Technical Report

UCAM-CL-TR-947
ISSN 1476-2986

Number 947

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

CHERI C/C++ Programming Guide

Robert N. M. Watson, Alexander Richardson,
Brooks Davis, John Baldwin, David Chisnall,
Jessica Clarke, Nathaniel Filardo,
Simon W. Moore, Edward Napierala,
Peter Sewell, Peter G. Neumann

June 2020

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 Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 ("CTSRD") and HR0011-18-C-0016 ("ECATS"). 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. This work was supported in part by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 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 Advanced Grant ELVER. We also acknowledge the EPSRC REMS Programme Grant (EP/K008528/1), Arm Limited, HP Enterprise, and Google, Inc. Approved for Public Release, Distribution Unlimited.

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

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

ISSN 1476-2986

Abstract

This document is a brief introduction to the CHERI C/C++ programming languages. We explain the principles underlying these language variants, and their grounding in CHERI's multiple architectural instantiations: CHERI-MIPS, CHERI-RISC-V, and Arm's Morello. We describe the most commonly encountered differences between these dialects and C/C++ on conventional architectures, and where existing software may require minor changes. We document new compiler warnings and errors that may be experienced compiling code with the CHERI Clang/LLVM compiler, and suggest how they may be addressed through typically minor source-code changes. We explain how modest language extensions allow selected software, such as memory allocators, to further refine permissions and bounds on pointers. This guidance is based on our experience adapting the FreeBSD operating-system userspace, and applications such as PostgreSQL and WebKit, to run in a CHERI C/C++ capability-based programming environment. We conclude by recommending further reading.

Contents

1 Introduction

This document is a brief introduction to the CHERI C/C++ programming languages. We explain the principles underlying these language variants, and their grounding in CHERI's multiple architectural instantiations: CHERI-MIPS, CHERI-RISC-V, and Arm's Morello. We describe the most commonly encountered differences between these dialects and C/C++ on conventional architectures, and where existing software may require minor changes. We document new compiler warnings and errors that may be experienced compiling code with the CHERI Clang/LLVM compiler, and suggest how they may be addressed through typically minor source-code changes. We explain how modest language extensions allow selected software, such as memory allocators, to further refine permissions and bounds on pointers. This guidance is based on our experience adapting the FreeBSD operating-system userspace, and applications such as PostgreSQL and WebKit, to run in a CHERI C/C++ capability-based programming environment. We conclude by recommending further reading.

1.1 Definitions

CHERI Clang/LLVM and LLD implement the following new language, code-generation, and linkage models:

CHERI C/C++ are C/C++-language dialects tuned to requirements arising from implementing all pointers using CHERI capabilities. This includes all explicit pointers (i.e., those declared by the programmer) and all implied pointers (e.g., those used to access local and global variables). For example, they diverge from C/C++ implementations on conventional architectures by preventing pointers passed through integer type other than `uintptr_t` and `intptr_t` from being dereferenced. New Application Programming Interfaces (APIs) provide access to capability features of pointers, including getting and setting their bounds, required by selected software such as memory allocators. The vast majority of C/C++ source code we have encountered requires little or no modification to be compiled as CHERI C/C++.

Pure-capability machine code is compiled code (or hand-written assembly) that utilizes CHERI capabilities for all memory accesses – including loads, stores, and instruction fetches – rather than integer addresses. Capabilities are used to implement pointers explicitly described in the source program, and also to implement implied pointers in the C execution environment, such as those used for control flow. Pure-capability machine code is not binary compatible with capability-unaware code using integer pointers, not least due to the different size of the pointer data type.

While the focus of this document is CHERI C/C++, CHERI is an architectural feature able to support other software use cases including other C/C++-language mappings into its features. Another mapping is hybrid C/C++, in which only selected pointers are implemented using capabilities, with the remainder implemented using integers. We have primarily used hybrid C in systems software that bridges between environments executing pure-capability machine code and those running largely or entirely non-CHERI-aware machine code. For example, a largely CHERI-unaware CheriBSD kernel can host pure-capability processes using its CheriABI wrapper implemented in hybrid C (see Section 5). Hybrid machine code has stronger binary compatibility, but weaker protection, than pure-capability machine code. We do not consider hybrid C further in this document.

2 Background

CHERI extends conventional processor Instruction-Set Architectures (ISAs) with support for *architectural capabilities*. One important use for this new hardware data type is in the implementation of safer C/C++ pointers and the code or data they point at. Our technical report, *An Introduction to CHERI*, provides a more detailed overview of the CHERI architecture, ISA modeling, hardware implementations, and software stack [7].

2.1 CHERI capabilities

CHERI capabilities are twice the width of the native integer pointer type of the baseline architecture: there are 128-bit capabilities on 64-bit platforms, and 64-bit capabilities on 32-bit platforms. Each capability consists of an integer (virtual) address of the natural size for the architecture (e.g., 32 or 64 bit), and also additional metadata that is compressed in order to fit in the remaining 32 or 64 bits of the capability (see Figure 1 for an example; details vary across underlying architectures and word sizes). In addition, they are associated with a 1-bit validity "tag" whose value is maintained in registers and memory by the architecture, but not part of addressable memory. Each element of the additional metadata and tag of the capability contributes to the protection model:

Validity tag The tag tracks the validity of a capability. If invalid, the capability cannot be used for load, store, instruction fetch, or other operations. It is still possible to extract fields from an invalid capability, including its address.

Bounds The lower and upper bounds are addresses restricting the portion of the address space within which the capability can be used for load, store, and instruction fetch. Setting a capability's address (i.e., where it points) within bounds will retain the capability's validity tag. Setting addresses out of bounds is subject to the precision limits of the bounds compression model (see below and Section 4.3.5); broadly speaking, setting addresses

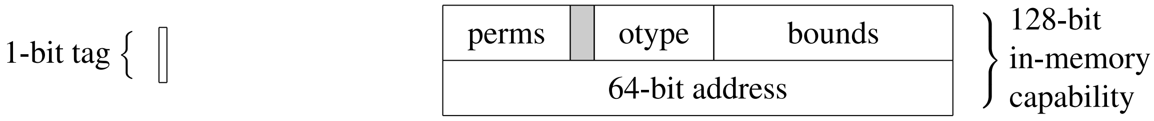

Figure 1: 128-bit CHERI Concentrate capability representation used in 64-bit CHERI-MIPS and 64-bit CHERI-RISC-V: 64-bit address and metadata in addressable memory; and 1-bit out-of-band tag.

"near" the capability's bounds will preserve the validity tag. (These out-of-bounds capabilities continue to authorize access only to memory within bounds.)

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

Object type If this value is not equal to the unsealed object type, the capability is "sealed" and cannot be modified or dereferenced, but can be used to implement opaque pointer types. This feature is not described further in this document, as it is primarily used to implement software compartmentalization rather than object-level memory protection.

When stored in memory, valid capabilities must be naturally aligned – i.e., at 64-bit or 128-bit boundaries, depending on capability size – as that is the granularity at which in-memory tags are maintained. Partial or complete overwrites with data, rather than a complete overwrite with a valid capability, lead to the in-memory tag being cleared, preventing corrupted capabilities from later being dereferenced.

In order to reduce the memory footprint of capabilities, capability compression is used to reduce the overhead of bounds so that the full capability, including address, permissions, and bounds fits within 64 or 128 bits (plus the 1-bit out-of-band tag). Bounds compression takes advantage of redundancy between the address and the bounds, which occurs because a pointer typically falls within (or close to) its associated allocation, and because allocations are typically well aligned. The compression scheme uses a floating-point representation, allowing high-precision bounds for small objects, but requiring stronger alignment and padding for larger allocations (see Section 7.5).

2.2 Architectural rules for capability use

The architecture enforces several important security properties on changes to this metadata:

Provenance validity ensures that capabilities can be used – for load, store, instruction fetch, etc. – only if they are derived via valid transformations of valid capabilities. This property holds for capabilities in both registers and memory.

Monotonicity requires that any capability derived from another cannot exceed the permissions and bounds of the capability from which it was derived (leaving aside sealed capabilities, used for domain transition, whose mechanism is not detailed in this report).

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

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

The hardware furthermore guarantees that capability tags and capability data is written atomically. For example, if one thread stores a valid capability and another writes arbitrary data to the same location, it is impossible to observe the arbitrary data with the validity bit set.

These architectural properties provide the foundation on which a capability-based OS, compiler, and runtime can implement C/C++-language memory safety. They have been made precise and have been proved, with machine-checked proof, to hold for the CHERI-MIPS architecture [6].

3 CHERI C/C++

The architectural-capability type can be used in a variety of ways by software. One particularly useful use case is in implementing *CHERI C/C++*. In this model, all C/C++ language-visible pointer types, as well as any implied pointers implementing vtables, return addresses, global variables, arrays of variadic-function arguments, and so on, are implemented using capabilities with tight bounds. This allows the architecture to imbue pointers with protection by virtue of architectural provenance validity, bounds checking, and permission checking, protecting pointers from corruption and providing strong spatial memory safety.

3.1 The CHERI C/C++ run-time environment

CHERI C code executes within a capability-aware run-time environment – whether "bare metal" with a suitable runtime, or in a richer, OS-based process environment such as CheriABI (see Section 5), which ensures that:

- capabilities are context switched (if required);
- tags are maintained by the OS virtual-memory subsystem (if present);
- capabilities are supported in OS control operations such as debugging (as needed);
- system-call arguments, the run-time linker, and other aspects of the OS Application Binary Interface (ABI) utilize capabilities rather than integer pointers; and
- the C/C++-language runtime implements suitable capability preservation (e.g., in `memcpy()`) and restriction (e.g., in `malloc()`).

In CheriBSD, our CHERI-extended version of the open-source FreeBSD operating system, CheriABI operates as a complete additional OS ABI. CheriABI is implemented in the style of a 32-bit or 64-bit OS personality, in that it requires its own set of suitably compiled system libraries and classes. We have also successfully adapted bare-metal runtimes, such as newlib, and embedded operating systems, such as FreeRTOS (CheriFreeRTOS) and RTEMS (CHERI-RTEMS), to support CHERI memory protection.

Outside of the OS and language runtime themselves, CHERI C/C++ require relatively few source-code-level changes to C/C++-language software. We explore those changes in the remainder of this document.

3.2 Referential, spatial, and temporal safety

CHERI C/C++ introduces a number of new types of protection not present in compilation to conventional architectures:

Referential safety protects pointers (references) themselves. This includes *integrity* (corrupted pointers cannot be dereferenced) and *provenance validity* (only pointers derived from valid pointers via valid manipulations can be dereferenced).

When pointers are implemented using architectural capabilities, CHERI's capability tags and provenance validity naturally provide this protection.

Spatial safety ensures that pointers may be used only to access memory within bounds of their associated allocation; dually, manipulating an out-of-bounds pointer will not grant access to another allocation.

This is accomplished by adapting various memory allocators, including the run-time linker for global variables, the stack allocator, and the heap allocator, to set the bounds on the capability implementing a pointer before returning it to the caller. Due to precision constraints on capability bounds, bounds on returned pointers may include additional padding, but will still not permit access to any other allocations (see Section 7.5). Monotonicity ensures that callers cannot later broaden the bounds to cover other allocations.

Referential safety and spatial safety are implemented in CheriBSD's pure-capability CheriABI execution environment and for bare-metal in CheriFreeRTOS and CHERI-RTEMS.

Temporal safety prevents a pointer retained after the release of its underlying allocation from being used to access its memory if that memory has been reused for a fresh allocation (e.g., after a fresh pointer to that memory has been returned by a further call to `malloc()` after the current pointer passed to `free()`).

Heap temporal safety is accomplished by preventing new pointers being returned to a previously allocated region of memory while any prior pointers to that memory persist in application-accessible memory. Memory will be held in *quarantine* until any prior pointers have been revoked; then the memory may be reallocated. Architectural capability tags and virtual memory allow intermittent *revocation sweeps* to accurately and efficiently locate and overwrite any capabilities implementing stale pointers. Spatial safety ensures that pointers cannot be used to reference other memory, including other freed memory.

Temporal safety is the object of ongoing experiments. A prototype that guards *heap* allocations has been developed for CheriABI on CheriBSD, but is not yet integrated with the main development branch. We currently have no plans to develop support for temporal memory safety in CheriFreeRTOS and CHERI-RTEMS, both due to the complexity of the temporal safety runtime, and also because of CHERI temporal safety's dependence on an MMU for performance.

4 Impact on the C/C++ programming model

Several kinds of changes may be required by programmers; the extent to which these changes impact a particular library or application will depend significantly on its idiomatic use of C. Our experience suggests that low-level system components such as run-time linkers, debuggers, memory allocators, and language runtimes require a modest but non-trivial porting effort. Similarly, support classes that include, for example, custom synchronization features, may also require moderate adaptation. Other applications may compile with few or no changes – especially if they are already portable across 32-bit and 64-bit platforms and are written in a contemporary C or C++ dialect. In the following sections, we consider various kinds of programmer-visible changes required in the CHERI C/C++ programming environment. In many cases, compiler warnings and errors can be used to identify potential issues compiling code as CHERI C/C++ (see Section 6).

4.1 Capability-related faults

When architectural capability properties are violated, such as by an attempt to dereference an invalid capability, access memory outside the bounds of a capability, or perform accesses not authorized by the permissions on a capability, this typically leads to a hardware exception (trap). Operating-system kernels are able to catch this exception via a trap handler, optionally delivering it to the run-time environment via OS-specific mechanisms.

However, the language-level behavior of CHERI C/C++ is considerably more subtle: existing undefined behavior semantics in C are retained. The compiler is free to assume that loads and stores will not trap (i.e., that any program is free of undefined behavior), and may optimize under this assumption, including reordering code. Architectural traps occur when dynamic loads and stores are attempted, and reordering could lead to potential confusing behavior for programmers.

In the CheriABI process environment, the operating system catches the hardware exception and delivers a `SIGPROT` signal to the user process; further information may be found in Section 5. In other environments, such as bare metal or under an embedded OS, behavior is specific to those environments, as it will depend both on how architectural exceptions are handled, and how those events are delivered to the C-language stack. Fail stop may be appropriate behavior in some environments, and is in fact the default behavior in CheriABI when `SIGPROT` is not handled.

4.2 Pointer provenance validity

CHERI C/C++ implement pointers using architectural capabilities, rather than using conventional 32-bit or 64-bit integers. This allows the provenance validity of language-level pointers to be protected by the provenance properties of CHERI architectural capabilities: only pointers implemented using valid capabilities can be dereferenced. Other types that contain pointers, `uintptr_t` and `intptr_t`, are similarly implemented using architectural capabilities, so that casts through these types can retain capability properties. When a dereference is attempted on a capability without a valid tag – including load, store, and instruction fetch – a hardware exception fires (see Section 4.1).

On the whole, the effects of pointer provenance validity are non-disruptive to C/C++ source code. However, a number of cases exist in language runtimes and other (typically less portable) C code that conflate integers and pointers that can disrupt provenance validity. In general, generated code will propagate provenance validity in only two situations:

Pointer types The compiler will generate suitable code to propagate the provenance validity of pointers by using capability load and store instructions. This occurs when using a pointer type (e.g., `void *`) or an integer type defined as being able to hold a pointer (e.g., `intptr_t`). As with attempting to store 64-bit pointers in 32-bit integers on 64-bit architectures, passing a pointer through an inappropriate type will lead to truncation of metadata (e.g., the validity tag and bounds). It is therefore important that a suitable type be used to hold pointers.

This pattern often occurs where an opaque field exists in a data structure – e.g., a `long` argument to a callback in older C code – that needs to be changed to use a capability-oblivious type such as `intptr_t`.

Capability-oblivious code In some portions of the C/C++ runtime and compiler-generated code, it may not be possible to know whether memory is intended to contain a pointer or not – and yet preserving pointers is desirable. In those cases, memory accesses must be performed in a way that preserves pointer provenance. In the C runtime itself, this includes `memcpy()`, which must use capability load and store instructions to transparently propagate capability metadata and tags.

A useful example of potentially surprising code requiring modification for CHERI C/C++ is `qsort()`. Some C programs assume that `qsort()` on an array of data structures containing pointers will preserve the usability of those pointers. As a result, `qsort()` must be modified to perform memory copies using pointer-based types, such as `intptr_t`, when size and alignment require it.

4.2.1 Recommended use of C-language types

As confusion frequently arises about the most appropriate types to use for integers, pointers, and pointer-related values, we make the following recommendations:

`int`, `int32_t`, `long`, `int64_t`, ... These pure integer types should be used to hold integer values that will never be cast to a pointer type without first combining them with another pointer value – e.g., by using them as an array offset. Most integers in a C/C++-language program will be of these types.

vaddr_t This is a new integer type introduced by CHERI C and should be used to hold virtual addresses. vaddr_t should not be directly cast to a pointer type for dereference; instead, it must be combined with an existing valid capability to the address space to generate a dereferenceable pointer. Typically, this is done using the cheri_address_set(c, x) function.

size_t, ssize_t These integer types should be used to hold the unsigned or signed lengths of regions of address space.

ptrdiff_t This integer type describes the difference of indices between two pointers to elements of the same array, and should not be used for any other purpose. It can be added to a pointer to obtain a new pointer, but the result will be dereferenceable only if the address lies within the bounds of the pointer from which it was derived.

Less standards-compliant code sometimes uses ptrdiff_t when the programmer more likely meant intptr_t or (less commonly) size_t. When porting code, it is worthwhile to audit use of ptrdiff_t.

intptr_t, uintptr_t These integer types should be used to hold values that may be valid pointers if cast back to a pointer type. When an intptr_t is assigned an integer value – e.g., due to constant initialization to an integer in the source – and the result is cast to a pointer type, the pointer will be invalid and hence non-dereferenceable. These types will be used in two cases: (1) Where there is uncertainty as to whether the value to be held will be an integer or a pointer – e.g., for an opaque argument to a callback function; or (2) Where it is more convenient to place a pointer value in an integer type for the purposes of arithmetic (which takes place on the capability's address and in units of bytes, as if the pointer had been cast to char *).

The observable, integer range of a uintptr_t is the same as that of a vaddr_t (or ptrdiff_t for intptr_t), despite the increased *alignment* and *storage* requirements.

intmax_t, uintmax_t According to the C standard, these integer types should be 'capable of representing any value of any (unsigned) integer type'. In CHERI C/C++, they are not provenance-carrying and can represent the integer *range* of uintptr_t/intptr_t, but not the capability metadata or tag bit. As the observable value of uintptr_t/intptr_t is the pointer address range, we believe this choice to be compatible with the C standard.

Additionally, due to ABI constraints, it would be extremely difficult to change the width of these types from 64 to 129 bits. This is also true for other architectures such as x86: despite Clang and GCC supporting an __int128 type, intmax_t remains 64 bits wide.

We generally do not recommend use of these types in CHERI C/C++. However, the types may be useful in printf() calls (using the %j format string width modifier) as the inttypes.h PRI* macros can be rather verbose.

max_align_t This type is defined in C as 'an object type whose alignment is the greatest fundamental alignment' and this includes capability types for CHERI C/C++. We found that some custom allocators use sizeof(long double) or sizeof(uint64_t) to align their return values. While this appears to work on most architectures, in CHERI C/C++ this must be changed to alignof(max_align_t).[1]

char *,... These pointer types are suitable for dereference, but in general should not be cast to or from arbitrary integer values. Valid pointers are always derived from other valid pointers (including those cast to intptr_t or uintptr_t), and cannot be constructed using arbitrary integer arithmetic.

It is important to note that uintptr_t is no longer the same size as size_t. This difference may require making some changes to existing code to use the correct type depending on whether the variable needs to be able store a pointer type. In cases where this is not obvious (such as for a callback argument), we recommend the use of uintptr_t. This ensures that provenance is maintained.

4.2.2 Capability alignment in memory

Because tags apply only to memory locations that are capability-aligned and capability-sized, unaligned storage of pointers will either generate a run-time hardware exception (if a capability-aware load or store is performed), or discard the tag (if a capability-oblivious memory copy is performed – e.g., using memcpy() to copy from an aligned location to an unaligned one). One example of this is Berkeley DB (BDB) when used as an in-memory implementation rather than as an on-disk database format. Even when patched to use memcpy() to copy objects stored as data, it does not ensure sufficient alignment in its internal storage to preserve tags. We therefore recommend against using BDB for this purpose. While unaligned pointer use is uncommon in C programs, as data-structure layouts are normally designed to keep them strongly aligned for performance and atomicity reasons, any code depending on unaligned pointers will need to be changed.

4.2.3 Single-origin provenance

In the CHERI memory protection model, capabilities are derived from a single other capability. However, in C code, expressions may construct a new intptr_t value from more than one provenance-carrying parent intptr_t – for example, by casting both a pointer and a literal value to intptr_t-s, and then adding them. In that case, the compiler must decide which input capability provides the capability metadata (bounds, permissions, ...) to be used in the output value. Consider for example the following code:

```
void *c1 = (void *)((uintptr_t)input_ptr + 1);
void *c2 = (void *)(1 + (uintptr_t)input_ptr);
uintptr_t offset = 1;
void *c3 = (void *)(offset + (uintptr_t)input_ptr);
```

[1] It is important to use alignof instead of sizeof since many common implementations, such as GCC and FreeBSD, define max_align_t as a struct and not a union.

In C with integer pointers, the values of c1, c2, and c3 might be expected to have the same value as input_ptr, except with the address incremented by one. In CHERI C, each expression includes an arithmetic operation between provenance-carrying types. While not visible in the source code, the constant 1 is promoted to a capability type, uintptr_t. In the current implementation, the compiler will return the expected provenance-carrying result for cases c1 and c2 but not c3.[2] For c1 and c2, the compiler sees that one of the sides is a non-provenance-carrying integer type that was promoted to uintptr_t and therefore selects the other operand as the provenance source. It is not feasible to infer the correct provenance source for the third case, so the compiler will emit a warning.[3] The current behavior for such ambiguous cases is to select the left-hand-side as the provenance source, but we are considering making this an error in the future. The recommended approach to resolve such ambiguous cases is to change the type of one operand to a non-provenance-carrying type such as size_t. Alternatively, if the variable declaration cannot be changed, it is also possible to use a cast in the expression itself.

```
size_t offset_size_t = 1;
void *c3_good1 = (void *)(offset_size_t + (uintptr_t)input_ptr);

uintptr_t offset_uintptr_t = 1;
void *c3_good2 = (void *)((size_t)offset_uintptr_t + (uintptr_t)input_ptr);
```

We also provide a new attribute cheri_no_provenance that can be used to annotate variables or fields of type intptr_t/uintptr_t where the underlying type cannot be changed:

```
struct S {
    uintptr_t maybe_tagged;
    uintptr_t never_tagged __attribute__((cheri_no_provenance));
}
void test(struct S s, uintptr_t ptr) {
    void *x1 = (void *)(s.maybe_tagged + ptr); // ambiguous, currently uses LHS
    void *x2 = (void *)(s.never_tagged + ptr); // not ambiguous, uses RHS
}
```

4.3 Bounds

CHERI C/C++ pointers are implemented using capabilities that enforce lower and upper bounds on access. In the pure-capability run-time environment, those bounds are normally set to the range of the memory allocation into which the pointer is intended to point. Because of capability compression, increased alignment requirements may apply to larger allocations (see Section 7.5).

[2] Historically, the CHERI compiler would select the left-hand-most pointer in the expression as the provenance source. While this model follows a single consistent rule, it can lead to surprising behavior if an expression places the provenance-carrying value to the right-hand-side. In the example above, the value of c1 would be a valid capability, but c2 and c3 would hold an untagged value (albeit with the expected address).

[3] We could add a data-flow-sensitive analysis to determine whether values are the result of promotion from a non-provenance-carrying type. However, this would add significant complexity to the compiler and we have not seen many cases where this would have avoided changes to the source code.

Bounds may be set on pointers returned by multiple system components including the OS kernel, the run-time linker, compiler-generated code, system libraries, and other utility functions. As with violations of provenance validity, out-of-bounds accesses – including load, store, and instruction fetch – trigger a hardware exception (see Section 4.1).

4.3.1 Bounds from the compiler and linker

The compiler will arrange that language-level pointers to stack allocations have suitable bounds, and that the run-time linker will return bounded pointers to global variables. Bounds will typically be set based on an explicitly requested allocation size (e.g., via the size passed to `alloca()`) or, for compiler-generated code or linker-allocated memory, by the C type mechanism (e.g., `sizeof(foo)`), adjusted for precision requirements arising from capability compression. In some cases, such as with global variables allocated in multiple object files, the actual size of the allocation may not be resolved until run time, by the run-time linker. These bounds will typically not cause observable changes in behavior – other than hardware exceptions when (accidentally) performing an out-of-bounds access.

4.3.2 Bounds from the heap allocator

`malloc()` will set bounds on pointers to new heap allocations. In typical C use, this is not a problem, as programmers expect to access addresses only within an allocation.

However, in some uses of C, there may be an expectation that memory access can occur outside the allocation bounds of the pointer via which memory access takes place. For example, if an integer pointer difference `D` is taken between pointers to two different allocations (`B` and `A`), and later added to pointer `A`, the new pointer will have an address within `B`, but permit access only to `A`. This idiom is mostly likely to be found with non-trivial uses of `realloc()` (e.g., cases where multiple pointers into a buffer allocated or reallocated by `realloc()` need to be updated). We note that the subtraction of two pointers from different allocations is undefined behavior in ISO C, and risks mis-optimization from breaking compiler alias analysis assumptions. Further, *any* operation on the pointer passed to `realloc()` is undefined upon return. Instead, we suggest that the programmer measure a pointer `P`'s offset into an object `A` *prior to* `realloc()` and derive new pointers from the `realloc()` result `B` and these offsets. (i.e., compute $\texttt{B} + (\texttt{P} - \texttt{A})$ rather than $\texttt{P} + (\texttt{B} - \texttt{A})$).[4]

4.3.3 Subobject bounds

CHERI C/C++ also supports automatically restricting the bounds when a pointer is taken to a subobject – for example, an array embedded within another structure that itself has been heap allocated. This will prevent an overflow on that array from affecting the remainder of the structure, improving spatial safety. Subobject bounds are not enabled by default as they

[4] While it may seem that `A` remains available after `realloc()`, our revocation sweeps which enforce temporal safety may have atomically replaced this with a non-pointer value. The scalar value $\texttt{D} = \texttt{P} - \texttt{A}$ will naturally be preserved by revocation.

may require additional source code changes for compatibility, but can be enabled using the `-Xclang -cheri-bounds=subobject-safe` compiler flag.

One example of C code that requires changes for subobject bounds is the `containerof` pattern, in which pointer arithmetic on a pointer to a subobject is used to recover a pointer to the container object – for example, as seen in the widely used BSD `queue.h` linked-list macros or the generic C hash-table implementation, `uthash.h`.

In these cases, an opt-out annotation can be applied to a given type, field or variable that instructs the compiler to not tighten bounds when creating pointers to subobjects. We currently define three opt-out annotations that can be used to allow existing code to disable use of subobject bounds:

Completely disable subobject bounds It is possible to annotate a typedef, record member, or variable declaration with:

```
__attribute__((cheri_no_subobject_bounds))
```

to indicate that the compiler should not tighten bounds when taking the address or a C++ reference. In C++11/C20 mode this can also be spelled as `[[cheri::no_subobject_bounds]]`.

```
struct str {
    /*
     * Nul-terminated string array -- pointers taken to this subobject will
     * use the array's bounds, not those of the container structure.
     */
    char                str_array[128];

    /*
     * Linked-list entry element -- because of the additional attribute,
     * pointers taken to this subobject will use the container structure's
     * bounds, not those of the specific field.
     */
    struct list_entry  str_le __attribute__((cheri_no_subobject_bounds));
} str_instance;

void
fn(void)
{
    /* Struct pointer gets bounds of str_instance. */
    struct str *strp = &str_instance;

    /* Character pointer gets bounds of the subobject, not str_instance. */
    char *c = str_instance.str_array;

    /* Struct pointer gets bounds of str_instance, not the subobject. */
    struct list_entry *lep = &str_instance.str_le;
}
```

Disable subobject bounds in specific expressions It is also possible to opt out of bounds-tightening on a per-expression granularity by casting to an annotated type:

```
char *foo(struct str *strp) {
    return (&((__attribute__((cheri_no_subobject_bounds))struct str *)
        strp)->str_array);
}
```

Use remaining allocation size In certain cases, the size of the subobject is not known, but we still know that data before the field member will not be accessed (e.g., variable size array members inside structs). Pre-C99 code will declare such members as fixed-size arrays, which will cause a hardware exception if the allocation does not grant access to that many bytes.[5] To use the remaining allocation size instead of completely disabling bounds (and thus protecting against buffer underflows) the annotation:

```
__attribute__((cheri_subobject_bounds_use_remaining_size))
```

can be used. When targeting C++11/C20:

```
[[cheri::subobject_bounds_use_remaining_size]]
```

is also supported. Examples of this pattern include FreeBSD's `struct dirent`, which uses `char d_name[255]` for an array that is actually of variable size, with the containing allocation (e.g., of the heap) being sized to allow additional space for array entries regardless of size in the type definition. For example:

```
struct message {
    int      m_type;

    /*
     * Variable-length character array -- because of the additional
     * attribute, pointers taken to this subobject will have a lower bound
     * at the first address of the array, but retain an upper bound of the
     * allocation containing the array, rather than 252 bytes higher.
     */
    char     m_data[252]
                 __attribute__((cheri_subobject_bounds_use_remaining_size));
};
```

The use of subobject bounds imposes additional compatibility constraints on existing C and C++ code. While we have not encountered many issues related to subobject bounds in existing code, it does slightly increase the porting effort.

4.3.4 Other sources of bounds

Bounds may also be set by other parts of the implementation. For example, the kernel may set bounds on pointers to new memory mappings (see Section 5), and the system library may set bounds on pointers into returned buffers from APIs – e.g., `fgetln()`. More detailed information on how C/C++ code can set bounds can be found in Section 7.

[5] If flexible arrays members are declared using the C99 syntax with empty square brackets, the compiler will automatically use the remaining allocation size.

4.3.5 Out-of-bounds pointers

ISO C permits pointers to go only one byte beyond their original allocation, but widely used code sometimes constructs transient pointer values that are further out of bounds. For example, `for` loops iterating over an array may increment a pointer into the array by the array entry size before performing an overflow check that terminates the loop. This temporarily constructs an out-of-bounds pointer without an out-of-bounds dereference taking place.

To support this behavior, capabilities can hold a range of out-of-bounds addresses while retaining a valid tag, and CHERI-enabled hardware performs bounds checks only on pointer use (i.e., dereference), not on pointer manipulation. Dereferencing an out-of-bounds pointer will raise a hardware exception (see Section 4.1). However, an out-of-bounds pointer can be dereferenced once it has been brought back in bounds, by adjusting the address or supplying a suitable offset in the dereference.

There is, however, a limit to the range of out-of-bounds addresses a capability can hold. The capability compression model exploits redundancy between the pointer's address and its bounds to reduce memory overhead (see Section 2.1). However, when a pointer goes out of bounds, this redundancy is reduced, and at some point the bounds can no longer be represented within the capability. The architecture prohibits manipulations that would produce such a capability. Depending on the architecture and context, this may lead to the tag being cleared, resulting in an invalid capability, or in an immediate hardware exception being thrown. Attempting to dereference the invalid capability will fail in the same manner as a loss of pointer provenance validity (see Section 4.2). The range of out-of-bounds addresses permitted for a capability is a function of the length of the bounded region and the number of bits used for bounds in the capability representation. With 27 bits of the capability used for bounds, CHERI-MIPS and 64-bit CHERI-RISC-V provide the following guarantees:

- A pointer is able to travel at least ¼ the size of the object, or 2 KiB ($2^{floor(bounds\_bits/2)-2}$), whichever is greater, above its upper bound.
- It is able to travel at least ⅛ the size of the object, or 1 KiB ($2^{floor(bounds\_bits/2)-3}$), whichever is greater, below its lower bound.

In general, programmers should not rely on support for arbitrary out-of-bounds pointers. Nevertheless, in practice, we have found that the CHERI capability compression scheme supports almost all in-the-field out-of-bounds behavior in widely used software such as FreeBSD, PostgreSQL, and WebKit.

4.4 Pointer comparison

In CHERI C/C++, pointer comparison considers only the integer address part of a capability. This means that differences in tag validity, bounds, permissions, and so on, will not be considered when by C operators such as ==, <, and <=. On the whole, this leads to intuitive behavior in systems software, where, for example, `malloc()` adjusts bounds on a pointer before returning it to a caller, and then expects an address-wise comparison to succeed when the pointer is later returned via a call to `free()`. However, this behavior could also lead to potentially confusing results; for example:

- If a tag on a pointer is lost due to non-provenance-preserving `memcpy()` (e.g., a `for` loop copying a sequence of bytes), the source and destination pointers will compare as equal even though the destination will not be dereferenceable.
- If a `realloc()` implementation returns a pointer to the same address, but with different bounds, a caller check to see if the passed and returned pointers are equal will return `true` even though an access might be permitted via one pointer but not the other.

However, practical experience has suggested that the current semantics produce fewer subtle bugs, and require fewer changes, than having comparison operators take the tag or other metadata into account.[6]

4.5 Implications of capability revocation for temporal safety

Heap temporal safety utilizes revocation sweeps, which, after some quarantine period, replace in-register and in-memory capabilities to freed memory with non-dereferenceable values. For performance reasons, that replacement may be substantially deferred, or, if there is little demand for fresh allocations, may never occur. Pointer value replacement may also permit some instances of a pointer to continue to be usable for longer than others, but the referenced memory will not be reallocated or otherwise reused until all instances have been rendered unusable. This model does permit non-exploitable *use-after-free* of heap memory, but prohibits exploitable memory aliasing by disallowing *use-after-reallocation*.

A pointer's value after `free()` is undefined, and so dereference is an undefined behavior. In practice, however, the value of a `free()`-d pointer may still be observed in a number of situations, including in lockless algorithms, which may compare an allocated pointer to a freed one.

Our systems have a choice of replacement values for revoked pointers; all that is required for correct temporal safety is that the replacement not authorize access to memory. Our prototype implementation clears the tag when replacing, as this certainly removes authority and possibly simplifies debugging and non-dereferencing operations, as the original capability bits are left behind. For example, pointer equality checks that compare only the addresses of the two pointers (and not their tag values) will continue to work as expected. With revocation performed this way, software making explicit use of tags must be designed to tolerate capability tag clearing by revocation.

Unfortunately, tag-clearing risks type confusion if programmers intend to use the capability tag to distinguish between integers and pointers in tagged unions (we have so far generally discouraged this idea, but understand why it may remain attractive). Therefore, we have considered other options for revocation, including tag-preserving *permission*-zeroing (but tag preservation) and wholesale replacement with `NULL` (i.e., the untagged all zero value). These

[6] The CHERI Clang compiler supports an experimental flag `-cheri-comparison=exact` that causes capability equality comparisons to also include capability metadata and the tag bit.

options may be more attractive for some software, and would have different implications for the C/C++ programming model.

We anticipate that revocation will remain a tag-clearing operation by default, as tag-clearing removes any risk of needlessly re-examining the capability in later revocations. However, it may be possible to allow coarse control over revocation behavior either per process or by region of the address space. In the latter case, `mmap()` may gain flags specifying which revocation behavior is desirable for capabilities pointing *into* the mapped region and/or `madvise()` may gain flags controlling the revocation behavior of capabilities *within* a target region. Which of these or similar mechanisms provide utility to software and can be offered at reasonable performance remains an open question.

4.6 Bitwise operations on capability types

In most cases bitwise operations – such as those used to store or clear flags in the lower bits of pointers to well-aligned allocations – will result in the expected `uintptr_t` value being created. However, there are some corner cases where the result may be a tagged (but out-of-bounds) capability when an integer value is expected. Dually, bitwise operations may also result in the loss of tags if intermediate results become unrepresentable (recall Section 4.3.5).[7] Most bitwise operations on `uintptr_t` fall into one of three categories for which we provide higher-level abstractions.

Aligning pointer values If the C code is attempting to align a pointer or check the alignment of pointers, the following compiler builtins should be used instead:

`T __builtin_align_down(T ptr, size_t alignment)` This builtin returns `ptr` rounded down to the next multiple of `alignment`.

`T __builtin_align_up(T ptr, size_t alignment)` This builtin returns `ptr` rounded up to the next multiple of `alignment`.

`_Bool __builtin_is_aligned(T ptr, size_t alignment)` This builtin returns `true` if `ptr` is aligned to at least `alignment` bytes.

One advantage of these builtins compared to `uintptr_t` arithmetic is that they preserve the type of the argument and can therefore remove the need for intermediate casts to `uintptr_t`. Moreover, using these builtins allows for improved compiler diagnostics and can result in better code-generation compared to hand-written functions or macros. We have submitted these builtins as part of the upstream Clang 10.0 release, so they can also be used for code that does not depend on CHERI.

[7] Previous versions of the compiler used the capability offset (address minus base) instead of the address for arithmetic on `uintptr_t`. This often resulted in unexpected results and therefore we switched to using the address in `uintptr_t` arithmetic instead. The old offset-based mode may be interesting for garbage collected C where addresses are less useful and therefore it can still be enabled by passing `-cheri-uintcap=offset`. However, this may result in significantly reduced compatibility with legacy C code.

Storing additional data in pointers In many cases the minimum alignment of pointer values is known and therefore programmers assume that the low bits (which will always be zero) can be used to store additional data.[8] Unused high pointer bits cannot be used for additional metadata since toggling them causes a large change to the address field, and capabilities that are significantly far out-of-bounds cannot be represented (see Section 4.3.5).

The compiler-provided header `<cheri.h>` provides explicit macros for this use of bitwise arithmetic on pointers. The use of these macros is currently optional,[9] but we believe that they can improve readability compared to hand-written bitwise operations. Additionally, the bitwise-AND operation is ambiguous since it can be used both to clear bits (which should return a provenance-carrying `uintptr_t`) and to check bits (which should return an integer value). In complex nested expressions, these macros can avoid ambiguous provenance sources (see Section 4.2.3) since it shows the compiler which intermediate results can carry provenance.

`uintptr_t cheri_low_bits_clear(uintptr_t ptr, vaddr_t mask)` This function clears the low bits of `ptr` in the same way as `ptr & ~mask`. It returns a new `uintptr_t` value that can be used for memory accesses when cast to a pointer. `mask` should be a bitwise-AND mask less than `_Alignof(ptr)`.

`vaddr_t cheri_low_bits_get(uintptr_t ptr, vaddr_t mask)` This function returns the low bits of `ptr` in the same way as `ptr & mask`. It should be used instead of the raw bitwise operation since it can never return an unexpectedly tagged value. `mask` should be a bitwise-AND mask less than `_Alignof(ptr)`.

`uintptr_t cheri_low_bits_or(uintptr_t ptr, vaddr_t bits)` This function performs a bitwise-OR of `ptr` with `bits`. In order to retain compatibility with a non-CHERI architecture, `bits` should be less than the known alignment of `ptr`.

`uintptr_t cheri_low_bits_set(uintptr_t ptr, vaddr_t mask, vaddr_t bits)`
This function sets the low bits of `ptr` to `bits` by clearing the low bits in `mask` first.

Computing hash values The compiler will also warn when operators such as modulus or shifts are used on `uintptr_t`. This usually indicates that the pointer is being used as the input to a hash function or similar computations. In this case, the programmer should not be using `uintptr_t` but instead cast the pointer to `vaddr_t` and perform the arithmetic on this type instead. This has the advantage that it can be slightly more efficient than `uintptr_t` arithmetic on a split-register file architecture such as CHERI-MIPS.

[8] CHERI actually provides many more usable bits than a conventional architecture. In the current implementation of 128-bit CHERI, any bit between the least significant and the 9th least significant bit may be toggled without causing the tag to be cleared in pointers that point to the beginning of an allocation (i.e., whose *offset* is zero). If the pointer is strongly aligned, further bits may be toggled without clearing the tag.

[9] Until recently, not using these macros could result in subtle bugs at run time since pointer equality comparisons included the tag bit in addition to the address.

4.7 Function prototypes and calling conventions

CHERI C/C++ distinguishes between integer and pointer types at an architectural level, which can lead to compatibility problems with older C programming styles that fail to unambiguously differentiate these types:

Unprototyped (*K&R*) functions Because pointers can no longer be loaded and stored without using capability-aware instructions, the compiler must know whenever a load or store might operate on a pointer value. The C-language default of using an integer type for function arguments when there is not an appropriate function prototype will cause pointer values to be handled improperly; this is also true on LP64 ABIs (e.g., most 64-bit POSIX systems).[10] To avoid these problems, the CHERI Clang compiler emits a warning (`-Wcheri-prototypes`) by default when a function without a declared prototype is called. This warning is less strict than `-Wstrict-prototypes` and can be used to convert *K&R* functions that may cause problems.[11] This should not be an issue for C code written in the last 20 years, but many core operating-system components can be significantly older.

Variadic arguments The calling convention for variadic functions passes all variadic arguments via the stack and accesses them via an appropriately bounded capability. This provides memory-protection benefits, but means that vararg functions must be declared and called via a correct prototype.

Some C code assumes that the calling convention of variadic and non-variadic functions is sufficiently similar that they may be used interchangeably. Historically, this included the FreeBSD kernel's implementation of `open()`, `fcntl()`, and `syscall()`.

4.8 Data-structure and memory-allocation alignment

CHERI C/C++ have stronger alignment requirements than C/C++ on conventional architectures. These requirements arise from two sources: that capabilities themselves must be aligned at twice the integer architectural pointer width, and that capability compression constrains the addresses that can be used for bounds on larger objects.

4.8.1 Restrictions in capability locations in memory

CHERI C/C++ constrain how and where pointers can be stored in memory in two ways:

Alignment CHERI's tags are associated with capability-aligned, capability-sized locations in physical memory. Because of this, all valid pointers must be stored at such locations, potentially disrupting code that may use other alignments.

[10] The forthcoming ISO C2x standard makes function declarations with an empty parameter list equivalent to a parameter list consisting of a single `void`.

[11] If the *K&R* function is defined within the same file, the compiler can determine the correct calling convention and will not emit a warning.

On the whole, for performance and atomicity reasons, pointers are strongly aligned even on non-tagged architectures – however, when C constructs such as `__packed` are used, unaligned pointers can arise, and will not work with CHERI. While the compiler and native allocators (stack, heap, ...) will provide sufficient alignment for capability-based pointers, custom allocators may align allocations to `sizeof(intmax_t)` rather than `alignof(max_align_t)`.

Size CHERI capabilities are twice the size of an integer able to describe the full address space. On 64-bit systems, this means that CHERI pointers will have a width of 128 bits – while maintaining the arithmetic properties of a 64-bit integer address. C code historically embeds assumptions about pointer size in a number of forms, all of which will need to be addressed when porting to CHERI, including:

- Assuming that a pointer will fit into the largest integer type.
- Assuming that the number of bits in a pointer type is the same as the number of bits indexing the address space it can refer to.
- Assuming that the number of bits in a pointer type is the same as the number of bits suitable for use in performing bit-wise manipulations of pointer values.
- Assuming that pointers must either be 32 or 64 bits.
- Assuming that aligning to `sizeof(double)` is sufficient to store any type.
- Assuming that high bits of the pointer address can be used for additional metadata. This is not true on CHERI since toggling high bits of a pointer can cause it to be so far out of bounds that it is no longer representable due to the compression of pointer bounds. However, it is still possible to use the low bits for additional metadata (see Section 4.6).

These portability problems will typically be found due to hardware exceptions thrown on attempted unaligned accesses of capability values (see Section 4.1). However, they can also arise in the form of stripped tag bits, leading to invalid capabilities that cannot be dereferenced, if, for example, pointer values are copied into inappropriately aligned allocations.

5 The CheriABI POSIX process environment

The CheriABI process environment implements a standard POSIX/UNIX API, but in some areas there are changes to API semantics (e.g., in the handling of tagged pointer values and I/O) or new functionality (such as relates to handling capability-related faults).

5.1 POSIX API changes

Writing and reading pointers via files In the CheriABI process environment, only untagged data (not tagged pointers) may be written to or read from files. If a region of memory containing valid pointers is written to a file, and then read back, the pointers in that region will no longer be valid. If a file is memory mapped, then pages mapped copy-on-write (MAP_PRIVATE) are able to hold tagged pointers, since they are swap-backed rather than file-backed, but pages mapped directly from the buffer cache (MAP_SHARED) are not.

Passing pointers via IPC In the CheriABI process environment, only untagged data, not tagged pointers, may be passed via various forms of message-passing Inter-Process Communication (IPC). Some existing software takes advantage of a shared address-space layout (via fork()) to pass pointers to elements of shared data structures (e.g., entries in dispatch tables). This code must be converted to use indexes into tables or other lookup mechanisms rather than passing pointers via IPC.

mmap() bounds In CheriABI, the mmap() system call returns a bounded capability to the allocated address space. To ensure the capability does not overlap other allocations, lengths that would otherwise be unrepresentable are rounded up and padded with a new type of guard pages. These guard pages fault on access and may not be mapped over. They are unmapped when the rest of the mapping is unmapped.

mmap() permissions The permissions of the capability returned by mmap() are determined by a combination of the requested page protections and the capability passed as an address hint (or fixed address with MAP_FIXED). When using the pattern of requesting a mapping with PROT_NONE and then filling in sections (as is done in run-time linkers, VM host environments, etc), it is necessary to ensure that the initial capability has the right permissions. The prot argument has been extended to accept additional flags indicating the maximum permission the page can have so that a linker might request a reservation for a library with the permissions (PROT_MAX(PROT_READ|PROT_WRITE|PROT_EXEC) | PROT_NONE), which would return a capability permitting loads, stores, and instruction fetch while mapping the pages with no (MMU) permissions.

5.2 Handling capability-related signals

When a capability hardware exception fires, the operating system will map it into the UNIX SIGPROT signal. By default, this signal terminates the process, but the signal can be caught by registering a SIGPROT handler. When the signal handler fires, siginfo.si_code will be set to describe the cause of the fault; available values, defined in signal.h, include:

PROT_CHERI_BOUNDS Capability bounds fault – an out-of-bounds access was attempted.

PROT_CHERI_PERM Capability permission fault – the attempted access exceeded the permissions granted by a capability.

PROT_CHERI_SEALED Capability sealed fault – dereferencing a sealed capability was attempted.

PROT_CHERI_TAG Capability tag fault – dereferencing an invalid capability was attempted.

6 CHERI compiler warnings and errors

The CHERI Clang compiler includes many diagnostic warnings to identify code that is incompatible with CHERI C/C++ or may result in behavioral differences. In many cases, a successful compilation that does not emit any CHERI-specific warnings will result in a functional spatially-safe program. However, some incompatibilities (e.g., memory allocators returning insufficiently aligned pointers) cannot yet be diagnosed statically. This section describes some of the more-commonly seen compiler warnings and provides suggestions on how to change the source code to be compatible with CHERI C/C++. All these warnings are enabled when the `-Wall` compiler flag is set.

6.1 Loss of provenance

This common compiler warning is triggered when casting a non-capability type (e.g., `long`) to a pointer. As mentioned in Section 4.2, the result of this cast is a `NULL`-derived capability with the address set to the integer value. As any `NULL`-derived capability is untagged, any attempt to dereference it will trap.

Usually, this warning is caused by programmers incorrectly assuming that `long` is able to store pointers. The fix for this problem is to change the type of the cast source to a provenance-carrying type such as `intptr_t` or `uintptr_t` (see Section 4.2.1):

```
char *example_bad(long ptr_or_int) {
    return strdup((const char *)ptr_or_int);
}
char *example_good(intptr_t ptr_or_int) {
  return strdup((const char *)ptr_or_int);
}
```

```
<source>:2:17: warning: cast from provenance-free integer type to pointer type will give
    pointer that can not be dereferenced [-Wcheri-capability-misuse]
  return strdup((const char *)ptr_or_int);
                ^
1 warning generated.
```

In some cases, this warning can be a false positive. For example, it is common for C callback APIs take a `void *` data argument that is passed to the callback. If this value is in fact an integer constant, the warning can be silenced by casting to `uintptr_t` first:

```
void invoke_cb(void (*cb)(void *), void *);
void callback(void *arg);
void false_positive_example(int callback_data) {
    invoke_cb(&callback, (void *)callback_data); // warning
    invoke_cb(&callback, (void *)(uintptr_t)callback_data); // no warning
}
```

```
<source>:4:24: warning: cast from provenance-free integer type to pointer type will give
    pointer that can not be dereferenced [-Wcheri-capability-misuse]
  invoke_cb(&callback, (void *)callback_data); // warning
                       ^
<source>:15:24: warning: cast to 'void *' from smaller integer type 'int' [-Wint-to-void-
    pointer-cast]
  invoke_cb(&callback, (void *)callback_data); // warning
                       ^
2 warnings generated.
```

6.2 Ambiguous provenance

For arithmetic and bitwise binary operations between uintptr_t/intptr_t, the compiler can generally infer which side of the expression should be used as the provenance (and bounds) source. However, as noted in Section 4.2.3, there are cases that are ambiguous as far as the compiler is concerned.

Consider for example a structure that holds a pointer and a small number of flags. In this case the pointer is known to be aligned to at least 8 bytes, so the programmer uses the lowest 3 bits to store additional data:

```
typedef struct { uintptr_t data; } pointer_and_flags;
void set_ptr(pointer_and_flags *p, void *value) {
    p->data = (p->data & (uintptr_t)7) | (uintptr_t)(value);
}
void set_flags(pointer_and_flags *p, unsigned flags) {
    p->data = p->data | (flags & 7);
}
```

```
<source>:3:40: warning: binary expression on capability types '__uintcap_t' and 'uintptr_t' (
    aka '__uintcap_t'); it is not clear which should be used as the source of provenance;
    currently provenance is inherited from the left-hand side [-Wcheri-provenance]
    p->data = (p->data & (uintptr_t)7) | (uintptr_t)(value);
              ~~~~~~~~~~~~~~~~~~~~~~~~ ^ ~~~~~~~~~~~~~~~~~~
1 warning generated.
```

Unlike the compiler, the programmer knows that inside set_ptr() capability metadata should always be taken from the value argument. The suggested fix for this problem is fix is to cast the non-pointer argument to an integer type:

```
void set_ptr(pointer_and_flags *p, void *value) {
    p->data = (size_t)(p->data & (uintptr_t)7) | (uintptr_t)(value);
}
```

6.3 Underaligned capabilities

This warning is triggered when packed structures contain pointers. As mentioned in Section 4.8.1, pointers must always be aligned to the size of a CHERI capability (16 bytes for a 64-bit architecture). This warning can be triggered by code that attempts to align pointers to at least 8 bytes (e.g., for compatibility between 32- and 64-bit architectures). For example:

```
struct AtLeast8ByteAlignedBad {
    void *data;
} __attribute__((packed, aligned(8)));
```

```
<source>:1:8: warning: alignment (8) of 'struct AtLeast8ByteAlignedBad' is less than the
    required capability alignment (16) [-Wcheri-capability-misuse]
struct AtLeast8ByteAlignedBad {
       ^
<source>:1:8: note: If you are certain that this is correct you can silence the warning by
    adding __attribute__((annotate("underaligned_capability")))
1 warning generated.
```

The simplest fix for this issue is to either increase alignment to be CHERI-compatible, or use a ternary expression to include `alignof(void *)`:

```
#include <stdalign.h>
struct AtLeast8ByteAlignedGood {
    void *data;
} __attribute__((packed,aligned(alignof(void *) > 8 ? alignof(void *) : 8))
    );
```

In the rare case that creating a potentially underaligned pointer is actually intended, the warning can be silence by adding a `annotate("underaligned_capability")` attribute:

```
struct UnderalignPointerIgnoreWarning {
    void *data;
} __attribute__((packed, aligned(4), annotate("underaligned_capability")));
```

7 C APIs to get and set capability properties

CHERI C/C++ supports a number of new APIs to get and set capability properties given a pointer argument. Although most software does not need to directly manage capability properties, there are some cases when application code needs to further constrain permissions or limit bounds associated with pointers. For example, high-performance applications may contain custom memory allocators and wish to narrow bounds and permissions on returned pointers to prevent overflows between its own allocations.

7.1 CHERI-related header files

A set of compiler built-in functions provide access to capability properties of pointers. Two new header files (distributed as part of the CHERI Clang compiler) provide access to further CHERI-related programming interfaces including more human-friendly macro wrappers around the compiler builtins, and also definitions of key CHERI constants:

`cheriintrin.h` defines interfaces to access and modify capability properties. It also defines constants for capability permissions that are portable across all implementations of CHERI.

`cheri.h` provides macros for slightly higher-level operations such as the manipulation of low pointer bits (Section 4.6).

When compiling for CheriBSD, the following header provides additional constants relating to OS use of capabilities – for example, software-defined permission bits:

`cheri/cheri.h` defines constants such as those used in the capability permission mask.

7.2 Retrieving capability properties

The following APIs allow capability properties to be retrieved from pointers:

`vaddr_t cheri_address_get(void *c)` Return the address of the capability `c`.

`vaddr_t cheri_base_get(void *c)` Return the lower bound of capability `c`.

`size_t cheri_length_get(void *c)` Return the length of the bounds for the capability `c`. The base plus the length gives the upper bound on `c`'s address.

`size_t cheri_offset_get(void *c)` Return the difference between the address and the lower bound of the capability `c`.

`size_t cheri_perms_get(void *c)` Return the permissions of capability `c`. (See Section 7.4.)

`_Bool cheri_tag_get(void *c)` Return whether capability `c` has its validity tag set.

7.3 Modifying or restricting capability properties

The following APIs allow capability properties to be refined on pointers:

`void *cheri_address_set(void *c, vaddr_t a)` Return a new capability with the same permissions and bounds as `c` with the address set to `a`. This can be useful to re-derive a valid pointer from an address.

`cheri_address_set()` is able to set an address `a` that is outside of the current bounds of `c`. The resulting capability is treated as an out-of-bounds pointer as described in Section 4.3.5. However, if the address `a` is not representable in the current bounds of `c` due to capability compression, `cheri_address_set()` returns a capability without the tag bit set.

`void *cheri_bounds_set(void *c, size_t x)` Narrow the bounds of capability `c` so that the lower bound is the current address (which may have been increased relative to `c`'s original lower bound), and its upper bound is suitable for a length of `x`.

Note that the effective bounds of the returned capability may be wider than the range [`cheri_address_get(c)`, `cheri_address_get(c) + x`) due to capability compression (see Section 7.5), but they will always be a subset of the original bounds.

`void *cheri_bounds_set_exact(void *c, size_t x)` Narrow the bounds of capability `c` so that the lower bound is the current address, and its upper bound is `cheri_address_get(c) + x`. This is similar to `cheri_bounds_set()` but will raise a hardware exception if the resulting capability is not precisely representable instead of rounding the bounds.

`void *cheri_perms_and(void *c, size_t x)` Perform a bitwise-AND of capability `c`'s permissions and the value `x`, returning the new capability (see Section 7.4).

`void *cheri_tag_clear(void *c)` Clear the tag on `c`, returning the new capability.

7.4 Capability permissions

A number of capability permissions are available for use; only those relating to CHERI memory protection are enumerated here:

`CHERI_PERM_EXECUTE` Authorize instruction fetch via this capability.

`CHERI_PERM_LOAD` Authorize data load via this capability.

`CHERI_PERM_LOAD_CAP` Authorize capability load via this capability. If the permission is not present, the tag on the loaded value will be silently cleared.

`CHERI_PERM_STORE` Authorize data store via this capability.

`CHERI_PERM_STORE_CAP` Authorize capability store via this capability. If the permission is not present, and the tag on the stored capability is valid, then a hardware exception will be thrown.

In addition to architectural permissions, CHERI capabilities have software-defined permissions. CheriBSD defines the following additional memory-protection-related permission:

`CHERI_PERM_CHERIABI_VMMAP` A CheriABI-specific user permission that the kernel uses to authorize modifications to virtual-memory mappings. If the permission is not present, system calls that alter the contents or the presentation of memory mappings will reject the request. As this is a CheriBSD-specific permission, it is not defined in `cheriintrin.h` and requires inclusion of `cheri/cheri.h`.

7.5 Bounds alignment due to compression

Bounds imprecisions may require a memory allocator to increase the alignment of an allocation, or increase padding on an allocation, to prevent bounds from spanning more than one object. When the length of an object exceeds $2^{floor(bounds\_bits/2)-1}$ (i.e., 4 KiB for CHERI-MIPS and 64-bit CHERI-RISC-V), additional alignment requirements apply to the lower and upper bounds. The alignment required for allocations exceeding the minimum representable range (4 KiB for CHERI-MIPS and 64-bit CHERI-RISC-V) is 2^{E+3} bytes, where E is determined from the length, l, by $E = 52 - \text{CountLeadingZeros}(l[64 : floor(bounds\_bits/2)])$.

Correctly computing the rounded size and minimum alignment for a given allocation is non-trivial and may require many instructions to compute, especially in the context of fast allocators such as the stack allocator. Moreover, the architectural constants used for bounds precision differ across architectures or their variations, and so alignment constraints also vary. For example, the number of bits available for bounds differs between 32-bit and 64-bit CHERI-RISC-V, and also between 64-bit CHERI-RISC-V and Morello.

To avoid overly specific software knowledge of alignment requirements, and also to allow efficient calculation of alignment constraints during (for example) stack allocation, the CHERI ISA provides instructions that allow determining precisely representable allocations. These instructions can be generated using compiler builtins that are provided by `cheriintrin.h`:

`size_t cheri_representable_length(size_t len)` returns the length that a capability would have after using `cheri_bounds_set` to set the length to `len` (assuming appropriate alignment of the base).

`size_t cheri_representable_alignment_mask(size_t len)` returns a bitmask that can be used to align an address downwards such that it is sufficiently aligned to create a precisely bounded capability.

The precisely representable base address can be computed using:

```
base = base & cheri_representable_alignment_mask(len);
```

When allocating from a contiguous buffer, the base needs to be aligned upwards instead of downwards. This can be done with the following code:

```
size_t required_alignment(size_t len) {
    return ~cheri_representable_alignment_mask(len) + 1;
}
struct Buffer {
    void *data;
    size_t allocated;
};
void *allocate_next(struct Buffer *buf, size_t len) {
    char *result = buf->data + buf->allocated;
    result = __builtin_align_up(result, required_alignment(len));
    size_t rounded_len = cheri_representable_length(len);
    buf->allocated = (result + rounded_len) - (char *)buf->data;
    return cheri_bounds_set_exact(result, rounded_len);
}
```

Software written to use these compiler builtins, rather than encoding alignment requirements directly, is more likely to be portable between CHERI-MIPS, CHERI-RISC-V, and Morello.

7.6 Implications for memory-allocator design

One use case of these APIs is high-performance applications that contain custom memory allocators and wish to narrow the bounds of returned pointers. Two kinds of modifications are typically required:

Changes to alignment to allow for capabilities and bounds Changes relating to alignment fall into two categories. First, those required to allow pointers to be stored within allocations, which requires that allocations be aligned to the pointer width (128 bits). Second, further alignment changes will be required to ensure that bounds can be represented precisely. This requires suitably aligning both the bottom and top bounds to exclude any other live allocations, as described in Section 7.5.

Reaching allocation metadata on `free()` It is often the case that allocators utilize the value of the pointer passed to their custom `free()` function to locate corresponding metadata – for example, by always placing that metadata immediately before the allocation, which would be outside of the allocation's bounds. Therefore, some additional work may be required to derive a pointer to the allocation's metadata via another global capability, rather than the one that has been passed to `free()`.

These two concerns may interact: When a custom allocator places metadata at the beginning of the allocation, care must be taken that the resulting pointer is still strongly aligned. While porting programs to run on CHERI, we found multiple sub-allocators that used 8 bytes of metadata after the result from `malloc()`. This causes the resulting pointer to no longer be sufficiently aligned to store capabilities without faulting or stripping tag bits.

Note that it is also possible to use the above APIs to validate inputs to `free()`, which is useful when the consumer of `free()` is, for example, an untrusted compartment or a component of a web browser that might be influenced by an attacker. In such cases, `free()` should validate that the passed-in capability is tagged, is in-bounds, and points to a legitimate, still-allocated allocation. For allocators engaged in revocation for temporal safety, concurrent revocation opens the door to TOCTTOU races within `free()`; additional care must be taken to prevent a double-`free()` using a stale pointer from freeing an object allocated after revocation.

8 Further reading

The primary reference for the CHERI Instruction-Set Architecture (ISA) is the ISA specification; at the time of writing, the most recent version is CHERI ISAv7 [8]:

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

Our technical report, *An Introduction to CHERI*, provides a high-level overview of the CHERI architecture, ISA modeling, hardware implementations, and software stack [7]:

`https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf`

We published a paper on idiomatic C and spatial memory protection at ASPLOS 2015 [1]:

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

We published a paper on CheriABI and the adaptation of a complete OS userspace and application suite to a pure-capability process environment at ASPLOS 2019 [2]:

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

We also released an extended technical-report version of this paper that includes greater implementation detail [3]:

`https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-932.pdf`

We published a paper on CHERI and temporal memory safety for the heap at Oakland 2020 [4]:

`https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/`
`2020oakland-cornucopia.pdf`

We published a paper on C-language pointer provenance, and the implications for software design, at POPL 2019; CHERI C was a case study in the practical enforcement of capability provenance-validity enforcement [5]:

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

9 Acknowledgements

We gratefully acknowledge the helpful feedback from our colleagues, including Hesham Almatary, Ruben Ayrapetyan, Silviu Baranga, Jacob Bramley, Rod Chapman, Paul Gotch, Al Grant, Brett Gutstein, Alfredo Mazzinghi, Alan Mycroft, and Lee Smith. This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 (“CTSRD”) and HR0011-18-C-0016 (“ECATS”). 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. This work was supported in part by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 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 Advanced Grant ELVER. We also acknowledge the EPSRC REMS Programme Grant (EP/K008528/1), Arm Limited, HP Enterprise, and Google, Inc. Approved for Public Release, Distribution Unlimited.

References

[1] David Chisnall, Colin Rothwell, Robert N. M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis and Peter G. Neumann. ‘Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine’. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’15. Istanbul, Turkey: ACM, Mar. 2015, pp. 117–130. DOI: 10.1145/2694344.2694367. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201503-asplos2015-cheri-cmachine.pdf.

[2] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son and Jonathan Woodruff. ‘CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment’. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2019. ASPLOS ’19. Providence, RI, USA: ACM, 2019, pp. 379–393. DOI: 10.1145/3297858.3304042. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201904-asplos-cheriabi.pdf.

[3] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son and Jonathan Woodruff. *CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment*. Technical Report UCAM-CL-TR-932. University of Cambridge, Computer Laboratory, Apr. 2019. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-932.pdf`.

[4] Nathaniel Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, Theo Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann and Robert Watson. 'Cornucopia: Temporal Safety for CHERI Heaps'. In: *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2020, pp. 1507–1524. DOI: `10.1109/SP40000.2020.00098`. URL: `https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2020oakland-cornucopia.pdf`.

[5] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson and Peter Sewell. 'Exploring C Semantics and Pointer Provenance'. In: *Proceedings of the ACM on Programming Languages*. Cascais, Portugal, Jan. 2019, 67:1–67:32. DOI: `10.1145/3290380`. URL: `https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201901-popl-cerberus.pdf`.

[6] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson and Peter Sewell. 'Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process'. In: *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2020, pp. 1007–1024. DOI: `10.1109/SP40000.2020.00055`. URL: `https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/202005oakland-cheri-formal.pdf`.

[7] 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, 2019. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf`.

[8] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son and Hongyan Xia. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7)*. Technical Report UCAM-CL-TR-927. University of Cambridge, Computer Laboratory, June 2019. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf`.