International Journal on Software Tools for Technology Transfer (2025) 27:225–237
https://doi.org/10.1007/s10009-025-00781-6

GENERAL

Special Issue: SOAP 2024

Static analysis to make the most of CHERI C/C++ for existing code: improving memory safety at scale

Irina Dudina[1] · Ian Stark[1]

Accepted: 23 January 2025 / Published online: 20 February 2025

Abstract
We describe and evaluate custom static analyses to support transitioning existing C/C++ codebases to *CHERI* hardware. CHERI is a novel architectural extension, implemented for RISC-V and AArch64, that uses *capabilities* to provide fine-grained memory protection and scalable software compartmentalization. While the existing CHERI toolchain can recompile large code collections for the platform with only a few source changes, those changes are nonetheless critical: we demonstrate that static analysis can help to identify where they are needed and what must be done to avoid later runtime faults. We provide custom checkers for the Clang Static Analyzer to handle capability alignment, copying through memory, and manipulation as integers. Beyond simply picking up problems in existing code, we also have checkers that identify where code can take advantage of capabilities to better enforce least privilege and improve spatial memory safety. We evaluate all implemented checkers on a sample of packages from the CheriBSD ports library (408 packages, 5.4 MLoC analyzed) and confirm by analyzing true-positive warning rates that the reports produced are sufficiently high quality for practical use.

Keywords Static analysis · CHERI · Software porting

1 Introduction

CHERI (Capability Hardware Enhanced RISC Instructions) is a contemporary capability architecture with multiple hardware implementations, including for RISC-V and AArch64, that provide low-overhead hardware support for memory protection and software compartmentalization [21, 25]. At the software level this is supported by a C/C++ dialect known as *CHERI C/C++* with the distinctive feature that all program pointers (explicit and implied) are implemented with architectural capabilities [22].

On CHERI hardware these capabilities are a complete replacement for pointers, extending purely numerical addresses with upper and lower bounds on access, permissions, and other fields for features like compartmentalization and control-flow integrity (Fig. 1). These make capabilities larger than the underlying address width, such as 128 bits for a 64-bit architecture. Every capability has a software-inaccessible out-of-band *validity tag* that follows it through registers and memory. This prevents forgery, with CHERI providing hardware-enforced limitations on modification and use of capabilities.

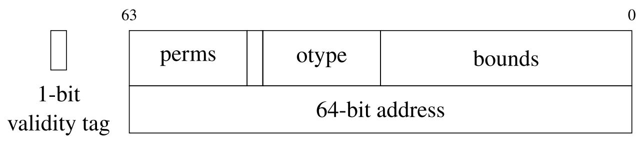

Fig. 1 A 128-bit CHERI capability with its out-of-band validity tag

A significant benefit of the CHERI architecture is the ease of transition: large amounts of existing C/C++ code recompile with minimal modification and immediately gain hardware-backed memory protection. CHERI design choices mean compilers can routinely use capabilities to implement C/C++ pointers, and there are Clang/LLVM and GCC versions supporting this. The precise meaning of the CHERI-C/C++ dialect is given by an executable mechanized semantics [30] that extends the existing Cerberus C semantics [14].

Moving existing C/C++ code to a CHERI platform provides strong guarantees of memory safety, but may still require code changes to ensure smooth execution – most notably, where minor violations of memory safety have previ-

✉ I. Dudina
irina.dudina@ed.ac.uk

I. Stark
ian.stark@ed.ac.uk

[1] The University of Edinburgh, Edinburgh, UK

Listing 1 Capability size and alignment
```
        // FIXME: type of ret must be compatible
        //        with void* (e.g., intptr_t)
        unsigned long ret, *retPtr = &ret;
        pthread_join(threadId, (void**)retPtr);
```

ously escaped detection. Certain legacy idioms also interact poorly with capabilities, and a very few need rethinking entirely.

CHERI encourages its users to adhere to the *principle of least privilege* [17]. Many memory access restrictions arising from this principle appear naturally in the language semantics and can be immediately enforced by a compiler. Some however depend on program logic and must be explicitly expressed in source code: for example, setting bounds on pointers returned by a custom memory allocator. Looking forward, embracing capabilities opens up opportunities for robust compartmentalization and other protections, but again requiring work to integrate these new features.

Tackling portability across architectural change is not new, moving from 32-bit to 64-bit architectures being a landmark example. It is therefore familiar that a key source of trouble is where C code makes assumptions, perhaps unwittingly, that were true in its original setting but no longer hold in the new context. For example, (1) assumptions on type size and alignment, in particular that certain types can be used interchangeably; (2) assumptions on pointer representation, used to manually inspect or modify address values; (3) assumptions on type representation, hard-coded into "magic constants". All these show up in transitioning C/C++ code to CHERI. Some are reliably detected in compilation, with suitable errors or warnings. What remains, though, presents a classic opportunity for static analysis – subtle errors, scattered sparsely across large volumes of code, where any eventual fault may be distant from its original cause. Some may show up in testing; but even where packages have high-coverage test suites those can share the same architectural assumptions and portability problems. To address this, we demonstrate custom static checkers that can identify up-front the crucial changes necessary to reliably adapt C/C++ codebases for CHERI.

As a next step, we extended our tool to include more checkers and, in particular, to go beyond simply picking up problems to identifying areas where code can directly take advantage of capabilities to enforce least privilege. For example, detecting coding patterns where explicitly narrowing capability bounds can improve memory safety. Another area is subobject bounds, where there already exist compiler switches to emit very tight capabilities but these are typically disabled by default because using them requires careful code changes for reliable porting.

1.1 Outline

The remainder of this section introduces the specific porting issues that our work addresses and notes some related work in the field. This is followed by sections setting out our central contribution: six new checkers for the Clang Static Analyzer tool (CSA) [12].

- §2 Capability Alignment
- §3 Capability Copy
- §4 Capability as Integers
- §5 Suballocation Bounds
- §6.1 Subobject Representability
- §6.2 Insufficient Pointer Size Check

Each section explains the issue being addressed and the action of the corresponding checker. Following this, Sect. 7 describes our implementation, its evaluation on a sample of the CheriBSD ports library (5.4 MLoC analyzed across 408 packages), and the results for each individual checker. Section 8 summarizes the outcome of our work.

1.2 Adapting code for CHERI C

The CHERI C/C++ Programming Guide [22] provides extensive commentary on potential issues in porting existing code to a capability platform. We pick out here the specific problems addressed by our work.

Capability size and alignment In CHERI C, all pointers are implemented as capabilities, typically 128 bits wide and requiring 128-bit alignment. This can cause problems when code assumes pointers are the same size as some other, smaller, type. For example, Listing 1 shows a pattern we have seen in several real projects.

The address of local variable `unsigned long ret` is passed as a `void**` value to the second argument of `pthread_join` function, which will then write a `void*` value to this address. As long as `unsigned long` has size and alignment requirements at least as strict as `void*`, this code will work fine. However, for CHERI this is no longer true: attempts to write a 128-bit `void*` capability value to the narrower `unsigned long` stack allocation will result in a capability bounds fault.

Copying objects containing capabilities To maintain integrity, CHERI capabilities must be copied in their entirety using special capability load/store instructions that propagate validity tags. In CHERI C/C++, these instruction are emitted when copying evident capability values (pointers and `(u)intptr_t`). However, custom implementations of general-purpose memory management routines may copy data in smaller units (e.g., `memcpy` from Listing 3 in Sect. 3). When used for copying objects containing capabilities these will silently invalidate the capabilities leading to errors downstream.

Pointers as integers This includes using noncapability integer types to hold pointer values and manipulating capabilities by using bitwise and arithmetic operations on `(u)intptr_t` values.

Capability representability CHERI implementations use a floating-point-style CHERI Concentrate [27] scheme to encode the 64-bit address, upper and lower bounds among other fields within the 128-bit capability structure (Fig. 1). This exploits the redundancies in representation of address and bounds observed in most cases to reduce the memory overhead brought by capabilities. With this scheme, capabilities for small objects (less than 16 KiB on Arm Morello [1]) can always be encoded with precise bounds. For a larger allocation to have a capability with representable bounds, its address and size must comply with stricter alignment requirements. When precise bounds are not representable, the tightest representable bounds covering the allocation are used instead, potentially overlapping with adjacent objects. This means such allocations need additional padding to avoid the exposure of adjacent objects. In a similar way, if the address value of an `(u)intptr_t` capability is set to a value too far out of bounds then it may lose some bounds precision. Representability must be taken into account when, for example, implementing CHERI-aware memory allocators, when enabling subobject bounds (discussed below in Sect. 6.1), and when performing arithmetic on `(u)intptr_t` address values.

Capability bounds Following the principle of least privilege, when passing a capability to another software component it is important to ensure that its bounds only cover the object to be accessed and not the whole of the allocated memory region within which it lies. Properly managing this is important for custom allocators and any other manual memory management on a stack, a heap, or in static memory. CHERI C/C++ provides an API to set capability properties with, for example, a function `cheri_bounds_set` to narrow bounds manually; but as ever this needs to be used correctly.

In general, adaptation of contemporary C/C++ source code to CHERI C/C++ is straightforward and requires only very small changes to the source code (e.g., 0.026% LoC reported for porting a desktop stack based on X11 and KDE [23]). Low-level system and heavily platform-specific C/C++ code tends to require more porting effort [5, 11, 13, 19], as does some code not complying with existing C standards.

While many CHERI-compatibility issues are detected and well explained by the compiler, there remain certain classes of issue that it does not report and that surface only at run-time. Uncovering these during porting depends on test coverage, which can be haphazard when the tests naturally predate CHERI. A further complication arises when a developer responds to compiler warnings with changes that silence the warning rather than fix the issue – for example, by introducing a type cast that may not be valid under CHERI but is enough to make the warning go away.

Taken together these mean that remaining CHERI-related faults in ported software may be small in number, widely spread across large bodies of code, possibly concealed behind attempted fixes, and yet cause immediate software failure if triggered on execution. This is where a static analysis tool adds clear value: with less strict performance and false-positive requirements than a compiler it can perform wider and more thorough analysis resulting in better coverage overall.

We have implemented our analysis in the Clang Static Analyzer (CSA) [12], a static analysis tool for C, C++, and Objective-C programs within the Clang/LLVM project. As standard CSA provides two analysis methods: static symbolic execution for path-sensitive interprocedural analysis and AST matchers for simple pattern-matching checks. Both analyses are extendable with custom checkers, which we provide. In addition, we have identified some of the existing CSA checkers as especially important in the context of porting code to CHERI: 1) Use of hardcoded address `(alpha.core.FixedAddr)`, 2) Subtraction of pointers to distinct objects `(alpha.core.PointerSub)`, 3) Allocator `sizeof` operand mismatch `(unix.MallocSizeof)`

1.3 Related work

As noted above, the CHERI Clang compiler provides its own diagnostics `[-Wcheri]` that flag up many types of CHERI-incompatible code patterns [16].

The *CHERIseed* tool is a software emulation of capabilities that supports running CHERI-C/C++ code on a conventional (noncapability) host machine [4]. This provides a dynamic analysis of portability issues by exposing unsafe code that could fault on real CHERI hardware.

The ESBMC model-checker has an extension *ESBMC-CHERI* to support C program verification on CHERI-enabled platforms [6]. This aims to statically detect memory safety violations and compatibility issues for CHERI-C, with a capability-aware memory model and CHERI Clang integration.

Several studies have investigated the application of static analysis techniques to facilitate seamless API migration. For example, the *Desynchronizer* tool relies on static analysis to determine where calls to synchronous JavaScript API functions can be replaced with their asynchronous counterparts [10]. Similarly, the *Meditor* API migration tool for Java libraries employs static analysis to identify control and data dependencies that allow it to align code correspondence across versions and from this infer API replacement plans [29].

Listing 2 Underaligned capability storage

```
typedef struct S_s {
        /* regular fields */
        // ...
        /* final flexible array member */
        // FIXME: ensure alignment
        uint8_t extra [1];
} S;

S* alloc ( ssize_t extra_size , void** extra ) {
        // ... lines skipped ...
        // Set up structure S
        S* g = /* ... */ ;
        // Pass back address of flexible array
        *extra = &g->extra[0];
        return g;
}

typedef struct T_s {
        /* has pointer fields */
} T;

T* p; /* T requires 16-byte alignment */
S* q = alloc ( sizeof (T), (void**)&p);
```

2 Capability alignment

Our first checker uses symbolic execution to identify misalignment (and potential misalignment) of capability storage in memory. We demonstrate the issue at hand, its impact, and then present our checker together with some CHERI-specific concerns.

2.1 Capability alignment restrictions

As previously highlighted, all pointers in CHERI C must be stored in memory at addresses aligned to a 128-bit boundary. This is because the out-of-band capability tags that ensure unforgeability of capabilities only apply to such capability-aligned and capability-sized memory locations. Attempts to read or write capabilities at underaligned addresses will result in either a trap or stripping of the validity tag.

Consider the code snippet from Listing 2 that we extracted from a library written in C++. Function `alloc` is called to allocate a memory for the structure of type `T`, which contains pointer fields and therefore requires capability alignment. It returns the address of the allocated memory by writing to a pointer provided as a second argument. The implementation of `alloc` sets up a structure `S` with a flexible array member **`uint8_t`** `extra` and passes the address of this field as the start address of the allocated memory for the user. Since **`uint8_t`** type has a trivial alignment requirement, this address may not be properly aligned for storing structure `T` there and objects of type `T` stored there will have their capabilities invalidated.

2.2 Impact of misaligned pointers

Accessing objects in memory at misaligned addresses is a problem that is not unique to CHERI C: however, the absolute requirement to maintain capability validity does raise its profile. On a conventional architecture, underaligned access can result in a fault, affect atomicity or performance. Nevertheless, for some C programs, even when alignment is not ensured by using the correct type with appropriate alignment requirements, this issue may not manifest in runtime due to several reasons. First, runtime alignment of the object may be enforced by platform-specific ABI alignment guarantees (e.g., on static variables). Second, correct object alignment may be an implication of unintentional and, therefore, fragile properties of a current data layout (e.g., structure field alignment may be guaranteed by the combination of aligned field offset and parent structure alignment, regardless of the field type). Similarly, other features of memory layout controlled by the compiler (e.g., order of static variables) may prevent the problem from manifesting, but minor changes in the source code or compilation process may break this. At the same time, even when manifesting in runtime, the issue may remain undiscovered on some architectures due to subtle ill effects. C compilers report can `-Wcast-align` warning for simple cases, but this is very noisy and is turned off by default.

2.3 Detection of misaligned pointers

To detect this bug, we developed a symbolic execution checker in CSA. For every analyzed path, we track the *smallest guaranteed alignment* for pointers and trailing-zeros count for integers.

For the case of tracked allocations (*Typed Memory Regions* in CSA), we use the type alignment of the objects allocated on stack, heap, or in static memory, taking into account `alignas` and similar attributes, if present for this allocation. For a symbolic pointer, CSA creates a *Symbolic Region* to represent the memory block pointed to by this symbolic pointer [28]. For this case, we cannot always rely on the pointer type, as we did not track the actual allocation of this object. We rely on pointer type only for pointer variables, which are function arguments at the top of the analyzed call stack or global variables, as these variables can be seen as arbitrary input values to the analyzed function. Exceptions are **`void`**`*` and **`char`**`*` pointer arguments and variables, as they are often used to pass pointer values of more than one type, depending on the other input values.

We track trailing-zeros count for integers to propagate alignment values for pointer arithmetic, including pointer shifts and manual pointer alignment via binary operations (e.g., `ALIGNUP` macro).

We report a warning when a pointer value with the smallest guaranteed requirement A is cast to a pointer to the type with an alignment requirement stricter than A or stored as such a pointer (as in Listing 2).

2.4 CHERI-specific issues

There are aspects of this issue specific to CHERI C. Firstly, in CHERI C, pointers require 16-byte alignments, which often breaks legacy code assumptions – aligning to **sizeof double**, **sizeof uint64_t**, and sometimes even **sizeof long double** is not enough. Secondly, capability-aware unaligned load or store of a capability will generate a run-time hardware exception, therefore, previously undetected misalignment, sometimes almost harmless on a conventional architecture, can result in a crash on CHERI. Finally, unlike other types, capabilities cannot be moved around as a sequence of bytes and stored at a misaligned address, even temporarily. Thus, attempting to `memcpy` a capability through unaligned memory will result in tag stripping during the first copying (storing); copying it back to a properly aligned address will not bring the tag back. An actual crash will happen later during an attempt to use this capability for memory access, making this issue very hard to investigate.

Therefore, we also need to track memory regions that may contain capabilities and report a warning when we detect a memory copy to/from such region from/to an underaligned memory region. To track memory regions potentially containing capabilities, we rely on pointer types for both *Symbolic* and *Typed* regions and propagate this property through `memcpy`-like library calls.

To be able to detect the usage of underaligned allocations for storing capabilities, like in the example in Listing 2, we need to detect a single intraprocedural path that contains both the initialization of the pointer with a misaligned value and usage of this pointer as capability storage. In practice, these two events are often distant; a single path covering both of them may be very long and span across multiple functions or even modules, making it very difficult to analyze. However, even if we do not have a call instruction in the analyzed path that explicitly uses the `alloc` function to allocate memory for a capability-aligned structure, we can still suspect the implementation of `alloc` as potentially not CHERI-friendly. We can argue that `*extra` pointer, passed by address for the initialization to this function, has type **void***, which indicates that it may be used to store arbitrary data there – including objects containing capabilities. As a heuristic, we report assigning addresses of underaligned memory with unknown content (potential storage allocations) to **void*** variables that are top-level function arguments, global variables, or structure fields (potential pointers to capability storage). Although this approach leads to a significant number of false warnings (see Sect. 7 for evaluation), it allows detection of a misaligned allocation even if it was never explicitly used to store or load capabilities, but could be so used.

Listing 3 Tag-stripping memcpy
```
// FIXME: increase block size on CHERI
#define BIGBLOCKSIZE (sizeof(long) << 2)

void *memcpy (void *dst,
              const void *src,
              size_t len) {
  if (!TOO_SMALL (len) &&
      !UNALIGNED (src, dst)) {
    long *aligned_dst = (long*)dst;
    const long *aligned_src = (long*)src;
    while (len >= BIGBLOCKSIZE) {
      *aligned_dst++ = *aligned_src++;   // Tag
      *aligned_dst++ = *aligned_src++;   // lost
      *aligned_dst++ = *aligned_src++;   // on
      *aligned_dst++ = *aligned_src++;   // copy
      len -= BIGBLOCKSIZE;
    }
  } else { ... }
}
```

3 Capability copy

The only way to construct a valid CHERI capability is from another valid capability, known as the *provenance validity* rule. Therefore, whenever there is a need to copy, move, or swap a capability, it must be copied as a whole using capability-aware instructions that preserve the validity tag. The compiler will emit such instructions whenever capability-sized and capability-aligned data is explicitly copied. However, if memory that contains a valid capability is copied in parts, piecemeal, then that will strip the tag to give an invalid capability; and any general-purpose routines that could potentially copy or move capabilities (`memcpy`, `realloc`, `qsort`, etc.) need to take care with this. Listing 3 shows an example of a capability-unaware memory copy routine we extracted from a popular C standard library implementation.

We developed a capability-copy checker to detect functions that move data in a way that could result in a tag-stripping capability copy. We report a capability copy warning for a copy operation if and only if:

1. A memory chunk of size S is loaded from memory pointed by a *source* pointer (+offset);
2. This memory chunk is then stored to memory pointed by a *destination* pointer (+offset);
3. The *source* and *destination* memory regions hold, or may hold, capabilities (discussed below);
4. Size S is less than capability size;
5. *Source* and *destination* may be capability-aligned; and

6. Copying is performed in a loop or a sequence of assignments long enough to copy a capability.

Conditions 5 and 6 are required to avoid false warnings for the typical implementation of `memcpy` where the case of unaligned or small copy is handled separately (line 18 in Listing 3).

As previously when detecting underaligned generic storage, we cannot rely on always having a call to a memory-copying function explicitly requesting capability-copying as a confirmation that this function will be used to move around capabilities. In this case this problem can be even more pronounced, as such functions are often implemented as library functions and such invocation may not be present in the analyzed codebase, let alone in the same module. In particular, we again assume that top-level **`void`**`*` function arguments may point to objects containing capabilities.

We did also hypothesize that this assumption can be extended to **`char`**`*` arguments, as **`char`**`*` pointers can be used to iterate over arbitrary data. The obvious exception would be functions that expect C strings as **`char`**`*` arguments. We therefore implemented a detector that would disable this assumption if there were signs of C-string-like manipulation involving this pointer (e.g., checking bytes against character literals or `NULL`-terminator, passing the pointer to string processing functions like `strcpy`, etc.). Sadly, this approach did not provide a significant supply of true warnings and added many false reports related to C strings (the detector multiplied warnings by around 10 times, but only 2% were true positives). String detection could be improved, but the number of actual issues detected using this assumption does not provide strong motivation for that.

Since this checker highlights code that loads only fragments of a capability value, it will also report where individual bytes of a capability are used in arithmetic, for example, in computing a hash value. This too is worth detecting, as in this situation the CHERI C/C++ Programming Guide recommends extracting address value from a capability and using it instead for the sake of performance.

4 Capabilities as integers

According to the C standard, **`(u)intptr_t`** is an integer type capable of holding object pointers. Therefore, in CHERI **`(u)intptr_t`** must be implemented with capabilities as well.

4.1 `NULL`-derived capabilities

If **`(u)intptr_t`** holds an integer value rather than a pointer value, then it is represented with so-called `NULL`*-derived capability*, which is an untagged (invalid, not dereferenceable) capability with its address value set to the given integer value.

Assumptions about integer-pointer round-trips that affect portability are often encountered in the legacy code. For example, it is commonly thought that **`long`** variables can hold values of pointer type. On CHERI, when casting a valid capability (pointer or **`(u)intptr_t`**) to a noncapability integer type (e.g., **`long`**, **`ptrdiff_t`** or **`size_t`**), only an address value will be preserved, and the validity tag, bounds and metadata will be lost. If this value is converted back to a pointer type (either directly or via **`(u)intptr_t`**), it will become a `NULL`-derived invalid capability. The compiler can detect this in the simplest case of a direct cast; static analysis can report more general cases.

We implemented a symbolic execution checker for tracking **`(u)intptr_t`** values provenance. It tracks `NULL`-derived capabilities and detects those that are used as pointers. It reports two types of warnings: for a general case of `NULL`-derived **`(u)intptr_t`** capability being cast to a pointer type, and for the specific case of provenance loss, when the analyzer tracked the fact that a noncapability integer value originated from a valid capablitity, indicating an attempt of a round-trip pointer conversion through a noncapability integer type.

4.2 Ambiguous provenance

For binary arithmetic or bitwise operations on **`(u)intptr_t`** values, CHERI-C applies a *single-provenance rule* that the resulting capability has its validity derived from just one of the arguments. Unless one of the arguments is a noncapability (integer) type promoted to **`(u)intptr_t`**, the compiler cannot know from which argument to derive the resulting capability and will emit a `[-Wcheri-provenance]` warning. Currently, for these cases the result will be derived from a left-hand-side by default. Fixing this warning requires explicitly telling the compiler what side to pick by casting the other argument to a noncapability type. A static analyzer can help resolve these warnings by tracking the origin of the **`(u)intptr_t`** values, highlighting which will be a valid capability, and suggesting the appropriate fix. We developed a checker that tracks integer and pointer values stored as **`(u)intptr_t`** type and reports the following warnings for **`(u)intptr_t`** binary operations with ambiguous provenance source.

Both operands are derived from pointers This code is not compatible with CHERI and should be rewritten. Real-world examples include using XOR of pointers for branchless select and XOR-linked lists.

LHS is `NULL`-derived, and RHS is derived from a pointer Under compiler defaults, the resulting capability will be `NULL`-derived. The RHS should likely be used to derive the result instead.

Other cases Those are fine with the default compiler behavior and are not reported if `[-Wcheri-provenance]` is disabled in the original project build.

Listing 4 Suballocations escape on assign

```
 1 typedef struct {
 2   S1 *s1; size_t n1;
 3   S2 *s2; size_t n2;
 4 } T;
 5
 6 T* allocate_T(size_t n1, size_t n2) {
 7   T* t = malloc( sizeof(T)
 8                        + n1 * sizeof(S1)
 9                        + n2 * sizeof(S2) );
10   t->s1 = (S1*) &t[1];
11   t->n1 = n1;
12   \\ FIXME: narrow the bounds for s1
13
14   t->s2 = (S2*) &t->s1[n1];
15   t->n2 = n2;
16   \\ FIXME: narrow the bounds for s2
17
18   return t;
19 }
```

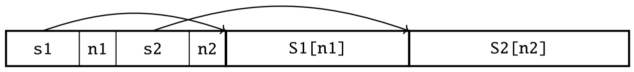

Fig. 2 Structure T

5 Suballocation bounds

When taking an address of the automatic or static variable, the compiler is responsible for setting the bounds of the resulting capability to the memory range of the corresponding object. Similarly, a CHERI-aware allocator should set the appropriate bounds for the returned capability for allocated memory.

For the cases when user code prefers to allocate a memory buffer (on the heap, stack, or in static memory) and manually place distinct objects there (e.g., for the sake of performance), it should use the CHERI C `cheri_bounds_set` intrinsic to narrow the bounds to achieve the same level of spatial memory safety.

Let us look at the example in Listing 4 extracted from the source code of a popular database. This function allocates a buffer for three consecutive objects: structure `T` and two arrays `S1[n1]` and `S2[n2]`, which addresses are stored in the structure `T` (Fig. 2). The issue with this implementation is the fact that all three capabilities `t`, `t->s1`, and `t->s2` will have the same bounds initially set by `malloc`. Therefore, it will be possible to, for example, access elements of the second array `S2[n2]` using the capability `t->s1` for the first array and vice versa, which would be a violation of the least privilege principle. One could fix this by adding the following after line 10:

```
t->s1 = cheri_bounds_set(t->s1, sizeof(S1)*n1);
```

and similar code after line 14.

We have developed a checker to detect such cases and suggest bounds narrowing. The idea is to detect adjacent objects of unrelated types placed manually (i.e., not as fields or elements of a parent type) within a single allocation (e.g., buffer). In this case, we assume these objects are meant to belong to distinct suballocations and their bounds should not overlap.

This checker reports warnings when a capability for suballocation with wide bounds escapes to another context: is returned from a function, stored in the memory reachable from the outer context, or passed to a function.

6 AST checks

6.1 Subobject representability

When an address of a structure field is taken, the CHERI C/C++ compiler can narrow the bounds for the resulting capability so that it covers the memory range of this field alone. This feature is often referred to as *subobject hardening* or *subobject bounds* [16]. It improves spatial safety and can prevent vulnerabilities related to intraobject overflows (e.g., [15]). However, this behavior is turned off by default in CHERI Clang, as it can break certain coding patterns, for example, when an address of a field is intentionally used to access other fields or to obtain the pointer to the whole parent structure. Nevertheless, for the critical parts of software, like OS kernel, it is reasonable to enable subobject bounds by default when porting code to CHERI, and explicitly opt-out for individual structures when necessary using annotations provided by the CHERI C API.

As was discussed in Sect. 1.2, due to the floating-point style representation of capabilities, a capability pointing to a large (sub)object may not have precise bounds if its address or size is not aligned well enough for the precise bounds to be representable. If the compiler is responsible for placing an object in memory, it will respect this representability requirement by choosing a properly aligned address and inserting proper padding after the object so the allocation will have representable bounds that do not overlap with other objects. CHERI-aware memory allocators also take care of proper alignment and tail-padding when asked for a large allocation with nontrivial representability requirements. However, there is no such flexibility for the layout of the fields within a structure that is strictly defined by their types (and attributes). Therefore, they may not have representable bounds, and this can undermine subobject bounds safety guarantees for some structure types.

For example, consider the structure in Listing 5. On Arm Morello, field `buf` is 65,536 bytes long and requires 32-byte alignment to have representable bounds. The alignment of this structure is at least 16 (due to the third field). The offset

Listing 5 Structure field with unrepresentable bounds

```
struct S {
  int fd;
  // FIXME: field buf has unrepresentable bounds
  char buf[65536];
  char *rp;
  /* other fields */
} *s;
```

(and alignment) of `buf` is 4. Therefore the actual bounds of the capability `&s->buf` are imprecise and span from the start of the structure to the end of the third field, exposing the first field as well as the capability field `s->rp`, violating the principle of least privilege.

We have implemented an AST-checker that checks every data field `f` for every declared structure or class type and issues a warning if this field has unrepresentable bounds, assuming a proper alignment of a parent structure.

Let A be the alignment required for the object of size `sizeof(f)` to have precise bounds. Note that larger objects have stricter representability requirements, and a structure's size is not smaller than the size of any of its fields. Hence, while checking the field `f`, we can assume that the alignment and tail-padding of its parent structure satisfy A (for top-level objects, this is guaranteed by an implementation; for nested structures that violate this assumption, a separate warning for the parent field will be issued). Thus, it is enough to check that the offset of `f` within its parent structure and the distance from the start of `f` to the next field are multiple of A. This will ensure that `f`'s bounds do not overlap with other fields.

If this property is violated, the checker will report a warning, calculate the actual bounds and highlight the adjacent fields of `f` that get exposed with a capability for `f`. In case there are capabilities within exposed bytes (e.g., field `s->rp` on Listing 5), which is potentially a more serious issue, the checker will point this out by emitting a special note with this warning.

6.2 Insufficient pointer size check

We also implemented a simple AST-checker to detect the code pattern where the **`sizeof`** a pointer is explicitly checked but the case of 128-bit pointers is not considered. Usually, this leads to a default branch and an assumption of, say, 64 bits with consequent incorrect behavior.

7 Implementation and evaluation

7.1 Implementation

The *CheriBSD* operating system is an extension of FreeBSD that takes advantage of capability hardware to implement memory protection and software compartmentalization. It supports CHERI extensions to both RISC-V and AArch64 (Arm Morello) and has matured through several releases: we work with CheriBSD 23.11 [8, 18].

CheriBSD provides versions of third-party software in the *CheriBSD ports* collection, a fork of the FreeBSD ports collection with CHERI-related changes. It uses the FreeBSD concept of *compatibility layers* to allow programs compiled for different ABIs to coexist within one system: (i) *CheriABI* (also known as *pure-capability ABI*), which permits memory accesses only via capabilities, (ii) *hybrid ABI*, which supports capabilities as well as raw legacy pointers to facilitate incremental software adaptation, and (iii) *benchmark ABI*, a modification of CheriABI designed to work around performance limitations of the prototype Morello architecture (which are to be addressed by microarchitectural enhancements in a commercially deployed CHERI architecture) and is intended solely for the purpose of pure-capability performance overhead analysis [24]. The current CheriBSD ports collection has ~9K CheriABI (and benchmark ABI) packages and ~24K hybrid ABI packages [26].

We implemented our checkers within the Clang Static Analyzer (all source on GitHub [9]) and added this tool as a port to the CheriBSD ports collection. Our implementation is based on the Clang/LLVM forks with CHERI support for AArch64 (Morello) [3] and RISC-V [7]. Initial support for CHERI in CSA required just a few minor fixes related to the width of the symbolic pointer values. Furthermore, we had to support a few missing cases in the CSA core related to integer-pointer conversion for the corresponding checker, pointer arithmetic for the alignment and suballocation checkers. Additionally, we introduced some CHERI API support in the CSA. During the development of the discussed checkers, we found that the default pointer value analysis was sufficient for our purposes and did not encounter the need to track the bounds of capabilities and other metadata.

Building projects for CheriABI on Morello requires a Morello LLVM toolchain that can be installed with the `llvm-base` package. This package provides `cc`, `c++`, and similar utilities that should be used to invoke the compiler and tools with the correct flags to build the CheriABI binary for the current version of CheriBSD. CSA build interception tool `scan-build` uses a similar interposition scheme to run `clang` with analysis flags alongside the original compilation command. These two layers of wrapper scripts did not naturally work together: `llvm-base` utilities were adding ABI-specific flags to the compilation command *after* `scan-build` had already captured the original command and reused it for the analysis. As a result, while the original build was performed for the CheriABI, CSA analyzed the project as if it was built for the hybrid ABI. To address this issue, additional wrapper scripts, similar to those provided by `llvm-base`, were introduced to the CheriBSD package for CSA to make `scan-build` mimic the behavior of the original build on CheriBSD.

Table 1 Analyzed projects

Category	Projects	CSA debug statistics + `cloc`				Build		Build+Analysis		Slowdown	
		Lines	Blocks	Functions	Files	Real, s	User, s	Real, s	User, s	Real	User
databases	27	454,877	143,505	7172	872	507	461	4605	6383	9.1	13.8
lang	25	480,468	133,881	7498	824	259	331	3187	4582	12.3	13.8
sysutils	162	1,503,648	391,981	23,364	3421	1991	1735	7458	12,362	3.7	7.1
devel	166	2,547,970	857,525	46,705	5342	2568	3419	15,699	28,366	6.1	8.3
editors	28	447,811	161,800	7663	1029	275	265	2926	4470	10.7	16.9
Sum [Avg]	**408**	**5,434,774**	**1,688,692**	**92,402**	**11,488**	**5599**	**6212**	**33,874**	**56,162**	**[6.1]**	**[9.0]**

7.2 Evaluation on CheriBSD ports collection

For evaluation, we ran the analyzer on a selection of pure-capability packages from the CheriBSD ports collection. Almost all of these packages had been merely recompiled for CheriABI without extensive testing or necessarily even fixing all the CHERI compiler warnings.

We selected 408 pure-capability packages from the `databases`, `lang`, `sysutils`, `devel`, and `editors` categories, as we believe those might be not trivial to port (Table 1). We chose projects that are mainly written in C/C++ (containing at least 50% and 500 lines of C/C++ code) and were successfully captured for analysis with `scan-build`.

According to the `cloc` utility, this selection contains a total of 11.7 MLoC of C/C++ in 21.5K files. However, according to CSA internal statistics, only 11.5K C/C++ modules were analyzed. The main reason why some of the C/C++ files were not seen by the analyzer was that they were not involved in the build process for this particular build configuration (e.g., platform-specific code for other platforms, implementation of features disabled in the default build configurations, tests, etc.). Some of the compilations might have evaded the build interception (if the port does not respect ports build system configuration variables), and therefore, those modules were not included for the analysis either.

Across all 408 projects, 5.4 MLoC of C/C++ source code (not including headers) in 11.5K files in total were analyzed. C++ source code is used in 53 projects and constitutes about 7.8% LoC of the analyzed lines of code.

Project build and analysis for this evaluation were performed on the Morello evaluation board (CHERI-extended ARMv8-A processor, 4 cores, 16 GB RAM) [2, 20]. Both the original compiler toolchain and CSA only support hybrid mode and do not have any capabilities introduced to their implementations (i.e., they are conventional ARM executables). Thus, our goal is to evaluate the analysis time compared to build time, and, although conducted on Morello, the evaluation itself does not involve any runtime CHERI protection.

On average, build+analysis of a port is around six times slower than a project build alone. For every intercepted C/C++ compilation, CSA runs its own instance of `clang` to perform an analysis of this module. Therefore, projects with a higher ratio of C/C++ code experience a more significant slowdown. If the original build process does not utilize all available CPU cores, then the analysis runs in parallel with the build, making the real-time slowdown less noticeable. The user-time slowdown does not depend on the degree of parallelism of a build process and is 9 times on average.

Table 2 Warning density

Warning category	C code		C++ code	
	W	*D*	*W*	*D*
Capability Alignment	381	76.05	39	91.78
Capability Copy	120	23.95	1	2.35
Capabilities as Integers	373	74.45	4	9.41
Suballocation Bounds	396	79.04	14	32.95
Subobject Representability	62	12.38	10	23.53
Insufficient Pointer Size Check	2	0.40	0	0.00

Table 2 shows the number of warnings of each category for C and C++ source code. Columns *W* correspond to the total number of warnings, column *D* represent the density – average number of warnings of this category per 1 MLoC.

Although this codebase includes 7.8% of C++, and our tool does analyze that, in practice, most of the warnings are for C code, and the density of the warnings for the C++ code is significantly lower. None of the portability issues we check are C++-specific, and most occur in low-level code anyway – true C++ will likely avoid them.

7.3 Evaluation of warnings

The results of the warning assessment are presented in Table 3. The first two columns list warning types, column *Total* gives the overall number of warnings for the type, and the next column shows the absolute and relative number of the reviewed warnings for this type.

For the next column, *Unique*, multiple instances of the same code pattern among reviewed warnings spread across

Table 3 Evaluation of Warnings

#	Warning	Total	Reviewed	Unique	True	TPR 95% CI,%
	Capability Alignment (§2)					
W1	Cast increases required alignment to capability alignment	138	85 (61%)	35	30 (85%)	74–97
W2	Not capability-aligned pointer used as capability storage	13	13 (100%)	3	3 (100%)	–
W3	Not capability-aligned pointer stored as 'void*'	143	143 (100%)	47	10 (21%)	9–32
W4	Copying capabilities through underaligned memory	126	97 (76%)	52	28 (53%)	40–67
	Capability Copy (§3)					
W5	Tag-stripping copy of capability	121	121 (100%)	25	10 (40%)	20–59
W6	Part of capability value used in binary operator	0	0	0	0 –	–
	Capabilities as Integers (§4)					
W7	NULL-derived capability: loss of provenance	300	300 (100%)	27	26 (96%)	–
W8	NULL-derived capability used as pointer	9	9 (100%)	8	6 (75%)	–
W9	CHERI-incompatible pointer arithmetic	0	0	0	0 –	–
W10	Capability derived from wrong argument	17	17 (100%)	3	3 (100%)	–
W11	Binary operation with ambiguous provenance	51	51 (100%)	22	22 (100%)	–
	Suballocation Bounds (§5)					
W12	Allocation partitioning	56	56 (100%)	26	15 (57%)	38–76
W13	Unknown allocation partitioning	354	63 (17%)	42	32 (76%)	63–89
	Subobject Representability (§6.1)					
W14	Field with imprecise subobject bounds (offset)	63	42 (66%)	33	33 (100%)	90–100
W15	Field with imprecise subobject bounds (length)	9	9 (100%)	9	9 (100%)	–
	Insufficient Pointer Size Check (§6.2)					
W16	Only a limited number of pointer sizes checked	2	2 (100%)	2	2 (100%)	–

the project code were considered duplicates and counted as a single instance. Column *True* gives the absolute and relative number of unique true warnings. Finally, the last column gives a 95% confidence interval for the True Positive Rate (TPR) where there are enough unique warnings of this type to calculate it. The overall TPR across all warnings is 69%, with a 95% confidence interval ±5% around that.

7.3.1 Capability alignment

Warnings of the first group (W1–W4) were produced by the Capability Alignment checker. The most reliable warning type from this group (85% TPR) is the one related to alignment-increasing casts – W1. The checker produces warnings for all pointer types, but for this evaluation, we included reports related to capability alignment only. Warnings reported for the tracked allocations (*Typed Regions*), perhaps not surprisingly, turned out to be more reliable than those reported for the *Symbolic Regions* associated with the top-level symbolic pointers (e.g., global variables or top-level function arguments). In the latter case, a pointer variable with not capability-aligned pointer type (other than explicitly ignored `char*` and `void*`) is used to pass a pointer to capability-aligned type, and the consistency of the alignment may be guaranteed by the path conditions that the analyzer has failed to track precisely. Such cases account for most of the false positive warnings of this type.

Reports for storing a noncapability-aligned pointer as a capability pointer are split into two warning types: the case when the analyzer is sure that the memory pointed by the underaligned pointer is or will be used to store capabilities (W2) and the case when the analyzer just suspects that it may be used for this purpose, basing on its generic type (W3). The former is more reliable: it produced only three warnings, but all of them were true issues. The latter has a significantly lower True Positive Rate (21%), owing to the quite unreliable assumption that any top-level `void*` variable may be intended to hold a pointer to capability. However, this heuristic can identify alignment issues in cases that are hard to detect otherwise: for example, when the possibility of storing pointers at this address is implied but never exercised in the analyzed code.

Finally, the last type, W4, is for copying capabilities through underaligned memory. It showed an acceptable 53%

True Positive Rate since it relies on the same assumption on the `void*` variables. Reports of this type are often fired simultaneously with the warnings for assignments (W2/W3), as these two events are often adjacent.

7.3.2 Capability copy

True warnings related to the tag-stripping capability copying (type W5) were mainly fired for the custom implementations of `qsort` and `memcpy`-like functions, as expected. False positive reports were issued for the functions that accept `void*` arguments and perform partial copying but are not intended to be used with capabilities. As with the previous checker, using an assumption on `void*` arguments led to a slightly underwhelming 40% True Positive Rate but made it possible to identify some hard-to-uncover issues.

In the selected codebase, there were no reports of a hash value calculation of a capability representation (type W6). However, we have encountered very few such warnings in other projects, and they were all true issues.

7.3.3 Capabilities as integers

Pointer vs. integer conversion is a pattern specific only to particular projects. Typically, there is either no warning of this type in the project, or many of them reported for the same idiom. For example, for the accidental loss of provenance due to the round-trip conversion via noncapability integer type (W7), we encountered 300 reported warnings but only 27 unique warning patterns, the vast majority of which correspond to real issues. For a similar pattern, when the analyzer failed to track the origin of an integer converted to a pointer – W8, we had an extra 8 unique warnings and slightly lower TPR (75%).

The rest of the warning types in this group, W9–W11, are related to the binary arithmetic on `intptr_t` values. We have not seen any warning for the CHERI-incompatible arithmetic with capabilities as both arguments (W9) in the evaluation codebase. There were three unique patterns of binary arithmetic with pointers on the RHS that will break with the default provenance derivation from the LHS (W10), all of them were true. The last warning type from this group, W11, also produced all true warnings, but they can be safely ignored with the default compiler behavior.

7.3.4 Suballocation bounds

W12 and W13 warning types are reported when a suballocation pattern is detected for a tracked (static, automatic, or heap) allocation or an allocation with an unknown origin, respectively. We classified these reports as true issues when narrowing the bounds for the suballocation would be reasonable in the current context and would not break the code. Common patterns of false positives encountered were cases when narrowing the bounds before exposing the suballocation capability would be excessive (e.g., zeroing a suballocation with `memset`); cases where the adjacent suballocations are supposed to be reachable from the current suballocation (e.g., allocation metadata residing before the allocated memory).

7.3.5 Subobject representability

We have encountered 63 warnings for structure or class data fields with unrepresentable bounds caused by unaligned offsets (W14) and 9 reports for the same issue caused by the unaligned size of the field itself (and the lack of tail-padding after the field) (W15). The largest structure field reported was 10 Mb in size and required 4 Kb alignment and 4 Kb tail-padding for precise bounds on Morello. As this is a simple AST check, it does not produce false alarms.

7.3.6 Insufficient pointer size check

For the pattern with insufficient pointer size check, W16, there were two matches, and both were true alarms.

8 Conclusion

Our contributions are the following:

- Four capability-specific CSA checkers for detecting CHERI-compatibility issues: for capability alignment (§2), tag-stripping capability copying (§3), pointer vs. integer conversions (§4), and insufficient pointer size checks (§6.2);
- Two CSA checkers for capability-related memory safety enhancements: suballocation bounds (§5) and subobject representability (§6.1);
- Implementation and evaluation of these checkers against a body of samples from the CheriBSD ports library (§7).

Among the CHERI-compatibility checkers, we believe that the capability alignment checker is the most useful: the problems it finds are numerous; they are not reported especially well by a compiler; and are often hard to investigate when they cause a run-time fault.

Detectors from the second group produce suggestions for enforcing least privilege by carefully tightening the capability bounds. These reports turned out to be sufficiently reliable from our perspective, as they correctly identify code patterns that we believe might benefit from bounds tightening. However, whether the potential security benefit is worth the effort of introducing CHERI-specific changes in each case depends on the program logic and should be decided by the code maintainers.

Funding information 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).

For the purpose of Open Access, a CC-BY public copyright licence has been applied by the authors to the present document and will be applied to all subsequent versions up to the Author Accepted Manuscript arising from this submission.

References

1. Arm Ltd: Arm Architecture Reference Manual Supplement – Morello for A-profile Architecture (2019–2022). https://developer.arm.com/documentation/ddi0606
2. Arm Ltd: Arm Morello program (2019–2024). https://www.arm.com/architecture/cpu/morello
3. Arm Ltd: LLVM Toolchain for Morello (2020). https://git.morello-project.org/morello/llvm-project-releases
4. Arm Ltd: CHERIseed – port effortlessly to CHERI. https://www.morello-project.org/resources/cheriseed-port-effortlessly-to-cheri (2022). Accessed 2 May 2024
5. Bramley, J., Jacob, D., Lascu, A., Singer, J., Tratt, L.: Picking a CHERI allocator: security and performance considerations. In: ISMM 2023: Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management, pp. 111–123. Association for Computing Machinery, New York (2023). https://doi.org/10.1145/3591195.3595278
6. Brauße, F., Shmarov, F., Menezes, R., Gadelha, M.R., Korovin, K., Reger, G., Cordeiro, L.C.: ESBMC-CHERI: towards verification of C programs for CHERI platforms with ESBMC. In: ISSTA 2022: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 773–776. Association for Computing Machinery, New York (2022). https://doi.org/10.1145/3533767.3543289
7. CTSRD-CHERI: The CHERI LLVM compiler infrastructure (2019). https://github.com/ctsrd-cheri/llvm-project
8. Davis, B., Watson, R.N.M., Richardson, A., Neumann, P.G., Moore, S.W., Baldwin, J., Chisnall, D., Clarke, J., Filardo, N.W., Gudka, K., Joannou, A., Laurie, B., Markettos, A.T., Maste, J.E., Mazzinghi, A., Napierala, E.T., Norton, R.M., Roe, M., Sewell, P., Son, S., Woodruff, J.: CheriABI: enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment. In: ASPLOS'19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 379–393. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3297858.3304042
9. Dudina, I.: CHERI CSA (2024). https://github.com/rems-project/llvm-project
10. Gokhale, S., Turcotte, A., Tip, F.: Automatic migration from synchronous to asynchronous JavaScript APIs. Proc. ACM Program. Lang. **5**(OOPSLA) (2021). https://doi.org/10.1145/3485537
11. Gutstein, B.: Memory safety with CHERI capabilities: Security analysis, language interpreters, and heap temporal safety. Tech. Rep. UCAM-CL-TR-975, University of Cambridge Computer Laboratory (2022). https://doi.org/10.48456/tr-975
12. LLVM Project: Clang static analyzer (2016). https://clang-analyzer.llvm.org/
13. Lowther, D., Jacob, D., Singer, J.: Morello MicroPython: a python interpreter for CHERI. In: MLPR 2023: Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, pp. 62–69. Association for Computing Machinery, New York (2023). https://doi.org/10.1145/3617651.3622991
14. Memarian, K.: The Cerberus C semantics. Tech. Rep. UCAM-CL-TR-981, University of Cambridge Computer Laboratory (2023). https://doi.org/10.48456/tr-981
15. Ormandy, T.: This shouldn't have happened: a vulnerability postmortem. https://googleprojectzero.blogspot.com/2021/12/this-shouldnt-have-happened.html (2021). Accessed 9 September 2024
16. Richardson, A.: Complete spatial safety for C and C++ using CHERI capabilities. Tech. Rep. UCAM-CL-TR-949, University of Cambridge Computer Laboratory (2020). https://doi.org/10.48456/tr-949
17. Saltzer, J.H.: Protection and the control of information sharing in multics. Commun. ACM **17**(7), 388–402 (1974). https://doi.org/10.1145/361011.361067
18. SRI International and the University of Cambridge: CheriBSD 23.11 release notes (2023). https://www.cheribsd.org/release-notes/23.11
19. Wang, K., Kasatkin, D., Ahlrichs, V., Auer, L., Hohentanner, K., Horsch, J., Ekberg, J.E.: Cherifying Linux: a practical view on using CHERI. In: EuroSec'24: Proceedings of the 17th European Workshop on Systems Security, pp. 15–21. Association for Computing Machinery, New York (2024). https://doi.org/10.1145/3642974.3652282
20. Watson, R.: The Arm Morello board. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-morello.html (2019). Accessed 12 September 2024
21. Watson, R.N.M., Moore, S.W., Sewell, P., Neumann, P.G.: An introduction to CHERI. Tech. Rep. UCAM-CL-TR-941, University of Cambridge Computer Laboratory (2019). https://doi.org/10.48456/tr-941
22. Watson, R.N.M., Richardson, A., Davis, B., Baldwin, J., Chisnall, D., Clarke, J., Filardo, N., Moore, S.W., Napierala, E., Sewell, P., Neumann, P.G.: CHERI C/C++ programming guide. Tech. Rep. UCAM-CL-TR-947, University of Cambridge Computer Laboratory (2020). https://doi.org/10.48456/tr-947
23. Watson, R.N.M., Laurie, B., Richardson, A.: Assessing the viability of an open-source CHERI desktop software ecosystem. Technical report, Capabilities Limited (2021) https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/20210917-capltd-cheri-desktop-report-version1-FINAL.pdf
24. Watson, R.N.M., Clarke, J., Sewell, P., Woodruff, J., Moore, S.W., Barnes, G., Grisenthwaite, R., Stacer, K., Baranga, S., Richardson, A.: Early performance results from the prototype Morello microarchitecture. Tech. Rep. UCAM-CL-TR-986, University of Cambridge Computer Laboratory (2023). https://doi.org/10.48456/tr-986
25. Watson, R.N.M., Neumann, P.G., Woodruff, J., Roe, M., Almatary, H., Anderson, J., Baldwin, J., Barnes, G., Chisnall, D., Clarke, J., Davis, B., Eisen, L., Filardo, N.W., Fuchs, F.A., Grisenthwaite, R., Joannou, A., Laurie, B., Markettos, A.T., Moore, S.W., Murdoch, S.J., Nienhuis, K., Norton, R., Richardson, A., Rugg, P., Sewell, P., Son, S., Xia, H.: Capability hardware enhanced RISC instructions: CHERI instruction-set architecture (version 9). Tech. Rep. UCAM-CL-TR-987, University of Cambridge Computer Laboratory (2023). https://doi.org/10.48456/tr-987

26. Witaszczyk, K.: CheriBSD ports and packages: Pure-capability third-party software for Arm Morello and CHERI-RISC-V CheriBSD. FreeBSD J., 12–22 (2023). https://freebsdfoundation.org/wp-content/uploads/2023/05/CheriBSD_ports.pdf
27. Woodruff, J., Joannou, A., Xia, H., Fox, A., Norton, R.M., Chisnall, D., Davis, B., Gudka, K., Filardo, N.W., Markettos, A.T., Roe, M., Neumann, P.G., Watson, R.N.M., Moore, S.W.: CHERI concentrate: practical compressed capabilities. IEEE Trans. Comput. **68**(10), 1455–1469 (2019). https://doi.org/10.1109/TC.2019.2914037
28. Xu, Z., Kremenek, T., Zhang, J.: A memory model for static analysis of C programs. In: ISoLA 2010: Leveraging Applications of Formal Methods, Verification, and Validation. LNCS, vol. 6415, pp. 535–548. Springer, Berlin (2010). https://doi.org/10.1007/978-3-642-16558-0_44
29. Xu, S., Dong, Z., Meng, N.: Meditor: inference and application of API migration edits. In: ICPC'19: Proceedings of the 27th International Conference on Program Comprehension, pp. 335–346. IEEE Press, New York (2019). https://doi.org/10.1109/ICPC.2019.00052
30. Zaliva, V., Memarian, K., Almeida, R., Clarke, J., Davis, B., Richardson, A., Chisnall, D., Campbell, B., Stark, I., Watson, R.N.M., Sewell, P.: Formal mechanised semantics of CHERI C: capabilities, undefined behaviour, and provenance. In: ASPLOS'24: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 1, pp. 181–196. Association for Computing Machinery, New York (2024). https://doi.org/10.1145/3617232.3624859