Notes about CHERI

prepared by Michal Borowski

November 15, 2021

Contents

1 Preface **2**

2 Protection mechanisms of CHERI **2**

3 Capability contents **2**
3.1 Capability permissions . . . 3
3.2 Capability bounds . . . 3
3.3 Capability otype (object type) . . . 4
3.3.1 What it means that a capability is "sealed"? . . . 5
3.4 Capability validity bit . . . 5

4 Evolving nature of CHERI **5**

5 More about safety types + examples **5**
5.1 Referential safety . . . 5
5.2 Spatial safety . . . 6
5.3 Temporal safety . . . 7

6 Definitions of recurring terms in CHERI documents **8**

7 Notes about using CHERI-RISC-V Qemu emulator on cseekdmsim1.essex.ac.uk server **8**
7.1 Helper scripts . . . 9
7.2 Collecting traces . . . 9
7.2.1 Extracting program behaviour traces . . . 9
7.2.2 Changing log file . . . 10
7.2.3 Closing emulator . . . 10

1 Preface

As of 03/11/2021, the most recent published version of Cheri ISA is version 8 (published in 11/2020). According to CHERI RISC-V website, version 9 is/was expected to be ready in 2nd half of 2021 (3rd/4th quarter).

2 Protection mechanisms of CHERI

CHERI offers referential, spatial and temporal safety. It implements various mechanisms to do so (that are constantly evolving). Mechanisms I learned about, are the following:

- **pointer provenance validity** - a pointer can't be used as pointer anymore (can't be dereferenced) if any part of it gets overwritten as different (non-pointer) type (e.g byte, int, float, long). 3.4 Capability validity bit section describes how it happens.
- **pointer permissions** - assigned to pointers in accordance with the least privilege principle (e.g. permission to read/write/execute, for example *$PCC* - program counter capability - or function pointers have 'execute' permission)
- **pointer bounds** - pointer becomes invalid if it's set to value out of bounds (meaning it can't be dereferenced)
- **immutable function entry/exit points** - using *sealed entry capabilities* ("sentry"). So overflowing a buffer on stack to overwrite return address will make it invalid because return addresses are *sealed* upon calling a function (it's based on the 1994 Hardware Support for Fast Capability-based Addressing), it prevents jumping mid-function (return/jump oriented programming attacks as mentioned on page 99 of CHERI ISA V8 - 'Sealed Entry Capabilities')
- **encapsulation at architectural level** - using *sealed capability pairs* (1 cap to object methods, 1 cap to object fields) that can only be used together. I'm not completely sure how that works in detail, but both of such caps are supplied to *CInvoke* instruction to make them unsealed (which happens only if their *otype* (object type) matches.
- **periodical scanning of the heap** - method called Cornucopia

3 Capability contents

I think a good way to get intuition about a system is to know what data structures it uses and what operations are performed on them.

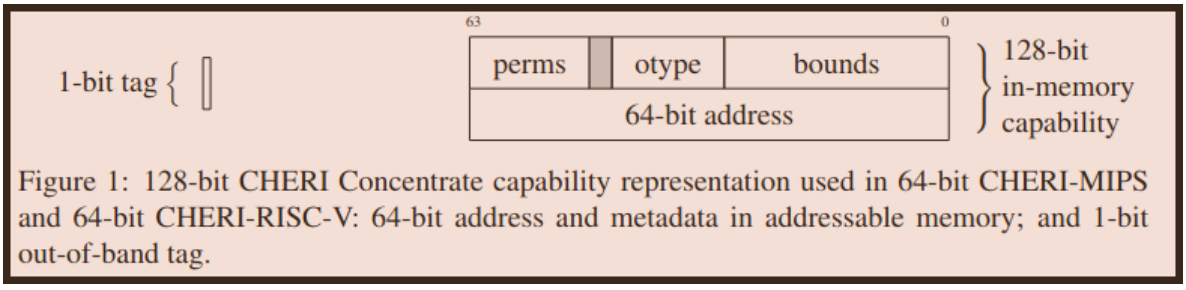


Figure 1: 128-bit CHERI Concentrate capability representation used in 64-bit CHERI-MIPS and 64-bit CHERI-RISC-V: 64-bit address and metadata in addressable memory; and 1-bit out-of-band tag.

From CHERI C/C++ programming guide (page 6)

3.1 Capability permissions

The following permissions can be given to a capability:

Bit	Name	Tag?	Seal?	Bounds?
0	GLOBAL	✓	-	-
1	PERMIT_EXECUTE	✓	Unsealed	Address
2	PERMIT_LOAD	✓	Unsealed	Address
3	PERMIT_STORE	✓	Unsealed	Address
4	PERMIT_LOAD_CAPABILITY	✓	Unsealed	-
5	PERMIT_STORE_CAPABILITY	✓	Unsealed	-
6	PERMIT_STORE_LOCAL_CAPABILITY	✓	Unsealed	-
7	PERMIT_SEAL	✓	Unsealed	Object Type
8	PERMIT_CINVOKE	✓	Sealed	-
9	PERMIT_UNSEAL	✓	Unsealed	Object Type
10	PERMIT_ACCESS_SYSTEM_REGISTERS	✓	Unsealed	-
11	PERMIT_SET_CID	✓	Unsealed	CID

Table 3.1: Architectural permission bits for the **perms** capability field, along with checks usually used alongside that permission: *Tag?* Require a valid tag; *Seal?* Require the capability to be sealed or unsealed; *Bounds?* Perform a bounds check authorizing access to the listed namespace. See the instruction-set reference for detailed per-instruction requirements.

From CHERI ISA V8 - 'Capability Contents' (page 75)

See CHERI ISA V8 - 'Permissions on Capabilities' for more info (about how and by 'who' permissions are set).

3.2 Capability bounds

The purpose of bounds (boundaries) is simple, if the pointer is set to address outside of them, it becomes invalid and can't be dereferenced anymore.

A simple way to implement bounds would be to use absolute 2 addresses like:

```
upper_bound = 0xFFFFFFFFFFFFFFFF;
lower_bound = 0xFFFFFFFFFFFFFF00;
address =     0xFFFFFFFFFFFFFF08;
```

Before CHERI ISA V5 (2016), the capabilities were implemented this way. A capability containing 64-bit address would have width of 256-bits, as shown below. That was inefficient in terms of space.

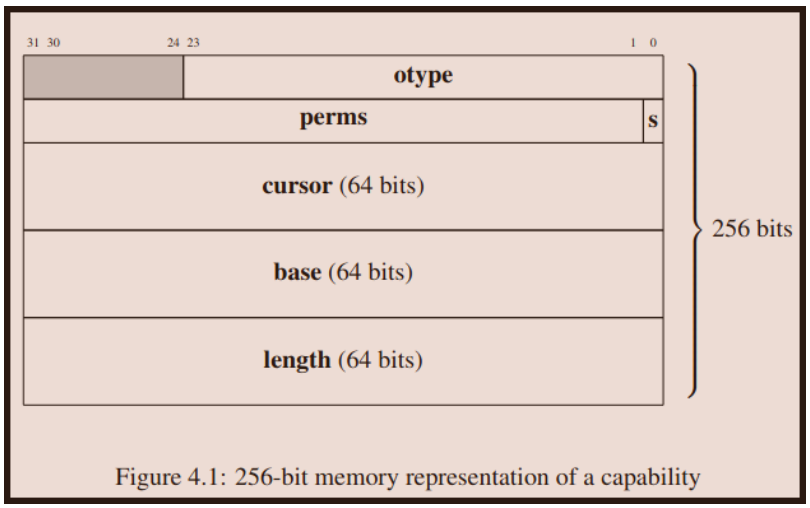


Figure 4.1: 256-bit memory representation of a capability

From CHERI ISA V4 (2015) (page 67)

So a compression/encoding got introduced and refined over the last few years, the method was called CHERI Concentrate. I don't fully understand how it works exactly, but it uses relative address and float representation. Because of using float and compressed representation, the boundaries are not perfectly accurate for large boundaries. This has several implications:

- area between bounds is higher than needed (e.g. 1000**448** bytes are allocated when using `char c[1000000];`)
- more memory is allocated than is needed
- surplus memory pose a threat of **leaking information**
- thanks to surplus memory, it's **not possible to overwrite other objects**

MSRC Security analysis of CHERI ISA (page 7) state:

> " *With CHERI Concentrate and capabilities represented on 128 bits, capabilities representing objects of more than 0x1000 bytes will be compressed. In such a situation, the length of the block allocated for the capability will be slightly larger than the length requested.*"

3.3 Capability otype (object type)

Object type has 18-bits on CHERI-RISCV64 (and 4-bits on CHERI-RISCV32).

otype value	Interpretation
$2^{XLEN} - 1$	Unsealed capability
$2^{XLEN} - 2$	Sealed entry ("sentry") capabilities; see Section 3.8
$2^{XLEN} - 3$	Reserved (experimental "memory type tokens"; see Appendix D.12)
$2^{XLEN} - 4$	Reserved (experimental "indirect enter capabilities"; see Appendix D.9)
$2^{XLEN} - 5$ through $2^{XLEN} - 16$	Reserved
other	Capability sealed by `CSeal`

Table 3.2: Object types and their architecture-specified roles.

From CHERI ISA V8 - 'Object type' (page 78)

3.3.1 What it means that a capability is "sealed"?

It means that it can't be modified or dereferenced.

3.4 Capability validity bit

The validity bit isn't stored only for pointers/registers. All physical memory is tagged. On 64-bit system, every 128-bit capability has its' own validity tag.

From Cheri ISA V8 - 'Tagged Memory':

> "*CHERI relies on tagged physical memory: the association of a 1-bit tag with each capability-sized, capability-aligned location in physical memory.*"
> (...)
> "*When a capability-sized value in a capability register is written to a capability-aligned area of memory using a capability store instruction, and the capability via which the store takes place has suitable permissions, the tag bit on the capability register will be stored atomic-ally in memory with the capability value.* ***Other stores of untagged capability values or other types (e.g., bytes, half words, words, floats, doubles, and double words) across one or more capability-aligned locations in memory will atomically clear the corresponding tag bits for that memory.***"

In this context, clearing of the validity bit is how CHERI recognizes that the value shouldn't be treated as a pointer anymore.

4 Evolving nature of CHERI

I asked a question on https://cheri-cpu.slack.com * because I found specification of *otype* hard to understand. It got clarified and then the author/contributor Robert Watson wrote:

> *It's not clear that the 'otype' feature is in its final shape for CHERI yet. It's quite expensive on bits, and while it enables some interesting programming models, it's not yet clear whether that investment of bits is worth it (vs using other techniques). This is a piece of feedback Arm would particularly like from the Morello effort – whether using that rather than sentry-style protections is of strong value.*

* cheri slack can be joined by requesting invitation from: cheri-slack@cl.cam.ac.uk as mentioned on the cl.cam.ac.uk Cheri-slack page. There are not many members (around 150) but include authors/contributors (e.g. Franz Fuchs who wrote dissertation about transient-execution attacks on CHERI-RISC-V)

5 More about safety types + examples

5.1 Referential safety

Only pointers derived from valid pointers can be dereferenced. CHERI recognizes what is a pointer, what isn't a pointer and therefore shouldn't be treated as such. For example, not allowing int/long to be treated as a pointer as shown in the type confusion exercise:

```
union long_ptr {
    long l;
    const char *ptr;
}

void inc_long_ptr(union long_ptr *lpp)
{
    lpp->l++;
}
```

RISC-V can use *inc_long_ptr* to manipulate pointer value. If CHERI-RISC-V attempts to dereference the pointer after manipulation, exception will be raised.

```
// #include <cheriintrin.h> contains cheri_get_tag function
printf("lp.ptr=%s tag=%d\n", lp.ptr, cheri_tag_get(lp.ptr)); // tag=1
inc_long_ptr(&lp);
printf("tag=%d\n", cheri_tag_get(lp.ptr)); // tag=0
printf("lp.ptr=%s\n", lp.ptr); // exception is raised (attempt to dereference a
    pointer with validity tag equal to 0)
```

```
root@cheribsd-riscv64-purecap:/cheri-exercises # ./union-int-ptr_cheri
lp.ptr=Hello World! tag=1
tag=0
In-address space security exception (core dumped)
```

In the Uninitialized stack frame to manipulate control flow mission, stack frames of 3 function calls (called one after another) occupy areas intersecting each other. These functions are:

- **init_cookie_pointer** - sets local variable function pointer to a function
- **get_cookies** - reads user input
- **eat_cookies** - calls uninitialized function pointer which normally leads to function specified in *init_cookie_pointer* function

Interestingly CHERI-RISC-V doesn't have a problem with calling uninitialized function pointer that was set by other function (in a valid way), the program runs and the function gets called. However, if we modify the memory value as if it wasn't a pointer (within *get_cookies*), it will lose its' validity tag and the program will raise exception as shown below.

```
=> 0x1027fc <eat_cookies+180>:      jalr    a0
(gdb) info reg $ca0
ca0            0xd117200008db8006000000000010223c       0x10223c <success> [rxR,0x100000-0x104da0] (sentry)
(gdb) stepi

Program received signal SIGPROT, CHERI protection violation
Capability tag fault caused by register ca0.
```

This program can be exploited on RISC-V but not on CHERI-RISC-V.

It's worth to notice that it's possible to use the following CHERI clang flags when compiling CHERI-RISC-V to mitigate uninitialized stack problems:
`-ftrivial-auto-var-init=zero`
`-enable-trivial-auto-var-init-zero-knowing-it-will-be-removed-from-clang`

5.2 Spatial safety

It's relating to space. Pointers have bounds that limit where they can point, functions have immutable entry/exit points (thanks to function pointers and return addresses being *sealed entry* capabilities)

- mitigation of common buffer overflow attacks
- mitigation of jumping mid-function (return/jump oriented programming)
- mitigation of return address manipulation
- encapsulation implemented at architectural level

The inter-object buffer overflow exercise presents how CHERI bounds checking prevents from too large user input overwriting another variable by overflowing its' own buffer.

```
root@cheribsd-riscv64-purecap:/cheri-exercises # ./buffer-overflow_riscv64
c = c
c = b
root@cheribsd-riscv64-purecap:/cheri-exercises # ./buffer-overflow_cheri
c = c
In-address space security exception (core dumped)
```

RISC-V lets the program run and allows the *c* variable to be overwritten. CHERI-RISC-V doesn't let that happen. We can see why using gdb.

Upper boundary is not inclusive (as outputted by gdb). We can see that the **buffer** upper boundary **0x104390** prevents the buffer from accessing variable **c** located at that address.

```
0x104310 <buffer> [rwRW,0x104310-0x104390]
 0x104390 <c> [rwRW,0x104390-0x104391]
```

The exact instruction causing the *bounds fault* is the **sb** (store byte) which attempts to put 0x64 ('b') at the **a1** capability (buffer variable) which is out of its' own bounds (because the upper bound is not inclusive as outputted by gdb).

```
=> 0x101e56 <fill_buf+130>: sb      a0,0(a1)
(gdb) info reg $ca0 $ca1
ca0            0x62     0x62
ca1            0xf17d000004e583140000000000104390       0x104390 <c> [rwRW,0x104310-0x104390]
(gdb) stepi

Program received signal SIGPROT, CHERI protection violation
Capability bounds fault caused by register ca1.
```

5.3 Temporal safety

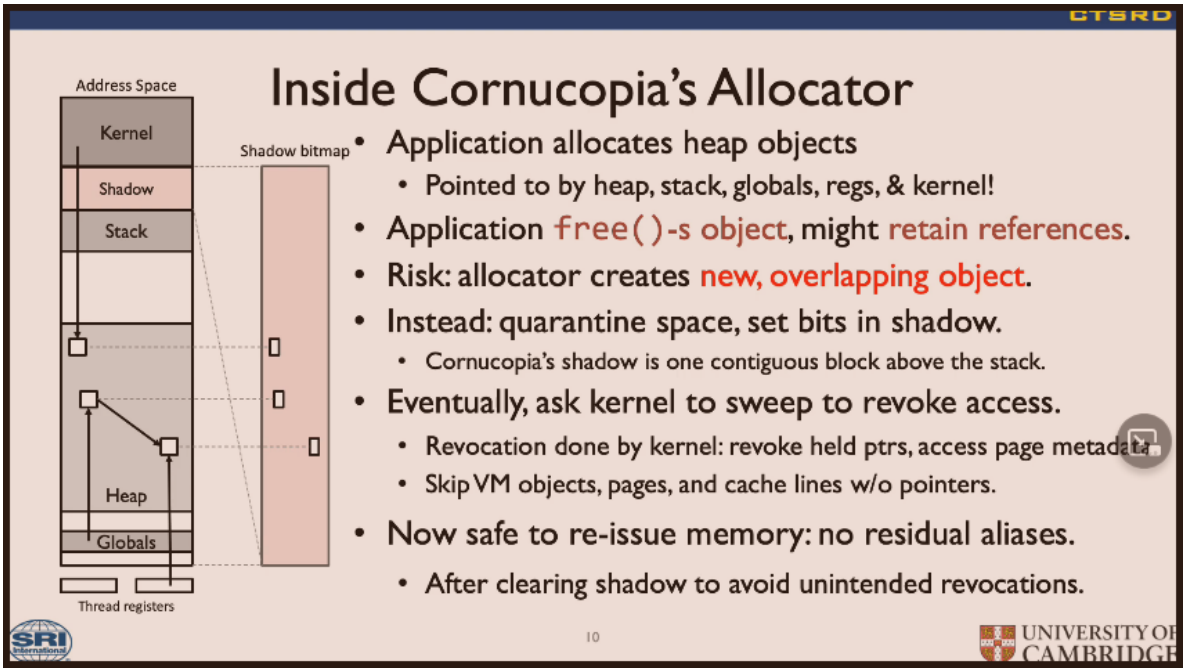
It's relating to time. It's ensuring that objects (and memory in general) are not used outside of their lifetime. For example, it involves mitigation of *use-after-free* or *uninitialized-stack* based attacks.

CHERI originates from 2010 but temporal safety wasn't provided by CHERI until around 2019 from what I understand when CHERIvoke method was published/incorporated. In 2020 it got adapted by another method called Cornucopia (faster/extended version of CHERIvoke method).

If I understand correctly, both these methods periodically scan the heap (in optimized way by using a *shadow map*) looking for *dangling pointers* (pointing to objects/memory that got deallocated).

Why isn't this task simple (involving sweeping/scanning methods)? Why wouldn't pointers just become automatically invalid when *free(ptr)* is used?
The problem is that very often, multiple pointers can point to the same object/memory, and the data on heap doesn't store any references to pointers that point to it.

Cornucopia: Temporal Safety for CHERI Heaps youtube video explains it in more details.



From 6:48 of the youtube video referenced above.

6 Definitions of recurring terms in CHERI documents

Definitions copied from CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment:

Provenance validation - *ensures that only capabilities derived via valid transformations of valid capabilities using capability instructions can be used.*

Capability integrity - *prevents direct in-memory manipulation of architectural capability encodings.*

Monotonicity - *prevents the permissions or bounds associated with a capability from being increased.*

7 Notes about using CHERI-RISC-V Qemu emulator on cseekdm-sim1.essex.ac.uk server

The emulator is installed at:
/tools/RISC-V/emulator

The **clang** program used to compile programs for cheri is located at:
/tools/RISC-V/emulator/cheri/output/sdk/bin

However, the list of parameters to compile for RISC-V or CHERI-RISC-V is long, and scripts described further in this section can be used to easily compile (and transfer programs).

As mentioned on the cheribuild Github repository, we can install CHERI-RISC-V with Qemu and all dependencies using:
./cheribuild.py -d run-riscv64-purecap

The **-d** parameter installs all dependencies which may take hours, so it should really be used once (to install) and then be omitted (to run the emulator).

7.1 Helper scripts

I wrote few short scripts (**/tools/RISC-V/emulator/scripts**) to make compilation, file transfer and running of the emulator easier. These are:

- **compile_cheri.sh file.c -o file**
- **copy_to_cheri.sh file /destination_at_cheri_risc_v** - uses `scp -P 10019` to copy files to absulute path on CHERI-RISC-V through ssh
- **run_cheri_riscv_purecap_with_qtrace_log_file.sh logfile_name.log** - runs qemu and creates specified filename in **/tools/RISC-V/emulator/log_files** (I prefer using this to **./cheribuild.py run-riscv64-purecap**, it contains the same command that cheribuild runs to start Qemu, but with log file specified and without running updates so it's faster)
- **ccc2.sh file.c** - used for cheri-exercises (compiles the file for RISC-V and CHERI-RISC-V, creates llvm-objdump for both, then transfers both to qemu CHERI-RISC-V along with the source file for debugging with source code using gdb)

There is also a ccc compilation script provided for cheri-exercises by their creators (which is also in the *scripts* directory).

Adding the following lines at the end **/home/user/.profile** will allow to use these files from anywhere without specifying absolute/relative path:

```
if [ -d "/tools/RISC-V/emulator/scripts" ] ; then
    PATH="/tools/RISC-V/emulator/scripts:$PATH"
fi
```

All scripts have file permissions set to allow execution by anyone. The same applies to the whole **/tools/RISC-V/emulator** folder. I suggest that whenever a new file/directory is created inside it, we run `chmod o=u emulator/some_new_dir_or_file -R` to allow everyone have the same permission for it as the creator.

7.2 Collecting traces

To collect program traces from the CHERI-RISC-V we can use the following command:
qtrace -u exec ./program_name

If emulator was ran with the **run_cheri_riscv_purecap_with_qtrace_log_file.sh logfile_name.log** then the trace will be stored in that file. If emulator is ran using cheribuild.py itself, then I don't know where the log file will be created (tried to find it, googled a lot but didn't succeed).

7.2.1 Extracting program behaviour traces

The program trace is around 300MB for a small file, including mostly system calls invoked before the main function itself and after it returns.
I noticed that the main begins just after the third occurrence of **0x402AA5C6** program counter from the begining of the file. The main ends just over the first occurence of **0x402AAFA0** program counter from the back of the file. So we could extract the trace of the program itself (without routines/syscalls responsible for running the program on the OS).
* (I checked it only with 2-3 programs so I'm not sure if it's a reliable way, checked it after emulator reboot as well, but more tests may be needed)

7.2.2 Changing log file

By default, when running qtrace multiple times, the traces are appended to the same file. The log file can be changed from inside emulator by using **ctrl+a** and pressing **c**, then typing **logfile new_name.log** and using ctrl+a and c again to go back into CHERI-RISC-V.
(the new file may appear in **emulator/cheribuild/** instead of **emulator/log_files/**).
After changing the log file name, **ctrl+a** and **c** will switch back to the emulated CHERI-RISC-V system.

7.2.3 Closing emulator

Emulator can be closed by using **ctrl+a** and pressing **x**.