Picking a CHERI Allocator: Security and Performance Considerations

Jacob Bramley
Arm Limited
Cambridge, United Kingdom

Dejice Jacob
University of Glasgow
Glasgow, United Kingdom

Andrei Lascu
King's College London
London, United Kingdom

Jeremy Singer
University of Glasgow
Glasgow, United Kingdom

Laurence Tratt
King's College London
London, United Kingdom

Abstract

Several open-source memory allocators have been ported to CHERI, a hardware capability platform. In this paper we examine the security and performance of these allocators when run under CheriBSD on Arm's prototype Morello platform. We introduce a number of security attacks and show that all but one allocator are vulnerable to some of the attacks — including the default CheriBSD allocator. We then show that while some forms of allocator performance are meaningful, comparing the performance of hybrid and pure capability (i.e. 'running in non-CHERI vs. running in CHERI modes') allocators does not currently appear to be meaningful. Although we do not fully understand the reasons for this, it seems to be at least as much due to factors such as immature compiler toolchains and prototype hardware as it is due to the effects of capabilities on performance.

CCS Concepts: • **Security and privacy** → Virtualization and security; • **General and reference** → **Performance**; • **Software and its engineering** → **Virtual memory**; *Access protection*; *Empirical software validation.*

Keywords: memory allocators, capabilities, CHERI, software implementation, validation

ACM Reference Format:
Jacob Bramley, Dejice Jacob, Andrei Lascu, Jeremy Singer, and Laurence Tratt. 2023. Picking a CHERI Allocator: Security and Performance Considerations. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management (ISMM '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3591195.3595278

Authors' URLs: D. Jacob [https://www.dcs.gla.ac.uk/~jacobd/], A. Lascu [https://andreil.eu/], J. Singer [https://www.dcs.gla.ac.uk/~jsinger/], L. Tratt [https://tratt.net/laurie/].

ISMM '23, June 18, 2023, Orlando, FL, USA

1 Introduction

CHERI (Capability Hardware Enhanced RISC Instructions) provides and enforces hardware capabilities that allow programmers to make strong security guarantees about the memory safety properties of their programs [14]. However, capabilities are not magic — programmers must first decide which memory safety properties they wish to enforce and then write their software in such a way to enforce those properties. Mistakes or oversights undermine the security guarantees that programmers believe their code possesses.

In this paper we study memory allocators (henceforth just "allocators") in the context of CHERI. Apart from some embedded systems (which preallocate fixed quantities of memory), allocators are ubiquitous, because they allow us to write programs that are generic over a variety of memory usage patterns. Allocators' security properties and performance are a fundamental pillar supporting the security properties and performance of software in general — any security flaws and/or performance problems in allocators thus have significant, widespread, consequences.

In this paper we show that most CHERI allocators are subject to surprisingly simple attacks. We then show that while some aspects of allocator performance can be meaningfully compared, it does not currently appear to be meaningful to compare the performance of hybrid and pure capability (roughly speaking: "running in non-CHERI vs. running in CHERI modes") allocators. We analyse some of the likely factors for this latter case, which suggest that this may be at least as much due to factors such as immature compiler toolchains and prototype hardware as to the effects of capabilities on performance. We do not claim that our work is definitive, though it does suggest two things: that some allocators undermine the security properties one might reasonably expect from software running on pure capability CHERI; and that it is currently difficult to reason about the performance impact of CHERI on software. Our experiments are repeatable and can be downloaded from https://archive.org/details/cheri_allocator_ismm.

This paper is structured as follows. First, we introduce the necessary background: a brief overview of capabilities and CHERI (Section 2); and our running example, a trivial bump allocator (Section 3). We then introduce our study:

the allocators under consideration (Section 4); our attacks (Section 5); a partial performance evaluation (Section 6) and an analysis of some of the performance discrepancies we uncovered (Section 7).

2 CHERI Overview

In this section, we provide a simple overview of CHERI, and its major concepts. Since CHERI has developed over a number of years, and is explained across a range of documentation and papers, some concepts have acquired more than one name, or names that subtly conflict with mainstream software development definitions. We use a single name for each concept, sometimes introducing new names where we believe that makes things clearer.

A *capability* is a token that gives those who bear it *abilities* to perform certain actions. By restricting who has access to a given capability, one can enforce security properties (e.g. "this part of the software can only read from memory address range X to Y"). Capabilities have a long history: [7] provides a narrative overview of capability architectures, and may usefully be augmented by more recent work such as [11]. A good first intuition is that CHERI is a modern version of this longstanding idea, with finer-grained permissions, and adapted to work on recent processor instruction sets.

We use the term *CHERI* to refer to the 'abstract capability machine' that software can observe: that is, the combination of a capability hardware instruction set, an ABI (roughly speaking, the interface between userland and kernel e.g. [2]), a user-facing library that exposes capability-related functions, and a CHERI-aware language. (e.g. CHERI C [15], an adaption, in the sense of both extending and occasionally altering, of C). Except where we use the name of a different hardware implementation (e.g. CHERI RISC-V), we assume the use of Arm's 'Morello' hardware, which is a prototype ARMv8 chip extended with CHERI instructions.

Conceptually, a CHERI system starts with a 'root' capability that has the maximum set of abilities. Each new *child* capability must be derived from one or more *parent* capabilities. A child capability must have the same, or fewer, abilities than its parent: put another way, capabilities' abilities monotonically decrease. An *authentic*[1] capability is one that has been derived from authentic parents according to CHERI's rules. Attempts to create capabilities that violate CHERI's rules cause the hardware to produce an *inauthentic* result, guaranteeing that capabilities cannot be forged. On Morello and CHERI RISC-V, capabilities behave as if they are 128 bits in size, but also carry an additional (129th) bit that records the authenticity of each capability. Software can read, and unset, the authenticity bit, but cannot set it: only a child capability derived, correctly, from authentic parent capabilities can itself be authentic.

[1]CHERI calls these 'tagged' or 'valid' (and their inauthentic counterparts 'untagged' or 'invalid').

A capability consists of a memory *address*[2], and its abilities: a set of *permissions* (only a subset of which we consider in this paper); and *bounds*, the memory range on which the capability can operate.

Permissions include the ability to read / write from / to memory. A *permissions check* is said to be successful if the permission required for a given operation is provided by a given capability.

A capability's bounds are from a *low* (inclusive) to a *high* (exclusive) address: when we refer to a capability's bounds being of 'x' bytes we mean that $high - low = x$. An address is *in-bounds* for a given capability if it is contained within the capability's bounds, or *out-of-bounds* otherwise; a capability is in (or out) of bounds if its address is in (or out) of bounds[3]. A *bounds check* is said to be successful if a given capability, address, or address range, is in-bounds for a given capability.

A processor instruction that operates on a capability requires at least one of: an authenticity check, a permissions check, or a bounds check. If a capability fails the relevant checks, then either: the hardware produces a `SIGPROT` exception (similar to `SIGSEGV`) that terminates the program; or produces an inauthentic capability (where this is not in violation of the CHERI rules).

CHERI allows both double-width capabilities and single-width addresses-as-pointers to exist alongside each other at any time. Conventionally, a program which uses both traditional addresses and capabilities is said to be operating in *hybrid* mode while a program which uses only capabilities is in *pure capability* – henceforth "purecap" – mode. Some caution is necessary with these terms, because different parts of a system may be hybrid or purecap. For example, kernels often use hybrid mode while some parts of userland may use purecap: at least one of the sides in such a relationship must translate between the hybrid and purecap worlds as necessary. In this paper we make two simplifying assumptions: that we only have to consider userland, and that userland is entirely purecap or entirely hybrid.

CHERI does not presuppose a particular Operating System (OS). While there is a CHERI Linux port, at the time of writing the most mature OS for CHERI hardware is CheriBSD, a FreeBSD descendent. In this paper we use CheriBSD.

3 A Basic Pure Capability Allocator

To illustrate how CHERI affects allocators, in this section we adapt a simple non-CHERI aware allocator to become CHERI aware. For simplicity's sake, we assume that this means at least running successfully on, and preferably taking advantage of, purecap CHERI.

[2]This portion of a capability does not *have* to store an address, though typically it does so, and the CHERI API calls it `address`. In the context of this paper, since it always stores an address, we stick with this name.

[3][16] shows why authentic capabilities can have an out-of-bounds address.

```
#include <string.h>
#include <sys/mman.h>

char *heap = NULL;
char *heap_start = NULL;
size_t HEAP_SIZE = 0x1000000000;

void *malloc_init() {
  heap = heap_start = mmap(NULL, HEAP_SIZE,
    PROT_READ | PROT_WRITE,
    MAP_PRIVATE | MAP_ANON, -1, 0);
  return heap;
}

void *malloc(size_t size) {
  if (heap == NULL && !malloc_init())
    return NULL;
  size = __builtin_align_up(size,
    _Alignof(max_align_t));
  if (heap + size >
      heap_start + HEAP_SIZE)
    return NULL;
  heap += size;
  return heap - size;
}

void free(void *ptr) { }

void *realloc(void *ptr, size_t size) {
  void *new_ptr = malloc(size);
  if (new_ptr == NULL) return NULL;
  if (ptr) memcpy(new_ptr, ptr, size);
  return new_ptr;
}
```

Listing 1. A simple, but complete, non-CHERI aware, bump pointer allocator: `malloc` works as per normal; `free` is a no-op; and `realloc` always allocates a new chunk, copying over the old block. `__builtin_align_up(v, a)` is an LLVM / clang primitive which rounds `v` up to the next smallest multiple of `a`; `_Alignof(max_align_t)` returns an alignment sufficiently large for any scalar type (i.e. integers and pointers).

```
1  void *malloc(size_t size) {
2    if (heap == NULL && !malloc_init())
3      return NULL;
4
5    char *new_ptr = __builtin_align_up(
6      heap,
7      -cheri_representable_alignment_mask(
8        size));
9    size_t bounds =
10     cheri_representable_length(size);
11   size_t size_on_heap =
12     __builtin_align_up(
13       size, _Alignof(max_align_t));
14
15   if (new_ptr + size_on_heap >
16       heap_start + HEAP_SIZE)
17     return NULL;
18   heap = new_ptr + size_on_heap;
19   return cheri_bounds_set_exact(
20     new_ptr, bounds);
21 }
22
23 void *realloc(void *ptr, size_t size) {
24   void *new_ptr = malloc(size);
25   if (new_ptr == NULL) return NULL;
26   if (ptr)
27     memcpy(new_ptr, ptr,
28       cheri_length_get(ptr) < size
29       ? cheri_length_get(ptr) : size);
30   return new_ptr;
31 }
```

Listing 2. Replacing the non-CHERI aware `malloc` from Listing 1 with a CHERI-aware alternative using the idioms suggested in [15, p. 30]. This `malloc` returns a capability whose bounds are sufficient to cover `size` bytes starting at the capability's address (calculated in lines 5–10), such that two callers to `malloc` cannot read or write from another block. We also have to update `realloc` so that it never tries to copy more data from the old block than the `ptr` capability gives it access to.

Listing 1 shows a simple, complete, example of a C bump allocator: `malloc` works as per normal; `free` is a no-op; and `realloc` always allocates a new chunk of memory. The allocator reserves a large chunk of memory using a single `mmap` call then doles out chunks on each `malloc` / `realloc` calls. The bump pointer moves through the `mmap`ed chunk until it reaches the upper limit, at which point the allocator returns `NULL`. Though intentionally simplistic, `realloc` is correct even when the block is increased in size.

Adapting the Allocator to CHERI: Perhaps surprisingly, our simple bump allocator compiles, and `malloc` runs correctly, on a purecap CHERI system. As this suggests, CHERI C is largely source compatible with normal C code, though pointer types are transparently 'upgraded' to become *capability types* (on Morello occupying exactly twice the space of a non-capability pointer). CHERI also implies changes in libraries: on CheriBSD, for example, `mmap` returns a capability whose bounds are at least those of the size requested: from that capability our bump allocator derives new capabilities that differ in their address but not their bounds. In other words, calling `malloc` just once gives the caller the ability to read and write from all past and future blocks returned by `malloc`!

As this suggests, using CHERI without careful thought may give no additional security benefits. This then raises the question: how should a secure 'CHERI aware' allocator behave? There can be no single answer to this question, but we believe that most programmers would at least expect

`malloc` to return a capability whose bounds are restricted to the block of memory allocated. Listing 2 shows how to adapt `malloc` to do this.

The code to create the capability (using the idioms suggested in [15, p. 30]) is more involved than one might first expect. The underlying cause is that there aren't, and cannot reasonably be, enough bits in CHERI's bounds to precisely represent every possible address and size. Modern CHERI therefore uses an encoding for bounds that allows small bounds to be precisely represented, at the expense of larger bounds becoming progressively less precise [16]. On Morello, the smallest bound that cannot be precisely represented is 16,385 bytes, which is rounded up to 16,392 bytes[4]. Our capability aware `malloc` thus has to ensure that both the capability's low and high bound addresses are rounded down and up (respectively) in a way that ensures that the address and size can be fully covered.

The two versions of our allocator have meaningfully different security properties, even when we run both on a purecap system. For example, consider this simple C snippet which models a buffer overrun:

```
1 char *b = malloc(1);
2 b[0] = 'a';
3 b[1] = 'b';
```

On a non-CHERI system, or a CHERI system with Listing 1 as an allocator, this snippet compiles and runs without error. However, using the allocator from Listing 2 on a purecap CHERI system, while the snippet compiles, the tighter capability bounds returned by `malloc` cause a `SIGPROT` when executing line 3. This demonstrates how CHERI can prevent programmer errors becoming security violations.

However, just because a program compiles with CHERI C does not guarantee that it will run without issue: when run on CHERI, the `realloc` in Listing 1 causes a `SIGPROT` when asked to increase the size of a block. This occurs because `memcpy` tries to copy beyond the bounds of the input capability (e.g. if the existing block is 8 bytes and we ask to resize it to 16 bytes, `memcpy` tries to read 16 bytes from a capability whose bounds are 8 bytes). On a non-CHERI system, this is not a security violation, but it is treated as one on CHERI. In Listing 2 we thus provide an updated `realloc` which uses `cheri_length_get` (which returns a capability's bounds in bytes) to ensure that it never copies more data than the input capability's bounds allow.

4 CHERI Allocators

In this paper we consider a number of allocators that are available for CheriBSD. We first explain the set of allocators we use, before exploring in more detail how the allocators have been adapted (if at all) for CHERI.

[4]For CHERI RISC-V the first unrepresentable length is 4,097 bytes, which is rounded up to 4,104.

Table 1. The allocators we examined, their size in Source Lines of Code (SLoC), and the number of lines changed (as an absolute value and relative percentage) to adapt them for purecap CheriBSD. The top portion of the table shows the allocators which passed a basic test and are used in our experiments; the bottom portion shows the allocators which failed a basic test.

Allocator	Version	SLoC	Changed	
			LoC	%
bump-alloc	21cb5f38	61	31	50.81
dlmalloc-cheribuild	9cfbb169	3 475	231	6.65
jemalloc	cc4e4c05	28 755	116	0.40
libmalloc-simple	62175107	408	43	10.54
snmalloc-cheribuild	888d182b	14 669	180	1.23
snmalloc-repo	0a5eb403	21 342	212	0.99
dlmalloc-pkg64c	2.8.6	-	-	-
ptmalloc	3.0_2	-	-	-

4.1 The Allocators under Consideration

A number of allocators are available for purecap CheriBSD, installable via three different routes: as part of the base distribution; via CheriBSD *packages*; or via external sources. We examined allocators available from all three sources. We excluded allocators aimed at debugging (e.g. *ElectricFence*). We then ran a simple validation test, `malloc`ing a block of memory, copying data into the block, and then `free`ing the block: we excluded any allocator which failed this test.

On that basis, the allocators we consider in this paper, and the names we use for them for the rest of this paper, are as follows:

jemalloc, a modified version of the well-known allocator [3]: this is the default allocator for CheriBSD.

libmalloc-simple[5], a port of the allocator in the FreeBSD utility `rtld-elf`[6], based on Kingsley's malloc from 4.2BSD.

snmalloc-cheribuild, a version of *snmalloc* [8] that can be installed via `cheribuild`. We found this version to have several problems which we rectified by manually building a newer version from *snmalloc*'s GitHub repository. We term this more recent version `snmalloc-repo`.

dlmalloc-cheribuild, a modified version of the well-known allocator [5], installable via `cheribuild`. *dlmalloc-pkg64c* is an unmodified version of the allocator, available as a

[5]https://github.com/CTSRD-CHERI/cheribsd/commit/e85ccde6d78d40f130ebf126a001589d75d60473, accessed 23rd February 2023

[6]https://github.com/freebsd/freebsd-src/blob/releng/4.3/libexec/rtld-elf/malloc.c, accessed 23rd of February 2023

package in CheriBSD. Both versions are based on dlmalloc 2.8.6.

ptmalloc [4] is an extension of *dlmalloc*, with added support for multiple threads.

bump-alloc-nocheri is the simple, non-CHERI-aware bump allocator from Listing 1. Conversely, the CHERI aware version is *bump-alloc-cheri*, presented in Listing 2.

Table 1 shows the version of each allocator we used. We have not included two other major memory allocators that have only been partly ported to CHERI: the *Boehm-Demers-Weiser* conservative garbage collector; and the *WebKit* garbage collector.

4.2 How Much Have the Allocators Been Adapted for CHERI?

As we saw from Listing 1, simple allocators may not need adapting for CHERI, though they are then likely to derive only minor security gains. In practice, we expect most allocators to incorporate at least the capability bounds enforcement of Listing 2. Indeed, more sophisticated allocators tend to crash on CHERI without at least some modifications. For example, most of the allocators available via CheriBSD's package installer (e.g. dlmalloc-pkg64c) have had no source-level changes for CHERI: they compile correctly but crash on even the most trivial examples.

Understanding the details of the CHERI modifications to all of the allocators under consideration is beyond the scope of this work. Instead, Table 1 shows what proportion of an allocator's LoC are 'CHERI specific' by calculating the percentage of lines of code contained between `#ifdef CHERI` blocks and similarly guarded code. This count is an under-approximation, as some code outside such `#ifdef` blocks may also have been adapted, but it gives a rough idea of the extent of changes.

With the exception of the extremely small `bump-alloc` and `libmalloc-simple`, the pure capability memory manager libraries in Table 1 have a mean 2.31% of their SLoC changed. Although this is a relatively small portion, it is two orders of magnitude larger than the 0.026% lines that were adapted when porting a desktop environment (including X11 and KDE) [13]. It is a reasonable assumption that the lower-level, and more platform dependent, nature of allocators requires more LoC to be adapted.

5 The Attacks

Our definition of CHERI in Section 2 might suggest that software running on CHERI hardware is invulnerable to attack. Rather, CHERI gives us the tools to make secure software, but it is up to us to use them correctly — and wisely. We must decide which attack model is relevant to our use-case, and then write, or adjust, the software, to withstand such attacks. In our context, allocators are subject to spatial (e.g. buffer overrun) or temporal (e.g. a sequence of function calls) attacks, and those attacks can either target an allocators' internals (e.g. corrupting private data-structures) or its interface (e.g. allowing user code to bypass security checks).

Table 2. Attacks per allocator: × indicates that an allocator is vulnerable to an attack; ✓ that the allocator is invulnerable; and ⊘ a failure for other reasons (e.g. a segfault).

Allocator	EscInauth	EscPrms	NrwWide	Overlap	Undef
bump-alloc-cheri	✓	×	✓	✓	✓
bump-alloc-nocheri	✓	⊘	✓	×	✓
dlmalloc-cheribuild	✓	×	×	✓	✓
jemalloc	✓	×	×	✓	×
libmalloc-simple	✓	✓	×	✓	×
snmalloc-cheribuild	✓	✓	✓	⊘	✓
snmalloc-repo	✓	✓	✓	✓	✓

In this section we introduce a number of simple 'attacks' on CHERI allocators (4 temporal and 1 spatial) and then run those attacks on the allocators from Section 4, with the results shown in Table 2. Even the default CheriBSD allocator is vulnerable to some attacks: only snmalloc is invulnerable.

In the rest of this section we explain each attack, giving C code using the CHERI API. Our code examples assume that we start with a 'non-attacker' who allocates memory and hands it over to another part of the system which has been taken over by an ' attacker ' (whose code has a light grey background). For each (allocator, attack) pair, we state whether it is vulnerable, invulnerable, or whether the attack fails for other reasons. We model this via a series of `assert`s: if all the `assert`s pass, the attack is successful. The code we show in the paper is elided relative to the version we run, which contains changes that makes it possible for us to automate the running of the attacks over multiple allocators. The full code (available as part of our experiment) must be considered the definitive source of truth for Section 4.

Our descriptions use the following CHERI functions:

`void *cheri_address_set(void *c, ptraddr_t a)` Takes a capability `c` as input and returns a capability that is a copy of `c` except with the address `a`. `ptraddr_t` is a CHERI C integer type that is guaranteed to be big enough to represent addresses but, unlike `intptr_t`, is not big enough to represent capabilities.

`ptraddr_t cheri_base_get(void *c)` Returns the address of the lower bound of a capability `c`.

`void *cheri_bounds_set(void *c, size_t s)` Takes a capability `c` as input and returns a new capability that is a copy of `c` except with bounds `s`.

`size_t cheri_length_get(void *c)` Returns the bounds of a capability `c`.

`_Bool cheri_tag_get(void *c)`
Returns true if the capability `c` is authentic.

`void* cheri_perms_and(void *c, size_t perms)` Returns the capability `c` with its permissions bitwise-ANDed with `perms`.

`size_t cheri_perms_get(void *c)`
Returns the permissions of capability `c`.

5.1 NarrowWiden: Narrowing then Widening Can Allow Access to Hidden Data

In the simple bump allocator of Listing 1, `realloc` always allocates a new block of memory. While this is always correct, it is inefficient, in part because it requires copying part of the block's existing content. Most allocators thus try to avoid allocating a new block of memory if: the new size is the same as, or smaller than, the existing size; or if the new size would not lead to the block overwriting its nearest neighbour. The latter optimisation is dangerous for a CHERI allocator.

Consider the case where `realloc` wants to increase a block in size, and there is sufficient room to do so without moving the block. `realloc` needs to return a capability whose bounds encompass the new (larger) size. However, such a capability cannot be derived from the input capability, as doing so would lead to an inauthentic capability, and we would violate the property that a capability's abilities must monotonically decrease. Thus, the allocator needs access to a 'super' capability which it can use to derive a capability representing the new bounds. Let us call the 'super' capability `SC` and introduce a function `size_of_bucket` which tells us the maximum space available for the block starting at `ptr`. Eliding extraneous details (e.g. about alignment), `realloc` will then look as follows:

```
1  void *realloc(void *ptr, size_t size) {
2    if (size <= size_of_bucket(ptr)) {
3      // No need to reallocate.
4      void *new_ptr =
5        cheri_address_set(SC, ptr),
6      return cheri_bounds_set(
7        new_ptr, size);
8    } else {
9      // Allocate a larger region of memory
10     // and copy the old contents.
11   }
12 }
```

Lines 4–7 need to deal with the case where the block is to be increased in size but will still fit in its current bucket. We first use `cheri_address_set` to derive a new capability from `SC` whose address is the same as `ptr` but whose bounds will be those of `SC` (lines 5 and 6) before narrowing those bounds to `size` (lines 6 and 7).

When implemented in this style, an allocator can be subject to the following attack:

```
1 uint8_t *arr = malloc(256);
2 for (uint8_t i = 0; i < 256; i++)
3   arr[i] = i;
4 arr = realloc(arr, 1);
5 arr = realloc(arr, 256);
6 for (uint8_t i = 0; i < 256; i++)
7   assert(arr[i] == i);
```

We first allocate a block of memory, receiving a capability with a bounds of 256 bytes (line 1). We fill the block up with data (lines 2 and 3) then `realloc` the block down to a single byte, receiving back a capability whose bounds are 1 byte[7] (line 4).

At this point, we expect to have permanently lost access to the values written to bytes 2-255 in lines 2 and 3 — if an attacker `realloc`s the block back to its original size they should not have access to the values written to bytes 2-255. However, allocators using the optimisation above will often return a capability that covers the same portion of memory as the original block, without zeroing it, allowing an attacker to read the original bytes out unchanged (lines 6 and 7).

It might seem merely undesirable for `realloc` to allow an attacker access to the original data, but in a capability system this attack is particularly egregious if that data contains capabilities, since an attacker can read those and gain new abilities.

Mitigations: Mitigating this attack is relatively simple. When `realloc` shrinks a block, any excess storage should be zeroed. Note that we consider this safer than the seemingly similar alternative of zeroing excess storage when `realloc` enlarges a block, because that implies a delay in zeroing that might give an attacker other unexpected opportunities to read data.

5.2 EscPerms: Escalate Permissions

When an allocator uses a 'super' capability (as seen in subsection 5.1), there may be potential to upgrade a capability's permissions as shown in the following simple attack:

```
1 uint8_t *arr = malloc(16);
2 assert(cheri_perms_get(arr)
3   & CHERI_PERM_STORE));
4 arr = cheri_perms_and(arr, 0);
5 assert((cheri_perms_get(arr)
6   & CHERI_PERM_STORE) == 0);
7 arr = realloc(arr, 16);
8 assert(cheri_perms_get(arr)
9   & CHERI_PERM_STORE);
```

We first allocate a block and check that the capability returned is allowed to store data (`CHERI_PERM_STORE`) to that block (lines 1–3). We deliberately remove the store permission (line 4), checking that this permission really has been removed (lines 5 and 6). We then call `realloc` (without changing the block's size) and check whether we have regained the ability to store data via the capability.

[7] Some allocators return bounds bigger than 1 byte, though none we tested returned a bounds of 256 bytes or more.

Mitigations: There are two ways of mitigating such an attack. The simplest is to AND the output capability's permissions with the input capability's permissions: doing so guarantees that the output capability has no more permissions than the input capability.

However, in some cases, one should consider validating the input capability to decide whether any action should be possible. For example, if handed a capability whose address is genuinely an allocated block, but where the capability has the read and write permissions unset, should `realloc` refuse to reallocate the block? Perhaps `realloc` should check that the capability handed to it has exactly the same permissions as the capability handed out by the most recent `malloc` or `realloc`? There are no universal answers, but some cases may be easier to rule upon than others.

5.3 EscInauthentic: Escalate Inauthentic Capabilities

An important variant on EscPerms is to see whether an allocator will reallocate a block pointed to by an inauthentic capability and return an authentic capability:

```
uint8_t *arr = malloc(16);
assert(cheri_tag_get(arr));
arr = cheri_tag_clear(arr);
assert(!cheri_tag_get(arr));
arr = realloc(arr, 16);
assert(cheri_tag_get(arr));
```

Interestingly, none of the allocators we examined was vulnerable to this attack. However, several fall into something of a grey zone: despite not explicitly checking the capability's authenticity, the allocators cause a `SIGPROT` when they try to perform an operation on the inauthentic capability. It is difficult to know whether this was an expected outcome or not, in the sense that programs often explicitly rely on null pointer dereferencing causing `SIGSEGV` to maintain certain security properties. However, since the outcome is a reasonable one, we have chosen to give the allocators the benefit of doubt in this case, and have classified them as invulnerable to this attack.

5.4 Undef: Authentic Capabilities from Undefined Behaviour

It is easy to assume that authentic capabilities can only be derived if one follows CHERI-C's rules correctly. However, it is possible for an attacker to use undefined behaviour at the language level to trick an allocator into returning authentic capabilities that it should not possess as shown in this attack:

```
uint8_t *arr = malloc(256);
for (uint8_t i = 0; i < 256; i++)
  arr[i] = i;
arr = realloc(arr, 1);
free(arr);
arr = malloc(256);
for (uint8_t i = 0; i < 256; i++)
  assert(arr[i] == i);
```

This follows a similar pattern to NarrowWiden. We first allocate a block and fill it with data (lines 1–3). Although not strictly necessary to demonstrate the attack, we then reallocate the block down to a single byte, modelling the case where we pass a capability with few abilities to an attacker (line 4). The attacker then frees that block (line 5) and immediately allocates a block of the same size (line 6) hoping that the new block is allocated in the same place as the old block. If that is the case, they will obtain a capability spanning the same memory as the old block, which allows them access to secret data (lines 7 and 8).

Interestingly, this attack places both the 'non-attack' and 'attack' portions into undefined behaviour. Most obviously, the attack portion of the code reads data via a capability / pointer that it cannot ensure has been initialised. Less obviously, both attacker and non-attacker have an equivalent capability (with the same address and bounds) but, due to capability / pointer provenance rules, the non-attackers version of the capability is, technically speaking, no longer valid by those rules. This outcome is unlikely to trouble an attacker.
Mitigations: There are no general mitigations for Undef. For the particular concrete example, a partial mitigation is for `free` to scrub memory so that, at least, whatever was present in the buffer cannot be read by the attacker. More generally, attackers may sometimes be able to use undefined behaviour to 'alias' a capability. In most such cases, the only solution is to scan memory looking for all references to a capability whose bounds encompass an address and render them inauthentic (so-called 'revocation' [17]), downgrading a security leak into a denial-of-service.

5.5 Overlap: Capabilities Whose Memory Bounds Overlap with Another's

A capability's bounds span a portion of memory from a low to a high address. If an allocator returns two distinct capabilities whose bounds overlap (e.g. because of the bounds imprecision we saw in Section 3 as a result of [16]), an attacker might be able to read or write memory they should not have access to. An example attack in this mould is:

```
void *b1 = malloc(16);
void *b2 = malloc(16);
assert(
  cheri_base_get(b1) >= cheri_base_get(b2)
  && cheri_base_get(b1) <
  cheri_base_get(b2) + cheri_length_get(b2)
);
```

In practice, such a simple attack is unlikely to succeed on all but the most basic allocators, as the most likely attack vector is when an allocator fails to take into account bounds imprecision. The 'full' Overlap attack initially finds the first 512 lengths that are not precisely representable as bounds

Table 3. Our benchmark suite. We list the benchmark name, the source of the benchmark, and a brief characterisation of it as a workload. *Alloc* is short-hand for 'allocator intensive' (i.e. frequent allocation and deallocation).

Benchmark	Source	Characterisation
barnes	mimalloc	Floating-point compute
binary-tree	boehm	Alloc & pointer indirection
cfrac	mimalloc	Alloc & int compute
espresso	mimalloc	Alloc & int compute
glibc-simple	mimalloc	Alloc
mstress	mimalloc	Alloc
richards	richards	Pointer indirection

and then randomly allocates multiple blocks to see if any of the resulting capabilities overlap.

5.6 Failed Attacks

Two attacks in Table 2 failed (⊘) due to what we believe are unintended bugs (as distinct from subsection 5.3, where we could not distinguish unintended bugs from careful, efficient, programming), and thus we cannot state whether the allocator is vulnerable or invulnerable.

EscPerms fails on *bump-alloc-nocheri* because `realloc` causes a `SIGPROT` when trying to increase a block in size. By design, *bump-alloc-cheri* contains a version of `realloc` which fixes this issue (see Section 3).

Overlap fails on *snmalloc-cheribuild* due to what we believe is an internal snmalloc bug. Because of this, we included a newer version of snmalloc as *snmalloc-repo* which is invulnerable to Overlap.

6 Performance Evaluation

We wanted to understand both the relative performance of allocators under CHERI, and also the performance impact of porting allocators to CHERI. The former measures performance within hybrid and purecap and is straightforward (subsection 6.2). The latter measures performance across hybrid and purecap and shows far greater differences than we expected (subsection 6.2). Because of that, in Section 7 we attempt to understand possible causes for these differences.

6.1 Methodology

We conducted our experiments on Arm's prototype Morello hardware: a quad-core Armv8-A 2.5GHz CPU, with 64KiB L1 data cache, 64KiB L1 instruction cache, 1MiB L2 unified cache per core, two 1MiB L3 unified caches (each shared between a pair of cores), and 16GiB DDR4 RAM. We ran CheriBSD 22.12 as the OS, with both purecap (`/usr/lib/`) and hybrid (`/usr/lib64/`) userlands installed.

We wanted a benchmark suite that contains benchmarks written in C, that have minimal library dependencies (so that we best understand what is being run), and that execute fixed workloads (rather than those that execute for fixed time). We selected 5 benchmarks from the *mimalloc-bench* suite [6] that meet this criteria, as well as the 'classic' binarytrees [1] and richards [12] benchmarks. Table 3 shows our complete benchmark suite. Our benchmark suite deliberately contains a mix of allocation-heavy benchmarks and non-allocation-heavy benchmarks, where the latter can serve as a partial 'control' to help us understand the effects of allocators on performance against other factors.

We compile each benchmark with clang's `-O3` optimisation level. We use `LD_PRELOAD` at runtime to dynamically switch between allocators. We measure wall-clock time on an otherwise unloaded Morello machine. Since jemalloc and snmalloc were the highest performing allocators, we concentrate on them for the rest of this section.

6.2 Results within Hybrid and Purecap

With the geometric mean, snmalloc is faster than jemalloc by 1.25x and 1.24x in hybrid and purecap respectively. Overall, the wall-clock time roughly correlates with instruction counts and L1I instruction cache misses. snmalloc retires a similar number of instructions compared to jemalloc in hybrid and 1.19x fewer in purecap. snmalloc also has 1.73x and 1.6x fewer L1-I cache misses than jemalloc in hybrid and purecap respectively. It is difficult to know whether this difference is because snmalloc is faster in general, or because it has been more carefully tuned for purecap than has jemalloc — indeed, a combination of both factors is plausible. Because the overall performance differences are clear and because, as we will soon see, there are much greater differences to consider elsewhere, we do not dwell further on these results.

6.3 Results across Hybrid and Purecap

Figure 1 shows a comparison of *jemalloc* and *snmalloc* across hybrid and purecap. We expected to see some performance difference between hybrid and purecap: capabilities are likely to incur some costs due to their increased size (e.g. increased cache pressure); and purecap allocators make use of additional security properties, such as bounds information, which require executing more instructions. However, the differences are far greater than we expected: using the geometric mean, jemalloc and snmalloc purecap are 1.56x and 1.61x slower respectively than their hybrid counterparts. This is a larger slowdown than we believe can be explained by the additional costs of using capabilities in the allocator — indeed, richards, a benchmark which performs little allocation, also slows down by more than 2x.

7 Analysing the Disparity between Hybrid and Purecap Performance

The disparity in performance between hybrid and purecap surprised us, and we thus explored three different avenues

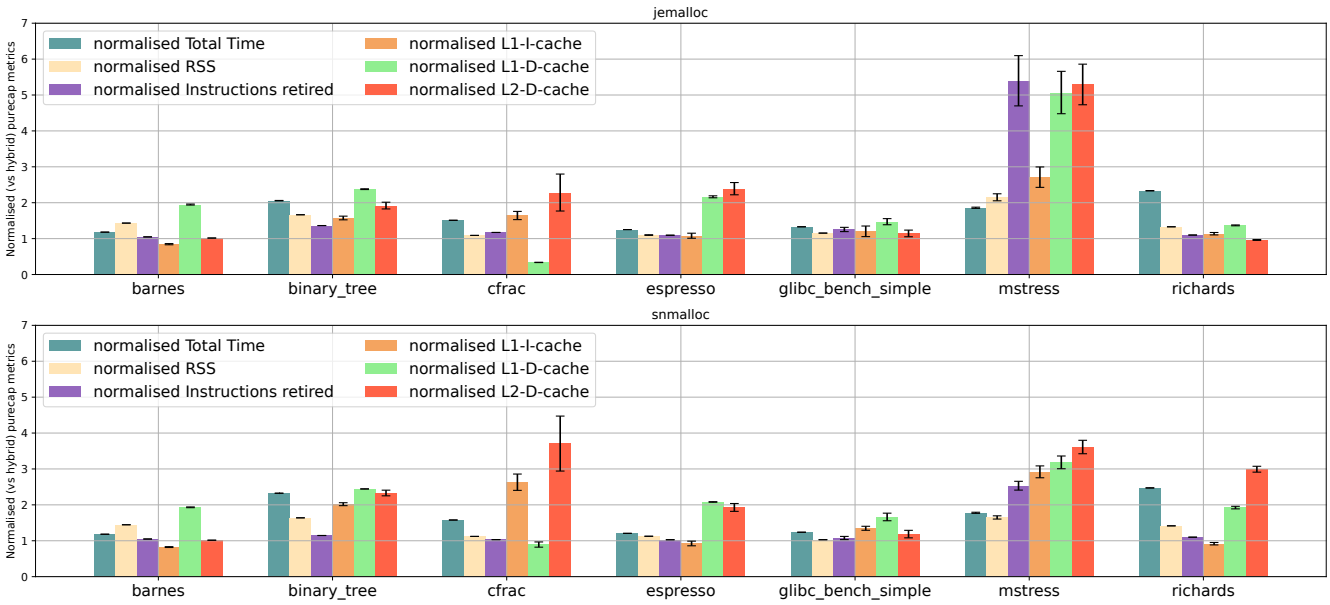

Figure 1. *jemalloc* (top) and *snmalloc* (bottom) running on purecap, normalised to their respective hybrid allocators. For example, the wall-clock execution time *total-time* of barnes with *jemalloc* on purecap is 1.22x greater than on hybrid. To understand why purecap is slower than we expected, we recorded several performance metrics: *rss-kb* is maximum memory utilisation; *INST_RETIRED* the number of instructions retired while executing the benchmark; and *{L1-I, L1-D, L2-D}_CACHE* the L1 instruction, L1 data, and L2 data, cache misses respectively. None of these factors provides obvious clues as to why purecap is so much slower than hybrid.

in an attempt to understand the causes. In this section, we explain these avenues in ascending order of difficulty.

Our results suggest that it is likely that much of the difference we see is not inherent to the use of capabilities in a CPU, but is the result of immature tool-chains and micro-architectural anomalies common in a first prototype implementation of an ambitious platform such as Morello. Our results should be interpreted in that context: it is likely that further research, and future evolutions of CHERI hardware, will change our view of capabilities' effect on performance. Indeed, since we submitted this paper we have become aware of other researchers who are also investigating similar issues[8].

7.1 Simple Metrics

We started by measuring several simple metrics, including the resident set size (RSS), and several CPU performance counters: the results are shown in Figure 1. We hoped that this might highlight possible causes such as increased cache pressure in purecap. Unsurprisingly, overall purecap has higher costs across all metrics. L2 data cache accesses increase in almost all cases in purecap, suggesting increased pressure on the L1 data cache with capabilities. In most cases memory usage is only slightly worse in purecap, though in the allocator-intensive mstress RSS doubles. However, overall, these factors do not suggest an obvious cause for the slowdown we see.

7.2 Class and Quantity of Instructions

We wondered whether the higher instruction counts on purecap benchmarks could be explained by the extra CHERI instructions that an allocator needs (compare Listings 1 and 2). To understand this, we wanted to count how often different 'classes' of instructions were executed. Unfortunately, there is currently no direct way on either CheriBSD or Morello to obtain such a count, so we had to cobble together two approaches: a fast, somewhat imprecise, approach based on QEMU; and a slow, but more precise, approach based on Arm's Morello Platform FVP [9, p. 23]. In essence, the latter serves as a sanity check for the former.

We wrote a custom QEMU plugin that maintains counts for each class of instructions as a Morello instance is running. This includes the kernel booting and shutting down as well as the benchmark we are interested in. We repeatedly ran CheriBSD without any meaningful workload, so that we could find the instruction counts that together constitute the 'head' and 'tail' of execution. When we ran a benchmark, we then subtracted the head/tail instruction counts to obtain the 'benchmark only' instruction counts.

[8]Although not available at the time of writing, we have been informed that a detailed performance white paper will soon be available at https://www.arm.com/-/media/Files/pdf/white-paper/prototype-morello-performance-results

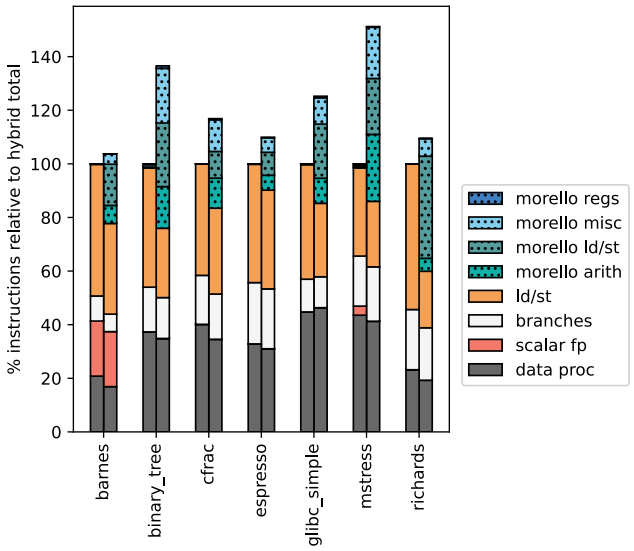

Figure 2. Dynamic instruction mix for hybrid (left-hand bar) and purecap (right-hand bar) benchmark runs. Most of these are as expected. For example, branches are the same in hybrid and purecap; some pointer arithmetic (a subset of 'data proc' in hybrid) moves from AArch64 to Morello ('morello arith'). Similarly, some loads and stores move from AArch64 to Morello. There is a small but noticeable increase in the overall quantity of loads and stores (AArch64 and Morello combined). The 'Morello misc' category captures instructions related to capability bounds, tag checks, and so on that we would only expect to see in any quantity in purecap.

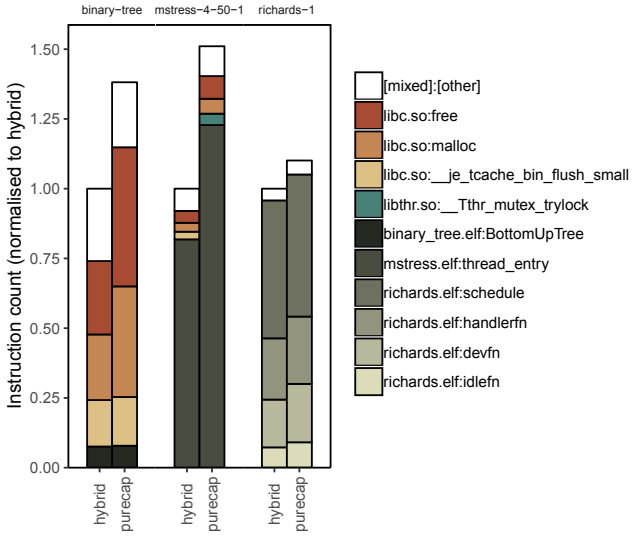

Figure 3. Instruction counts per symbol, based on FVP traces for three benchmarks using the CheriBSD system allocator (jemalloc). mstress and richards were parameterised to perform fewer iterations than the default. Note that whilst binary_trees is dominated by the effects of `malloc` and `free`, the others are dominated by their own code.

To validate these results, we used the Morello Platform FVP, which can emit *tarmac* traces [10, p. 5452] representing complete records of execution. We altered the FVP so that when we executed an otherwise unused instruction, it toggled tracing on and off. We executed a complete run of the binary_tree benchmark, examined the traces, and counted the instructions contained therein, which were a close match to our QEMU instruction counts. This gives us confidence that our QEMU figures are representative. Since our QEMU approach only has to track a few integers, whereas FVP produces tarmac traces that are often a TiB long for our workloads, our two approaches differ in performance by about 3 orders of magnitude. Running our full benchmark suite for its full duration would be infeasible in FVP, so the results are from our QEMU approach.

The instruction mixes in Figure 2 look largely sensible: some aspects (e.g. branches) are identical in hybrid and purecap; some vary where capabilities are sometimes used (e.g. loads and stores); and purecap uses Morello instructions. Overall, while there are some minor oddities (e.g. mstress uses Neon vector instructions – classified as floating point – to a much greater degree on hybrid compared to purecap), there are no obvious smoking guns.

We then wrote an analysis tool for our FVP traces (which contain virtual addresses) to provide per-function profiling, and ran this for one full benchmark, and shortened versions of two others. Figure 3 shows that some functions execute many more instructions in purecap (vs. hybrid) than others. `malloc` and `free` are particularly strongly affected, but the effect on other functions is fairly uniform.

7.3 Low-Level Differences

Hybrid and purecap CHERI imply certain differences which we would expect to account for some of the performance changes we see. In this section we analyse the following factors in detail:

$F_{Hardware}$ At the hardware level, pointer operations (including arithmetic, loads, and stores) have different semantics for capabilities, which are likely to have different performance characteristics relative to operations on normal pointers. Similarly, since capabilities are double word width, we would expect them to put greater pressure on caches and other system resources.

F_{ABI} At the ABI level, the purecap CheriBSD ABI is different than the hybrid ABI (where the latter is largely the same as a non-CHERI ABI), sometimes requiring different code generation.

$F_{Toolchain}$ At the compiler level, both hybrid and purecap modes require altered versions of LLVM. Neither is as mature as "mainstream" LLVM, and thus are unlikely to optimise code as fully as expected. Although the purecap LLVM has received more attention than the hybrid LLVM, it is also more different from "mainstream" LLVM, so it is possible that the purecap LLVM will produce less optimal code than the hybrid LLVM.

F_{User} At the user level, code that wants to take advantage of CHERI will tend to use different execution paths when compiled for purecap (e.g. the bump allocator of Listing 1).

Our expectation is not that we can identify causal relationships for performance oddities, but that we can at least understand some of the 'beneath the surface' factors that might explain part of the performance story.

7.3.1 $F_{Hardware}$: Hardware Pointer Operation (Micro-Benchmarks).

To attempt to understand some of the performance characteristics of Morello, we wrote a series of simple microbenchmarks, in a variety of C and assembly, that run under both hybrid and purecap CheriBSD. The results are shown in Figure 4.

We split our microbenchmarks into two. First, those microbenchmarks which show little or no difference between hybrid and purecap:

random-graph-walk-L1 performs a random walk through a graph that fits in Morello's 64KiB L1 data cache. This suggests that there is no overhead in reading from a capability.

random-graph-walk-fixed is similar, but uses a larger set that consumes 1MiB in hybrid and (due to capabilities being double word width) 2MiB in purecap. Despite this, there is little performance difference between hybrid and purecap.

ptr-add-asm measures pointer addition, a common operation. Although capability addition is more complex (due to bounds checking), it has a dedicated instruction and hybrid and purecap perform similarly.

busy-loop is an empty C loop, to act as a control. The C loop uses no capabilities, and compiles to identical code for hybrid and purecap targets. Hybrid and purecap have the same performance.

Second, those benchmarks which do show noticeable differences:

factorial-asm-minimal is a tail-recursive factorial implementation that, despite not using capabilities in the loop, shows bimodality in hybrid. We assume this is a microarchitectural artefact.

factorial-asm-indirect is similar, but uses an indirect tail call of a normal pointer in hybrid and a capability in purecap. This benchmark shows a consistent overhead of just under 10% in calling via a capability.

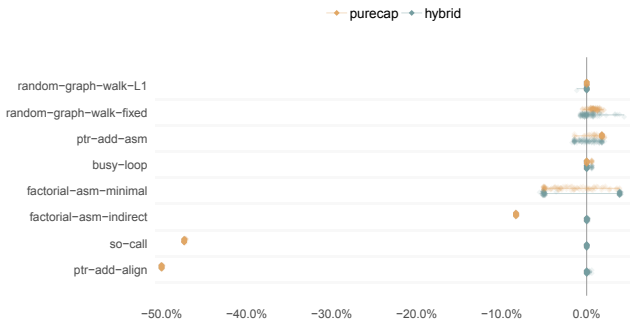

Figure 4. Microbenchmark performance results, with purecap normalised to median hybrid performance (slower to the left, faster to the right). This shows that many operations have similar performance on hybrid and purecap but some, such as calling a function in a shared object, are significantly slower on purecap.

so-call measures the PLT overhead by having a simple loop call an empty function in another shared object. This operation is significantly slower in purecap. We assume this is a microarchitectural artifact.

ptr-add-align performs pointer addition followed by an align-down operation, a common operation in allocators. The capability variant has to check bounds encodability and so uses a dedicated instruction, while the hybrid version uses a simple bitwise operation. As a result, the purecap version is much slower than the hybrid version.

Since the *so-call* and *ptr-add-align* microbenchmarks show significant differences between purecap and hybrid, we analysed how often the same idioms occurred in our larger benchmarks. In the three benchmarks shown in Figure 3, these two factors account for 0.00%-1.25% of executed instructions — they are thus likely to play a correspondingly small part in the performance difference seen in Figure 1.

7.3.2 F_{ABI}: ABI Implications.

We observed some ABI-related differences between hybrid and purecap code. For example, storing zero to a global variable in hybrid compiles to the following:

```
1     adrp x1, #+0x20000
2     str xzr, [x1, #2472]
```

However, the purecap ABI requires another level of indirection in order to obtain a capability with tight bounds:

```
1     adrp c1, #+0x10000
2     ldr c1, [c1, #3776]
3     str xzr, [c1]
```

It is difficult for us to tell if this behaviour is important for security in all, or merely some, situations. However, analysing the traces behind Figure 3 showed that in the richards benchmark, for example, we were able to observe that such code

was executed frequently in some functions, though we cannot precisely quantify the performance impact.

7.3.3 $F_{Toolchain}$: Toolchain Maturity. We noticed that the same C code compiled for hybrid and purecap by LLVM could sometimes result in very different machine code. The differences are far too extensive to admit a simple analysis, but we hypothesise that they might be due to capability code: either preventing LLVM from performing some of its normal optimisations when capability code is present; or LLVM simply not having been taught how to optimise capability code. Two small examples demonstrate the overall point.

In hybrid code, zeroing memory is compiled to a single instruction:

```
stp     xzr, xzr, [x0]
```

whereas in purecap it is compiled to two instructions:

```
movi    v0.2d, #0000000000000000
stp     q0, q0, [c0]
```

The purecap version could have used a single instruction with the `czr` register. This may or may not improve performance, but serves as an example of how quickly minor code generation differences can make it difficult for humans to comprehend differences between hybrid and purecap code generation.

Another example relevant to allocators is in a hot path within jemalloc's `malloc` function, where a byte-sized thread-local is loaded from memory. In hybrid this is compiled to:

```
ldr     x2, [...]
mrs     x1, TPIDR_EL0
ldrb    w0, [x2, x1]
```

In purecap this is compiled to a load and a bounds restriction:

```
ldp     x2, x3, [...]
mrs     c1, CTPIDR_EL0
add     c2, c1, x2
scbnds  c2, c2, x3
ldrb    w0, [c2]
```

These additional instructions will almost certainly have a measurable impact on performance, but do not appear to have any security benefit: the `c2` register is not indexed by a variable, and a greater capability is already available in `c1`. It seems likely that this is a missed optimisation opportunity by the compiler rather than a deliberate security restriction.

7.3.4 F_{User}: The Impact of CHERI Security on User Code. CHERI aware allocators often improve security by restricting capability bounds and other permissions. Doing so requires generating and running more code. For example, after allocating space internally, a purecap `malloc` may need to derive a capability from a 'super' capability:

```
...     c0, ...  # Calculate address
...     x1, ...  # Calculate length
scbndse c0, c0, x1
mov     x2, #0xffffffffffff....
movk    x2, #0x...., lsl #16
clrperm c0, c0, x2
```

In this fragment, `scbndse` sets the bounds, and `clrperm` removes permissions that `malloc` results should not have. Although it is difficult for us to say with certainty, it appears that such instructions are not as well optimised as they could be: we observed situations where it would seem more efficient to reorder some of these instructions, exposing further opportunities for optimisation. Either way, it would be interesting to understand the performance impact of these CHERI-aware aspects in isolation from other aspects of code generation, but this would require a very challenging analysis that is beyond the scope of our work.

8 Conclusions

CHERI holds great promise for securing software in general: allocators are a key part of that story. In this paper we have shown that many CHERI allocators, including the current CheriBSD default allocator, suffer from simple security vulnerabilities. We have also shown that measuring the effect of capabilities on performance is challenging, as it is difficult to understand, let alone factor out, the impact of immature compiler toolchains and prototype hardware. We expect ongoing and future research to tease apart these factors.

Despite all of this, one allocator has shone throughout: snmalloc is not susceptible to any of our attacks, and is faster than the default CheriBSD allocator in our benchmarks. We suggest that snmalloc be considered to be the default CheriBSD allocator going forward.

Acknowledgments

We thank Ruben Ayrapetyan, David Chisnall, Jessica Clarke, and Richard Grisenthwaite for comments. This work was funded by the Digital Security by Design (DSbD) Programme delivered by UKRI (grants EP/V000349/1 and EP/V000373/1).

References

[1] Hans-Juergen Boehm. 2014. An Artificial Garbage Collection Benchmark. https://www.hboehm.info/gc/gc_bench.html.
[2] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James 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. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment. In *ASPLOS*. 379–393.
[3] Jason Evans. 2006. A scalable concurrent malloc(3) implementation for FreeBSD. In *BSDCan*.
[4] Wolfram Gloger. 2006. ptmalloc. http://www.malloc.de/en/index.html.
[5] Doug Lea. 2000. A memory allocator.
[6] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. Mimalloc: Free List Sharding in Action. In *APLAS*. 244–265.
[7] Henry M Levy. 1984. *Capability-based computer systems*. Digital Press.

[8] Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J. Parkinson, Alex Shamis, Christoph M. Wintersteiger, and David Chisnall. 2019. snmalloc: A Message Passing Allocator. In *ISMM*. 122–135.

[9] Arm Limited. 2021. Morello Platform Model Reference Guide. https://developer.arm.com/documentation/102225/0200, version 2.0.

[10] Arm Limited. 2022. Fast Models Reference Guide. https://developer.arm.com/documentation/100964/1120, version 1120.

[11] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph. D. Dissertation. Johns Hopkins University.

[12] Martin Richards. 1999. Bench. https://www.cl.cam.ac.uk/~mr10/Bench.html.

[13] Robert N. M. Watson, Ben Laurie, and Alex Richardson. 2021. Assessing the Viability of an Open-Source CHERI Desktop Software.

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

[15] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, and Peter G. Neumann. 2020. *CHERI C/C++ Programming Guide*. Technical Report UCAM-CL-TR-947. University of Cambridge.

[16] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. 2019. CHERI Concentrate: Practical Compressed Capabilities. *Transactions on Computers* 68, 10 (April 2019), 1455–1469.

[17] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety. In *MICRO*. 545–557.

Received 2023-03-03; accepted 2023-04-24