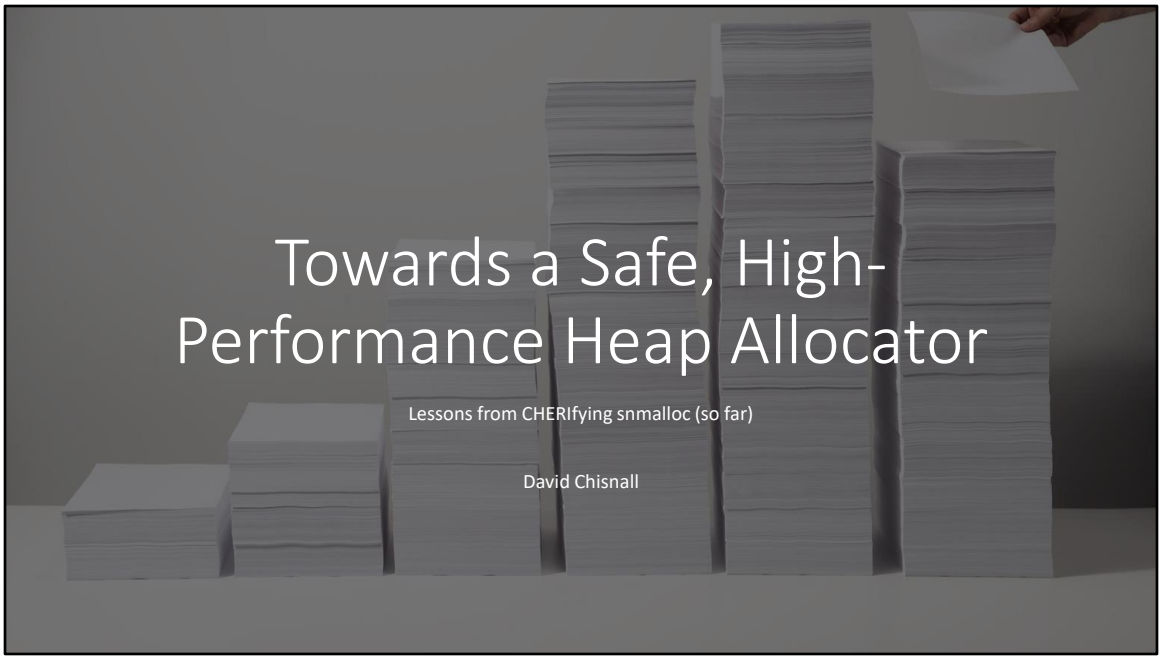


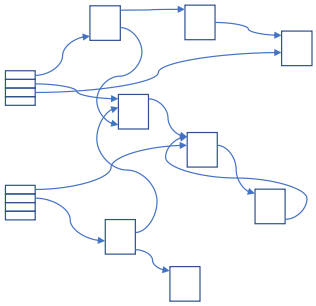
Towards a Safe, High-Performance Heap Allocator

Lessons from CHERIfying snmalloc (so far)

David Chisnall

Many programming languages, from C through Haskell, have some kind of “object” notion.

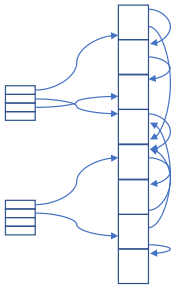
Building the object abstraction



3

These objects are largely defined by their in-language connectivity to each other and from some privileged notion of roots, such as thread stacks. These objects intrinsically have spatial extent and an extrinsic temporal extent (with the limits being either explicitly programmer-managed or abstracted away, say through garbage-collection).

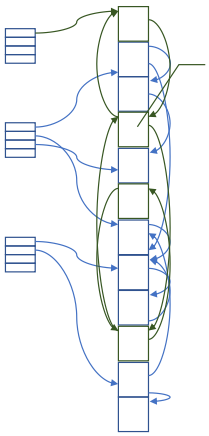
Building the object abstraction



4

To lower that abstraction to a von Neumann or Harvard architecture, we use a *heap allocator*, which places those objects in contiguous ranges of memory. This *introduces* a novel relationship between objects: *spatial adjacency.*

Building the object abstraction

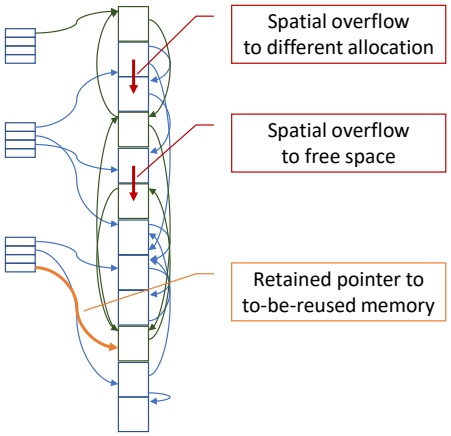


Free regions for reuse must be tracked by allocator

5

But, of course, as we said, objects come and go, and so our heap allocator must track which regions are *allocated* for the program's use or *free*, which is really another way of saying *allocated for future use*. This tracking usually involves some *in-band* metadata, such as linked lists plumbed through the "free" space in the heap. By necessity, the allocator must, itself, keep its own root pointers to these "free" region(s).

Less-than-full abstraction



6

When combined with other unsafe aspects of the language or bugs in safe languages' runtimes, spatial adjacency can lead to *overflow* or *out of bounds* accesses, which are particularly exciting when the objects are from different facets of the application. The use of *heap grooming techniques* can increase the probability that a given overflow hits an intended target.

1. Notably, overflowing into free space is possible, which can corrupt the allocator's state. Of course, there's nothing requisite about immediate proximity, and wider traipsing through the heap is certainly possible.

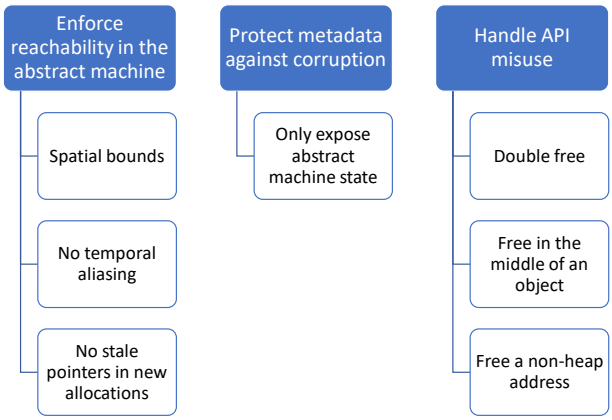
2. Reuse of memory for new objects also creates yet another new relationship between objects, one of *temporal aliasing*. This, in turn, creates its own family of risks, including disclosing old application state to a different facet and use of retained pointers to freed objects. Again, grooming techniques can increase the probability of leaking interesting state or performing accesses to reincarnated memory whose new and old types are especially interesting in combination.

This list is hardly exhaustive and there are myriad other challenges that emerge as well.

All of this might be fine, in principle: whatever is going to go wrong is going to happen *inside one application*. But applications rarely handle only data produced by the user or agent running the program, and so different facets of the application naturally encounter data from less-trusted sources, and the less-than-full abstraction we have obtained by lowering our object graph to integer-indexed memory this way turns parser bugs into weird machines for arbitrary code execution.

Phrack Magazine: Vudo Malloc Tricks - 2001
Bugtraq: The Malloc Maleficarum (seclists.org) - 2005

Towards full abstraction for the heap



7

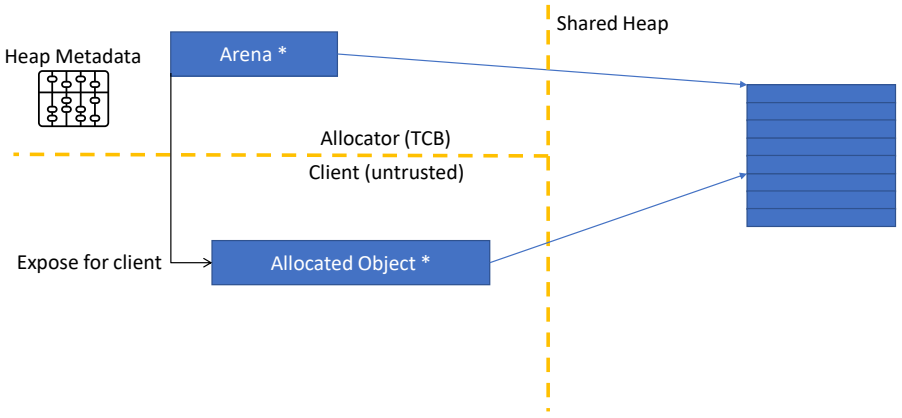
What would we need to be true of our heap implementation for it to be a fully abstract realization of the object graph abstract machine view of the world?

1. We'd need the objects it allocated to behave like abstract objects: with defined spatial extent, without surprise mystery pointers inside, and without temporal confusion.

2. The allocator does not exist within the abstract machine, and since it doesn't exist, it can't have metadata.
   When we implement an allocator, the system needs to continue to act like the allocator doesn't exist.
   Most challengingly, this means that our standard trick of in-band metadata is on very thin ice.

3. And then there's the possibility of API misuse, and in particular "Incorrect free". Thankfully, that's almost all the abuse we need to consider, but, historically, it's been a huge problem, fouling up allocator metadata in yet more exciting ways!

Heap allocator is a core part of the TCB



8

The picture that emerges is one in which the allocator is part of the TCB, isolated from untrusted client(s). The allocator holds and protects, by design, authority to the entire shared heap, as well as its own internal metadata. The allocator defines the degree to which this shared heap is a leaky or fully abstract implementation of the graphical object model.

What we are looking for, then, is a set of mechanisms for isolating the allocator and for enforcing invariants within the shared heap region(s).

(@ 5m00)

snmalloc is, first and foremost, a high-performance allocator that has added security mechanisms while aiming to preserve its performance. We have used it as our CHERIfication baseline because we had extensive local expertise and opportunity for co-design and because we are aware that CHERI will impose *some* overheads.

So, let's quickly go over some of the inexpensive defenses that we can add without CHERI.

snmalloc pre-CHERI defenses

Threat		Pre-CHERI	
Spatial separation		General	Canaries
Information disclosure			
Temporal aliasing			
Metadata access or corruption	Out-of-band		
	In-band		
Incorrect free			

Canary free objects and occasional tests of liveness

Randomized defenses | **Deterministic w/ issues** | **Solved (!?)**

10

To detect out-of-bounds stores, we intersperse random “canary” objects within the heap and occasionally validate that those canaries are still alive and with us. In principle, we could, without much effort, leave the occasional gaps between pages as well, but only at a quite coarse granularity in the heap.

snmalloc pre-CHERI defenses

Threat		Pre-CHERI	
Spatial separation		General	Canaries
		`memcpy`	Checked
Information disclosure			
Temporal aliasing			
Metadata access or corruption	Out-of-band		
	In-band		
Incorrect free			

Canary free objects and occasional tests of liveness

Look up allocator metadata for source & dest!

Randomized defenses | **Deterministic w/ issues** | **Solved (!?)**

11

In the specific context of memcpy, a popular gadget since it may have a controllable length parameter, we do have another trick up our sleeve: we can have memcpy (and memmove) integrate with the allocator and look up internal metadata. That way, at least these very common byte-slinging functions enforce heap spatial bounds!

snmalloc pre-CHERI defenses

Threat		Pre-CHERI	
Spatial separation		General	Canaries
		`memcpy`	Checked
Information disclosure		0 on alloc (optional)	
Temporal aliasing			
Metadata access or corruption	Out-of-band		
	In-band		
Incorrect free			

Canary free objects and occasional tests of liveness

Look up allocator metadata for source & dest!

Opt-in zero when allocating

Randomized defenses | **Deterministic w/ issues** | **Solved (!?)**

12

snmalloc can, optionally, zero memory before handing it out. That's a pretty good defense against revealing someone else's data, but being opt-in isn't great and it's also not actually a guarantee that allocations return zero, in the face of temporal aliasing, but still, it's pretty good.

snmalloc pre-CHERI defenses

Threat		Pre-CHERI	
Spatial separation		General	Canaries
		`memcpy`	Checked
Information disclosure		0 on alloc (optional)	
Temporal aliasing		Randomized free queues	
Metadata access or corruption	Out-of-band	Randomized location & guard pages	
	In-band		
Incorrect free			

- Canary free objects and occasional tests of liveness
- Look up allocator metadata for source & dest!
- Opt-in zero when allocating
- Randomization to frustrate attacker's attempts to locate objects of interest

Randomized defenses | **Deterministic w/ issues** | **Solved (!?)**

13

We can deploy randomization of heap layouts to help with two different parts of the problem.

First, snmalloc shuffles the order in which it allocates or reallocates objects (and takes steps to ensure that it won't shuffle a very small number of objects). This aims to frustrate heap grooming.

Second, snmalloc randomly places its metadata amongst guard pages, and preserves the type distinction between data and metadata.

snmalloc pre-CHERI defenses

Threat		Pre-CHERI	
Spatial separation		General	Canaries
		`memcpy`	Checked
Information disclosure		0 on alloc (optional)	
Temporal aliasing		Randomized free queues	
Metadata access or corruption	Out-of-band	Randomized location & guard pages	
	In-band	Pointer obfuscation & lightweight MAC	
Incorrect free		↑ & (opt-in) check of ptr to object start	

- Canary free objects and occasional tests of liveness
- Look up allocator metadata for source & dest!
- Opt-in zero when allocating
- Randomization to frustrate attacker's attempts to locate objects of interest
- Optional "encrypt and MAC" on in-band metadata: minimizes disclosure and detects tampering (whp)
- "Same object twice" DF, not "temporally aliased"

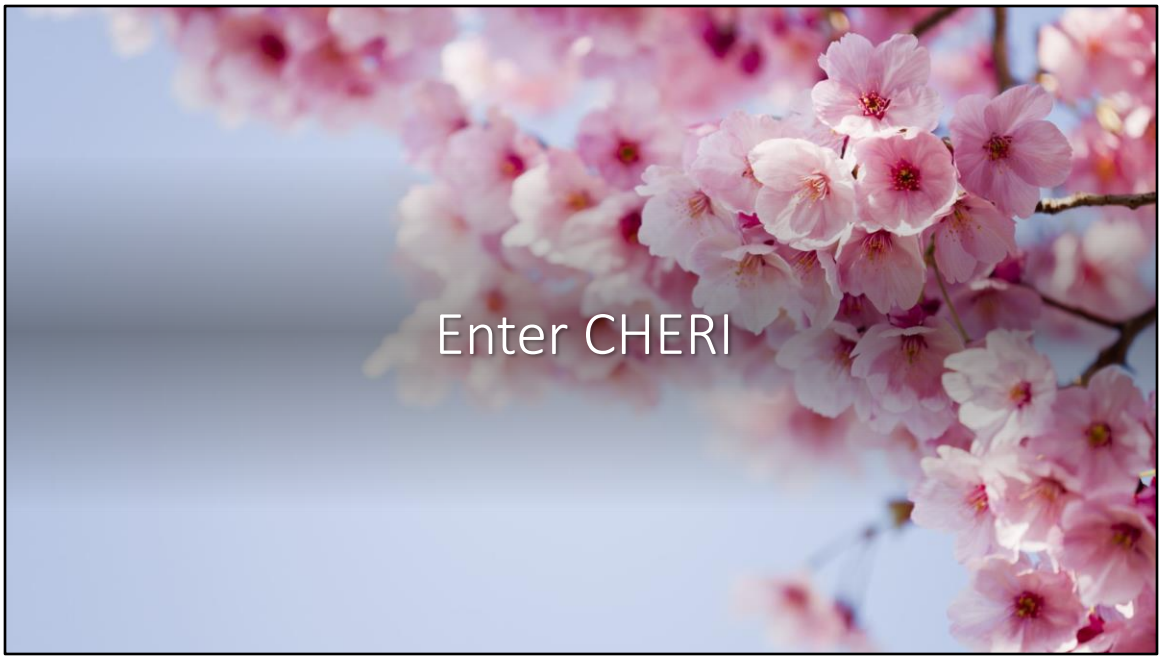
Randomized defenses | **Deterministic w/ issues** | **Solved (!?)**

14

snmalloc also (optionally) has a (novel?) lightweight "encryption with MAC" scheme for its in-band metadata.

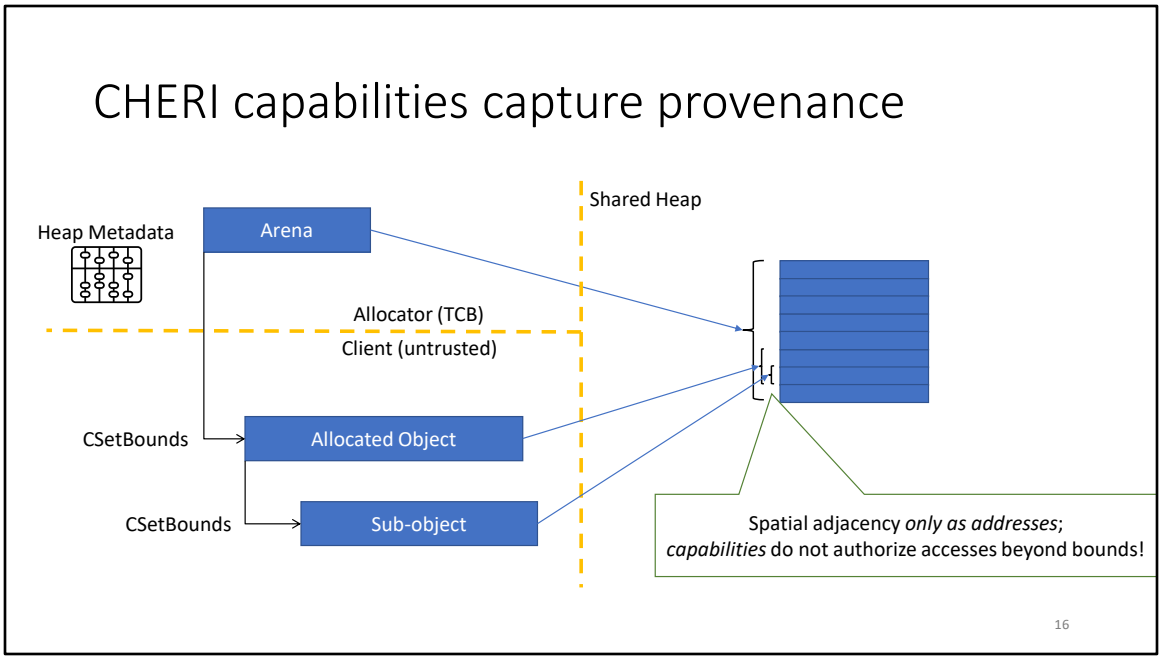
By obfuscating pointer bits underneath an allocator-private secret, it minimizes disclosure of heap pointers.
The MAC half also relies on an allocator-private secret and validates links between in-band metadata objects, and, so, it serves to detect most forms of corruption as well as double frees (because you can't have two distinct predecessors in what should be a linear order).

(Again, because integer pointers are just integers, snmalloc's double free detection only catches the "same object twice" case. It cannot tell, for an allocated object, if the pointer passed to free traces its provenance back to the most recent allocation or a prior one.)



(@ 7m30)

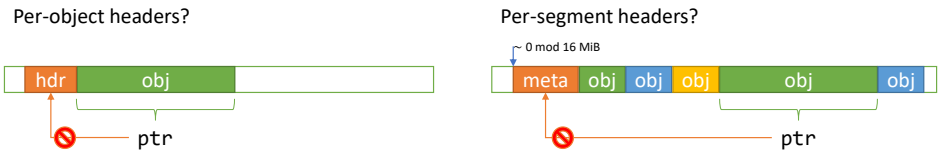
So, what does CHERI get us, and why are we so eager to CHERIfy snmalloc?



Key property: A heap pointer passed back to free will have bounds less than or equal to the bounds of an allocated object.
Going back to our earlier picture, of the allocator as part of the TCB, we see that CHERI is immediately useful, as it gives us an architectural mechanism to ensure that clients of the allocator cannot *exploit* the spatial adjacency or proximity of heap objects.

Addresses are not secret – it would be impractical to do so in C – so software can *tell* that objects happen to be adjacent. That is, our heap implementation is not quite *fully* abstract, but no amount of address arithmetic will transmute a pointer to one heap object into another.

What about `free()`?



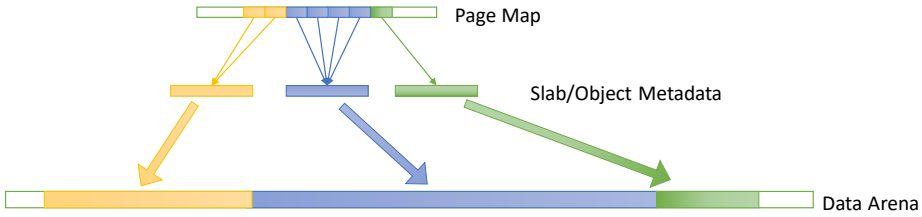
- Bounding in `malloc()` means `free()` can't use argument as pointer!
  - Need to reach metadata via allocator-private state (global/TLS/handle)

17

However, bounding capabilities handed out by a CHERIfied heap allocator frustrates very common design patterns. Traditionally, *the allocator* was *allowed* and *expected* to take heap object pointers out of bounds to access the allocator's own metadata.

1. CHERI does not give us a way to have our heap allocators directly *amplify* the returned pointer, and so we must appeal to some other construction.

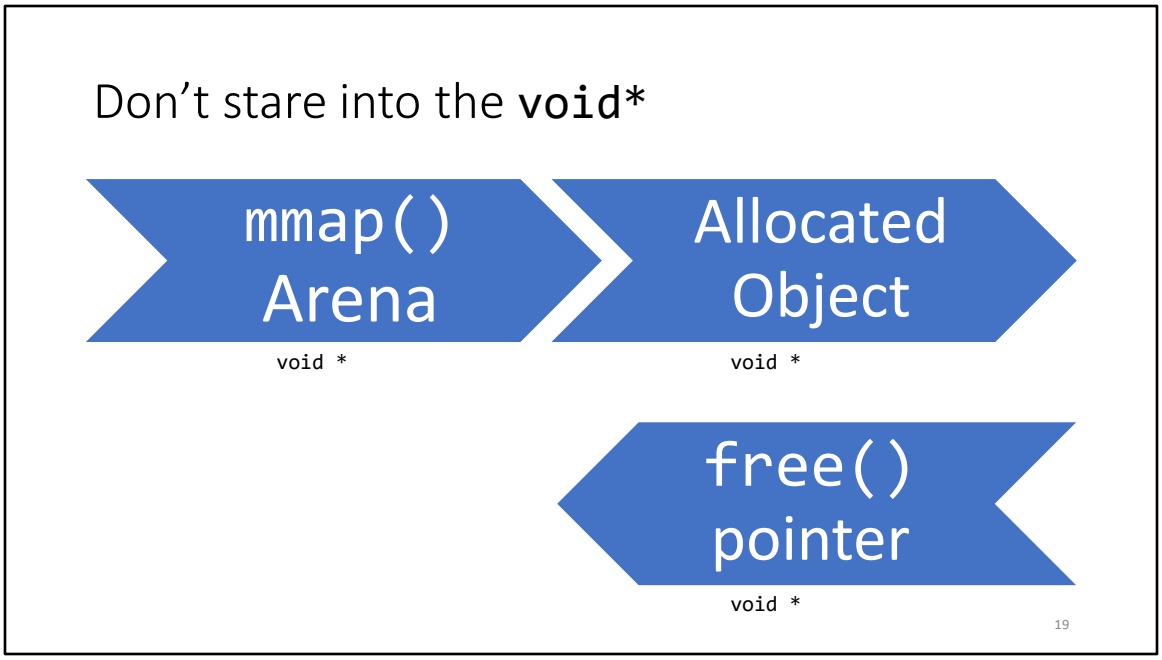
`free()` requires *amplification*



- snmalloc's central internal data structure is its "Page Map"
  (VA / 16KiB) ↦ Per-Slab or Per-large-object Metadata
- Convenient place to stash widely-bounded pointers

18

During the repeated rounds of co-design that took place between snmalloc and its current CHERI-enabled form, snmalloc's global data structure, its "Page Map", changed from storing *data* about 16MB chunks to storing *pointers to metadata* at a 16KB granularity. This *optimization*, which reduced the number of loads and branches on the free hot path on non-CHERI architectures, turns out to also be very convenient for CHERI: we can hold widely-bounded pointers within the slab/object metadata structures for allocated regions and within the pagemap itself for deallocated regions.
Already need to do these loads to find the correct freelist to add the object to, so no additional memory accesses.

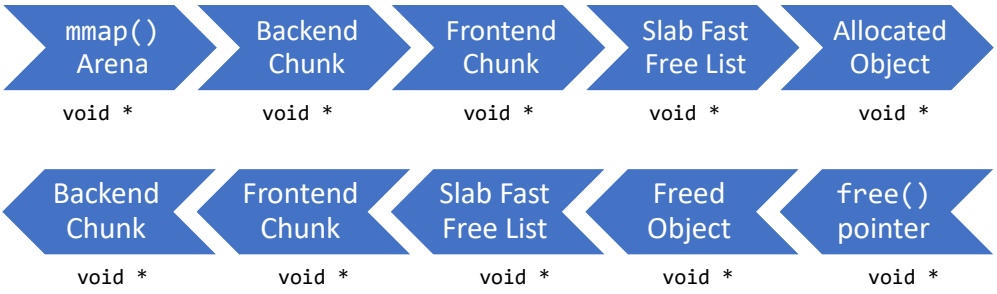


But notice that we now speak of “widely-bounded” or “high-authority” pointers, drawing a distinction between pointers to the *allocation arena* and those pointing to individual objects within.

1. These *dramatically different things* both have the same C type: “void *”.
   It is *crucial* for security, especially on CHERI, that we not reveal an arena-bounded pointer to the client; if they are both “void *”, auditing that is harder.

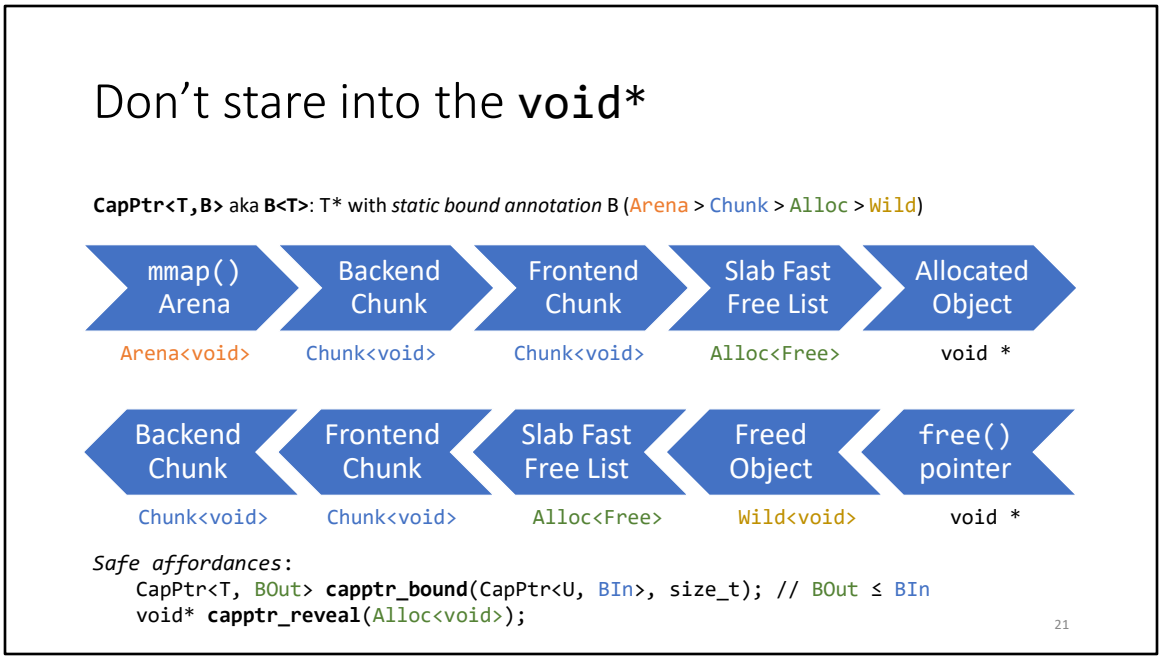
2. In the other direction, anything the client *claims* to be a pointer to a (to be) freed object, is, again, a void *.

Don’t stare into the `void*`



20

Worse, that’s just a fraction of the real story. As with most high-performance heap allocators, snmalloc has a pipeline of stacked allocators internally, partitioning the large arenas into chunks, parceling chunks out to threads, turning some chunks into slabs of objects, and so on. Again, all of these are probably represented in C as… “void *”.



`snmalloc` is written in C++, so we can use a richer type system! We introduce "`CapPtr<T,B>`", or "B<T>", a pointer to type T with static bound annotation B. Our bounds ontology captures the *role* of the pointer (whether its CHERI bounds are expected to be the whole arena, an individual allocation, or something in between) and its *domestication* (ensuring that we pass *wild* user-sourced or user-exposed pointers through defensive measures before they are presumed *tame* for dereference). Our ontology is reasonably extensible to track other aspects as well.

In addition to discouraging *disclosure* bugs, where we have failed to apply bounds to pointers given to clients, CapPtr also ensures that we use suitably *amplified* views of larger chunks as slabs are repurposed internally within snmalloc. That is, we are guided by the type system to fetch the widely bounded internal pointers held in our metadata, since we cannot pass too-narrow pointers backwards through our internal pipeline.

1. While there are, out of necessity, "unsafe" operations for constructing and destructing `CapPtr` wrappers, these are used sparingly and generally within *safe* affordances like `capptr_bound`, which ensures that the spatial role of its output is *more restrictive* than that of its input, or like `capptr_reveal`, which exposes a

`void *` from a `CapPtr` that has been refined down to being a single allocation. These safety requirements are imposed *statically*, even on non-CHERI architectures; getting them wrong is a *compilation failure* and their portability should help to ensure that snmalloc remains CHERI-aware even as it undergoes third-party development.

Initial CHERIfication of snmalloc

Threat		Pre-CHERI		CHERI
Spatial separation		General	Canaries	Set bounds
		`memcpy`	Checked	
Temporal aliasing		Randomized free queues		←
Information disclosure		0 on alloc (optional)		←, CapPtr
Metadata access or corruption	Out-of-band	Randomized location & guard pages		Capability reachability
	In-band	Pointer obfuscation & lightweight MAC		Seal* & MAC
Incorrect free		↑ & (opt-in) check of ptr to object start		↑ & ←'s check

CHERI spatial bounds do their thing!

Still just randomized defenses

`CapPtr` eases auditing

Clients not linked to snmalloc globals, but arena caps can still be leaked

Capability bits are precious; can't obfuscate, but can *seal* (not yet done) and can still MAC

Randomized defenses | **Deterministic w/ issues** | **Solved (!?)**

23

(@ 13m00)

So, here's our current state of having CHERIfied snmalloc.

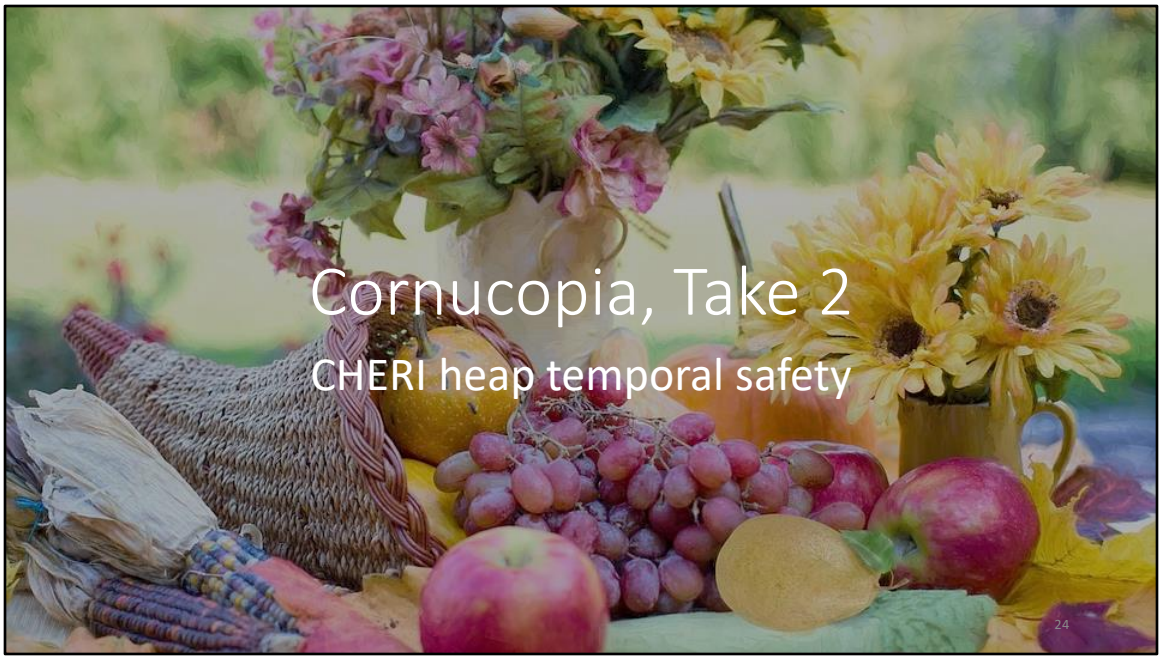
1. CHERI gives us some green in the table, since spatial bounds are CHERI's bread and butter.

2. CapPtr gives us increased assurance that snmalloc is correctly CHERI-aware.

3. Capability reachability should mean that client programs cannot see library-private globals, and so the internal metadata is reasonably protected at the language level. However, the ABI still has snmalloc sharing stacks with the client, so that's an open challenge.

4. And last, while we lose the ability to encrypt pointers, we gain the ability to seal them, and we can still MAC links within metadata.

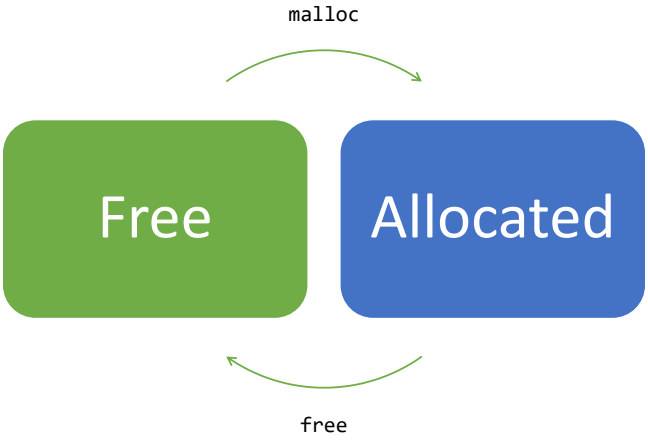
5. Unfortunately, we're still left with just randomization as our defense against temporal abuses.



(@ 14m00)

Which brings us to Cornucopia, our approach for *deterministic* mitigation of temporal aliasing.

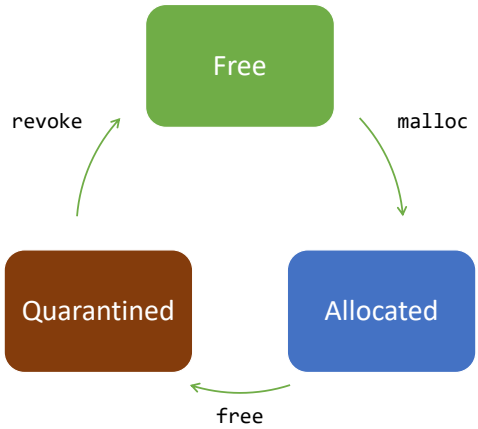
Address-space quarantine



25

To pull off a deterministic defense against temporal aliasing, we're going to expand the usual view of heap memory, in which things are either free or allocated and just bounce back and forth…

Address-space quarantine

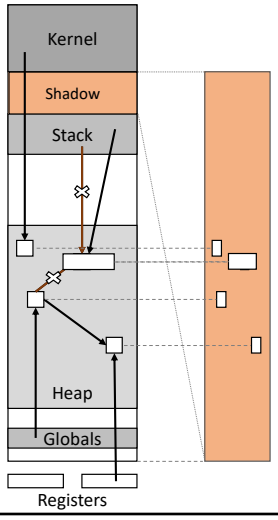


26

By introducing a new state – quarantined. Address space becomes quarantined when the application calls free() and only actually becomes free (ready for allocation) again after a global sweep through the application's memory.

This sweep will remove capabilities pointing into quarantine. Since sweeping is global and involves testing every capability in the address space, we allow quarantine to accumulate for a while and make each revocation pass process a *batch* of quarantined address space at once.

Cornucopia quarantine & revocation



- Application `free()`-s object, might retain references.
- Express quarantine by painting *shadow* bitmap
  - Live and free objects have 0 shadow bits.
- Eventually, ask *kernel* to revoke stale caps
  - Sweep AS & remove caps w/ base address shadow bit set
- After revocation, stale caps gone,
  - Now safe to clear shadow bits, &
  - re-issue *unaliased* address space!

27

So, how do we do this?

1. Cornucopia exposes a *shadow bitmap* that allocators can use to mark address space held in quarantine.

2. Eventually, userspace requests the kernel to interpret that bitmap and remove all capabilities pointing to memory with set shadow bits.

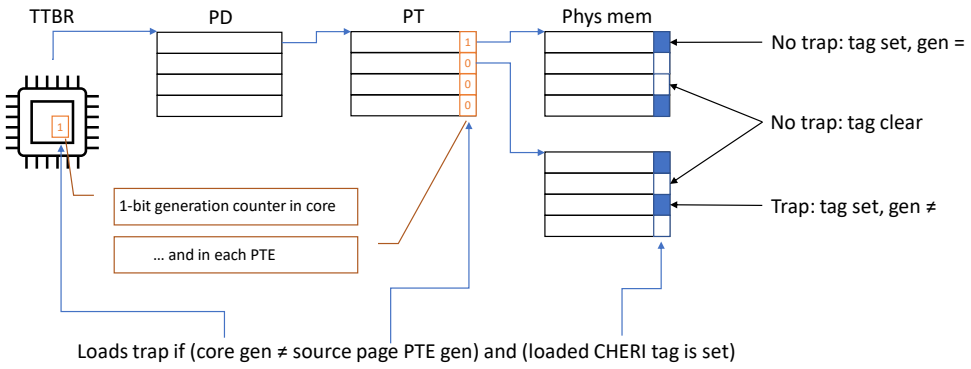
3. Once the shadow has been *imposed* on the set of capabilities, it is safe for the allocator to…
4. Clear the shadow bits, and…
5. Re-issue alias-free address space!

Of course, revocation is expensive, so we don't call it after every free(). Instead, we "quarantine" address space until a significant fraction of the heap is quarantined. The cost of a revocation pass is roughly independent of the quarantined address space or number of capabilities revoked; the bulk of the time is spent in finding and

testing capabilities.

The shadow bits also give us a defense against Incorrect free, even in highly decoupled/per-thread allocators like snmalloc: we LL/SC CAS the first word of the shadow that needs to change and bail if the bits are already set.
The same atomic sequence also guards us against concurrent revocation, where the shadow is clear but the pointer given to free is revoked.

New architecture
Per-page capability load generations



28

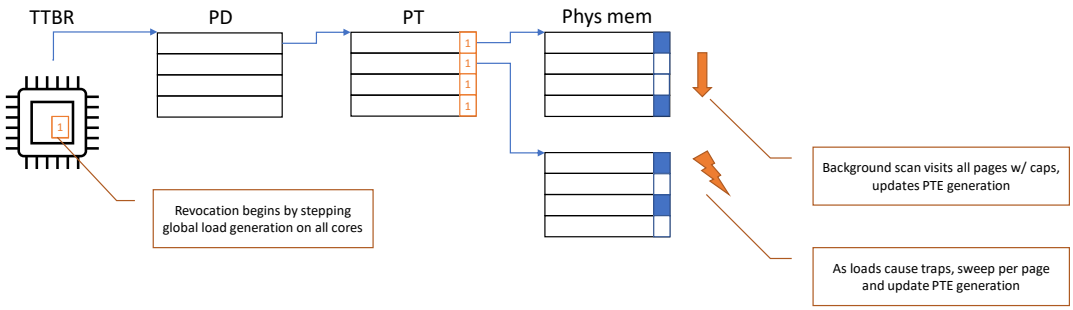
The first Cornucopia design, published at IEEE S&P 2020, demonstrated viability but suffered from large pause times. We have subsequently introduced an architectural notion of *capability load generations*. This feature is available in both CHERI-RISC-V and Morello (where it was a pretty late addition, so thanks to Arm!) and our implementation of revocation will use it by default.

1. The system tracks a generation bit in each CPU core and in each PTE.

2. If the core loads a tagged capability through a PTE whose generation bit does not match that of the core, the processor raises a trap.

Data dependence means independent instructions after load may not retire until CHERI tag value is available to be checked. PTE generation value available as part of the translation.

Revoking with capability load generations



29

This feature lets us revoke a page of capabilities *just in time*, as the application accesses that memory.

Steady state, when we're not revoking, is all generation bits equal

1) Revocation begins by incrementing the in-core generation. Now *all capabilities* are considered untested.

2) Sweep on pages as traps arrive, mark them as up to date.

3) Visit pages in background as well (currently done with a dedicated thread, so takes advantage of SMP systems w/ idle core)

4) Eventually, back in the steady state with all generations equal.

(Optimization: pages known to not contain capabilities are not brought up to date, but generation bits can't matter)

Threat assessment w/ CHERI & cornucopia

Threat		CHERI	CHERI+Revocation
Spatial separation		Set bounds	←
Temporal aliasing		Randomized free queues	Quarantine & revocation
Information disclosure		0 on alloc	0 on de-quarantine
Metadata access or corruption	Out-of-band	Capability reachability	←
	In-band	Seal* & MAC	Reuse only after revocation
Incorrect free		↑ & (opt-in) ptr check	Interlocks w/ quarantine

Address space quarantine & revocation eliminates dangling pointers

Zeroing post quarantine implies 0 at alloc, but leaves quarantine full of junk.

Quarantine tracked out-of-band; metadata in-band *only once unaliased*

At entry to quarantine: pointer validation & atomic claim of AS

Randomized defenses | **Deterministic w/ issues** | **Solved (!?)**

30

Very early revocation benchmarks

- *An unoptimized implementation, no statistical power; do not quote!*
- SPEC CPU2006 on Morello w/ load generations

Wall time	gobmk 13x13	astar BigLakes2048	omnetpp	xalancbmk
Single core	0.69%	1.7%	24%	23%
Revocation offload (SMP)	0.44%	1.1%	12%	20%

31

So, we have all this machinery implemented… how much does it cost to run?

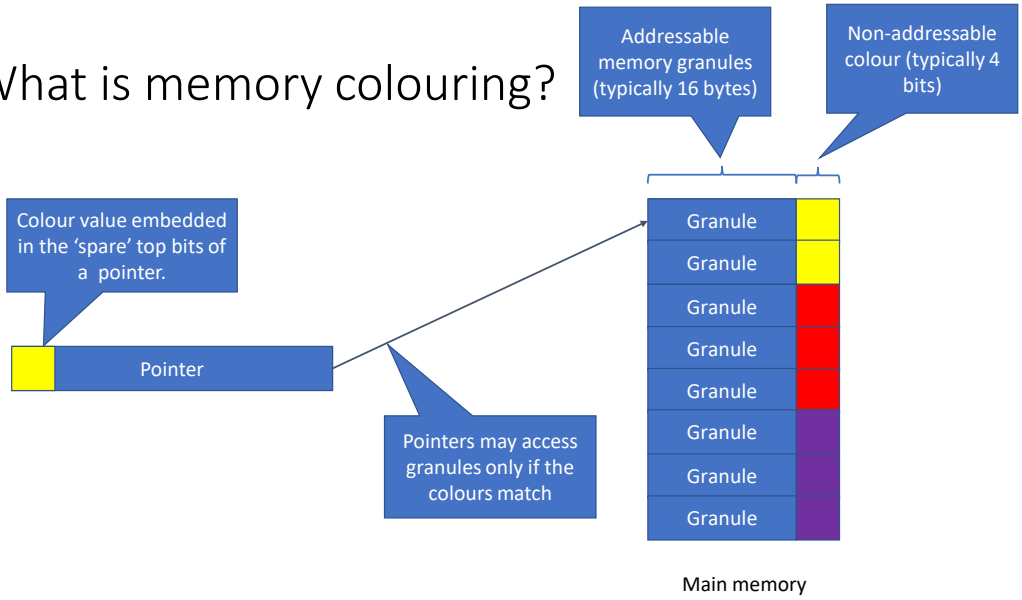
Looking at some of the worst-case tests of SPEC CPU2006, we see that even in some "worst cases" it costs barely anything at all, while for others we're starting to see significant overheads approaching 20% wall clock.

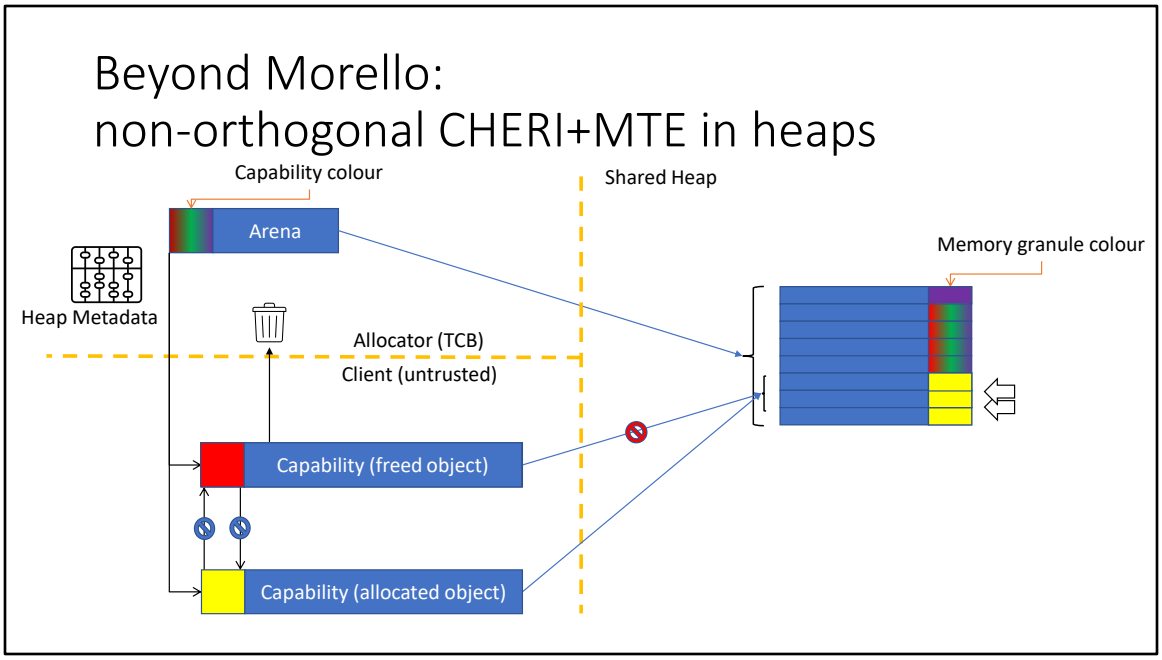
OK, that's not "turn it on in production" levels of performance, but it's likely useful for high-value targets.

We think some engineering can bring the costs down some, but we have also been thinking about an architectural extension that may simplify software and give an order of magnitude overhead improvement.

Which brings us to memory colouring.

What is memory colouring?





Change colour: inline metadata

In our proposed non-orthogonal composition of MTE and CHERI, colour bits become part of the capability metadata, and so are protected against tampering. Let's see how this new feature could improve Cornucopia.

1) On allocation, allocator derives a bounded, coloured capability to heap memory and grants this to the client.
   The distinguished "rainbow" colour value is allowed to derive capabilities of any colour and change the colour of memory.
   Other colours can only produce the same colour progeny and cannot change memory's colour.

2) The client is then free to use the derived capability, and eventually frees it.

3) The allocator uses its elevated authority to recolour memory, preventing the client's *valid capability* from reaching memory. (Can zero memory itself with little/no additional cost at the same time.)

4) Recolour-on-free closes UAF window, which gives a better debugging story, and enables secure *in-band metadata*, simplifying allocator design.

5) CHERI handles the spatial safety concerns, so adjacent heap objects can have the same colour without loss of security.

6) Re-allocation proceeds as last time, with the allocator constructing a new capability of the right colour for the client. (Any in-band metadata cleared before return.)

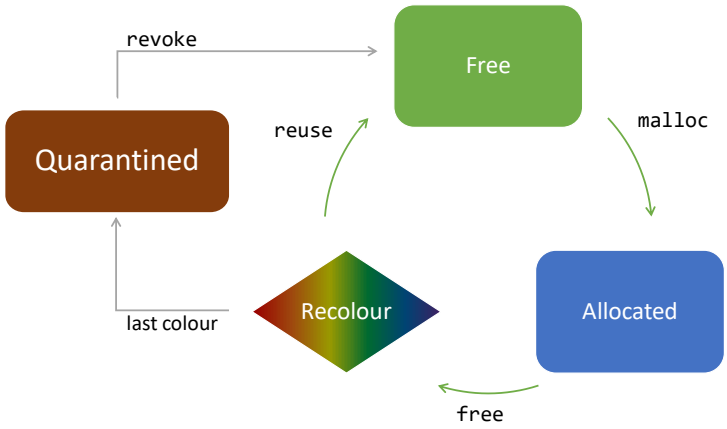
7) Clients cannot change the colour of their capabilities, nor can they recolour memory.

- Mismatching loads trap; data dependence may delay retirement of subsequent independent instructions, but no other costs.
  Mismatching stores can *fizzle*: can retire immediately and will be dropped from the store buffer rather than updating in L1. Some complexity around store-to-load forwarding (wait, don't trap, if colours mismatch?), but should hide latency of fetch for colour comparison.

- A purpose-built atomic compare-and-decrement-colour instruction catches would-be double-frees and handles concurrent interaction with the revocation.

- Even 1-bit "scaled down" MTE has useful security properties (closes UAF window) and simplifies software design (allows in-band metadata) but loses performance win of delayed revocation

Colouring & Revocation



Putting everything together, CHERI and memory colouring let us give out spatially-bounded pointers to heap objects at particular colours.

1. When those objects are freed, we can recolour their backing memory, invalidating pointers to free memory.

2. If we have not exhausted the colour space, memory can be queued for reuse immediately including in-band metadata! This means that address space enters quarantine at roughly 1/colours the rate it used to!

3. When we have exhausted the colour space for a given piece of memory, it is instead unmapped, if possible, and the address space held in quarantine.

4. Only when we are close to exhausting address space or when mapped memory is sufficiently fragmented must we run the revoker, which then makes address space safe for reuse.

Therefore, address space enters quarantine at a rate inversely proportional to the number of colours available.

snmalloc CHERI+MTE threat assessment

Threat		CHERI+Revocation	CHERI+Rev+MTE
Spatial separation		Set bounds	←
Temporal aliasing		Quarantine & revocation	Recolour & ←
Information disclosure		0 on de-quarantine	0 on free
Metadata access or corruption	Out-of-band	Capability reachability	←
	In-band	Reuse only after revocation	Reuse only after recolouring
Incorrect free		Interlocks w/ quarantine	Interlocks w/ recolouring

Recolouring reduces quarantine pressure

Zero-on-free combines w/ recolouring, clears stale caps, & is safe from client

Quarantine/free state in-band again!

Similar atomic sequence

Randomized defenses | **Deterministic w/ issues** | **Solved (!?)**

36

As we just saw, the ability to recolor memory saves us from having to quarantine memory except when colours are exhausted. Sound recolouring requires a very similar atomic sequence as was needed in Cornucopia, and so we continue to catch incorrect frees as part of our operation.

1. Because we can safely reuse memory immediately, it makes sense to zero the memory while recolouring. This has the advantages of clearing out stale capabilities while being safe from client tampering, so memory is definitely still zero once we hand it back out to the client.

2. Another benefit of safe immediate reuse is that the allocator can, even when client-useable colours are exhausted, repurpose returned memory for its own use. This means we no longer need to track quarantine out of band and can safely use in-band metadata even to track quarantine.

snmalloc with CHERI summary

Threat		Pre-CHERI		CHERI	CHERI+Revocation	CHERI+Rev+MTE
Spatial separation		General	Canaries	Set bounds	←	←
		`memcpy`	Checked			
Temporal aliasing		Randomized free queues		←	Quarantine & revocation	Recolour & ←
Information disclosure		0 on alloc (optional)		←	0 on de-quarantine	0 on free
Metadata access or corruption	Out-of-band	Randomized location & guard pages		Capability reachability	←	←
	In-band	Pointer obfuscation & MAC		Seal* & MAC	Reuse only after revocation	Reuse only after recolouring
Incorrect free		↑ & (opt-in) check of ptr to object start		↑ & ←'s check	Interlocks w/ quarantine	Interlocks w/ recolouring

Randomized defenses | **Deterministic w/ issues** | **Solved (!?)**

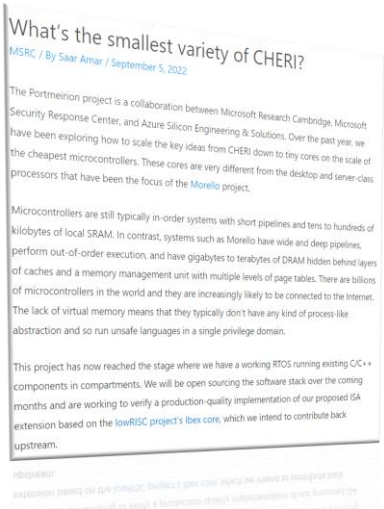
37

So, by way of reminder, here's what we have done and hope to do with further research and engineering of revocation: a high-performance heap allocator that attempts to deliver a deterministic, mostly-abstract lowering of the object graph model.

Current state of CheriBSD temporal safety

- snmalloc has baseline CHERI support (no quarantine)
  - Composes with "`mrs` wrapper" library providing a form of quarantine
  - Active work towards *integrated* quarantine
- Available for experimentation now, from source:
  - Kernel, userspace support, `mrs`, & integrated dlmalloc
- Next CheriBSD release (October) should have initial support:
  - Baseline support in userspace, bypassed on default non-Cornucopia kernels
  - 2nd, Cornucopia-enabled kernel as boot option
  - `LD_PRELOAD` malloc(s) and `mrs` wrapper as optional packages

One more thing…



What's the smallest variety of CHERI?

MSRC / By Saar Amar / September 5, 2022

The Portmeirion project is a collaboration between Microsoft Research Cambridge, Microsoft Security Response Center, and Azure Silicon Engineering & Solutions. Over the past year, we have been exploring how to scale the key ideas from CHERI down to tiny cores on the scale of the cheapest microcontrollers. These cores are very different from the desktop and server-class processors that have been the focus of the Morello project.

Microcontrollers are still typically in-order systems with short pipelines and tens to hundreds of kilobytes of local SRAM. In contrast, systems such as Morello have wide and deep pipelines, perform out-of-order execution, and have gigabytes to terabytes of DRAM hidden behind layers of caches and a memory management unit with multiple levels of page tables. There are billions of microcontrollers in the world and they are increasingly likely to be connected to the Internet. The lack of virtual memory means that they typically don't have any kind of process-like abstraction and so run unsafe languages in a single privilege domain.

This project has now reached the stage where we have a working RTOS running existing C/C++ components in compartments. We will be open sourcing the software stack over the coming months and are working to verify a production-quality implementation of our proposed ISA extension based on the lowRISC project's Ibex core, which we intend to contribute back upstream.

https://aka.ms/smallestcheri