CHERI
Capability Hardware Enhanced RISC Instructions
User's Guide

Version 1.7

This interim document is not released for public consumption

Robert N. M. Watson, David Chisnall, Brooks Davis,
Wojciech Koszek, Simon W. Moore, Steven J. Murdoch, Jonathan Woodruff
SRI International and the University of Cambridge[1]

March 5, 2014

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

Contents

Abstract

This document is the *User's Guide* for the research-prototype implementation of the Capability Hardware Enhanced RISC Instructions (CHERI) Instruction Set Architecture (ISA) developed by SRI International and the University of Cambridge. It complements the *BERI Hardware Reference* by providing information required by hardware and software developers working with the prototype. The document is intended to capture our evolving architecture, which is being refined, tested, and formally analyzed.

This report is targeted at system developers bringing up an operating system and compiler stack for CHERI; future versions of this document will also address end users. The guide describes the CheriBSD operating system, the CHERI-adapted Clang/LLVM compiler suite, and the Deimos demonstration microkernel.

Acknowledgments

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

Peter G. Neumann	Ross J. Anderson	Jonathan Anderson	Gregory Chadwick
Nirav Dave	Khilan Gudka	Jong Hun Han	Alex Horsman
Alexandre Joannou	Asif Khan	Myron King	Ben Laurie
Patrick Lincoln	Anil Madhavapeddy	Ilias Marinos	Ed Maste
Andrew Moore	Will Morland	Alan Mujumdar	Prashanth Mundkur
Robert Norton	Philip Paeps	Michael Roe	Colin Rothwell
John Rushby	Hassen Saidi	Muhammad Shahbaz	Stacey Son
Richard Uhler	Philip Withnall	Bjoern Zeeb	

The CHERI team wishes thank its external oversight group for significant support and contributions:

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

Finally, we are grateful to Howie Shrobe, MIT professor and past DARPA CRASH program manager, who has offered both technical insight and support throughout this work.

Chapter 1

Introduction

This is the *User's Guide* for the Capability Hardware Enhanced RISC Instructions (CHERI) prototype that complements the *CHERI Architecture Document*, which specifies the CHERI architecture and ISA, and the *BERI Hardware Reference*, which describes the hardware prototype, and the *BERI Software Reference*, which describes the BERI software development environment.

The User's Guide describes the CheriBSD prototype, a version of the Clang/LLVM/gas toolchain used with CHERI, and the Deimos demonstration microkernel. The Guide is intended to address the needs of hardware and software developers who are prototyping new hardware features, bringing up operating systems, language runtimes, and compilers on CHERI, rather than literal end users. Future iterations will continue to flesh out operational aspects of the CHERI processor.

1.1 Background

Capability Hardware Enhanced RISC Instructions (CHERI), developed by SRI International and the University of Cambridge, are security extensions for the MIPS64 Instruction Set Architecture (ISA). The CHERI ISA provides direct hardware 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 compartmentalization, CHERI's ISA features support fast and easy application compartmentalization of C-language systems software. These improvements are made possible by integrated hardware support for continuous memory protection and enforcement using memory segments and the object capability model. Contemporary software trusted computing bases (TCBs) such as operating system kernels and language runtimes are particularly interesting, 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 may be found in the *CHERI Architecture Document*, including new co-processor 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. CHERI is based on the Bluespec Extensible RISC Implementation (BERI) FPGA soft core, and implemented as an additional coprocessor. The distinction

between BERI and CHERI is evolving; however, the long-term hope is that BERI will be a reusable platform across multiple research projects in the hardware-software interface.

1.2 Licensing

CheriBSD and the CHERI-enhanced version of the Clang/LLVM have been released under BSD licenses. Modifications to the `gas` assembler have been released under GPLv2.

The Deimos demonstration microkernel is currently proprietary, although we intend to release it under a BSD license in the future.

1.3 Version History

This is the eighth version of the *CHERI User's Guide*.

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 is now 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 has been moved to the *BERI Software Reference*. A short chapter describing CheriBSD has been retained, and has been updated to reflect a migration from Perforce to Github. Information on the CHERI Clang/LLVM compiler has been updated to include new C-language extensions. CheriBSD build instructions are extended as the CHERI Clang/LLVM compiler is now required; information on CheriBSD extensions to FreeBSD has been expanded.

1.4 Document Structure

This document is an introduction and user manual for Version 1 of the Capability Hardware Enhanced RISC Instructions (CHERI) CPU prototype:

Chapter 2 describes CheriBSD.

Chapter 3 describes CHERI extensions to Clang/LLVM and the C programming language.

Chapter 4 describes Deimos, a prototype operating system software stack for CHERI. This microkernel is able to run on CHERI in simulated and synthesized forms, illustrating some of CHERI's protection and compatibility properties.

Chapter 2

CheriBSD

FreeBSD/BERI is a port of the open-source FreeBSD operating system that extends support for the Bluespec Extensible RISC implementation (BERI). General crossbuild and use instructions for FreeBSD/BERI may be found in the *BERI Software Reference*.

CheriBSD extends FreeBSD/BERI to support CHERI, and is described in this chapter. In general, procedures for building and using FreeBSD/BERI should entirely apply to CheriBSD, except as documented in this chapter.

2.1 CheriBSD Modifications

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

- Platform boot code has been extended to enable the capability co-processor.
- 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's handling of user exceptions has been extended to provide additional debugging information when userspace protection faults occur.
- The kernel rejects attempts to perform system calls frok user threads whose C0 register does not hold ambient authority, preventing sandboxes from using systrm services. In the future, this will be accomplished by a user permission bit rather than a kernel-determined policy. Also, we 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 along the lines of Capsicum.

- The kernel implements CCall and CReturn fast exception handlers which unseal invoked object capabilities, pushing the caller state onto the trusted stack, and restoring it on return. If a fault occurs in the invoked object, control is returned to the caller.

- The kernel is extended to allow processes implementing sandboxing to export class, method, and object statistics.

- 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, as well as implement a set of system-class objects that can be delegated to sandboxes. Currently this consists of a singleton system object providing the ability to print to stdout, and a file-descriptor class allowing individual kernel-provided file descriptors to be delegated to sandboxes.

- A new library, libc_cheri(3), has been added to provide core C-language APIs and services within sandboxes; it 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 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.

2.2 Obtaining FreeBSD/BERI and CheriBSD Source Code

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`

Development takes place on the master branch which will eventually become FreeBSD 11.x. CheriBSD may be retrieved from GitHub as follows:

```
$ cd /pool/users/${USER}
$ git clone https://github.com/CTSRD-CHERI/cheribsd
```

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
sys/mips/cheri/	CHERI-specific code: coprocessor 2 initialisation 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 2.1: CheriBSD directories

2.3 About FreeBSD/BERI and CheriBSD

CheriBSD contains additions to FreeBSD/BERI to support the CHERI capability coprocessor. Table 2.1 contains a list of directories affected by CHERI-specific behaviour. Table 2.2 lists CHERI-specific files in the common MIPS configuration directory.

2.4 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.

Portions of the CheriBSD tree rely on our CHERI-aware clang/LLVM extensions (described in Chap-

Filename	Description
CHERI_DE4_MDROOT	CheriBSD kernel configuration to use a memory root file system on the Terasic DE4
CHERI_DE4_SDROOT	CheriBSD kernel configuration to use an SD Card root file system on the Terasic DE4
CHERI_SIM_MDROOT	CheriBSD kernel configuration to use a memory root file system while in simulation
CHERI_SIM_SDROOT	CheriBSD kernel configuration to use a simulated SD Card root file system

Table 2.2: CheriBSD files in `src/sys/mips/conf`.

ter 3). You must tell `make buildworld` where to find the extended clang/LLVM by adding this argument to the build command line:

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

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

Some utility software and software used for demos is stored in the `ctsrd` and `tools/tools/atsectl` directories. They can 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
```

Support for the capability coprocessor is an optionally compiled extension enabled using `cpu CPU_CHERI` in CheriBSD. 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.

Chapter 3

CHERI Clang/LLVM

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

3.1 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 is gradually being deprecated in favour of the LLVM integrated assembler, but is still used in a number of places including the CHERI test suite and CheriBSD kernel build.

3.2 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
```

3.3 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
```

NB: You will need a recent version of CMake (at least 2.8.8).

Further NB: the `std*.h` files included with Clang/LLVM appear to be incompatible with those shipped in the FreeBSD base, so 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/3.4/include/std*
```

If this is not done, then the CheriBSD build may experience compiler errors relating to variable-argument functions and type. This will hopefully be resolved in the future by upstreaming 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
```

Linking 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
```

3.4 Building a Complete SDK

On a FreeBSD host system, if you have access to the CHERI svn repository, you can run `cherilibs/trunk/tools/build_sdk.sh`. 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.

3.5 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
```

And can generate native code directly, like this:

```
$ 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 are to 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 `nops` in delay slots with instructions from before the branch. It 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.

3.6 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}
```

3.7 Capability Extensions to C

Clang predefines a `__capability` macro. If you want to make your code portable to non-CHERI platforms, then you can begin it with:

```
#if !defined(__CHERI__) && !defined(__capability)
#   define __capability
#endif
```

You can then use the `__capability` qualifier on any pointer. This pointer will be treated as a capability with the following semantics:

- Casts to integers return the base address.
- Casts from integers are treated as C0-relative[1].
- Relative addressing will use the load/store via capability instructions
- Pointer increments will return a new capability, which may be invalid if it exceeds the upper bound of the original.
- Pointer decrements will always fail.

Clang also provides a number of built-in functions for accessing aspects of these semantics:

```
size_t __builtin_cheri_get_cap_base(__capability void *);
size_t __builtin_cheri_get_cap_length(__capability void*);
size_t __builtin_cheri_get_cap_perms(__capability void*);
size_t __builtin_cheri_get_cap_type(__capability void*);
_Bool  __builtin_cheri_get_cap_tag(__capability void*);
_Bool  __builtin_cheri_get_cap_unsealed(__capability void*);
__capability void* __builtin_cheri_inc_cap_base(__capability void*,
   size_t);
__capability void* __builtin_cheri_set_cap_length(__capability void*,
   size_t);
__capability void* __builtin_cheri_and_cap_perms(__capability void*,
   size_t);
__capability void* __builtin_cheri_set_cap_type(__capability void*,
   size_t);
__capability void* __builtin_cheri_seal_cap_code(__capability void*);
__capability void* __builtin_cheri_seal_cap_data(__capability void*,
   __capability void*);
__capability void* __builtin_cheri_unseal_cap(__capability void*,
   __capability void*);
```

Most of these aspects correspond directly to the relevant capability inspection / modification instructions.

[1]This means that a `__capability void*` to `uintptr_t` to `__capability void*` round trip will only work if the capability references something inside the C0 address space

3.7.1 Querying Reserved Capability Registers

Several capability registers are set aside in the ISA and ABI for specific functions. These can be queried using the following builtins:

```
__capability void *__builtin_cheri_get_global_data_cap(void);
__capability void *__builtin_cheri_get_invoke_data_cap(void);
__capability void *__builtin_cheri_get_kernel_cap1(void);
__capability void *__builtin_cheri_get_kernel_cap2(void);
__capability void *__builtin_cheri_get_kernel_code_cap(void);
__capability void *__builtin_cheri_get_kernel_data_cap(void);
__capability void *__builtin_cheri_get_exception_program_counter_cap(
   void);
```

Only builtins querying the global and invoke data capabilities will be available to regular userspace code.

3.7.2 Other Builtins

Clang provides builtins corresponding to the `CGetCause` and `CSetCause` instructions:

```
size_t __builtin_cheri_get_cause(void)
void __builtin_cheri_get_cause(size_t)
```

These are only useful for writing kernel code, as the instructions that they correspond to are privileged.

3.7.3 Inline Assembly

Clang adds a `"C"` constraint to inline assembly. This allows capability registers to be used as input and output for inline assembly blocks. For example:

```
__capability void* getC0(void)
{
    __capability void *c0;
    __asm__("cmove␣%0,␣$c0" : "+C" (c0));
    return c0;
}
```

This will return the current **C0** value.

3.7.4 Const and Capabilities

In C/C++, you can explicitly cast a `const` pointer to a non-`const` pointer. This casting is not supported for `__capability` pointers. For example, consider the following two `__capability` pointers:

```
__capability void *a = something;
const __capability void *b;
```

This action will implicitly clear the store and store-capability flags:

```
b = a;
```

So far, this casting is almost exactly like the C behavior, except that now you will get a hardware trap instead of just a compiler warning if you try to store through `b`. If you perform the assignment in the other direction, however, the constness is still preserved:

```
a = (__capability void)b;
```

Any attempts to store through `a` will cause a trap.

3.7.5 Output-Only Capabilities

Clang also now supports an `__output` qualifier on capabilities. These qualifiers are the twin of **`const`** capabilities: they can only be written to, not read. You can create a write-only capability by casting any other capability. The compiler will statically check attempts to write through an `__output` capability, and code that does so via an explicit cast will abort at run time.

```
int readFail(__capability __output int *x)
{
    *x = 12;
    (*x)++;
    return *x;
}
```

This code contains two bugs: two attempts to read through the pointer passed as a parameter, even though it is declared as being solely for output. Attempting to compile it will raise the following warnings:

```
writetest.c:4:2: error: write-only variable is not readable
        (*x)++;
        ^~~~
writetest.c:5:9: error: write-only variable is not readable
        return *x;
               ^~
```

You can call this function transparently, without an explicit cast:

```
int caller(__capability int *x)
{
    return readFail(x);
}
```

In this case, the caller will automatically modify the capability to remove load and load-capability permissions before passing it.

3.7.6 Capability Implicit Range Checking

At optimization level 1 and above, LLVM will attempt to automatically limit the range of capabilities. At **`-O0`**, the compiler will not transform the code at all. If static code flow analysis can determine that the capability was constructed from a global or a stack allocation, LLVM will initially begin with its range limited to the size of that allocation. This will also work for various heap allocation functions, including any allocations that are the direct result of casting from the result of one of the standard C allocation functions. Note that this analysis is *not* interprocedural, and so will not work in the general case. It is primarily intended as a way to provide a small amount of extra checking.

3.7.7 Opaque Types

CHERI-clang supports a new pragma, opaque, for linking a type to a key. This pragma is intended to provide enforcement of opaque types in C: no code outside of a compilation unit (where the key is visible) can dereference the opaque pointers, even after casting. This pragma is intended to be used as follows:

```
1   // In a public header (one or more of the following):
2   typedef __capability struct foo* foo_c;
3   typedef struct foo* foo_t;
4
5   // In either the implementation file or a private header:
6
7   // If we're in private header, these should be __attribute__
8   // ((visibility("hidden"))), otherwise they should be static:
9   void *ptrKey;
10  __capability void *capKey;
11
12  struct foo
13  {
14      int a, b;
15  };
16
17  #pragma opaque foo_c capKey
18  #pragma opaque foo_t ptrKey
```

In an implementation file, any function returning a `foo_t` will have its value xor'd with the value of `ptrKey` before return and any function returning a `foo_c` will have the value sealed with `capKey` before return. Similarly, any function receiving a `foo_c` or `foo_t` as an argument will have the inverse applied. Within the function, `foo_t` and `foo_c` behave exactly as they would without the pragma.

3.7.8 Object-Capability Invoke Calling Convention

By default, code generated for functions taking capability arguments or returning capability values will conform to the normal CHERI calling convention. However, functions prototypes may be tagged with `__attribute__((cheri_ccall))` to indicate to the compiler that the first two capability arguments should be placed in C1 and C2 rather than the normal argument registers. This is currently used when the function being called will perform a CCall to another protection domain, which requires the object-capability code and data capabilities to be in those registers. Typically, the invoked function will be an assembly function, `cheri_invoke`, which saves and clears registers as required before invoking potentially untrustworthy code, and then properly restores register state following CReturn. It would be very rare for end-user code to employ this attribute directly, but it will be of value to developers creating new sandboxing models.

3.7.9 Stack Spills and Safety

Functions declared with the `sensitive` attribute deal with some kind of sensitive data. C already provides tools (**`volatile`** in older versions and explicit memory operations in C11) to ensure that sensitive data is

not left in areas of memory managed by the programmer. C does not provide a mechanism for making the same guarantee for stack spills.

The traditional way of avoiding this problem is to use **`volatile`** variables everywhere and explicitly zero them. However, this effectively disables all compiler optimization for a function, purely to avoid the possibility of a stack spill. In general, however, higher optimization levels are more likely to reduce the need to ever spill registers to the stack. The correctness of such code is difficult to determine in the presence of optimizations such as common subexpression elimination. These optimizations may cause accidentally storage of intermediate results on the stack for future reuse that could persist beyond the end of the function.

Marking a function as sensitive does not disable *any* optimizations. It does, however, ensure that any values spilled to the stack during the function are zeroed at the end. This zeroing is especially important for CHERI systems, where stack spills include capabilities. The pass does not attempt to destroy the entire capability; it simply writes a 64-bit zero value over the start. This write is enough to destroy the base address (without which the rest of the information in the capability is largely useless) and, more importantly, invalidate the tag. With the tag invalidated, the capability is rendered invalid.

Note that the current implementation may still spill values to the stack if they were left in a callee-save register across function calls and the callee spills them. To avoid this issue, the compiler will warn if you call a function that is not marked as sensitive from one that is. One possible approach would be to store all callee-save registers to the stack and invalidate them before every call, but that would impose a significant overhead. Instead, we aim to provide tools that allow programmers to write secure code and to decide when these trade-offs are appropriate.

In typical use, the overhead of this attribute is a few instructions in the function epilog (or epilogs, if it has multiple return paths). Of course, this overhead depends on the complexity of the function and the size of its working set. In general, it does improve significantly at higher optimization levels where the register allocator eliminates most, if not all, stack spills. Spill slots are often reused, so the number of invalidations required is often lower than the total number of spills over the program.

We do not invalidate the spills in the function prolog, because these spills are unlikely to contain any sensitive data. They contain the initial stack pointer, the globals pointer and the return address. Once we begin using capabilities for these, invalidating them may be desirable.

3.8 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.

3.8.1 Capability Move

The `cmove` pseudo operation expands to a `CIncBase` 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.

3.8.2 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:

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

This will expand to:

```
clw   $1, $zero, 32($c2)
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 4

The Deimos Demonstration Operating System

Deimos is a demonstration microkernel operating system that uses the CHERI ISA's capability features to sandbox untrustworthy applications. For the purposes of this demonstration, the CHERI prototype CPU was implemented in the Terasic tPad platform using an Altera FPGA; the tPad includes a VGA touchscreen, which is used for the Deimos user interface.

This chapter is largely historical in nature, reflecting our early experimentation with the CHERI ISA; Deimos as demonstrated in November 2011 is not able to run on current CHERI hardware due to ISA changes. It was also unable to take advantage of a number of contemporary CHERI features, such as access to a CHERI-aware compiler.

4.1 Demonstration Narrative

Figure 4.1 illustrates the demonstration user interface: two untrusted applications, a touchscreen drawing application and a weather status bar, as well as a trustworthy status bar across the top of the display maintained by the operating system. All applications run within the kernel ring and a single address space that is traditionally reserved only for privileged applications. Using CHERI's memory capability features, it is partitioned into a supervisor and two sandboxes. Access to I/O devices (such as touchscreen input and portions of the frame buffer) is delegated to the sandboxes using the capability mechanism to provide hardware-enforced control over display access. Attempts to draw outside of the delegated display area trigger a hardware exception, returning control to the supervisor.

In the demonstration, all touchscreen input is sent to the on-screen drawing application – drawing outside of its delegated area using a stylus causes the application to "misbehave"; attempting to write outside its screen region triggers an exception. The supervisor displays an exception indicator that shows when a disallowed access has been requested. The sandbox is then restarted on the next instruction, allowing it to continue. Over 80% of code in the Deimos kernel and applications is portable C and stock 64-bit MIPS assembly code, requiring only a very small amount of CHERI-specific assembly in order to manage additional register state, delegate capabilities, and perform I/O access via capabilities.

The demonstration shows off a number of features of the CHERI architecture:

- A hybrid capability model, allowing conventional C and MIPS assembly to coexist, transparently, with a hardware-enforced sandbox model.

Figure 4.1: The Deimos touchscreen interface

- Selective delegation and containment of I/O access within the kernel ring, illustrating how device drivers (for example) can be contained using the capability model.
- The flexibility of CHERI hybridisation: the ability to run an operating system and applications using solely capability protection features, without employing the TLB.
- How CHERI features, combined with hardware user interface elements and software UI components, can be used to implement a trusted path.

4.2 Deimos Design and Implementation

Figure 4.2 illustrates the rough structure of the Deimos demonstration:

- $10,500$ lines of Bluespec that implement a fully pipelined, 64-bit MIPS CPU, including a capability coprocessor as CP2.
- $2,950$ lines of C and assembly code that implement a preemptive, multi-tasking microkernel, which uses capabilities for sandboxing of untrustworthy applications.
- 640 lines of C and assembly code that implement supporting libraries for sandboxed applications, including graphics and string management.
- 140 and 90 lines, respectively, of C and assembly code for two touchscreen applications.

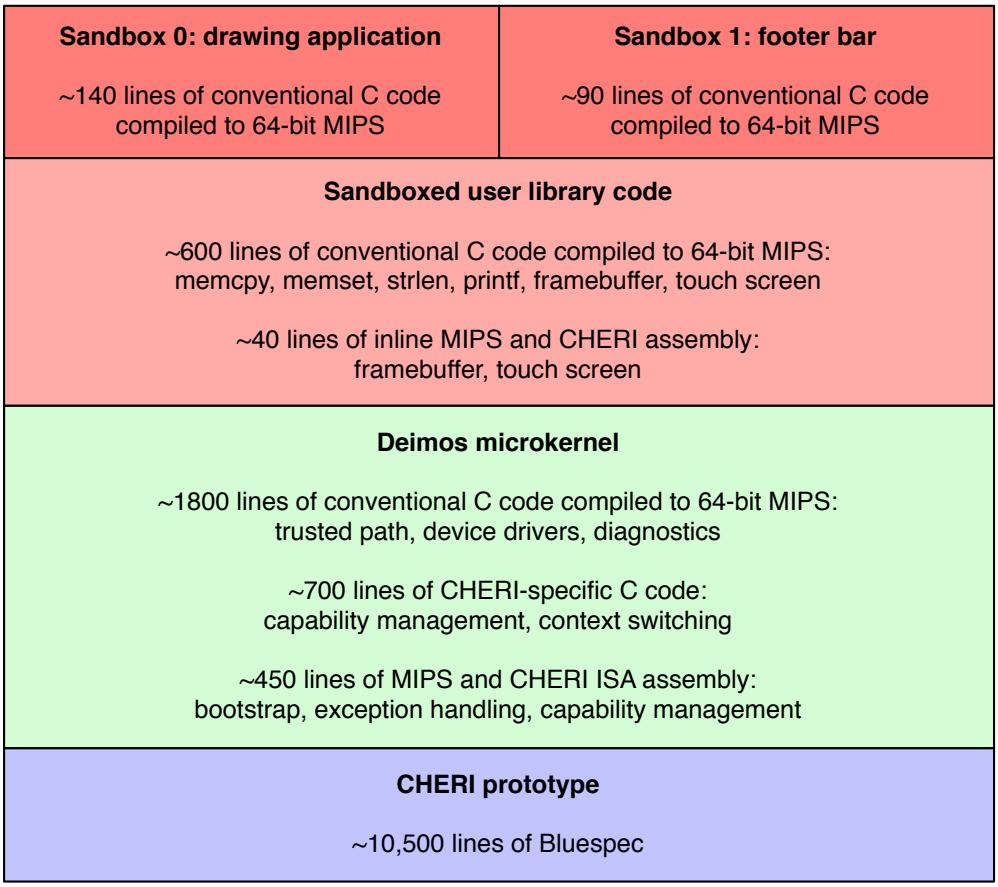

Figure 4.2: The Deimos software architecture: a blend of C code compiled to stock 64-bit MIPS machine code, stock 64-bit MIPS assembly, and a small amount of CHERI-specific assembly.

4.2.1 Supervisor

Deimos uses only the capability coprocessor for sandboxing, eschewing use of virtual addressing. The supervisor runs entirely within exception-handling context – that is, with $\mathbf{EXL} = 1$; a small bootstrap routine initializes the Deimos kernel stack and data structures, and then triggers an exception to enter the supervisor. The supervisor consists of an exception handler that implements a small microkernel. On an exception, the supervisor saves "userspace" general-purpose and capability registers to the "process"'s context structure; when the supervisor has completed execution, user context is restored – possibly after a context switch that selects a new process.

Deimos handles a number of exception types, including userspace preemption using the MIPS cycle counter, explicit system calls from the user process, and capability exception handling faults. Deimos system calls include simple UART I/O routines, a voluntary yield call, and a call to retrieve capabilities using a string-based namespace; capabilities delegating access to different portions of user-addressable screen space, as well as touchscreen input, are available to processes, subject to a simple access control policy.

4.2.2 Memory

Deimos uses capability registers **PCC** and **EPC** to lay out per-sandbox address space with respect to conventional MIPS memory access instructions, providing a small dead space at address 0, code segment, heap, and stack. Using this technique, unmodified MIPS application code can be isolated transparently. Access to additional memory outside of the compatibility address space (such as to the frame buffer) is possible through delegated capabilities. Application code is linked to run at alternative addresses, and some care must be taken during context switch to ensure that general-purpose and capability registers are saved and restored in appropriate order to allow both user applications and the supervisor to run correctly – especially as most of the supervisor is implemented in C code compiled to stock MIPS machine code.

4.2.3 CHERI-Aware ABI

Each user process is described by context structures holding preserved general-purpose and capability register context that is saved and restored when the supervisor is entered and exited. Deimos implements an enhanced Application Binary Interface (ABI) that allows capabilities to be passed to and returned from functions, including system calls. As with general-purpose registers, this calling convention designates specific capability registers for use as arguments, returned values, and temporary values. Most code in CHERI applications is not aware of the enhanced ABI and hence preserves capability registers; certain kernel and application code is aware of capability registers and manages them per these conventions. Application frame buffer library code, for example, queries frame buffer capabilities when starting, and then preserves them for later use.

4.3 Building Deimos

The Deimos source code may be found in the `deimos/trunk` tree in the CHERI Subversion repository. Table 4.1 shows a number of useful make targets for Deimos. Table 4.2 shows options that may be passed when building Deimos.

Target	Description
`run`	Build Deimos and run in the CHERI simulator
`verilog`	Build Deimos and generate a memory image for FPGA use
`clean`	Clean the Deimos build; the same options must be passed to clean as were used to build Deimos.

Table 4.1: Deimos make targets

Option	Description
`CP2=YES`	Enable capability coprocessor support, sandboxing applications
`PREEMPTION=YES`	Enable cycle timer-based preemption
`SIMULATOR=YES`	Modify runtime timing parameters to improve performance in a simulated environment
`TRACE=YES`	Enable UART tracing of Deimos system events
`UART_DISABLE_FC=YES`	Disable flow control support for the Deimos console on the UART – useful for some FPGA targets where a remote endpoint may not be connected

Table 4.2: Deimos make options

4.4 Conclusion

Deimos is a simplistic operating system, implementing an elementary process model and minimalist system services. However, it is sufficiently capable to illustrate a number of key features of the CHERI processor – especially, its support for a hybrid capability model in which some portions are explicitly aware of capabilities, but much is not. Deimos also illustrates how capability delegation can be used even within a privileged ring, to contain software components with delegation of limited I/O access. This ability suggests various applications of CHERI features that would be useful in conventional operating systems. We hope to build on Deimos for further demonstration and testing purposes, and use lessons learned from the experience of building Deimos to inform our larger-scale experiments with CHERI, such as the adaptation of the FreeBSD operating system to exploit capabilities within both its kernel and userspace.