Technical Report

UCAM-CL-TR-988
ISSN 1476-2986

Number 988

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

CHERI C semantics as an extension of the ISO C17 standard

Vadim Zaliva, Kayvan Memarian,
Ricardo Almeida, Jessica Clarke, Brooks Davis,
Alex Richardson, David Chisnall,
Brian Campbell, Ian Stark,
Robert N. M. Watson, Peter Sewell

October 2023

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

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

This work was supported by the UK Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme, to deliver a DSbDtech enabled digital platform (grant 105694).

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 789108, ERC AdG ELVER).

Distribution Statement A: Approved for public release; distribution is unlimited. This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts HR0011-22-C-0110 ("ETC") and HR0011-23-C-0031 ("MTSS"). The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

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

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

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-988*

Abstract

This document provides a specification for CHERI C, adhering to the style, conventions, and terminology found in the ISO C17 standard. Alongside the ISO/IEC 9899:2018 standard text and the "A Provenance-aware Memory Object Model for C" draft specification, it offers a comprehensive specification of the CHERI C language.

1 CHERI C prose semantics

Our CHERI C semantics prose adheres to the language, style, and terminology found within the ISO text. Instead of presenting it as an extensive list of modifications to the standard text, we have opted to describe the CHERI-specific aspects of the semantics through a series of sections with bullet points. Consequently, the complete CHERI C semantics definition comprises the ISO text [2] and the PNVI-ae-udi memory object model [1], along with the CHERI supplement, with this text taking precedence in cases of disagreement.

1.1 Capabilities

In the CHERI C language, we will use an architecture-neutral definition of *abstract capabilities*. Where it is allowed, the ISA-specific implementation details of the abstract capabilities will be designated as *implementation-defined behaviour*. An abstract capability is a data structure which consists of a set of fields. Those relevant to this specification are the address, bounds interval, validity tag, object type, and set of boolean permissions. Informally speaking, a capability allows to access the memory location designated by its *address*, but only if the address (and the rest of the footprint of the access) is within the *bounds*. The type of allowed access (e.g. *read* or *write*) is described by the permissions. Access is granted only if the capability's tag is true. Normal, non-privileged, code cannot set the capability tag, but could clear it. See [3] for a more detailed introduction to CHERI capabilities.

In the abstract C machine in addition to the standard capability fields each capability value is associated with a *ghost state*. It consists of two boolean flags. The first one, *tag_unspecified*, denotes that the given capability tag may be non-deterministically `true` or `false`. The second flag in the ghost state is called *bounds_unspecified* and if set, it means that capability bounds are not known.

A special *null capability* constant is always untagged, has empty permissions, bounds covering all abstract address space, and has the address of zero.

The implementation-defined capability encoding might not permit some combinations of capability addresses and bounds to be encoded. An attempt to set a capability address field with such *non-representable value* will result in a capability which will have the new address set and *tag_unspecified* and *bounds_unspecified* both set to `true` in the ghost state. It is guaranteed, however, that all addresses within the capability bounds and one byte past the upper bound are representable[1].

1.2 Types

The following C types are either introduced in CHERI C or differ from their definition in ISO C.

[1]Capability invalidation caused by *non-representable values* could not arise in well-defined pointer arithmetic but can happen when manipulating `(u)intptr_t` values.

1.2.1 `ptraddr_t` type

This is a new *integer type* representing an *abstract address* as defined in N3005 [1]

(1) This is the type of the *address* field of a capability.

(2) It is unsigned.

(3) It is not a pointer type and cannot be used directly to reference objects.

1.2.2 Pointer types

In the CHERI C abstract machine, a *pointer value* is a capability. The `address` field of this capability holds the *abstract address* (or one-past) of the object it points to or 0 for a *null pointer*.

(1) The object representation of pointer values is *implementation defined*.

(2) The *null pointer constant* corresponds to *null capability* constant.

(3) A *null pointer* is any pointer with the *abstract address* (the capability's `address` field) 0.

(4) The capabilities representing function pointers must be *sealed* by setting an appropriate *object type* (see [3] for further details on sealed capabilities).

(5) Any modification of a function pointer's address or its capability fields via CHERI intrinsics will set the `tag_unspecified` flag to to `true` in the corresponding capability ghost state.

1.2.3 `intptr_t` and `uintptr_t` types

As in ISO C, these are signed and unsigned integer types respectively, with the property that any valid pointer to `void` could be converted to either of these types and back and will compare *exactly equal* (see Section 1.3 for a definition of *exact equality*) to the original. Additionally, it will have the same provenance as the original. These two types together with all pointer types are collectively referred to as *capability types*.

The values of these types are capabilities, whose integer value is the value of the *address* field of the capability. Internally, the address field has an unsigned integer type `ptraddr_t` which is used unchanged in case of `uintptr_t` or interpreted as a signed integer of the same width in case of `intptr_t`.

(1) The object representations of these types and their storage size reported by `sizeof` are *implementation defined*.

(2) The *width* of these types is not guaranteed to be the same as their storage size in bits.

(3) The *width* of these types must be the same as the width of `ptraddr_t`.

(4) No other *standard integer type* shall have a higher *integer conversion rank* than `intptr_t` and `uintptr_t`.[2]

(5) The equality operators `==` and `!=`, and relational operators `<`, `>`, `<=`, `>=` compare `intptr_t` and `uintptr_t` as integer values, and do not take into account additional capability fields.

(6) Converting values of these types to other integer types is performed using standard integer conversions. The capability meta-information will be lost.

(7) Converting values of other integer types to values of these types is performed by first converting to `ptraddr_t` value using standard integer conversions and then assigning this value to a copy of the *null capability*.

[2]From that follows that no other signed integer type shall have greater *precision* than `intptr_t` and no other unsigned integer type shall have greater *precision* than than `uintptr_t`.

(8) Reading values of these types using *byte input/output functions* (e.g. `fread`, `fscanf`) or `sscanf` is allowed. However, converting the resulting value to a pointer will always result in an untagged pointer.

(9) Direct manipulation of the representation bytes of an `intptr_t` will set `tag_unspecified` flag to to `true` in the corresponding capability ghost state.

(10) Per *PNVI-ae-udi*, provenance is not tracked through values of `intptr_t` and `uintptr_t` types. When converted to pointers, the provenance will be reconstructed, if possible.

Arithmetic operations (e.g. unary `~` and binary `+`), when one or both of the operands has `intptr_t` or `uintptr_t` type, is performed as follows:

1. The operation is performed on address fields of operands.

2. The resulting `intptr_t` or `uintptr_t` object is a capability, derived from one of the operands and assigned the arithmetic result of the operation. If the result of the operation is *non-representable* in combination with the derived capability bounds, the resulting capability will have the new value set in the address field and *tag_unspecified* and *bounds_unspecified* flags both set to `true` in the ghost state.

3. If only one of the operands is capability-carrying, the derivation algorithm will use it.

4. If both operands are capability-carrying, the derivation algorithm will use the one on the left-hand side.

If the capability in a `(u)intptr_t` value is *sealed* any modification of its address as the result of arithmetic operations will set `tag_unspecified` to `true` in the ghost state and any other modification via CHERI intrinsics will clear the tag.

1.3 Pointer comparison

1.3.1 Standard pointer equality

Pointers are equal (per `==`) if their abstract addresses (the `address` fields of their respective capabilities) are the same. Thus pointers may compare as equal even when some of their capability fields differ.

1.3.2 Exact pointer equality

Additionally, the *exact equality* is defined, and implemented by *cheri_is_equal_exact* intrinsics. According to it, the two pointer objects are equal if object representations and tags of their corresponding capabilities match. If any of the flags in the ghost state is true for either of the capabilities being compared, the exact equality intrinsic will return an *unspecified* boolean value.

1.3.3 Relational operators

Relational operators (`<`, `>`, `>=`, `<=`) between pointers are only defined when the two pointers have the same provenance. They are defined by the relative position of the abstract addresses [1].

1.4 Conversion between integers and pointers

All *capability types* (`intptr_t`, `uintptr_t`, and pointer types) share the same representation and conversion between them leaves the underlying capability unchanged.

The rules for conversion between pointers and other integer types are described below.

1.4.1 Pointer to integer

The address field of the pointer capability is converted to the target integer type. If the result cannot be represented in the integer type, the behaviour is undefined.

1.4.2 Integer to pointer

The integer is wrapped to fit `ptraddr_t` by repeatedly adding or subtracting one more than the maximum value that can be represented in `ptraddr_t` until the value is in the range of the `ptraddr_t`. Then, the pointer is constructed from the *null capability*, assigning its address to the wrapped number.

It should be noted that the resulting capability will have the tag cleared and the resulting pointer cannot be dereferenced. The provenance of the resulting pointer will be recovered following the *PNVI-ae-udi* rules. If the value of the integer is 0, the result of this conversion will be a *null pointer*.

1.5 Pointer arithmetic

1.5.1 Pointer addition and subtraction

When an expression that has an integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. The resulting value is calculated as follows:

(1) The arithmetic operation is performed on `ptraddr_t` address fields following the pointer addition rules of ISO C [2].

(2) If the resulting address is not within (or one-past) the storage instance the behaviour is undefined.

(3) The resulting pointer value is constructed by taking a capability value of the pointer operand and setting its address field to the result of the arithmetic operation.

(4) The provenance of the pointer operand will be preserved in the result.

1.5.2 Pointer difference

(1) Pointer difference is only defined for pointers with the same provenance and within the same array.

(2) The implementation-defined type of the result, `ptrdiff_t`, is not capability-carrying signed integer type.

1.6 Pointer synthesis

While there are several ways to modify existing pointers and derive new ones from them, there are very few ways to construct a pointer from scratch (commonly referred to as "pointer synthesis").

(1) An attempt to read a pointer using *byte input/output functions* (e.g. `fread`, `fscanf`) will always result in a pointer which is untagged.

(2) An attempt to construct a pointer using `sscanf` will also result in a pointer which is untagged.

(3) The result of modifying the representation bytes of a pointer is *implementation-defined* and could result in a *trap representation*. If it does not, it is guaranteed that the resulting pointer value will have `tag_unspecified` flag set to `true` in the ghost state.

(4) The unary `&` operator applied to a *function designator* returns tagged, *sealed* function pointer.

(5) The unary `&` operator applied to *lvalue* that designates an object returns a pointer whose capability is tagged, the address is the virtual address of the object, and the bounds do not exceed the ones the storage instance for this object.

(6) The unary `&` operator applied to *lvalue* that designates an object with non-pointer type that is `const`-qualified returns a pointer whose capability does not have *write* permissions[3].

1.7 Alignment of objects

All capability objects must be properly aligned. The alignment value is implementation-defined.

An *implementation-defined* capability encoding may impose an additional optional alignment requirement on the objects holding capabilities. For example, an additional alignment may be required to have exact object bounds encoded in a capability (as used in Section 1.9). This additional alignment may be larger than the standard alignment, and larger than the one of `max_align_t` type.

1.8 Pointer validity

In this section, we will briefly explore subtleties of the relation between pointer virtual address, *storage instance*, object size and location, pointer *provenance*, and *capability bounds*.

To recall, from [1], in the abstract C machine each object is in a *storage instance*. Its footprint cannot cross the instance boundaries. The *provenance* of a pointer is an ID of a storage instance (or empty). Additionally, in CHERI C the capabilities which represent pointers have *bounds*. By construction, such bounds never exceed storage instance bounds (but could be narrower or narrowed later via a call to the intrinsics). Finally, capabilities have a tag and a ghost state.

A ***valid pointer*** is a pointer which points to a live object (or one-past), has a provenance of a storage instance which include this address, has the address within *bounds* (or one-past), has bounds within the address range of the storage instance, is properly aligned for the type of object it points to, and has the tag is `true`. Additionally, both *tag_unspecified* and *bounds_unspecified* flags in the ghost state should be *false*.

A ***dereferenceable pointer*** definition is similar to *valid pointer's*, except it does not allow one-past values. A dereferenceable pointer could be dereferenced but the intended use (read, write or function call) may or may not succeed depending on *permissions*.

An ***untagged pointer*** is a pointer with the tag cleared or *tag_unspecified* is `true`. Dereferencing an untagged pointer is an *undefined behaviour*.

Pointers constructed using integer-to-pointer casts (per Section 1.4.2), pointer synthesis (described in Section 1.6) are either dereferenceable or untagged.

Pointer arithmetic (as specified in Section 1.5.1) on valid and untagged pointers will produce either valid or untagged pointers or could result in an *undefined behaviour*.

[3]Pointers to *const*-qualified types do not guarantee that their values will not have *write* permissions. For example if `char*` was cast to `const char*` the resulting pointer will have `write` permissions.

However, converting a valid pointer to `intptr_t` or `uintptr_t`, performing arithmetic operations, and then converting it back to a pointer type could result in a pointer which is neither valid nor untagged. This pointer may have the address pointing outside one-past of its bounds and may have empty provenance. We will call such pointer an ***out of bounds pointer*** or ***OOB pointer***. Dereferencing an OOB pointer is an *undefined behaviour*.

1.9 Memory allocator

Memory allocation mechanisms in C include an *object allocator*, which allocates objects with static or automatic storage durations such as global and local variables, and a *heap allocator*, which is used by functions like `malloc`, `calloc`, `realloc`, and `aligned_alloc` to allocate memory regions. They are commonly referred to as an *allocator*.

(1) For any addressable storage instance created by the allocator, it should be possible to create a capability which has exact bounds matching instance memory footprint and an address for any address within this instance and one byte past it. This means that the implementation-dependent capability encoding may impose additional alignment of the region starting address and padding requirements.

(2) The content of regions returned by `malloc` are *unspecified*. If they contain any capabilities, their tags are also unspecified (via `tag_unspecified` in the ghost state).

(3) `realloc` must preserve all complete capabilities (including their tags) in the content of the original object. That means it must use a capability-friendly copy as byte-copy will clear tags of all stored capabilities.

(4) The content and tags of the newly allocated (not copied from the old one) region returned by `realloc` is *unspecified*.

(5) If the argument of `realloc` or `free` is not *exactly equal* to a pointer earlier returned by a memory allocation function the behaviour is undefined.

(6) `calloc` clears tags if they were set and leaves them *unspecified* if they weren't.

(7) `free` is not required to clear tags.

(8) The full set of permissions returned by the heap allocator (e.g. by `malloc`) is implementation-specific but it must include: *read* and *write*, and must not include *exec* permissions.

1.10 Intrinsics

There are special built-in intrinsic functions for accessing and manipulating fields of capability values. These functions, defined in the `cheriintrin.h` header, can be used on any capability type without additional type casts.

Intrinsics which retrieve capability fields accept any capability type. Because of this special type handling, it is not possible to write a C function declaration for them. For example, the `cheri_address_get` intrinsic will accept any pointer type with any qualifier, as well as `intptr_t` and `uintptr_t`. In Listing 1 for illustration purposes, we provide fake declarations for these functions using `void*` type for the capability parameter.

Intrinsics that modify capability fields, in addition to accepting any capability type as an argument, also return a capability value of the same type. In other words, their return type is derived from the type of their capability argument. For example, `cheri_address_set` when called with `void*` will have the return type of `void*` and when called with `uintptr_t` will have the return type of `intptr_t`. Again, for illustration purposes in our fake declarations for these functions, we will use `void*` return type.

```
ptraddr_t cheri_address_get(const void *);
void *cheri_address_set(const void *, ptraddr_t);
ptraddr_t cheri_base_get(const void *);
size_t cheri_length_get(const void*);
size_t cheri_offset_get(const void *);
void *cheri_offset_set(const void *, size_t)
void *cheri_tag_clear(const void *);
bool cheri_tag_get(const void *);
bool cheri_is_valid(const void *);
bool cheri_is_invalid(const void *);
bool cheri_is_equal_exact(const void *, const void *);
bool cheri_is_subset(const void *, const void *);

/* Bounds setting intrinsics: */
size_t cheri_representable_length(size_t);
size_t cheri_representable_alignment_mask(size_t)
void *cheri_bounds_set(const void *, size_t);
void *cheri_bounds_set_exact(const void *, size_t);

/* Object types, sealing and unsealing: */
cheri_otype_t cheri_type_get(const void *);
bool cheri_is_sealed(const void *);
bool cheri_is_sentry(const void *);
bool cheri_is_unsealed(const void *);

/* Builtins for indirect sentries */
void *cheri_sentry_create(const void *);
void *cheri_seal(const void *, void *);
void *cheri_unseal(const void *, void *);

/* Reconstruct capabilities from raw data: */
void *cheri_cap_build(x, y);
void *cheri_seal_conditionally(const void *, const void *);
void *cheri_type_copy(const void *, const void *);

/* Capability permissions: */
size_t cheri_perms_get(const void *);
void *cheri_perms_and(const void *, size_t);
void *cheri_perms_clear(const void *, size_t);
```

Listing 1: C prototypes of CHERI intrinsics

It should be noted that `cheri_offset_get`, `cheri_base_get`, and `cheri_length_get` will return an *unspecified* value of appropriate type if `bounds_unspecified` flag was set in the ghost state of the argument.

Similarly, `cheri_tag_get` will return an *unspecified* boolean value if `tag_unspecified` was set.

Finally `cheri_is_equal_exact` will return an *unspecified* boolean value if either of `bounds_unspecified` or `tag_unspecified` flags were set for either of the arguments.

1.11 Standard library

The following functions from the standard library have additional requirements with CHERI:

- `malloc`, `calloc`, `realloc`, `aligned_alloc`, and `free` - see Section 1.9
- `memcpy` and `memmove` must preserve tags and ghost state of values of capability types, but only if both source and destination addresses of such values are properly aligned (see Section 1.7).
- `qsort` must preserve tags of values of capability types but only if both the original and the final (sorted) addresses of such values are properly aligned (see Section 1.7).

References

[1] Jens Gustedt, Peter Sewell, Kayvan Memarian, Victor BF Gomes, and Martin Uecker. A Provenance-aware Memory Object Model for C, 2022. Working draft ISO Technical Specification TS6010.

[2] ISO WG14. *Programming languages – C*, ISO/IEC 9899:2018 edition, July 2018.

[3] Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. An Introduction to CHERI. Technical Report UCAM-CL-TR-941, University of Cambridge, Computer Laboratory, September 2019.