAM-CL-TR-877 This version of the *CHERI Programmer's Guide* was made available as a University of Cambridge Technical Report. It accompanies and is consistent with the *CHERI Instruction-Set Architecture* Version 1.15 [13].

Clarify that `std*.h` shipped with CHERI Clang/LLVM cannot be used with the CheriBSD build.

Provide additional information on downloading the source code for FreeBSD/BERI, which is now available via the FreeBSD Project subversion repository.

Extend the description of CHERI-specific kernel source files, including those relating to debugging, exception delivery, signal handling, system-call handling, and CheriABI.

Further information on publication history, with cross references to papers and other technical reports on BERI and CHERI, has been added.

1.6 Document Structure

This document is an introduction and user manual for the Capability Hardware Enhanced RISC Instructions (CHERI) system prototype, encompassing the CHERI software and its use. The Guide is split into two parts: Part I describes the compiler and its implementation; Part II describes the OS and its implementation.

Chapter 2 describes how to build and use CHERI Clang/LLVM.

Chapter 3 describes the abstract model by which C pointers are mapped into CHERI capabilities.

Chapter 4 describes CHERI Clang: implementation details for how CHERI-aware C code is processed, built-in functions, and also compiler assistance for domain crossing.

Chapter 5 describes CHERI LLVM, including changes to the generic code, and those to the MIPS backend.

Chapter 6 describes CHERI code generation and its use of ABIs, both in 'hybrid' mode, in which MIPS pointers and CHERI capabilities coexist, and 'pure-capability ABI' (or 'sandbox') mode, in which only CHERI capabilities are used in implementing C pointers.

Chapter 7 describes how to obtain the CheriBSD source code, build the CheriBSD userspace, and build the CheriBSD kernel.

Chapter 8 describes the CheriBSD kernel, focusing on elements required to support in-address-space memory protection, and the CheriBSD compartmentalization model.

Chapter 9 describes the CheriBSD userspace.

Part I

Compiler

Chapter 2

Building and Using CHERI Clang

This chapter describes CHERI-specific modifications to the Clang/LLVM compiler suite [4] and the GNU assembler, as well as our extensions to the C programming language to support explicit capability use.

2.1 Cross-Compiling for CHERI

For cross-compiling code that targets CHERI, we provide a modified LLVM back end and Clang front end for [Objective-]C[++]. The back end can generate CHERI assembly and object code from LLVM's intermediate representation (IR). The front end generates the IR from C family languages and supports some capability extensions to C.

For assembly language programming, we also provide a modified version of the GNU binutils (including the GNU assembler `gas`) that has support for the capability instructions. This approach is gradually being deprecated in favor of the LLVM integrated assembler, but is still used in a number of places including the CHERI test suite and CheriBSD kernel build.

2.1.1 Building a Complete SDK

On a FreeBSD host system you can run `cherilibs/trunk/tools/build_sdk.sh` from the base CHERI distribution. This should be run in a new directory, as it will check out and build both CheriBSD and CHERI/LLVM.

After running this script, you should have subdirectories for the various projects that must be built, and an `sdk` directory containing the SDK. The SDK includes all of the core FreeBSD libraries and headers along with all of the tools required to cross build C/C++ programs for CHERI.

If you don't have a FreeBSD host system, the following sections describes how to build individual components of the toolchain.

2.1.2 Building the Assembler

To build the assembler, you will need to have Git installed. Check out the source code and build it like this:

```
$ git clone git://github.com/CTSRD-CHERI/binutils.git
$ cd binutils
$ ./configure --target=mips64 --disable-werror
$ make
```

2.1.3 Building the Compiler

To build the compiler, you will need to have Git, CMake, and Ninja installed. Check out the code and build like this:

```
$ git clone git://github.com/CTSRD-CHERI/llvm.git
$ cd llvm/tools
$ git clone git://github.com/CTSRD-CHERI/clang.git
$ cd ..
$ mkdir Build
$ cd Build
$ cmake -G Ninja -DCMAKE_BUILD_TYPE:STRING=Debug \
    -DBUILD_SHARED_LIBS:BOOL=ON \
    -DLLVM_DEFAULT_TARGET_TRIPLE:STRING=cheri-unknown-freebsd ..
$ ninja
```

Note: A recent version of CMake (at least 2.8.8) is required. If you are targeting a version of CHERI with 128-bit capability registers then you will need to add **`-DLLVM_CHERI_IS_128=ON`** to your `cmake` command line.

The `std*.h` and `limits.h` files included with Clang/LLVM are incompatible with those shipped in the FreeBSD base; they cannot be used for a CheriBSD crossbuild. The SDK build will automatically trim them, but manual builds of Clang/LLVM require those files to be deleted from your build tree:

```
$ rm lib/clang/3.*/include/std* lib/clang/3.*/include/limits.h
```

If these files are not deleted, the CheriBSD build will experience compiler errors relating to variable-argument functions and type. These errors will hopefully be resolved in the future by upstreamed improvements to the integrated headers so that they are appropriate for use in FreeBSD.

By default, Ninja will select a number of processes to run in parallel on the build based on your number of processors. You can increase or decrease this number with the -j flag. Building LLVM is somewhat memory intensive, with compilation steps taking around 300MB of RAM and linking steps taking 1-2GB, so you may wish to reduce this number if you have less than 1GB of RAM per core. If you are on a 32-bit system, you may want to pass the following option to CMake to build a release build with asserts, rather than a debug build:

```
-DLLVM_ENABLE_ASSERTIONS:BOOL:=ON
```

Attempting to Link a debug build of LLVM can run out of address space in the linker in a 32-bit system.

Building Clang also requires a recent version of `gcc`. To compile Clang with itself (or to compile the CTSRD-modified version of Clang with an unmodified clang) pass the following options to `cmake`:

```
-DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++
```

You can run the LLVM test suite, including CHERI-related tests, using:

```
$ ninja check
```

2.2 Using Clang

Once you have built LLVM, you (mostly) have a working cross-compiler. You can generate CHERI assembly code from [Objective-]C[++] source code with this command:

```
$ clang -S {source file} -target cheri-unknown-freebsd \
    -msoft-float
```

You can also generate native code directly:

```
$ clang --sysroot={cheribsd sysroot} {source file(s)} \
    -B{cheribsd sdk directory} \
    -target cheri-unknown-freebsd -msoft-float \
    -o {output executable}
```

The **`-target`** flag specifies CHERI as the architecture and FreeBSD as the platform. If you have built the CHERI SDK, you will have a `cheri-unknown-freebsd-clang` which you can use instead of the `clang` and the **`-target`** flag.

The **`-sysroot`** and **`-B`** flags tell the compiler where to look for various things. The first specifies where to search for headers (when compiling) and libraries (when linking). The second specifies where to search for other parts of the toolchain, specifically the linker and (if you're not using the integrated assembler) the assembler.

The **`-msoft-float`** flag ensures that, if your version of CHERI has no FPU, we will emit calls to emulated FPU functions rather than causing illegal instruction traps.

If you do a debug build of LLVM, then Clang will default to using the simple register allocator. To see significantly better code, add the following CFLAGS:

```
-mllvm -regalloc=greedy -O3 -mllvm -enable-mips-delay-filler
```

These flags will enable a better register allocator and will attempt to replace the `nop`s in delay slots with instructions from before the branch – which also turns on the full set of LLVM optimizations. Note that not all of these optimizations are well tested with CHERI; a lower optimization level may be required for the generation of correct code.

2.3 Disassembling CHERI Binaries

Disassembly of some instructions is common during debugging. You can do this for individual instructions with the `llvm-mc` tool:

```
$ echo 0x48 0x02 0x08 0x02 | llvm-mc -disassemble - \
    -triple=cheri-unknown-freebsd
    .section TEXT,__text,regular,pure_instructions
    CGetType $2, $c1
```

This tool expects a string of hex bytes and will write out the corresponding assembly. To disassemble entire object code files, use the `llvm-objdump` tool:

```
$ llvm-objdump -disassemble -triple=cheri-unknown-freebsd \
    {something.o}
```

2.4 Assembly Extensions

The LLVM integrated assembler (used by default by Clang, unless **`-no-integrated-as`** is passed) provides some mnemonics for ease of assembly programming.

2.4.1 Capability Move

The `cmove` pseudo operation expands to a `CIncOffset` instruction with **$zero** as the increment size. This is a shorthand for moving a value between capability registers. Its use is discouraged, as the similarity of this instruction and `cmov` makes code difficult to read.

2.4.2 Capability-Relative Floating-Point Loads and Stores

The assembler provides `clwc1`, `cldc1`, `cswc1`, and `csdc1` pseudoinstructions. These load and store 32- or 64-bit floating point values and are the same format as integer load and store operations. For example, to load a single-precision floating point value from offset 32 inside **C3** you would do:

```
1  clwc1 $f2, $zero, 32($c2)
```

This will expand to:

```
1  clw   $1, $zero, 32($c2)
2  mtc1  $1, $f2
```

A future revision of the CHERI ISA may add instructions for loading and storing floating point values directly, at which point assembly code using this pseudo will generate a real instruction.

Chapter 3

Abstract model

To understand CHERI as a programmer target, it is important to understand the abstract machine that CHERI exposes. This is intended to be compliant with C abstract machine, yet provide strong memory safety guarantees that can serve as foundations for security properties.

3.1 Capabilities as pointers

CHERI provides *memory capabilities* to limit access to *virtual memory*. Software can only access the subset of its virtual address space for which it has valid capabilities. Each load or store operation—including instructions and instruction fetch—rely on implicit or capabilities. MIPS load and store operations are indirected via the *ambient data capability* (**$c0**), CHERI load and store operations take explicit capability operands.

Capabilities grant some permissions on a range of memory, identified by a base and a length. The CHERIv3 ISA extends this to incorporate an *offset*, intended to make it possible to use capabilities, rather than integers, as pointers in C-like languages. You can perform arbitrary arithmetic on a capability's offset, but can only dereference it if it lies within the bounds and you have the relevant permissions.

3.2 Operations on capabilities

CHERI does not allow arbitrary privilege elevation. A piece of code can disclaim a capability, by simply overwriting it with some data. It can also limit a capability by reducing its length, increasing its base (which decreases its length by a corresponding amount), or reducing the set of permissions. All of these operations provide a monotonic decrease in the access that a capability grants. If you have a capability to an allocation, for example an array, then you can construct capabilities to a range within the array, or capabilities that can only read or write (but not both) to the array, but you can not use it to construct a capability that allows you to write past the end of the array.

The offset, as previously mentioned, is exempt from the monotonic property. It may be set to any value, but any load and store that attempts to access outside of the permitted range will fail.

3.3 Integers in capabilities

One important property of C is the existence of types such as `intptr_t`, that can store either a pointer or an integer and can be operated on as if they were integers. Integers can be stored in capabilities using the offset: a null capability grants no rights, but can have an offset anywhere in the virtual address space. The offset can therefore be used to store integer values.

The current implementation defines `intmax_t` to be **`long`**. This is allowed by the C specification, as the requirement for `intmax_t` is that it must have a *range* equal to the largest integer, not a *size*. We believe that this is likely to cause fewer problems in porting code, as most uses of `intmax_t` are for storing integer, not pointer, values.

3.4 The different ABIs

Incremental adoption is one of the goals of CHERI. All existing unmodified binaries for the base architecture are expected to work. More importantly, it should be possible to run unmodified programs with modified libraries and vice versa.

With this in mind, we support two ABIs, discussed more in chapter 6. One is a small extension to the base ABI, permitting *some* pointers to be capabilities, but with the expectation that **$c0** encompasses the entire (or majority of the) address space and that the memory protection can be deliberately bypassed. The second, the pure-capability ABI, is intended to be used when compatibility is not a concern and uses capabilities for all pointer, as well as for the stack.

With care, it is possible to mix the use of both ABIs within a single program. The layout of various data structures will be different, so the programmer is responsible for ensuring that data is correctly marshaled at boundaries.

Chapter 4

C compiler support

The CHERI version of *Clang*, the C language front end for LLVM, has been modified to expose capabilities at the C level.

4.1 Supported targets

When targeting CHERI, you must specify a *target triple* that contains **`cheri`** in the CPU part. The most common triple to use is **`cheri-unknown-freebsd`**. This triple contains three portions, in the form *cpu-vendor-os*. The vendor is irrelevant, only the CPU and OS matter for code generation.

There are two ways to pass the triple to Clang. The first is to add a **`-target`** flag, followed by your triple, to your compile flags. The second is to create a symbolic link to clang of the form *cheri-unknown-freebsd-clang*.

> **Note:** `build_sdk.sh`
>
> If you have the cross-compiler from the SDK, then it will automatically have its default target triple (and sysroot) configured correctly. You can omit the **`-target`** option. You can see the default target triple by passing **`-v`** to `clang`.

4.2 The pure-capability ABI

You can select the pure-capability ABI by passing **`-mabi=sandbox`** to clang. When compiling in this mode, *all* pointers (including function pointers) will be represented as capabilities.

The compiler will infer the size of all stack allocations automatically and expects memory allocators to set the size appropriately for their allocations.

C code in this model will not contain any non-capability load or store instructions. This means that it is safe to clear the ambient data capability (**$c0**) when in this mode. It could be used as a general-purpose capability register, but is not currently. In this mode, or one where the ambient data capability is very limited in scope, this provides full memory safety.

4.3 Pointer qualifiers

In the compatible ABI, the `__capability` qualifier indicates that a pointer should be represented by a memory capability. For example:

OPTION	MEANING
`push`	Save the current interpretation.
`pop`	Restore a previously saved interpretation.
`integer`	Pointers are represented by integers.
`capability`	Pointers are represented by capabilities.
`default`	Pointers are represented by whatever the default is for this target.

Table 4.1: The valid options for the pointer interpretation pragma.

```
void *a; // 64-bit integer interpreted as a pointer
__capability void *b; // 256-bit capability interpreted as a pointer
```

Implicit casts between capability and integer pointers are not permitted. This is an intentional design decision, as such casts require careful thought to ensure that they are valid.

Clang exposes a feature-test macro that allows you to easily check whether you're compiling for a target that supports capabilities and to **`#define`**-way the `__capability` qualifier for other platforms:

```
    #if !__has_feature(capabilities) && !defined(__capability)
    #   define __capability
    #endif
```

4.4 Pragmas for generating capabilities

Often is it useful to define a block of code where pointers will be interpreted as either capabilities or integers, rather than having to annotate them individually. This is possible with the `pointer_interpretation` pragma. As with other C99 pragmas, this can be used either with the **`#pragma`** or `_Pragma()` syntax.

The valid options for this pragma are listed in Table 4.1. Most uses of this pragma are likely to be via macros that explicitly push and pop. For example, the following definitions provide a mechanism for defining ranges:

```
#if __has_feature(pointer_interpretation)
#   define BEGIN_CAPABILITIES \
        _Pragma("pointer_interpretation␣push") \
        _Pragma("pointer_interpretation␣capability")
#   define END_CAPABILITIES \
        _Pragma("pointer_interpretation␣pop")
#   define BEGIN_NO_CAPABILITIES \
        _Pragma("pointer_interpretation␣push") \
        _Pragma("pointer_interpretation␣integer")
#    END_NO_CAPABILITIES
        _Pragma("pointer_interpretation␣pop")
#else
#   define BEGIN_CAPABILITIES
        _Pragma("GCC␣error␣\"Compiler␣does␣not␣support␣capabilities
           \"")
```

```
#   define END_CAPABILITIES
        _Pragma("GCC error \"Compiler does not support capabilities
            \"")
#   define BEGIN_NO_CAPABILITIES
#   define END_NO_CAPABILITIES
#endif
```

When compiled with a compiler that supports this pragma, they will have their expected meanings (including correctly handling nesting). When compiled with another compiler, the `NO_CAPABILITIES` versions will be silently ignored, but the `CAPABILITIES` versions will raise an error.

This allows code like this:

```
BEGIN_CAPABILITIES
struct test
{
    int *a,*b,*c;
} t;
END_CAPABILITIES
```

This is equivalent to explicitly annotating each field in the **`struct`** with the `__capability` annotation.

4.5 Built-in functions

The C front end provides a number of built-in functions for manipulating capabilities, listed in Table 4.2. These correspond directly to a single instruction, shown in the table. Most of these functions (those with `memcap` in their name) are generic and should apply to any architecture that supports memory capabilities. The remainder (those with `cheri` in their name) are specific to the current implementation.

4.6 Predefined macros

CHERI Clang provides a number of feature test macros, shown in Table 4.3. These are intended to be used to check for specific functionality and conditionally compile code.

Additionally, the permission flags are provided as symbolic constants in the form of predefined macros. These all start with `__CHERI_CAP_PERMISSION_` followed by a suffix from Table 4.4. For example, `__CHERI_CAP_PERMISSION_PERMIT_SEAL__` is the permission bit for sealing.

4.7 `__intcap_t`

CHERI provides two builtin types, `__intcap_t` and `__uintcap_t`. These are expected to be **`typedef`**'d to `[u]intcap_t` in the compatible ABI and `[u]intptr_t` in the pure-capability ABI. These types are the same sizes as capabilities (i.e. **`sizeof`**`(__intcap_t)==` **`sizeof`**`(__capability` **`void`**`*)`).

As with `intptr_t` in conventional C code, the goal for these types is to support any integer value or any pointer value, and to allow arbitrary arithmetic. The C standard requires that, if `intptr_t` exists, it should be possible to store a pointer in it and recover the same pointer. Most C code expects to be able to also perform arithmetic on the pointer value.

CLANG BUILTIN	CHERI INSTRUCTION
__builtin_memcap_length_set	CSetLen
__builtin_memcap_length_get	CGetLen
__builtin_memcap_bounds_set	CSetBounds
__builtin_memcap_base_increment	CIncBase
__builtin_memcap_base_get	CGetBase
__builtin_memcap_perms_and	CAndPerm
__builtin_memcap_perms_get	CGetPerm
__builtin_memcap_type_set	CSetType
__builtin_memcap_type_get	CGetType
__builtin_memcap_tag_get	CGetTag
__builtin_memcap_sealed_get	CGetSealed
__builtin_memcap_tag_clear	CClearTag
__builtin_memcap_seal	CSeal
__builtin_memcap_unseal	CUnseal
__builtin_memcap_perms_check	CCheckPerm
__builtin_memcap_type_check	CCheckType
__builtin_memcap_offset_increment	CIncOffset
__builtin_memcap_offset_set	CSetOffset
__builtin_memcap_offset_get	CGetOffset
__builtin_memcap_program_counter_get	CGetPCC
__builtin_memcap_global_data_get	CMove %0, $c0
__builtin_memcap_stack_get	CMove %0, $c11
__builtin_cheri_cause.get	CGetCause
__builtin_cheri_cause.set	CSetCause
__builtin_cheri_invoke_data_cap_get	CMove %0, $c26
__builtin_cheri_kernel_cap1_get	CMove %0, $c27
__builtin_cheri_kernel_cap1_get	CMove %0, $c28
__builtin_cheri_kernel_code_cap_get	CMove %0, $c29
__builtin_cheri_kernel_data_cap_get	CMove %0, $c30
__builtin_cheri_exception_program_counter_cap_get	CMove %0, $c31

Table 4.2: C built-in functions provided for CHERI.

MACRO	VALUE	MEANING
`_MIPS_ARCH`	`"cheri"`	The variant of the MIPS architecture in use.
`_MIPS_ARCH_CHERI`	1	CHERI is the target (MIPS-specific).
`__CHERI__`	1	CHERI is the target (may be used for non-MIPS implementations).
`_MIPS_SZCAP`	256	The size of a capability (in bits).
`__CHERI_SANDBOX__`	1	Set only if targeting the pure-capability ABI.

Table 4.3: Feature test predefined macros supported by CHERI Clang

MACRO SUFFIX	VALUE	MEANING
`_GLOBAL__`	1	Global "permission" flag value.
`_PERMIT_EXECUTE__`	2	Execute permission flag value.
`_PERMIT_LOAD__`	4	Load (data) permission flag value.
`_PERMIT_STORE__`	8	Store (data) permission flag value.
`_PERMIT_LOAD_CAPABILITY__`	16	Load-capability permission flag value.
`_PERMIT_STORE_CAPABILITY__`	32	Store-capability permission flag value.
`_PERMIT_STORE_LOCAL__`	64	Store ephemeral permission flag value.
`_PERMIT_SEAL__`	128	Seal permission flag value.
`_ACCESS_EPCC__`	1024	Access **$epcc** permission flag value.
`_ACCESS_KCC__`	4096	Access **$kcc** permission flag value.
`_ACCESS_KDC__`	2048	Access **$kdc** permission flag value.
`_ACCESS_KR1C__`	8192	Access **$kr1c** permission flag value.
`_ACCESS_KR2C__`	16384	Access **$kr2c** permission flag value.

Table 4.4: Suffixes of permission value predefined macros supported by CHERI Clang. `__CHERI_CAP_PERMISSION_` is prefixed to all of these.

When an integer value is cast to an `__intcap_t`, the compiler will set that integer as the offset of a canonical null capability. Any arithmetic on an `__intcap_t` value is performed by extracting its offset, manipulating it, and then setting it. This ensures that integer arithmetic on `__intcap_t` values derived from integers will work precisely as expected. Arithmetic on values derived from pointers (capabilities) is a little bit more complex. Masking to access the unused low and high bits should still work as expected, as should addition. Multiplication and division, however, will not.

Comparisons between `__intcap_t` values use special pointer comparison instructions. These have the semantics that any untagged capability (including those derived from the null capability) will compare before those with a valid tag. Within these two regions, capabilities are compared based on their absolute virtual addresses (i.e. base + offset). Comparisons between signed and unsigned `__intcap_t` values will use the signed and unsigned variants of the compare instructions.

Note: Pointer comparison and garbage collection

The C specification makes comparisons between pointers to different objects undefined. The CHERI C compiler aims to ensure that such comparison is stable and has the same result as pointer comparison with non-capability pointers.

The fact that the CHERI model makes it possible to differentiate pointers from integers (even in the case of `__intcap_t`) means that it is possible to implement a copying garbage collector for CHERI C. The compiler aims to ensure that virtual addresses do not leak into integer registers unless the programmer explicitly requests them to (for example, casting a capability to an `int` or explicitly accessing the base). This means that, as long as care is taken, memory referenced by capabilities but outside of the global data capability range can be safely collected, unless the code relies on undefined behavior or explicitly attempts to subvert the garbage collector.

4.8 Input and output

The CHERI capability model supports pointers with restricted rights. An early version of CHERI Clang used this to enforce `const` in hardware. This proved problematic, for example in idioms such as `strchr`:

```
char *strchr(const char *s, int c);
```

This function returns a non-`const` pointer derived from a `const` pointer. The invariant that this intends to document is that the `strchr` function will not modify the buffer `s`. In the capability model, it is impossible to derive a writable capability from a read-only capability, which broke any caller of this function that expected the returned pointer to be writable.

To address this, CHERI Clang provides an `__input` qualifier, which is similar to `const` but enforced in hardware. Capabilities can be implicitly cast to `__input`-qualified versions, including as function parameters, and the compiler will insert a `CAndPerms` instruction to disclaim write permissions.

A corresponding `__output` qualifier has similar behavior, creating a write-only capability. For example, consider the following simple snippet:

```
int in(__input __capability int *x);
int out(__output __capability int *x);
```

```
void inout(__capability int *x)
{
    in(x);
    out(x);
}
```

The `inout` function will compile to the following LLVM IR:

```
; Function Attrs: nounwind
define void @inout(i32 addrspace(200)* %x) #0 {
entry:
    %0 = bitcast i32 addrspace(200)* %x to i8 addrspace(200)*
    %1 = tail call i8 addrspace(200)* @llvm.mips.cap.perms.and(i8
       addrspace(200)* %0, i64 65495)
    %2 = bitcast i8 addrspace(200)* %1 to i32 addrspace(200)*
    %call = tail call i32 @in(i32 addrspace(200)* %2) #3
    %3 = tail call i8 addrspace(200)* @llvm.mips.cap.perms.and(i8
       addrspace(200)* %0, i64 65515)
    %4 = bitcast i8 addrspace(200)* %3 to i32 addrspace(200)*
    %call1 = tail call i32 @out(i32 addrspace(200)* %4) #3
    ret void
}
```

The two intrinsic calls construct two capabilities derived from the original argument. The first removes the store and store-capability permissions. The second removes the load and load-capability permissions. This means that the `in` function can not use its argument to write memory and the `out` function can not use its argument to read memory.

4.9 Inline assembly

Clang supports GNU-style inline assembly. When targeting CHERI, *"C"* can be used as a register constraint indicating that a particular operand is a capability. For example, to access the value in **$c0**, you might write:

```
__capability void *c0;
__asm__ __volatile__ ("cmove␣%0,␣$c0"
    : "+C"(c0) /* c0 is a capability output operand */
    : /* No input operands */
    : /* No clobbers */
    );
```

Note that the + prefix is required to indicate that this is an output register, for input operands *"C"* is correct.

4.10 `memcap.h`

CHERI Clang includes a header, `memcap.h`, which can be included by code targeting any architecture and exposes architecture-neutral definitions of types and functions for memory capabilities.

The new types and qualifiers provided by the compiler all use identifiers that start with two underscores, because this part of the namespace is reserved for the language implementation and so should not conflict with any user code. The `memcap.h` header provides more human-friendly versions of these, including `intcap_t` and `uintcap_t` types, and the `capability`, `input` and `output` qualifiers.

On architectures that don't support capabilities, the two types will be equivalent to `intptr_t` and `uintptr_t` and the qualifiers will be ignored.

The header also provides versions of the `__builtin_memcap` functions, without the `__builtin_` prefix. The versions of these that are exposed when compiling without capability support are no-ops for the set variants and define maximum permissions and bounds for the get variants.

Only the functions that are expected to be generic across all implementations of the CHERI model (as opposed to the MIPS-based concrete implementation) are exposed by this header.

4.11 Compiler assistance for cross-domain calls

It is important to avoid leaking rights (and other information) when calling between security contexts. CHERI implements cross-domain calls via a special instruction (`ccall`) that traps to a privileged handler (currently in the kernel) to perform the domain transition. The handler is responsible for saving and zeroing all callee-save registers and clearing all caller-save registers. On return, the same code in the handler will restore all callee-save registers and (again) zero caller-save registers.

The handler does not know precisely what the arguments are for any function that is involved in a cross-domain call and so can not clear the unused argument registers on call, nor the unused return registers on return.

On the other side of the cross-domain call, the code must determine which method to invoke. This is done by means of a method number, which is passed in a register that is normally unused for calls.

If a function is annotated with `__attribute__((cheri_ccall))` then the front end will replace calls to it with specially crafted calls to `cheriinvoke`, with the method number (identified by a global variable that is initialised by the sandbox loader) in the correct register. This annotation provides two declarations of the function, one with a suffix (specified by the `cheri_method_suffix` attribute) that takes an explicit CHERI class argument and one that automatically sets the class (code and data capability) arguments from a global variable specified by the `cheri_method_class` attribute.

Functions with the `cheri_ccallee` attribute will use the ccall calling convention, and so the compiler will zero any return registers. This is intended to allow functions to be declared that are usable both inside and between compartments.

Functions with the ccall calling convention have two extra capability argument registers, **$c1** and **$c2**, which contain the code and data capabilities. They also have an extra integer argument register (**$v0**), which contains the method number. These are not exposed to functions marked as using the ccallee calling convention, which accept the normal calling convention's argument registers.

Consider the following simple program:

```
__capability void *data;
__attribute__((cheri_ccallee))
__capability void *cgetdata(void)
{
    return data;
}

__attribute__((cheri_ccallee))
int cgetnumber(void)
{
    return 42;
}
```

This contains two simple functions that are expected to be invoked as part of a domain transition. The first returns a capability from inside its data capability, the second returns a constant. In the first,

the return value is a capability and so will be returned in capability register **$c3**. The two integer return registers (**$2** and **$3**, sometimes called **$v0** and **$v1**) are unused. The compiler will generate this code for the return:

```
clc     $c3, $zero, 0($c2)
move    $2, $zero
move    $3, $zero
```

The return value is loaded into **$c3**. The two integer return values are zeroed. In the second function, one of the integer return registers is used, but the capability registers are not. The compiler generates this code:

```
cfromptr  $c3, $c0, $zero
addiu     $2, $zero, 42
move      $3, $zero
```

In this case, the first integer return register is set to 42, but the other return registers are cleared.

Chapter 5

LLVM implementation

The changes in LLVM can be roughly split into two components: those specific to the MIPS back end and those in generic code. The MIPS back end changes are specific to the current CHERI implementation. The remainder are generic changes to support memory capabilities in the middle of the optimization pipeline and in the target-independent code generator.

5.1 Address space 200

We reserve address space 200 to be the address space used for capabilities. This is below 256 and so is a target-agnostic address space. We assume that other architectures providing support for memory capabilities would use the same address space.

5.2 Data layout

To support the pure-capability ABI, the data layout string is modified to define the address space for `alloca` instructions. It's assumed that, within a compilation unit, every `alloca` returns a pointer in the same address space. By default, this is address space 0, but when targeting the pure-capability ABI it is set to 200.

In the MIPS back end, a pass will replace each of these allocas (and all of their uses) with one in address space 0, followed by calls to two intrinsics. The first intrinsic derives a **$c11**-relative capability from the integer value. The second sets its length to the size of the alloca.

This allows the normal MIPS stack pointer to be used as an offset within **$c11**.

A lot of optimizations, in particular scalar evolution and the SLP vectorizer, assume that the size of a pointer is the size of an integer that can express its range. As a result, they will attempt to create `i256` operands to various pointer arithmetic operations. To avoid this, the `DataLayout` class now provides a few variants of `getPointerBaseSize()` methods, which get the size of the base of a fat pointer or capability. For non-capability pointer types, these simply return the size.

5.3 Alignment of types

Capabilities are *always* naturally aligned. This is a requirement of the hardware (there is one tag bit per capability-sized line of memory). The back end will assume that all pointers to capabilities are correctly aligned and will emit code that will trap at run time if not.

Capability-relative loads and stores have no equivalent of the MIPS `lwl` and `lwr` instructions for unaligned loads and stores. If capability-relative loads and stores have to be aligned, then this results

in a very inefficient sequence of loads and shifts. This is unfortunate, because in most cases loads and stores *are* naturally aligned, but the front end and mid-level optimizers lose the alignment information. This resulted in the back end emitting the inefficient sequence for all loads and stores.

The MIPS back end now assumes that CHERI is able to support unaligned loads and stores of every type. A sufficient number of loads and stores are correctly aligned (but have lost the alignment information) that this is a speed win. On recent versions of CHERI1, unaligned loads and stores within a cache line are also supported in hardware, so the slow path (trapping to the OS) is not required.

5.4 SelectionDAG types

In LLVM, one of the early parts of the target-independent code generator replaces all pointers with an integer of the correct size. On CHERI, this would be an `i256`, which is not a valid type for the target and would cause other problems if the definition allowed the rest of the code generator to assume that 256-bit integers were supported.

To avoid this, we add an `MVT::iFATPTR` machine value type to the code generator. This can be used for any non-integer pointer and can be pattern matched by any of the `tablegen`-generated matching code.

We also add `ISD::INTTOPTR` and `ISD::PTRTOINT` SelectionDAG nodes. These represent conversions between `MVT::iFATPTR` and integer types and are generated from any address space cast instructions between address spaces 0 and 200, as well as **`inttoptr`** and **`ptrtoint`** IR instructions. In the MIPS back end, these expand to `CToPtr` and `CFromPtr` instructions.

The conventional way of expressing pointer arithmetic in the SelectionDAG is via normal arithmetic nodes. This works because the `MVT::iPTR` type is lowered to an integer type (typically `MVT::i32` or `MVT::i64`) in the legalization phase. The `ADD` SelectionDAG node has the invariant that the operands must be the same type. This does not work for pointer addition with fat pointers, because one operand is the pointer and the other is the integer value that is being added to the pointer, which will typically be `MVT::iFATPTR` and `MVT::i64` respectively for CHERI.

To address this, we add another new SelectionDAG node: `ISD::PTRADD`. This is naÃŕvely lowered to `CIncOffset` in the MIPS back end, but may later be folded to use a complex addressing mode.

5.5 LLVM IR Intrinsics

The MIPS back end exposes a number of intrinsics in LLVM IR, listed in Table 5.1. Some of these, shown above the line in the table, may later be replaced by generic memory capability intrinsics. The remainder are specific to the MIPS implementation of the CHERI modell.

Most intrinsics map directly to a specific instruction. A small number fix some of the operands. For example, all of the accessors for the special registers will move the value of a specific register into an SSA register, which will then be replaced with a real register (whose number is not under programmer control) during register allocation.

The `llvm.mips.stack.to.cap` intrinsic is intended for internal use only. As mentioned in section 5.2, it is used by the MIPS target when converting the IR from a form where `alloca` instructions are in address space 200, to one where they are in address space 0. All `alloca` instructions are replaced with a short sequence of an `alloca` followed by a call to this intrinsic and then a call to the `llvm.mips.cap.length.set` intrinsic. All uses of the original `alloca` are then replaced by this value, which is a **$c11**-derived capability of a defined length.

LLVM INTRINSIC	CHERI INSTRUCTION
llvm.mips.cap.length.set	CSetLen
llvm.mips.cap.length.get	CGetLen
llvm.mips.cap.bounds.set	CSetBounds
llvm.mips.cap.base.increment	CIncBase
llvm.mips.cap.base.get	CGetBase
llvm.mips.cap.perms.and	CAndPerm
llvm.mips.cap.perms.get	CGetPerm
llvm.mips.cap.type.set	CSetType
llvm.mips.cap.type.get	CGetType
llvm.mips.cap.tag.get	CGetTag
llvm.mips.cap.sealed.get	CGetSealed
llvm.mips.cap.tag.clear	CClearTag
llvm.mips.cap.seal	CSeal
llvm.mips.cap.unseal	CUnseal
llvm.mips.cap.perms.check	CCheckPerm
llvm.mips.cap.type.check	CCheckType
llvm.mips.cap.offset.increment	CIncOffset
llvm.mips.cap.offset.set	CSetOffset
llvm.mips.cap.offset.get	CGetOffset
llvm.mips.stack.cap.get	CMove %0, $c11
llvm.mips.cap.cause.get	CGetCause
llvm.mips.cap.cause.set	CSetCause
llvm.mips.c0.get	CMove %0, $v0
llvm.mips.pcc.get	CGetPCC
llvm.mips.idc.get	CMove %0, $c26
llvm.mips.kr1c.get	CMove %0, $c27
llvm.mips.kr2c.get	CMove %0, $c28
llvm.mips.kcc.get	CMove %0, $c29
llvm.mips.kdc.get	CMove %0, $c30
llvm.mips.epcc.get	CMove %0, $c31
llvm.mips.stack.to.cap	CFromPtr $c11, %0

Table 5.1: LLVM intrinsics provided for CHERI.

Chapter 6

The CHERI ABIs

The CHERI compiler supports two ABIs, an extended version of the MIPS n64 ABI and the pure-capability ABI where every pointer is a capability. The pure-capability ABI is intended to evolve to a point where **$c0** is never used. Currently, **$c0** is used for globals, but not for any other data accesses.

6.1 Register usage

The compiler's use of capability registers is summarized in Table 6.1. Registers **$c16**–**$c24** are preserved across calls, as is **$c0**, which is never modified by the compiler. In the pure-capability ABI, so is **$c11** (the stack capability).

6.1.1 The pure-capability ABI

The pure-capability ABI uses **$c11** as a stack capability. The **$sp** and **$fp** registers contain offsets in the stack capability. It would make sense to replace **$sp** with the offset in **$c11**. This would simplify the addressing modes for the stack and make a lot of spills cheaper in the pure-capability ABI.

The pure-capability ABI uses the capability mechanism to protect the return address. In the MIPS ABI, function calls are implemented as `jalr $t9, $ra`. In the pure-capability ABI, they are `cjalr $c12, $c17`. This has two effects. The first is that **$t9** can no longer be used as a cheap way of getting the program counter for position-independent code. Instead, if this value is needed, the compiler will emit `CGetOffset $t9,$c12`, setting **$t9** to the **$pcc**-relative address of the program counter on function entry.

The second effect is that the return can be emitted as `cjr $c17`.If **$c17** is spilled to the stack, then it is protected by its tag bit. If it is overwritten by anything other than an executable capability, then the `cjr` instruction will trap.

6.2 Calling conventions

When targeting the n64 ABI, the only changes to the calling convention are to support capability arguments. Capability arguments are passed in registers **$c3** to **$c10**, with **$c3** also being used for capability return values.

The normal rules for composite types apply: the portions that will fit within a register are passed in registers and the remainder is passed on the stack.

REGISTER	COMPILER USAGE
$v0	Contains the method number for cross-domain calls.
$c0	Used implicitly for all non-capability memory accesses.
$c1–$c16	Used for arguments in the "fast" calling convention.
$c1–$c2	Code and data capability arguments with the "ccall" calling convention.
$c3	Capability return value.
$c3–$c10	Capability arguments (caller-save).
$c11	Stack capability (pure-capability ABI).
$c12	Used with `cjalr` as the destination register (pure-capability ABI).
$c13	Capability to on-stack arguments (variadic functions only).
$c11–$c15	Temporary (caller-save) registers.
$c17	Capability link register used with `cjalr` (pure-capability ABI).
$c16–$c24	Saved (callee-save) registers.
$c25–$c31	Not used by the compiler.

Table 6.1: Capability register usage.

6.2.1 Variadic calls

The n64 ABI requires that all arguments to variadic functions are passed either on the stack or in 8 integer registers. When a `va_list` is constructed, the 8 integer values are written out. This means that capability arguments in variadic functions are very difficult to support and are currently not expected to work. To fix them, we will have to ensure that all arguments after the first capability argument are passed on the stack and that the first capability argument has a mechanism for knowing whether it is in the first 64 bytes of the `va_list` (and, if so, incrementing the pointer until it's at the end).

In the pure-capability ABI, *all* variadic arguments are passed on the stack, with their natural alignment (non-variadic arguments to variadic functions are passed in registers or on the stack as the normal calling convention would dictate).

The **$c13** register holds a capability to the on-stack arguments. The `va_start` function copies the value that was stored in **$c13** on entry to the function.

The `va_list` is a capability to the (on-stack) variadic arguments and `va_arg` calls ensure correct alignment, load from the capability, and increment its offset past the value. The alignment requirements can result in large gaps in the variadic argument list if integer and capability arguments are interleaved.

Note: General usage

Being able to find the range of on-stack arguments can be useful in the general case, so it may be a good idea to extend the non-variadic ABI to store the range of on-stack arguments in a capability in the same way. This would require that all functions that do not pass values on the stack zero **$c13**, but that is a relatively small overhead.

6.3 Cross-domain calls: "ccall"

The `chericcallcc` calling convention uses registers **$c1** and **$c2** for the first two capability arguments and **$v0** for the method number. The front end will lower `struct`s that fit in registers to a

sequence of scalars, so this is typically generated from a two-capability `struct`. The remaining arguments are in the same place as the normal calling convention, allowing a simple jump to tail call functions that do not care about these arguments.

The back end tracks the argument registers that are used by callers of functions with this convention and the return registers that are used by the callee. At each call site, it will zero unused argument registers. In the callee, it will zero unused return registers.

> **Note:** Soft float
>
> The compiler currently assumes that the soft-float ABI is always used for calls into sandboxes and so does not zero any floating point registers. The CHERIBSD trampoline code in the kernel also makes this assumption, but it will need revisiting once floating point is enabled.

The attributes for generating this code are described in section 4.11. The compiler assumes that the runtime environment (library or kernel code) is responsible for preserving or zeroing all non-argument registers across security domain transitions. The code to do this is the same for all functions and so having only a single instance provides better cache utilization.

The compiler generates two special sections containing metadata about cross-domain calls. `__cheri_sandbox_required_methods` contains metadata about methods that are required by the binary. `__cheri_sandbox_provided_methods` contains metadata about methods that are provided by this binary. These are used by the `libcheri` runtime, to make method numbers in the caller and callee of cross-domain calls match.

For each method that is defined in the binary, the `__cheri_sandbox_provided_methods` section will contain an instance of the following structure:

```
struct sandbox_provided_method
{
    int64_t   flags;
    char     *class;
    char     *method;
    void     *method_ptr;
};
```

The `flags` field is currently always 0 and is reserved to permit future modifications to this structure without breaking compatibility. The `class` field points to the name of the class for which the method is provided. The `method` field points to the name of the method. The `method_ptr` field is a **$pcc**-relative address of the method.

Similarly, for each required method the compiler will emit a structure of the following form in the `__cheri_sandbox_required_methods` section:

```
struct sandbox_required_method
{
    int64_t   flags;
    char     *class;
    char     *method;
    int64_t  *method_number_var;
    int64_t   method_number;
};
```

Multiple compilation units in the same binary may require the same method, so each of these structures is emitted as a single *comdat*, to allow merging. Older versions of the cross-domain call ABI put

the method numbers in a separate section and required the runtime library to walk the ELF symbol table. The current structure is a hybrid, with the `method_number_var` field containing the address of that global, allowing code to be compiled and used with old and new versions of the runtime library.

The `flags` field is used by the runtime to indicate that a method has been resolved. It will also be used in a future version to indicate that the `method_number_var` field has been omitted and that the generated code expects the `method_number` field to contain the authoritative version of the method number. The top 32 bits of the `flags` field are reserved for use by the runtime, the low 32 bits for use by the compiler.

The `class` and `method` fields contain the names of the class and method, respectively.

6.4 Global initialization

Capabilities in globals require special handling. Capabilities can not be statically defined in the binary as other data, because doing so will not set the tag. Similarly, existing relocation types are not sufficient to describe a capability, which has a base, bounds, and permissions in addition to the location described by conventional pointers.

> **Note:** Dynamic initialization
>
> The initial implementation in clang simply emitted C++-style dynamic initialization code for all globals that contained non-null pointer values in the constant initializer. This is still the default, though this will change soon and can be disabled with the **`-Xclang -cheri-linker`** flag.

The LLVM back end will emit a special section in the ELF binary for these initializers. This section contains one entry for each capability that must be initialized at program launch. For example, consider the following program fragment:

```
extern int a[5];
int *b[] = {&a[2], &a[1], a};
```

The resulting binary will contain a `__cap_relocs` section with three instances of the following structure:

```
struct capreloc
{
    void *__capability capability_location;
    void               *object;
    uint64_t            offset;
    uint64_t            size;
    uint64_t            permissions;
};
```

The `capability_location` field contains the (relative) address of the capability that must be initialised at run time. The `object` field contains an address (and associated relocations) of the object that the capability refers to. The `offset` field contains the offset within this object. The `size` field contains the size of the underlying object. The `permissions` field contains the permissions that the capability should have and reserves space for other flags.

For the above example, the compiler will emit three structures, with the following values:

```
    { &b[0], &a, 8, 0, 0},
    { &b[1], &a, 4, 0, 0},
    { &b[2], &a, 0, 0, 0}
```

The compiler does not know the size of the object and so will set the size to 0. After linking, the ELF file will contain the size of the symbol. A capability-aware linker would then fill in the size field. In the absence of such a linker, the `fixcapsize` tool will set the size.

Currently, the permissions field is always 0. Future versions will indicate in this field whether the capability is relative to **$gdc** or **$pcc** and what permissions it should have.

You can check wither this has worked by using the **`-C`** flag to `llvm-objdump` :

```
$ llvm-objdump -C a.out

a.out: file format ELF64-mips

CAPABILITY RELOCATION RECORDS:
0x0000000000000a7a0 Base: a (0x0000000000000a700) Offset: 0000000000000008
Length: 0000000000000020
0x0000000000000a7c0 Base: a (0x0000000000000a700) Offset: 0000000000000004
Length: 0000000000000020
0x0000000000000a7e0 Base: a (0x0000000000000a700) Offset: 0000000000000000
Length: 0000000000000020
```

There are currently some significant limitations:

- We should have a flag to indicate whether this is a code (executable) or data capability, to indicate whether it should be derived from **$pcc** or **$gdc**.
- We have no way of enforcing permissions (for example, the `__output` or `__input` qualifier on pointers).
- Dynamically linked binaries will need the run-time linker to provide the symbol sizes.

6.5 Return address protection

RISC architectures typically provide a jump instruction that puts the return address in another register (e.g. `jalr` on MIPS, `bl[x]` on ARM). If the called function calls another function, it must spill the return address to the stack, where it can be reloaded later. This happens automatically on x86, where the `call` and `ret` instructions store the return address on the stack and read it from the stack, respectively.

If a buffer overflow allows the return address to be overwritten, then an attacker can control exactly where execution will continue after the return.

In the pure-capability ABI, this kind of attack is very difficult. Calls use the `cjalr` instruction, so the return address is a **$pcc**-relative capability. If this is overwritten with something that is not a capability, then the return will trigger a tag violation. If this is overwritten by a non-executable capability, then the return will trigger a permissions violation. For a successful exploit, the attacker would have to find an executable capability (e.g. a function pointer or a previous return address) that the program could be tricked into writing over the return address.

We would like to be able to provide the same benefits to the existing ABI. We achieve this by keeping the call sequence the same, but modifying the return sequence. In all non-leaf functions, when we spill the return address to the stack we also spill a *return capability*: **$pcc** with its offset set to the return address. This can then be used with `cjr` to return. Note that we still spill the return address, even though it is not used, because other tools (debugging tools and so on) sometimes rely on the position of the return address on the stack.

Part II

Operating System

Chapter 7

Building and Using CheriBSD

FreeBSD/BERI is a port of the open-source FreeBSD operating system [5] to the Bluespec Extensible RISC implementation (BERI), now available via the FreeBSD Project. CheriBSD extends FreeBSD/BERI to implement memory protection and software compartmentalization features supported by the CHERI ISA. General crossbuild and use instructions for FreeBSD/BERI may be found in the *BERI Software Reference*. Procedures for building and using FreeBSD/BERI should entirely apply to CheriBSD, except as documented in this chapter.

7.1 Obtaining FreeBSD/BERI and CheriBSD Source Code

FreeBSD/BERI has been merged to FreeBSD subversion repository and may be obtained from:

`https://svn.freebsd.org/base/head`

Development takes place on the master branch, which will eventually become FreeBSD 11.x.
Source code for CheriBSD is maintained on GitHub in the following repository:

`https://github.com/CTSRD-CHERI/cheribsd`

The CheriBSD development tree is branched from the FreeBSD GitHub repository at:

`https://github.com/freebsd/freebsd`

CheriBSD may be retrieved from GitHub as follows:

```
$ git clone https://github.com/CTSRD-CHERI/cheribsd
```

7.2 Building CheriBSD

CheriBSD follows the same build instructions as those found in the *BERI Software Reference* chapter on building FreeBSD/BERI, substituting source code from the above Git repository, as well as pathnames and kernel names in build commands.

7.2.1 CheriBSD Build Process

In order to build programs and libraries with support for CHERI as part of CheriBSD, support must be enabled with the `WITH_CHERI128` or `WITH_CHERI256` make options. The `CHERI_CC` must generally be set to the path to our CHERI-aware Clang/LLVM extensions (described in Chapter 2). If an up-to-date version of the `devel/llvm-cheri` package is installed then `CHERI_CC` may be omitted.

Otherwise, the following should be added to the `make buildworld` and `make installworld` commands (for build with 256-bit capabilities):

```
-DWITH_CHERI256 \
CHERI_CC=/path/to/cheri-unknown-freebsd-clang
```

Typically, this argument will be a pointer to the `Build/bin` directory in your CHERI Clang/LLVM build, or `bin` in the CHERI SDK.

Some utility and demonstration software is stored in the `ctsrd` and `tools/tools/atsectl` directories. They can be built with the world by adding the following to the `make buildworld` command line:

```
LOCAL_DIRS="ctsrd tools/tools/atsectl" \
LOCAL_LIB_DIRS=ctsrd/lib \
LOCAL_MTREE=ctsrd/ctsrd.mtree
```

You will similarly need to include the following lines for the `make installworld` target:

```
LOCAL_DIRS="ctsrd tools/tools/atsectl" \
LOCAL_MTREE=ctsrd/ctsrd.mtree
```

To simplify the common use case, the top-level `Makefile` of CheriBSD accepts the `CHERI` make option which may be set to `128` or `256` and sets all values described above except for `CHERI_CC`. For example:

```
make CHERI=256 CHERI_CC=/path/to/cheri-clang -j16 buildworld
```

7.3 Building the CheriBSD Kernel

Support for the capability coprocessor is an optionally compiled kernel extension enabled using `cpu CPU_CHERI`. Ensure that you have replaced a `BERI` kernel configuration-file name with a similar `CHERI` name to ensure that `nocpu CPU_BERI` and `cpu CPU_CHERI` lines have been used. An additional kernel option `CPU_CHERI128` is required to request compilation for 128-bit capabilities. Table 7.1 lists several sample 256-bit kernel configuration files for CHERI-enabled DE4 and simulator kernels. Table 7.2 lists similar configurations but compiled for 128-bit capabilities; these may be used only with a CheriBSD userspace compiled with 128-bit capability support.

A typical build command line for the CheriBSD DE4 kernel is:

```
make CHERI=256 KERNCONF=CHERI_DE4_USBROOT -j16 buildkernel
```

As with conventional FreeBSD/BERI kernels, additional steps are required to construct and configure a memory root filesystem image in CheriBSD kernels; see the *BERI Software Reference* for further information.

Filename	Description
CHERI_DE4_MDROOT	CheriBSD kernel configuration to use a memory root filesystem on the Terasic DE4; 256-bit capabilities
CHERI_DE4_NFSROOT	CheriBSD kernel configuration to use an NFS-based root filesystem on the Terasic DE4; 256-bit capabilities
CHERI_DE4_SDROOT	CheriBSD kernel configuration to use an SD Card root filesystem on the Terasic DE4; 256-bit capabilities
CHERI_DE4_USBROOT	CheriBSD kernel configuration to use a USB root filesystem on the Terasic DE4; 256-bit capabilities
CHERI_SIM_MDROOT	CheriBSD kernel configuration to use a memory root filesystem while in simulation; 256-bit capabilities
CHERI_SIM_SDROOT	CheriBSD kernel configuration to use a simulated SD Card root filesystem; 256-bit capabilities

Table 7.1: 256-bit CheriBSD files in src/sys/mips/conf.

Filename	Description
CHERI128_DE4_MDROOT	CheriBSD kernel configuration to use a memory root filesystem on the Terasic DE4; 128-bit capabilities
CHERI128_DE4_NFSROOT	CheriBSD kernel configuration to use an NFS-based root filesystem on the Terasic DE4; 128-bit capabilities
CHERI128_DE4_SDROOT	CheriBSD kernel configuration to use an SD Card root filesystem on the Terasic DE4; 128-bit capabilities
CHERI128_DE4_USBROOT	CheriBSD kernel configuration to use a USB root filesystem on the Terasic DE4; 128-bit capabilities
CHERI_SIM_MDROOT	CheriBSD kernel configuration to use a memory root filesystem while in simulation; 128-bit capabilities

Table 7.2: 128-bit CheriBSD files in src/sys/mips/conf.

Chapter 8

CheriBSD Kernel

The FreeBSD/BERI kernel has been modified in the following ways to support CHERI's protection features:

- Platform boot code has been extended to enable the capability coprocessor.
- The per-thread PCB context structure has been extended to hold a saved capability register file, as well as a per-thread *trusted stack* that tracks object-capability invocations to provide a reliable return path.
- Kernel context-switching code has been extended to save and restore the capability register file for userspace.
- The kernel debugger has been extended to be able to print CHERI-related information such as the contents of the capability register file.
- The virtual-memory subsystem has been extended to support preserving memory tags for anonymous (swap-backed) memory objects, for which memory mappings are permitted to set the CHERI TLB bits enabling tagged loads and stores. Other memory objects to not (yet) support tagged memory, and TLB bits will not be set – for example, for memory mapped files whose underlying filesystems will be unable to preserve tags.
- New kernel memory-copying routines that can preserve tags on memory have been added. These are used selectively (e.g., in copying register files and in explicitly tag-preserving copies in the VM system), but not for the majority of kernel memory copies. For example, memory copies used in message-oriented IPC, such as those performed to copy data to and from local domain socket buffers, will not preserve tags, as data messages are not intended to carry pointers.
- The kernel's handling of user exceptions has been extended to provide additional capability-related debugging information when userspace protection faults occur.
- The kernel rejects attempts to perform system calls from user threads whose **PCC** (program-counter capability) register does not have the `CHERI_PERM_SYSCALL` user-defined permission bit, preventing sandboxes from directly invoking system services. They must instead invoke a system class that is authorized to invoke system calls. We also hope to introduce new system calls that are safe within sandboxes and are authorized using special user capabilities – e.g., user capabilities that represent kernel file descriptors directly, avoiding the need for interposition – similar to the behavior of Capsicum.

Filename	Description
`sys/mips/cheri/`	CHERI-specific code: coprocessor 2 initialization and context management

Table 8.1: CheriBSD kernel source directories

- The kernel implements `CCall` and `CReturn` fast exception handlers that unseal invoked object capabilities, push the caller state onto the trusted stack, and restore it on return. If a fault occurs in the invoked object, control is returned to the caller.
- The kernel's `ktrace` facility has been extended to allow user object invocation and return to be traced.
- The kernel delivers capability-coprocessor faults in userspace processes as signals, extending the signal trap frame to include capability registers. This allows userspace software (and, in particular, language runtimes) to catch and handle software protection faults.
- The kernel is extended to allow processes that implement sandboxing to export class, method, and object statistics.
- The kernel now supports a new ABI and system-call interface, *CheriABI*, in which all pointers passed to and from the kernel are implemented as capabilities. This allows userspace processes to execute pure-capability-ABI binaries that have no dependence on conventional MIPS pointers.

8.1 CheriBSD Kernel Source Code

CheriBSD contains additions to FreeBSD/BERI to support the CHERI capability coprocessor. Table 8.1 contains a list of kernel directories added in introducing CHERI support.

The majority of kernel changes exist in the `src/sys/mips` subtree, with new CHERI-specific files added to `src/sys/mips/include` and `src/sys/mips/cheri`. The following headers have been added:

`cheri.h` C-language definitions relating to capabilities, usable with both CHERI-aware and CHERI-unaware compiler targets. These include kernel-only context structures such as `struct cheri_kframe` and `struct cheri_signal`, but also context structures shared with userspace, such as `struct cheri_frame` and `struct cheri_stack`. Macro wrappers for inline assembly are provided for CHERI-aware software implemented via CHERI-unaware C, such as the kernel.

`cheriasm.h` This header contains definitions for use in CHERI-aware assembly in both userspace and kernel, such as macros for various CHERI register names, but also kernel-specific code used in exception handling.

`cheric.h` This header provides programmer-friendly C macros wrapping compiler builtins for CHERI register-access, such as `cheri_getbase()` and `cheri_andperm()`. This is used only in userspace due to dependence on CHERI-aware Clang/LLVM.

`cherireg.h` This header provides C macros suitable for use in both C and assembly that specify low-level CHERI constants, such as permission-mask values. It is suitable for use in both kernel and userspace.

`sys/sys/cheri_serial.h` This header provides a structure and definitions supporting serialization of capabilities independent of their size and micro-architectural details. It is suitable for use in both kernel and userspace and is installed outside the sys/mips hierarchy to allow use on non-mips platforms.

The following new C files have been added:

`ccall.S` Assembly-language implementation of CCall and CReturn fast exception handlers; if an error is encountered, then the regular `MipsUserGenException` handler will be jumped to.

`ccall_ktrace.S` Assembly-language implementation of slow path exception handlers used to trace CCall and CReturn invocations.

`cheri.c` The majority of CHERI-specific C code including debugging features, `sysctl`s, initialization for the capability state of threads and processes, handling of `fork`, portions of signal handling, exception logging, and system-call authorization.

`cheri_bcopy.S` CHERI versions of `memcpy` and `bcopy` suitable for use throughout the kernel, but especially in `copyin` and `copyout` scenarios requiring no use of the stack due to the potential for memory-access exceptions.

`cheri_debug.c` CHERI commands for the in-kernel debugger.

`cheri_exception.c` Support for reporting CHERI exceptions and registers on the system console.

`cheri_signal.c` CHERI signal-handling infrastructure.

`cheri_stack.c` CHERI trusted-stack initialization, copying, and unwinding support, as well as `sysarch` system calls to get and set the current trusted stack.

`cheri_syscall.c` CHERI related system-call infrastructure.

`cheriabi_machdep.c` ISA dependent CheriABI support including system call vector declaration, argument parsing, return handling, signal handling, and process memory initialization.

`sys/compat/cheriabi/*` CheriABI ISA-independent implementation. The implementation is modelled on the support for 32-bit binaries in `sys/compat/freebsd32`.

8.2 Capability-Register-File Context Switching

CheriBSD extends the kernel's support for context management and switching to provide each user thread with its own capability-register file. This allows each thread to have its own complete capability state, compatible with the idea that capabilities are compiler-managed. The kernel preserves and restores this state in the user thread control block (PCB) on each context switch, and also performs transformations of that state in order to implement in-thread protection-domain transition via CCall/CReturn on object capabilities.

As the kernel itself makes only minor use of capabilities, and is not currently compiled with a capability-aware compiler, kernel threads do not have full capability-register-file state in their own PCBs. Instead, context switching to the kernel occurs in three ways: an initial saving of the userspace PCC and C0 performed by the hardware and low-level exception handling code, a more complete capability-register context save and restore when transitioning from a low-level exception handler to kernel C code, and finally, saving and restoring of a capability-register-file subset during voluntary kernel context switches.

8.2.1 Thread Control Block and Thread State

CheriBSD extends the kernel's thread control block, `struct pcb`, with four new fields: `pcb_cheriframe`, which saves userspace CHERI registers for the thread, `pcb_cheristack`, which holds the CHERI trusted stack for the thread, `pcb_cheri_signal`, which holds the signal-handling context to install when a signal is delivered within a sandbox, and `pcb_cherikframe`, which holds caller-save registers across a kernel voluntary context switch.

Currently, CheriBSD maintains a 'full' CHERI capability-register context only for userspace, not the kernel, as the kernel is compiled by a CHERI-unaware compiler. Instead, the kernel uses fixed global values for C0 and PCC, and its context switches maintain two capability registers for use in setting up userspace state, performing capability-aware memory copies, etc: C11 and C12.

8.2.2 Context-Switching Philosophy

As with FreeBSD/MIPS, fast exception handlers perform only a partial context switch to the kernel by using reserved registers for exception delivery; if execution of a full C function is required, then full userspace context is saved in the thread's PCB. While the MIPS exception-handling code does this by simply using the two reserved exception-handling registers (K0, K1), the CHERI exception-handling code instead saves the userspace C0 in KR2C so that it can install the kernel's own C0 so that MIPS memory-access instructions work.

Unlike MIPS floating-point support, lazy context switching of capability-coprocessor state is not currently implemented, requiring a full save and restore of the capability register file when entering kernel C code; this has been done in anticipation of larger-scale use of capabilities. One possible optimization to the current design might split the current 32-entry capability register file into two portions, one to be used by the kernel, and the other by userspace, reducing system-call overhead at a cost to register-file size.

8.2.3 PCB Setup and State Changes

A process's CHERI state is initialized in the kernel's `exec_setregs` routine, which is called in the kernel implementation of `execve`. `exec_setregs` calls `cheri_exec_setregs`, which sets up the first thread's live and signal-handling C0, C11 (stack), IDC, and PCC registers for ambient authority. It also calls `cheri_stack_init` to initialize an empty trusted stack.

MIPS machine-dependent state for new threads is set up in `cpu_set_upcall`. Currently, this function calls `cheri_context_copy`, literal cheri_signal_copy, and `cheri_stack_copy` to propagate full CHERI register context, signal-handling context, and the trusted stack from the parent thread to the child. It is not clear this is completely desirable behavior: possibly, new threads should get fresh trusted stacks?

Another important event is the `fork` system call, implemented in the MIPS machine-dependent code in `cpu_fork`. This function calls `cheri_context_copy` to propagate the normal CHERI capability state to the new thread, `cheri_signal_copy` to propagate the signal-delivery context, and `cheri_stack_copy` to copy the current trusted stack to the new thread. The result should be a thread in the child process that exactly replicates the CHERI state of the thread that invoked `fork` in the parent process.

8.2.4 Types of Context Switches

There are a number of types of context switch in the kernel, each affected by additional CHERI thread state:

Low-level exception enter/return When a kernel exception vector is entered, whether from userspace or kernel, `CHERI_EXCEPTION_ENTER` saves the preempted C0 and PCC; likewise, on exception return, `CHERI_EXCEPTION_RETURN` will restore them. This is sufficient to allow general-purpose MIPS loads and stores to be used by MIPS exception handlers without performing a larger context switch for fast-path handlers. If an actual thread context switch takes place later, these saved capability registers, copied to the exception-handling reserved registers, will be stored to an explicit in-memory register frame later.

User-to-kernel switch If the low-level exception handler must make a full context switch to the kernel (e.g., save all MIPS and CHERI registers), then `SAVE_U_PCB_CHERIFRAME` will be used, which populates the PCB's CHERI register frame; that state can later be restored by `RESTORE_U_PCB_CHERIFRAME`. This occurs if an exception delivered to user code cannot be satisfied without entering C code – e.g., to process a full VM fault rather than just a TLB miss that a low-level assembly handler can resolve.

Kernel involuntary switch If an exception preempts the kernel itself (e.g., a kernel TLB miss or interrupt), the full MIPS general-purpose state will be saved by `MipsKernGenException`, but only C11 and C12 will be preserved, as the kernel currently uses only these capability registers. In the future, this code will need to preserve a full kernel CHERI frame; as the kernel register frame is stored on the stack, this may require larger kernel stacks.

Kernel voluntary switch If a kernel thread sleeps, perhaps due to blocking on a mutex or waiting on I/O, then it will perform a voluntary context switch via `cpu_switch`, which in turn will call `SAVE_U_PCB_CHERIKFRAME` to `RESTORE_U_PCB_CHERIKFRAME` to save and restore caller-save registers in the PCB.

8.3 CCall/CReturn Fast Exception Handlers

The kernel implements CCall and CCReturn via a combined exception handler, `CHERICCallVector` that in turn branches to either `CHERICCall` or `CHERICReturn`. Both implementations perform the tests described in the *CHERI Instruction-Set Architecture*; in addition, CCall expects that an ABI in which the invoked code capability is always placed in C1, and the invoked data capability is always placed in C2, allowing the kernel to avoid software instruction decoding. If either path detects an error, CSetCause will be used to set the capability-cause register, and `MipsUserGenException` will be called to enter the kernel's general user exception handler. CCall will push IDC, PCC, and PC+4; CReturn likewise pops IDC, PCC, and PC. In the future, it may be desirable to save PC within PCC.

When the kernel is compiled with `ktrace` support, the trace flags of the calling process are checked before trusted stack manipulation and slow path handlers `CHERICCallKtrace` and `CHERICReturnKtrace` are called if tracing is enabled. These functions are near duplicates of `MipsUserGenException`. They differ in that they return to the fast path handlers in `ccall.S` on completion rather than directly user space and they call logging functions rather than `trap`. Enabling `ktrace` support adds a number of memory accesses to the CCall and CReturn fast path and it should be disabled in performance critical applications.

8.4 Trusted-Stack Manipulation

The trusted stack keeps a record of the CReturn part for each user thread, and is a key part of the kernel's CHERI state for each thread (`pcb_cheristack`). The trusted stack is initialized as empty when the first thread in a process is created. The trusted stack is inspected or modified in the following situations:

CCall exception Frames are pushed onto the trusted stack by invocations of the CCall instruction, which causes the kernel's exception handler to push values of PCC, C0, and PC+4 to restore when the frame is popped.

CReturn exception Frames are popped from the trusted stack by invocation of the CReturn instruction, which causes the kernel's exception handler to pop values of PCC, C0, and PC.

`CHERI_GET_STACK` Userspace code may query the trusted stack using the `CHERI_GET_STACK` operation on the `sysarch` system call. Currently, the kernel and userspace share the same representation of the stack; in the future, we will want to diverge the two, and also provide a way for userspace to query trusted-stack size.

`CHERI_SET_STACK` Userspace code may set the contents of the trusted stack using the `CHERI_SET_STACK` operation on the `sysarch` system call. The kernel will validate that the stack is approximately valid before installing it, and return an error if the stack is invalid. This call might be used by sandboxing frameworks or language runtimes to unwind the stack in the event of an exception delivery; this requires careful simultaneous rewriting of the current general-purpose and CHERI register frames, and (with care) is safe to do from a signal-handling context.

`cheri_stack_unwind` If a signal is delivered to a thread that is executing sandboxed code, and suitable signal-handling configuration has not been set up to safely receive the delivered signal, then for certain signals the kernel will automatically unwind the stack back to the caller of the sandbox. This occurs in `cheri_stack_unwind`.

DDB The `show cheristack` command will dump the current thread's trusted stack from the kernel debugger.

8.5 CHERI-Aware Signal Handling

UNIX signal handling is at best a tricky business, and CHERI does not simplify the problem. In a regular MIPS or even CHERI-aware process that does not make use of sandboxing (protection domains), signal handling is unmodified. However, things are more complex when a signal must be delivered while there is code executing in a sandbox in the target thread. There are several cases:

Sandboxed code, trap signal arises, no registered signal handler If a thread triggers a trap signal (e.g., `SIGSEGV` or `SIGTRAP`) due to an exception, the signal is uncaught (i.e., it doesn't have a registered handler), and it is tagged as `SIGPROP_SBUNWIND`, then the kernel can perform an automatic trusted-stack unwind using `cheri_stack_unwind`, returning control to the caller.

For the purposes of this case, 'sandboxed code' means that one or more frames are present on the trusted stack, rather than that the current execution context lacks privilege.

Sandboxed code, registered signal handler, no alternative signal stack If a thread will have a signal delivered to a registered handler, then we must install a suitable signal-handling context (typically, ambient authority). As we cannot trust the stack present in the sandbox context, we instead will use the UNIX alternative signal stack. If one is not defined for the thread, the process will be terminated as there is no safe way to handle the signal.

For the purposes of this and the following case, 'sandboxed code' means that the current executing context does not have ambient authority – i.e., that it cannot invoke system calls.

Sandboxed code, registered signal handler, alternative signal stack If an alternative signal stack is configured, then ambient authority will be temporarily restored and signal delivery will take place on the alternative stack. Currently, the kernel installs ambient-authority capabilities in **PCC**, **C0**, **C11** (stack capability), and **IDC** prior to executing the signal handler. When the signal handler returns, the kernel will restore capability-register state saved on the stack as it would general-purpose register state. This will release ambient authority if the saved (and possibly rewritten) register state does not hold it.

Configuring an alternative signal stack requires that a signal stack be allocated and registered with `sigaltstack` and the signal handler be registered to use it with the `SA_ONSTACK` flag in `sigaction`.

In addition to the general-purpose register frame installed on the signal handling stack by the kernel, CheriBSD also installs a copy of the capability-register file from the preempted thread. As with the general-purpose register frame, the kernel will copy modifications to the CHERI register frame back to the thread's capability register file on return from the signal handler, allowing rewriting of its capability state.

User code can access the saved capability-coprocessor register values, including the capability-cause register (`cf_capcause`), via a `struct cheriframe` pointed to by the `uc_mcontext.mc_cp2state` pointer in the `context_t` argument to the signal handler. The handler should check that the `mc_cp2state` pointer is non-`NULL`, and that the corresponding `uc_mcontext.mc_cp2state_len` field is equal to `sizeof(struct cheriframe)`, before proceeding. This ABI is currently immature, as the same data structure is used both for the kernel's internal representation of the capability register file and its on-stack representation; this will change in a future version of CheriBSD.

As ambient authority is installed, signal handlers are also able to rewrite the trusted stack. This allows more mature handling of exceptions within sandboxes or other invoked contexts – for example, unwinding of the trusted stack, garbage collection activities, etc. In CheriBSD's `cheritest` tool, this is used to handle timeouts triggered by `SIGALRM`, terminating sandboxes if they overrun their execution-time limit, for example.

A key design choice is that signal handlers are not invoked by a CCall-like mechanism. This is done for several reasons, not least that we wish to be able to handle trusted-stack overflow in userspace via a signal handler. Great care must be exercised in writing signal handlers that execute with ambient authority in order to not leak privileges to a non-ambient context.

8.6 Copying Memory

The kernel's `bcopy`, `memcpy`, `copyin`, and `copyout` routines are capability-unaware and will not preserve tag bits. New `cheri_bcopy`, `cheri_memcpy`, `copyincap`, and `copyoutcap` are used in situations where preserving tags is desirable – such as copying in or out of CHERI trusted stacks. Clearing tag bits across conventional IPC, system call arguments, and so on is import in preventing the accidental leaking of rights between address spaces where only data copies are intended.

8.7 Tracing Extensions

CheriBSD includes support for tracing capability exceptions, as well as CCall, and CReturn handlers via the `ktrace` framework. When the kernel is build with

```
options  KTRACE
```

then these points can be enabled with the `ktrace` and `kdump` commands' `-t` flag and the `e` and `C` arguments (for exceptions, and CCall/CReturn respectively.) Note that enabling `options KTRACE` imposes a significant performance impact on CCall and CReturn.

8.8 Kernel Debugger Extensions

CheriBSD includes a number of minor extensions to the FreeBSD kernel debugger:

show cheri Dump the current kernel thread's CHERI register file.

show cheriframe Dump the current user thread's saved CHERI register file.

show cheristack Dump the current user thread's CHERI trusted stack.

Normally, userspace CHERI exceptions are delivered as signals or trigger termination/core dumps. Sometimes it is useful to instead enter the kernel debugger, which makes it easier to inspect stack and register state. A set of `sysctl`s enables this in various situations:

`security.cheri.debugger_on_exception` Enter the kernel debugger when a thread triggers a CHERI exception or a system call is blocked due to an attempt to invoke it from a non-ambient context.

`security.cheri.debugger_on_sandbox_unwind` enter the kernel debugger when an automatic trusted-stack unwind would take place due to an unhandled trap exception within a sandbox.

The following commands, run as root, will enable these `sysctl`s:

```
# sysctl security.cheri.debugger_on_exception=1
# sysctl security.cheri.debugger_on_sandbox_unwind=1
```

Kernel execution can be restarted using the `continue` command from the debugger prompt.

8.9 CheriABI

The new CheriABI Application Binary Interface (ABI) introduces a new process execution environment in which pure-capability CHERI code can be executed. The kernel expects that all pointers passed via the system-call interface, and also other interfaces such as command-line arguments and environmental variables, ELF auxiliary arguments, signal handling, and so on, will also be via capabilities rather than MIPS pointers. This feature can be enabled by compiling `options COMPAT_CHERIABI` into the kernel, but is currently considered experimental.

Chapter 9

CheriBSD Userspace

The FreeBSD/BERI userspace has been modified in the following ways to support CHERI's protection features:

- The `libprocstat(3)` library and `procstat(1)` command have been extended to inspect exported sandbox statistics.
- A new library, `libcheri(3)`, has been added to provide a sandbox API, and to implement a set of system-class objects that can be delegated to sandboxes. Currently, this consists of a singleton system object that provides the ability to print to `stdout`, and a file-descriptor class that allows delegation of individual kernel-provided file descriptors to sandboxes.
- A new library, `libc_cheri(3)`, has been added to provide core C-language APIs and services within sandboxes. This library is able to use the system and file-descriptor classes to provide access to APIs such as `printf()`.
- A new command-line tool, `cheritest`, implements test cases for a variety of capability-related functions including sandboxing; `cheritest` relies on `cheritest-helper.bin` to provide sandboxed code.
- A new command-line tool, `cheri_tcpdump`, implements sandboxed packet sniffing and parsing; `cheri_tcpdump` relies on `tcpdump-helper` to provide sandboxed code.
- A new library, `libz-cheri(3)`, implements compression routines with fine-grained memory protection.

9.1 CheriBSD Userspace Source Code

CheriBSD contains additions to FreeBSD/BERI to support the CHERI capability coprocessor. Table 9.1 contains a list of directories affected by CHERI-specific behavior.

Filename	Description
`bin/cheritest/`	Command-line utility exercising CHERI and CheriBSD features, including sandboxing
`ctsrd/`	CTSRD-project demo code
`lib/libc_cheri/`	In-sandbox C library/runtime
`lib/libcheri/`	Library implementing the CHERI sandbox API; the CHERI system class implementation
`libexec/cheritest-helper/`	Sandboxed components for cheritest
`libexec/tcpdump-helper/`	Sandboxed components for cheri_tcpdump initialization and context management
`lib/libz-cheri`	Version of libz compiled with CHERI memory protection
`usr.sbin/tcpdump/cheri_tcpdump`	Version of tcpdump able to use CHERI sandboxing
`lib/libprocstat/`	Extensions to this library allow procstat(1) to monitor libcheri sandboxes
`usr.bin/procstat/`	procstat(1) command extended to monitor libcheri standboxes

Table 9.1: CheriBSD userspace source directories

Bibliography

[1] 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.

[2] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, S. J. Murdoch, P. G. Neumann, and A. Richardson. Clean application compartmentalization with SOAAP (extended version). Technical Report UCAM-CL-TR-873, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, Dec. 2015.

[3] K. Gudka, R. N. M. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015)*, October 2015.

[4] 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.

[5] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson. *The Design and Implementation of the FreeBSD Operating System, Second Edition*. Pearson Education, 2014.

[6] P. G. Neumann and R. N. M. Watson. Capabilities revisited: A holistic approach to bottom-to-top assurance of trustworthy systems. In *Fourth Layered Assurance Workshop*, Austin, Texas, December 2010. U.S. Air Force Cryptographic Modernization Office and AFRL. http://www.csl.sri.com/neumann/law10.pdf.

[7] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, P. G. Neumann, and J. Woodruff. Bluespec Extensible RISC Implementation (BERI): Software Reference. Technical Report UCAM-CL-TR-853, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, June 2014.

[8] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, P. G. Neumann, and J. Woodruff. Capability Hardware Enhanced RISC Instructions (CHERI): User's guide. Technical Report UCAM-CL-TR-851, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, June 2014.

[9] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, P. G. Neumann, and J. Woodruff. Bluespec Extensible RISC Implementation: BERI Software reference. Technical Report UCAM-CL-TR-869, University of Cambridge, Computer Laboratory, Apr. 2015.

[10] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, P. G. Neumann, and J. Woodruff. Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide. Technical Report UCAM-CL-TR-877, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, Nov. 2015.

[11] R. N. M. Watson, P. G. Neumann, J. Woodruff, J. Anderson, D. Chisnall, B. Davis, B. Laurie, S. W. Moore, S. J. Murdoch, and M. Roe. Capability Hardware Enhanced RISC Instructions (CHERI): Instruction-Set Architecture. Technical Report UCAM-CL-TR-850, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, June 2014.

[12] R. N. M. Watson, P. G. Neumann, J. Woodruff, J. Anderson, D. Chisnall, B. Davis, B. Laurie, S. W. Moore, S. J. Murdoch, and M. Roe. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture. Technical Report UCAM-CL-TR-864, University of Cambridge, Computer Laboratory, Dec. 2014.

[13] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S. W. Moore, S. J. Murdoch, R. Norton, and S. Son. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture. Technical Report UCAM-CL-TR-876, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, Nov. 2015.

[14] R. N. M. Watson, J. Woodruff, D. Chisnall, B. Davis, W. Koszek, A. T. Markettos, S. W. Moore, S. J. Murdoch, P. G. Neumann, R. Norton, and M. Roe. Bluespec Extensible RISC Implementation (BERI): Hardware Reference. Technical Report UCAM-CL-TR-852, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, June 2014.

[15] R. N. M. Watson, J. Woodruff, D. Chisnall, B. Davis, W. Koszek, A. T. Markettos, S. W. Moore, S. J. Murdoch, P. G. Neumann, R. Norton, and M. Roe. Bluespec Extensible RISC Implementation: BERI Hardware reference. Technical Report UCAM-CL-TR-868, University of Cambridge, Computer Laboratory, Apr. 2015.

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

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

[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.