"CHERI, a modern computing architecture centered around capabilities"

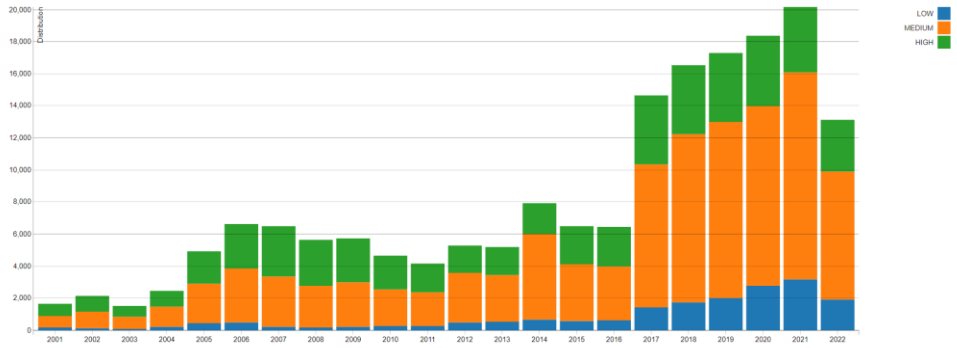
Preliminaries

- I work on CHERI for Microsoft Research, but…
  - I am not speaking on behalf of my employer. Opinions herein are mine.
  - This talk is about *science experiments* and does not constitute commitment or a promise of products.
- Questions during via Matrix; Q&A at the end.

A more inflammatory title for this talk.
Why might someone claim this?

CVEs and High Severity Bugs from (Lack of) Memory Safety

CVSS Severity Count Over Time (as of 22 Jul 2022)

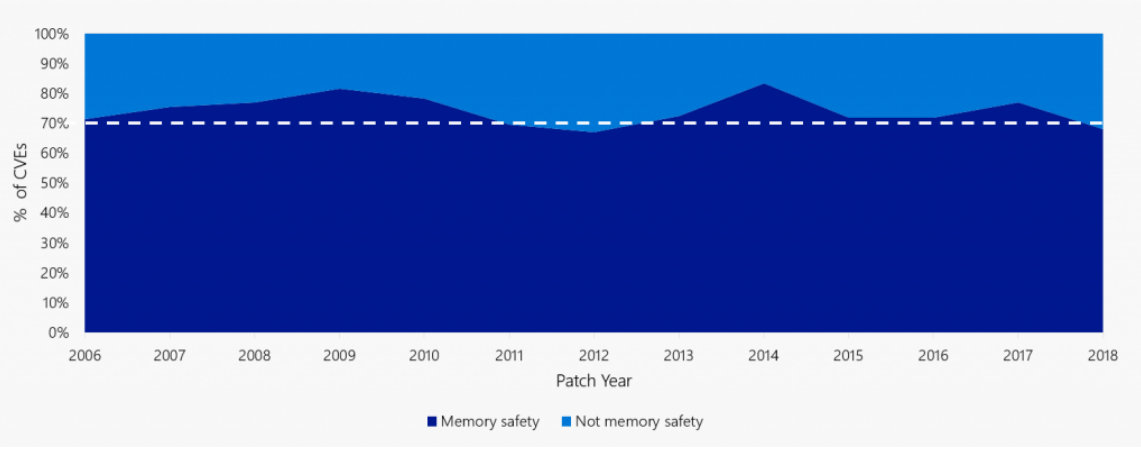


NVD - CVSS Severity Distribution Over Time (nist.gov)

4

A less contentious phrasing is that software security isn't great. As computers continue to infiltrate every aspect of existence, we are seeing a steep upwards trend in yearly CVEs…

CVEs and High Severity Bugs from (Lack of) Memory Safety

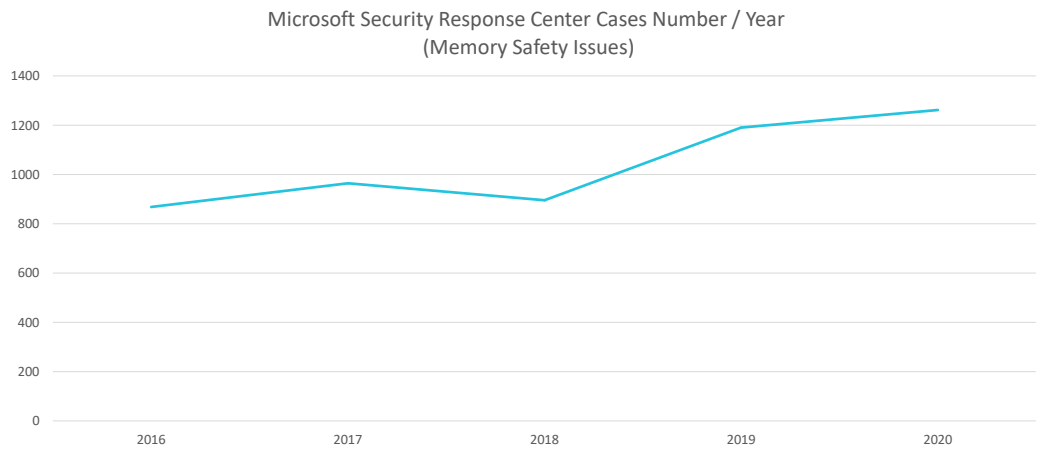


Matt Miller. *Trends, challenge, and shifts in software vulnerability mitigation.* (BlueHatIL 2019)

5

But, embarrassingly, these tend not to be new kinds of bugs in new kinds of domains. Rather, year after year, 70% of these turn out to be from memory safety problems: pointer injection, buffer overflows, use-after-free, and so on. These problems have been with us "since the beginning", at least of UNIX; so, 50 years, if not even longer; came to broader awareness in 1996 with "Smashing the Stack for Fun and Profit" by Aleph One, and even by that standard it's been 25 years and counting.

CVEs and High Severity Bugs from (Lack of) Memory Safety



Amar et al. *Security Analysis of CHERI ISA*. (Blackhat 2021)

6

So, if you take 70% of an increasingly bad time, it turns out to still be an increasingly bad time. Microsoft Security Response Center is handling more and more memory safety issues per year.

CHERI: A New Foundation for Software Security?

- Lots of people have tried lots of new things:
  - Software tricks: stack canaries, guard pages, ASLR, W^X, …
  - Static analyses: symbolic execution, fuzzing, …
  - Languages: Ada, ML, Haskell, Java, JavaScript, C#/.Net, Rust, …
  - Architectural features: PAC, MTE, BTI, continual excavation below ring 0, …
  - Computers: System/36, iAPX 432/BiiN, …

- CHERI
  - is radical, "new computer" approach: change *how pointers work*
    - A foundational shift on the same scale as *adding virtual memory*
  - is not so radical after all?
    - CHERI composes well with modern microarchitectures
    - Maybe C/C++ (and FFI) can be made safe(r)
  - has taped out; Arm's experimental "Morello" prototype SoC: 4-core, 2.5-GHz Armv8.2-A w/ CHERI extension

Chisnall et al. *Beyond the PDP-11: Architectural support for a memory-safe C abstract machine.* (ASPLOS 2015)

7

Of course, we're not the first to identify a 50-year-old problem. Lots of people have tried lots of things ranging from minor tinkering to vast, sweeping overhauls of everything. Unfortunately, nothing really seems to have moved us closer to "done" for commodity computers.

But, for all that, don't get discouraged! I'm going to try to convince you that there's hope, if we make a different kind of change.

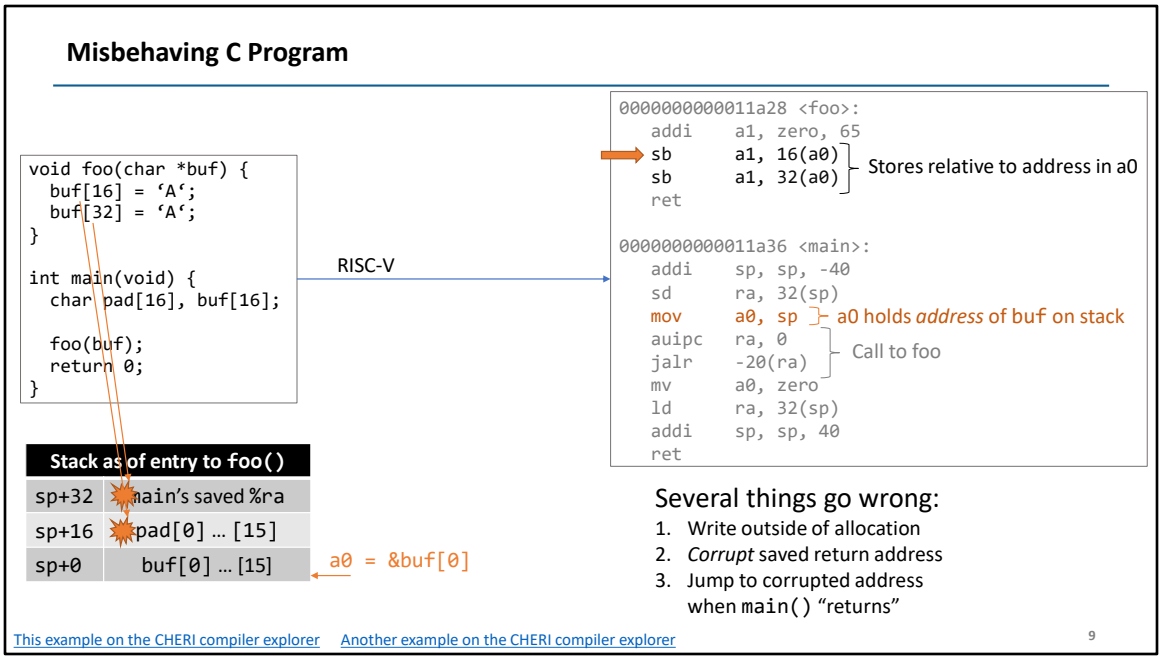
1. Enter CHERI, the, on the one hand, radical proposal that we *change how pointers work*, in both software and hardware.
   This is a shift to the von Neumann model of computing on the same scale as adding virtual memory (for what would become commodity platforms, this was around 1973; it's OK if you don't remember a time before).

2. On the other hand, I will try to convince you it's not so radical after all.
   I hope to show you that a computer with CHERI looks a lot like a computer without CHERI did, and that we might not have to throw away C/C++ and the enormous quantity of software (accumulated human effort) written in those languages to get to a better place.

3. Arm and its partners have engineered the “Morello” prototype SoC, a CHERIfied, quad-core, multi GHz extension of the Neoverse N1.

(@3m30)

To understand the changes CHERI makes to a computer architecture, it will be useful to have a small example of some kinds of unsafety that it is designed to inhibit.



Let's look at some memory safety problems. Here's a very basic C program with a stack allocation, a function call, and at least two rather glaring problems.

1. We can work out what the stack might look like when we make that function call, and there's nothing really surprising here. Reading upwards, we have our two buffers, buf then pad, and then the saved return address.

2. Here's one possible compilation of our program into RISC-V, a pretty boring RISC architecture.

3. Gazing into the assembler, we see that the stores done in foo() are performed relative to the register a0…

4. And, looking a little further down, we see that the compiler has inserted code before the call to foo that copies the stack pointer into a0. As we said, the stack had buf at its lowest address. So that's all as expected.

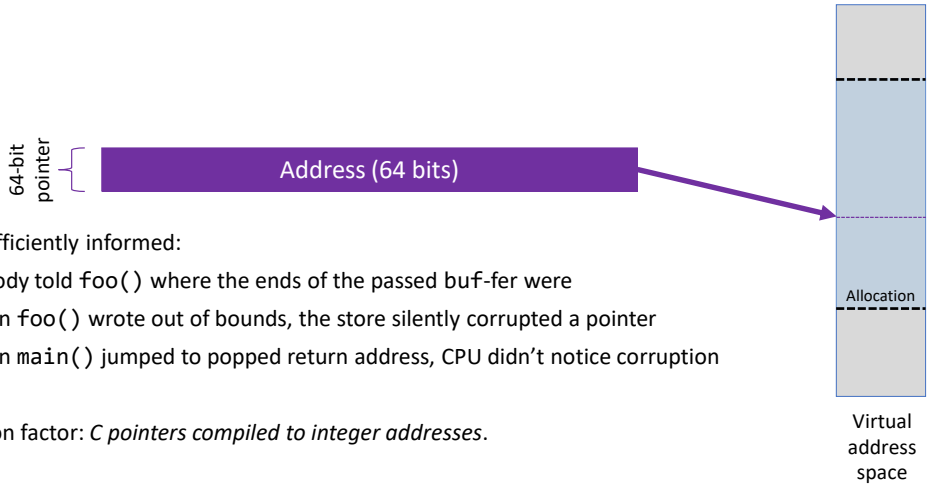
5. What happens when this program runs? Several things go wrong in escalating, rapid succession.

6. First, we write outside the intended allocation and into another object. That's bad, but it's at least something we can kind of explain using names of things visible in the language.

7. Then we write outside the language-visible allocations into something that's deeply magic – the compiler/ABI-inserted, anonymous return address. That's worse.

8. Then, when main goes to return, a while after our bug, it jumps off into the weeds to a corrupted address.

Misbehaving C Program: What went wrong?



CPU insufficiently informed:

1. Nobody told `foo()` where the ends of the passed `buf`-fer were
2. When `foo()` wrote out of bounds, the store silently corrupted a pointer
3. When `main()` jumped to popped return address, CPU didn't notice corruption

A common factor: *C pointers compiled to integer addresses*.

10

Let's spend a moment being sort of philosophical about what just happened.

I'd contend that each thing stems from the CPU not knowing enough about what's going on.

1. Nothing foo() had to hand (in registers) told it how big the buffer it was storing into was, just "here's an address, go for it"

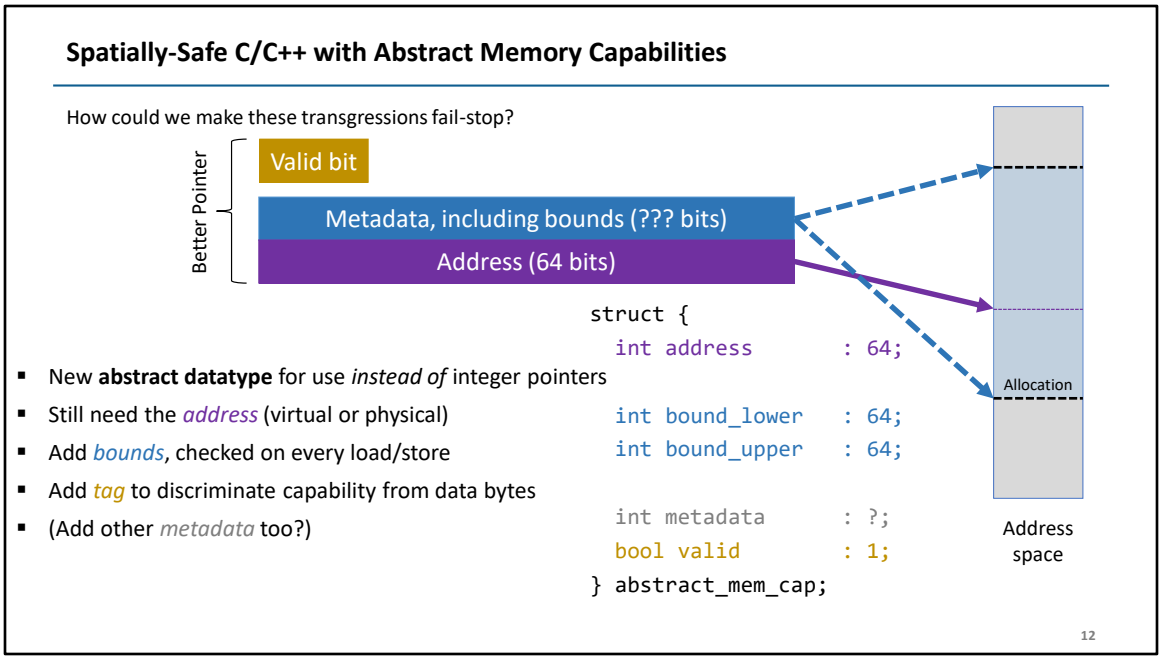
2. The out-of-bounds write silently corrupted a pointer main() cared about.

3. When main() does jump to its popped return address, the CPU didn't notice the corruption, because any sequence of bytes might be a pointer and so: off we go.

Note the common cause: we compiled a pointer, a semantic object, into just its runtime address, and there's nothing special about addresses: they're just fixed-width numbers.

(@6m30)

So, with that example in mind, what's CHERI going to do differently? What are "capabilities"?



Let's ponder what it would take to "fix" these problems. By "fix" I mean "cause to fail-stop", deterministically, and ideally close to the actual problem. What would we need to pull that off?

1. We'd need some new (abstract) datatype that we could use instead of integer pointers. Sometimes things like this go by the name "fat pointers", but we're aiming for something a little better than that phrase has usually meant.

2. Of course, it still needs to carry the address it's pointing to.

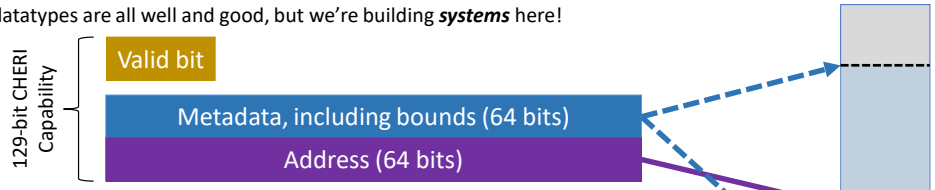
3. But we also need to carry bounds around with us; two more addresses: a lower base and an upper limit.

4. And, as we saw with the return address, we need to distinguish between valid better pointers and those that have been tampered with somehow (including by having some bytes overwritten).
   This has to be special, somehow, so we'll set it apart from the bytes that make up our better pointer.

5. And, of course, since we've opened the floodgates, we'll probably want to tack a bunch of other stuff in there, too. Everyone loves metadata.

Spatially-Safe C/C++ with CHERI Memory Capabilities

Abstract datatypes are all well and good, but we're building ***systems*** here!



- CHERI defines *architectural representation* of **capabilities**
  - 2x integer pointer size (+1 bit) via bounds *compression*
- Understood by CPU *hardware*!
  - Registers extended to hold capabilities
- Every load and store instruction executed must be to an address in bounds of a valid capability!
  - Or else the CPU *traps*: **capability fault**, like page fault
- (Add *permissions* and other *metadata* too)

```
struct {
  int address       : 64;

  int top_mantissa_exp    : 12;
  int bottom_mantissa_exp : 14;
  int bounds_denormalized : 1;

  int permissions         : 16;
  int flags               : 1;
  int seal                : 18;
  bool valid : 1; // out of band!
} CHERI_mem_cap;
```

13

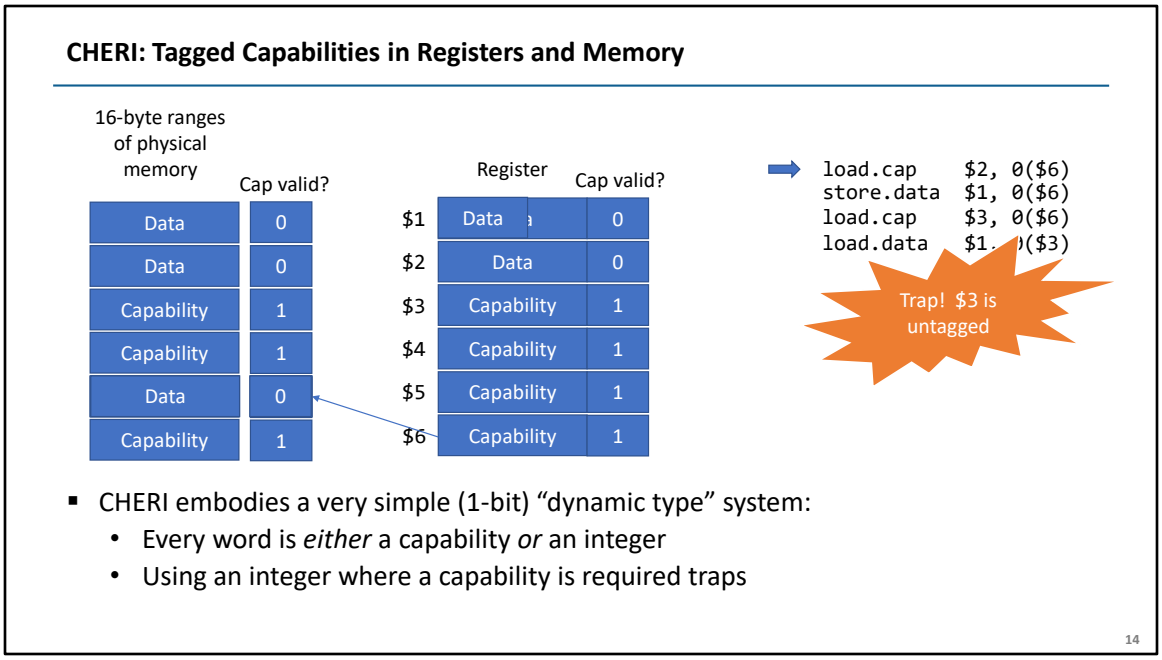
OK, while that's a nice abstraction, we're trying to build systems, here, so what would it look like, much more concretely?

1. CHERI defines an architectural representation for these "better pointers with mysterious valid bits on the side", which it calls "capabilities".
   The pointer bits are twice the size of an integer address, or 128 bits, but there's still that "+1" on the side.

2. These are understood by the CPU hardware. CHERI extends integer registers to hold capabilities ("129-bit registers").

3. Every load and store instruction checks that the address is within bounds of a valid capability.
   If that isn't true, the CPU will trap, raising a capability fault, rather like a page fault.

4. There are, concretely, permission bits as well as other metadata bits within the capability structure, as well. We'll get into that a bit more, later.



OK, let's talk about that Valid Bit that's been floating off in space. For historical reasons, we also call that bit the CHERI "tag", which is an overloaded word, but very short.

1. CHERI systems associate 1 bit of tag for every 16-byte granule (sized and aligned region) of physical memory.

2. Registers hold capabilities and their tags. Here, let's say that $6 points at a particular word in memory that happens to be holding a capability and $1 is holding some data.

3. Here are some (capability-authorized) load and store instructions, of capabilities and data. Let's run them and see what happens.

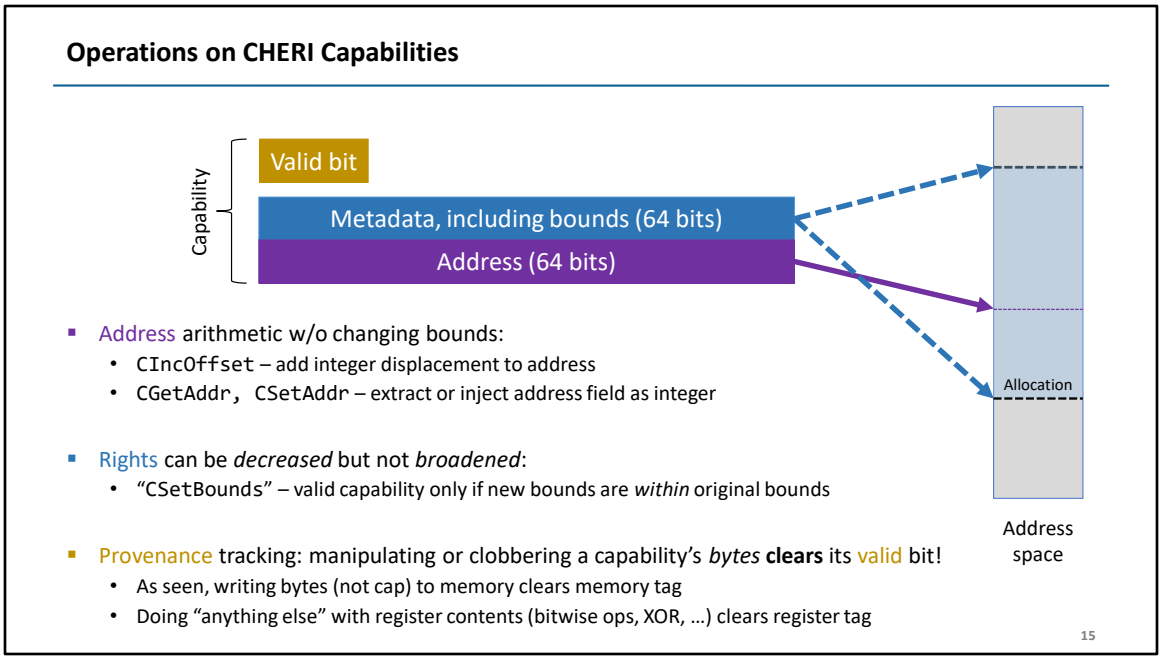
4. When we load a capability using another one, the tag comes along for the ride. Here, that tag was set in memory, and so the tag of $2 is set.

5. If we store the data component of a register to memory, it always clears the target's tag.

6. Now we try to load a capability and instead get data, so the tag in the target register, $3, is now clear.

7. If we try to access memory using an untagged register, the machine traps (like if we'd tried to read out of bounds)!

(Slide credit to Dr. David Chisnall; all errors introduced by nwf)



(@11m00)

Other than push capabilities around and load and store through them, what can we do?

1. One thing we'd better be able to do is manipulate the address just like we could when we were just using addresses without capability machinery.
   There's special handling for *offsetting* (signed addition), and for everything else there's a getter and a setter, so you can pull out the address, do whatever you need, and put it back.

2. We need a way to set bounds on a capability. If we want bounds to save us from ourselves, that should always be a monotone *non-increasing* operation, constructing a *narrower* capability from a *wider* one, not the other way around. (Uses current address as new lower bound, takes length to compute new upper bound)

3. And last, we can depend on the architecture to track *provenance* of a capability for us.

If the valid bit is set, we know that only approved operations have been performed on the capability.
Anything else – writing bytes, like we saw, or doing “anything else” with a capability in a register – results in a non-capability.

Misbehaving C Program

```
void foo(char *buf) {
  buf[16] = 'A';
  buf[32] = 'A';
}

int main(void) {
  char pad[16], buf[16];

  foo(buf);
  return 0;
}
```

RISC-V

```
0000000000011a28 <foo>:
   addi    a1, zero, 65
   sb      a1, 16(a0)
   sb      a1, 32(a0)
   ret

0000000000011a36 <main>:
   addi    sp, sp, -40
   sd      ra, 32(sp)
   mov     a0, sp
   auipc   ra, 0
   jalr    -20(ra)
   mv      a0, zero
   ld      ra, 32(sp)
   addi    sp, sp, 40
   ret
```

Stores relative to address in a0

a0 holds *address* of buf on stack

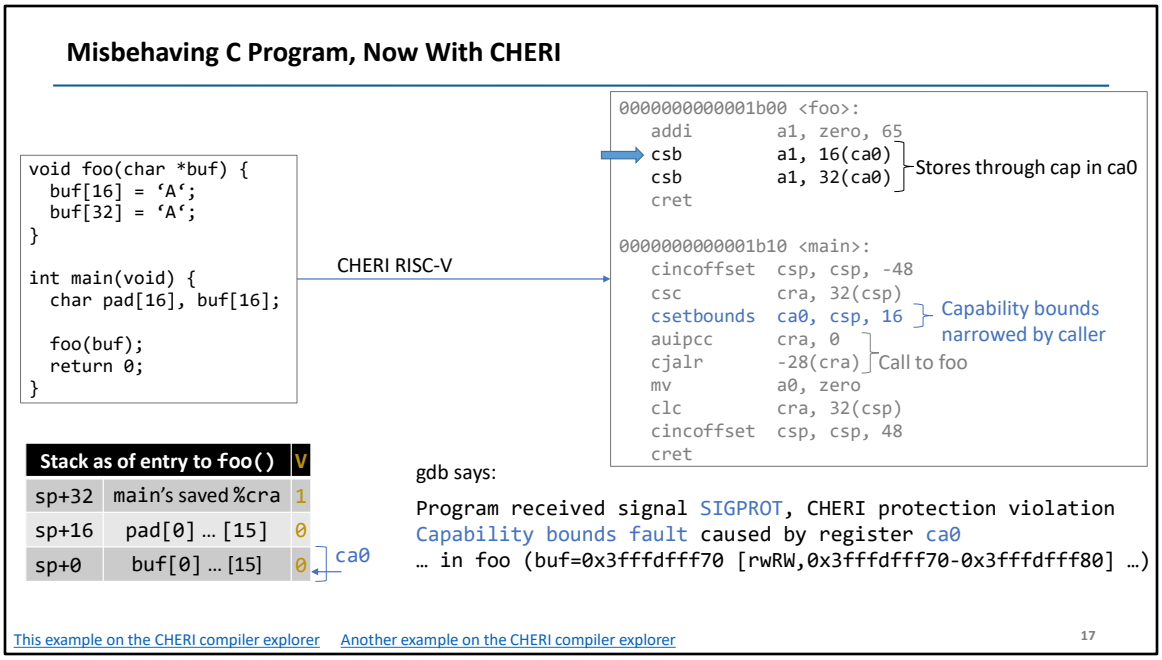
Call to foo

Stack as of entry to foo()	
sp+32	main's saved %ra
sp+16	pad[0] … [15]
sp+0	buf[0] … [15]

a0 = &buf[0]

16

So, with those operations in mind, and by way of reminder, here's what things looked like before, on a non-CHERI compilation target…

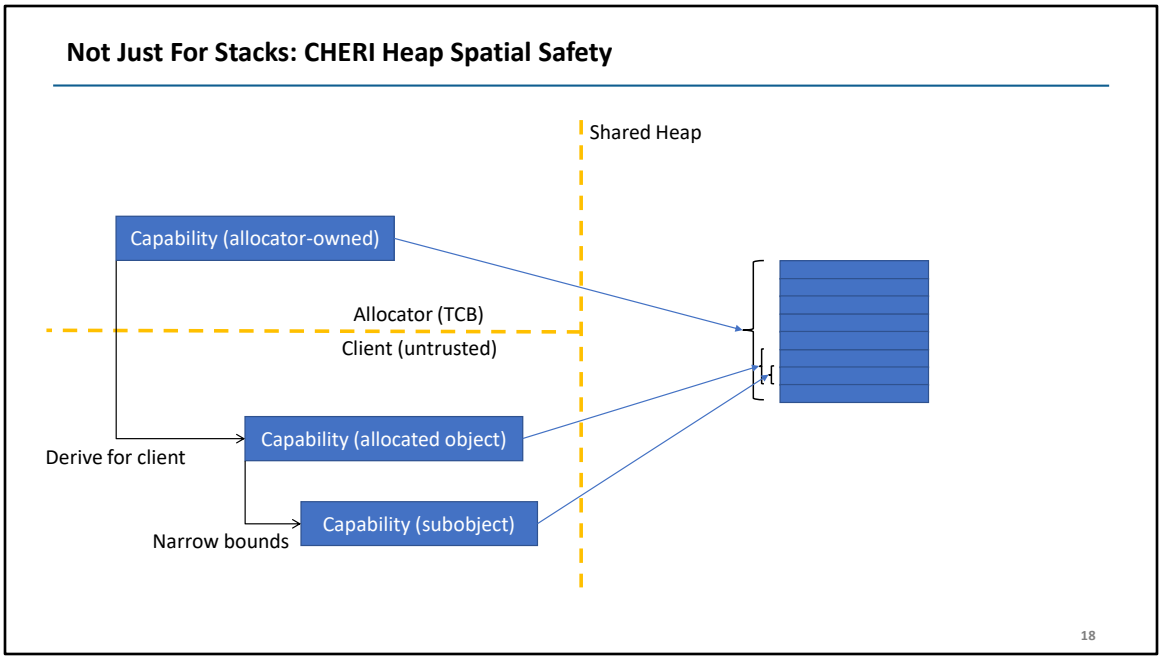


And now, if we compile to CHERI RISC-V… the program looks pretty similar.

1. The first thing to note is that our "store byte" instructions have become "capability-authorized store byte" instructions, citing the capability register ca0 (the augmented a0 register).

2. The second thing to see is that the *callsite* of foo has not simply copied the stack capability from csp into the argument register ca0, as it did the stack *address* before, but it now *builds a capability with narrower bounds* to pass as the argument.
   (The 16 here is an *immediate*, since we statically know the desired size. There is also a form that takes the length from a register.)

3. This program crashes when run, in foo: 16 up from ca0 is out of bounds.



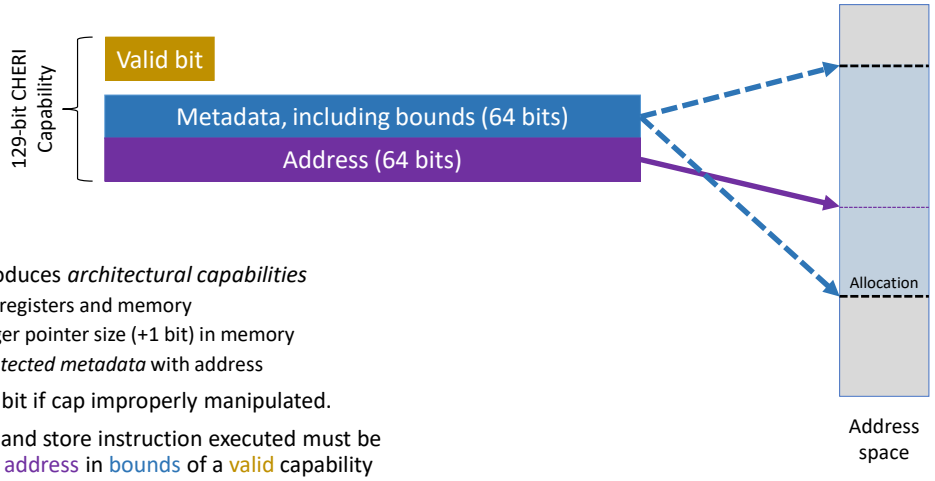
We can use capabilities for more than just stack allocations, too. Our malloc, for example, can return bounded capabilities to heap objects.
malloc internally has a capability allowing it to access the entire heap.

1) When deriving capabilities in response to client requests, allocator sets capabily bounds. No action by the client will let it use this capability to access beyond those initial bounds.

2) The client is free to derive its own sub-object pointers, but these must be to subsets of the original allocation.

Asides:
- When client returns capability to free(), bounds can only have been monotonically changed: must be a subset of the bounds given out, and so are a kind of architectural handle on the original allocation.
- This puts malloc in the TCB: enforces (and so, can violate) spatial memory safety.
- Existing CHERI-fied malloc implementations are imperfectly defensive and especially open to temporal attacks on their own metadata and/or allow clients to violate C's model by creating temporal aliases.

CHERI: The Big Idea™



- CHERI introduces *architectural capabilities*
  - Held in registers and memory
  - 2x integer pointer size (+1 bit) in memory
  - Pair *protected metadata* with address
- Clear valid bit if cap improperly manipulated.
- Every load and store instruction executed must be to an address in bounds of a valid capability
- Software can use capabilities to *implement* pointers

19

(@14m00)

So, that's the core of CHERI:

1. add architectural capabilities to the machine,
2. ensure that they come about only through legitimate operations,
3. check every load or store against the capability bounds,
4. and rewrite the compilers to use capabilities for pointers. Enlighten runtimes to take advantage of capabilities.

Secret-Free, Deterministic Mechanism

- CHERI is *secret-free* and *deterministic*.
- An adversary cannot forge a capability *even if they know every bit of system state*.
  - No ASLR slide, stack canaries, MTE colors, PAC secrets, …
  - Can't re-inject *data* as *pointers*: no more Smashing The Stack For Fun And Profit *even ignoring bounds*
- Out-of-bounds or invalid dereference *always* traps.
- Byte-level corruption or attempts to widen bounds *always* caught (clear tag or trap).

Amar et al. *An Armful of CHERIs*. (2022)

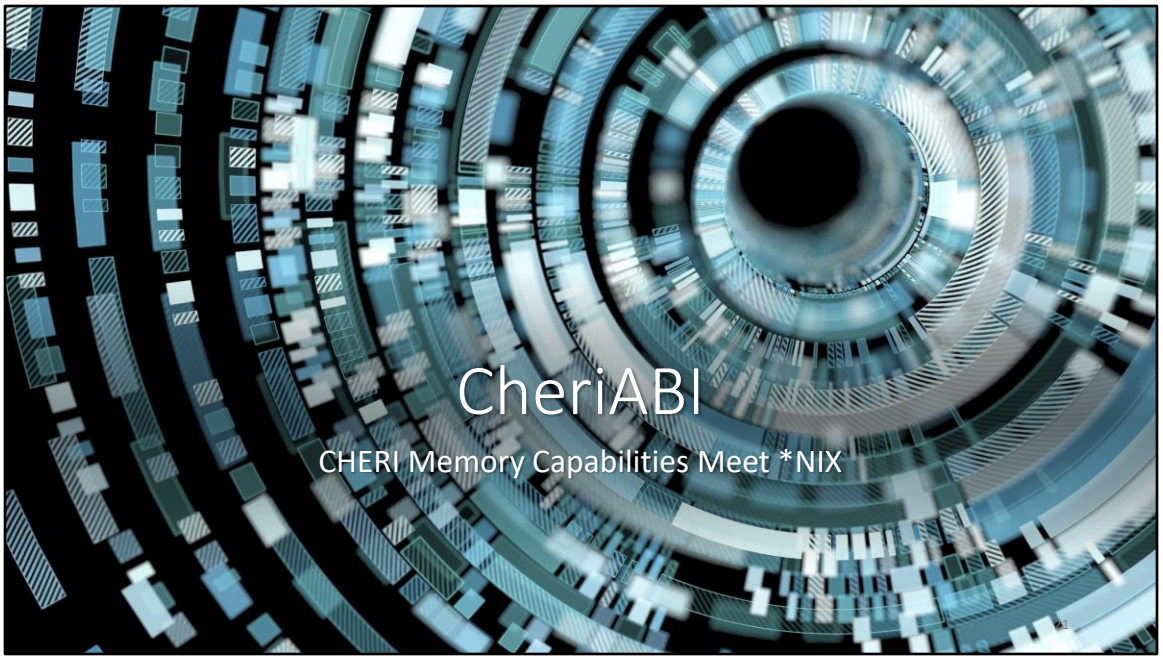
20

We should emphasize that CHERI is secret-free and completely deterministic.

1. A would-be attacker cannot forge a capability, like they could forge addresses, even if they know every bit of the system state. It's not that you can't leak addresses anymore, it's that knowing the address probably doesn't do you much good. Even if we ignore bounds, you can't ship a tagged capability across the network.

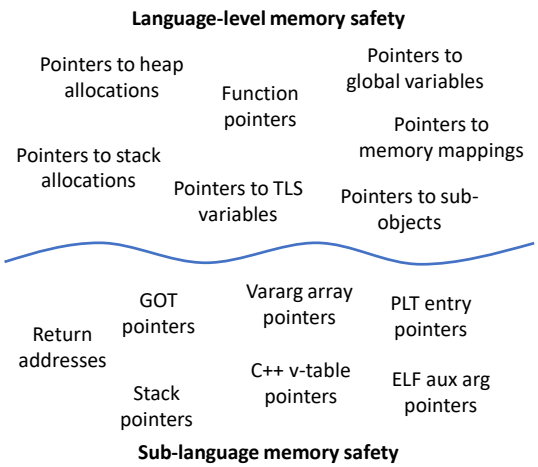
2. A CHERI system will always trap if the program tries to dereference out of bounds or through an invalid capability.

3. Byte-level corruption or attempting to widen bounds will always be caught and inhibited, either by clearing the tag of the result or by raising a trap.



So, now that we understand the architectural nature of CHERI, let's see what software can do with it.

Compiling C to CHERI



- CHERI capabilities used for both
  - **Language-level** pointers visible in source program
  - **Implementation** pointers *implicit* in source
- Compiler generates code to
  - bound address-taken stack allocs & sub-objects
  - build caps for vararg arrays
- Loader builds capabilities to globals, PLT, GOT
  - Derived from kernel-provided roots
  - Bounds applied during reloc processing
- Small changes to C semantics!
  - `intptr_t`, `vaddr_t`
  - `memmove()` preserves tags
  - Pointers have single provenance
  - Integer ↔ pointer casts require some care

See CHERI C/C++ Programming Guide.

22

We can take this idea to its logical conclusion: let's use capabilities for *everything*, or, at least, *all pointers* in a process.

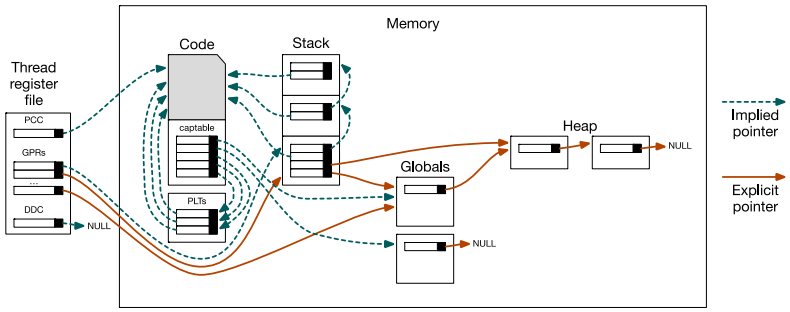
1. That means both the pointers you see and think about *in* the language as well as the ones you don't see or think (much) about *below* the language.

2. So, the compiler, loader, and even kernel also must be active participants in the implementation.

3. The C language semantics do have to change a little bit; please do see our programming guide for details.

CheriABI: Spatially safe UNIX Processes

- Compiler uses capabilities to implement *all* pointers in a process
- Result is a *capability graph* between allocations



Davis et al. *CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment*. (ASPLOS 2019)

23

(@16m00)

And if we do use capabilities to represent every pointer in a process, what we get is a capability graph between different objects, with the thread register files forming the roots. We call this environment “CheriABI”.

CheriABI System Call Interface

Significant *ambient authority* in modern *nix-like systems: system calls!

- Attacker might try to *trick kernel* into violating spatial safety ("confused deputy"). Consider:

```
char buf[1024];
… read(fd, buf, attacker_controlled_length /* == 2048 */); …
```

- CheriABI also makes system calls take and return *capabilities* instead of integer addresses!
  - Kernel uses passed-in capabilities to *limit its own behavior*.
  - `read(fd, buf, len)` won't write beyond `buf`'s capability bounds, even if `len` says to!
  - It *can't*, if it passes the *user's* `buf` to `copyout()`. *No new bounds-check instructions*!

Davis et al. *CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment*. (ASPLOS 2019) 24

But wait, user-space software does more than just follow user-space pointers. Sometimes it takes advantage of a huge ball of ambient authority, the kernel, and makes system calls.

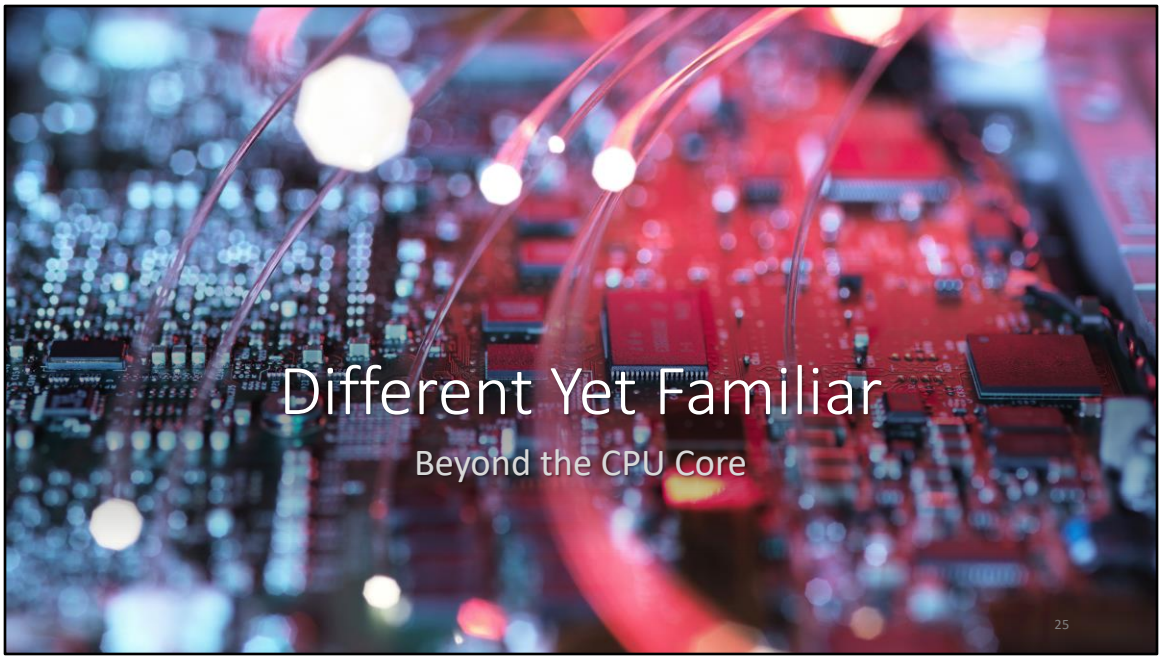
1. There's risk that the kernel could be tricked into violating our carefully orchestrated capability system, making it a "confused deputy". After all, it has, legitimate, intended access to the entirety of user-space.

   In this example, user-space has allocated a 1K buffer and is asking the kernel to write into that buffer some larger number of bytes, perhaps because an attacker has control over the length of the request. A completely implausible scenario, I know.

2. In order to deliberately limit its own behavior, a CheriABI-aware kernel changes the system call interface so that pointers are now passed *as capabilities*.

   This way, the overlong read request above will fail, gracefully, when the kernel copies data out.

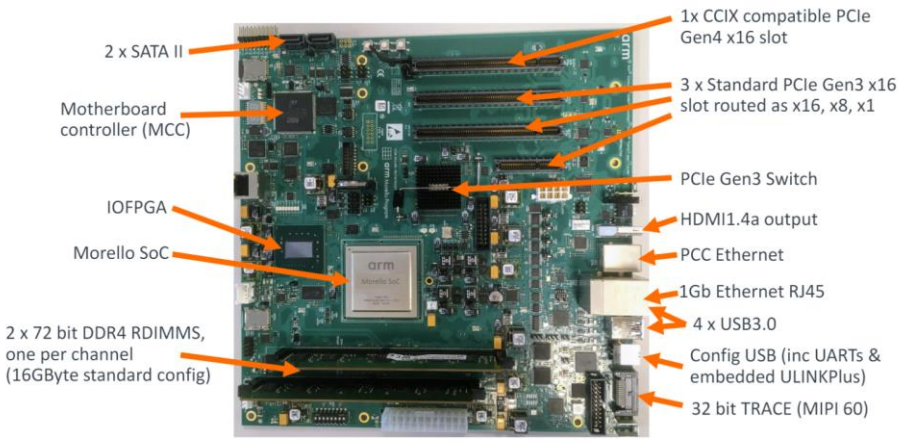
In fact, the implementation takes advantage of the existing, fault-safe copyout() logic in the BSD kernel: simply making sure that we pass the user's buffer capability to this routine makes sure that the store instructions that perform the copy down to the user will enforce the passed bounds. We did not need to add – manually or automatically! – any new instructions to *check* that the bounds were consistent!



(@18m00)

So, that's, in a nutshell, how CHERI's *different* than current architecture. But I also promised that it wasn't as disruptive as it might first have sounded…

Entering Reality: Morello, An experimental ARMv8 with CHERI



arm

Richard Grisenthwaite. Arm Morello: What Is It and Why Is It Important? (2022)

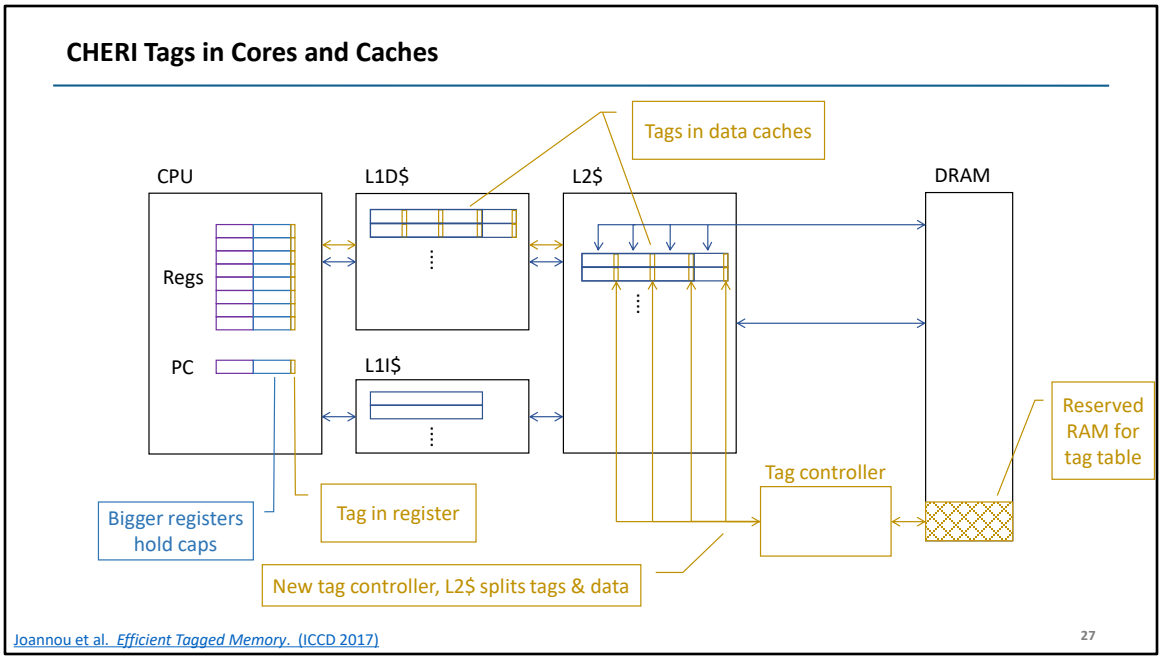
Morello Program – Arm® Morello Platform Open Source Software

26

Perhaps the greatest indication, so far, that CHERI is practical is that Arm and its partners (including us at Microsoft) are doing an industrial-scale science experiment, named Morello (it's a kind of cherry): ARMv8.2 with CHERI with a modern CPU microarchitecture, clocked at 2.5GHz.
Morello is (emphatically) not, but will hopefully influence, successors to ARMv9.

Given this, for present and future systems programmers: it is looking increasingly likely that CHERI will be part of the world we live in.

Photo from *DSbD Consortium Update*. (2021/05)



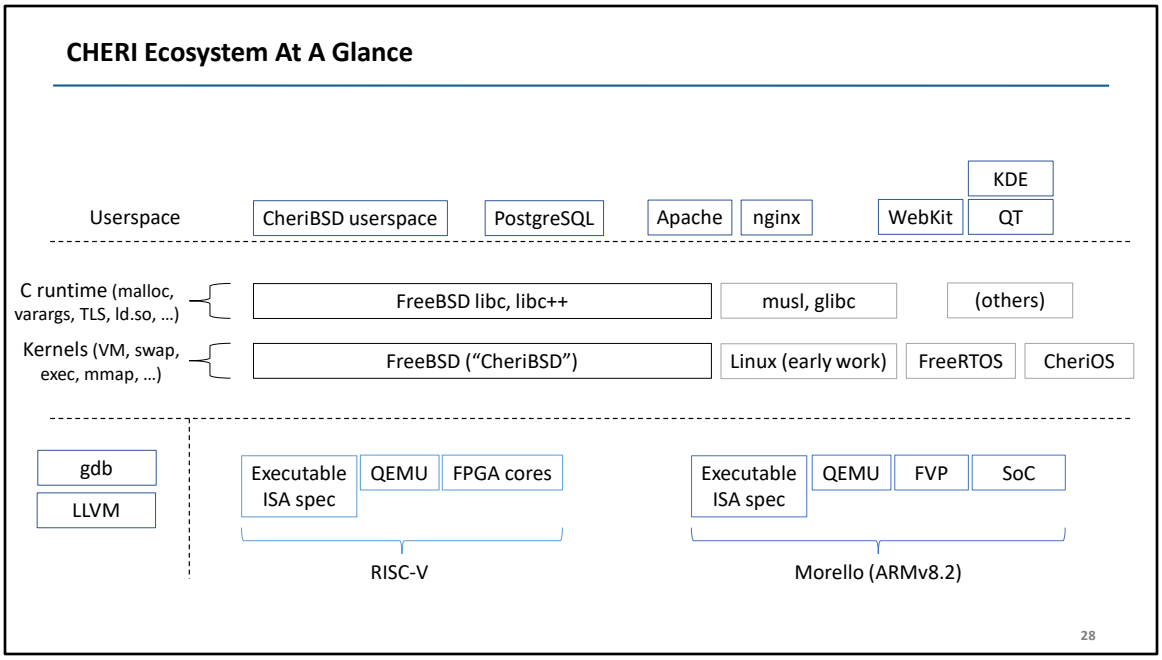
One of the central design objectives of CHERI was that it couldn't need a whole new *everything*. Importantly, it needed to be compatible with commodity memory (and busses and so on).
Peeking beneath the architectural covers, as it were, let's look at a slightly more detailed view of what happens in the memory hierarchy.

1. As said before, we augment the CPU core to hold capabilities in registers: so, roughly 2x the size plus tag bits.

2. Add tags to data cache lines; tags now move in tandem with data through the cache hierarchy.

3. L2$ splits lines, holding both data and tags, into separate channels, sends data directly to DRAM and tags to a new *tag controller*. This might be as simple as a bitmap, but prior work shows great gains from a small *hierarchical* caching scheme here.

   The tag controller is backed by a carve-out of RAM, which is *not accessible to software*: software cannot manipulate tags except as part of a capability.



(@20m00)

In fact, CHERI has two primary architectural incarnations these days: as Arm's Morello (SoC) and atop RISC-V (FPGA). Both also have executable, human-readable ISA specifications and emulators (QEMU and Arm's Morello "Fixed Virtual Platform"). Looking "above" the architecture, into software…

1. Atop these substrates, most of our software work takes place with a modified FreeBSD ("CheriBSD"). The kernel and C runtime components have been made CHERI-aware.

2. The whole software stack is built primarily (so far) in cross-compilation using a capability-aware branch of LLVM (so clang and lld); self-hosting on Morello is likely to be the norm soon.
   We have, similarly, educated gdb for native and cross-debug.

3. Continuing upwards, we have the FreeBSD userspace programs; servers like Apache, nginx, and PostgreSQL; and client software like WebKit, the QT library, and almost all of KDE ported.

CHERI Source Compatibility

Codebase kind	LoC Changes for CHERI
CheriBSD Kernel	0.2%
Low-level runtime libraries	< 0.5%
JSC JIT	1-2%
QT, KDE libraries	< 0.1%
CLI applications, libraries	≈ 0.02%
QT, KDE applications	< 0.05%

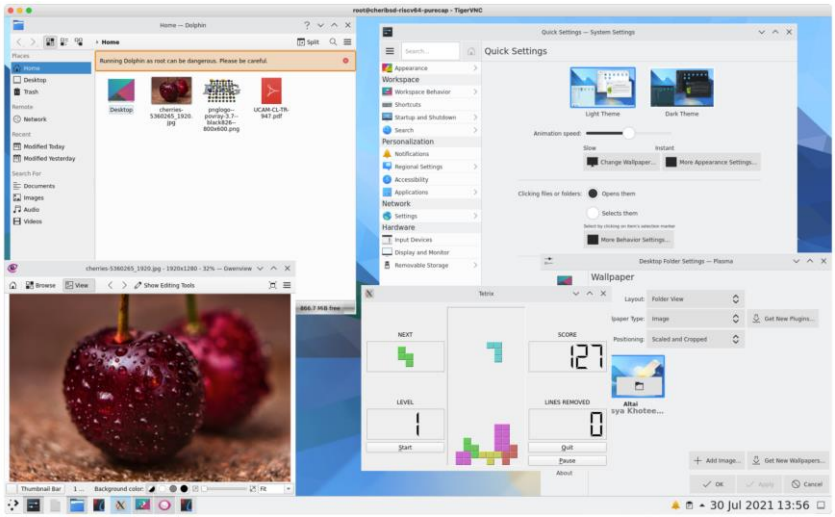
DSbD Consortium Update. (2021/05)

Capabilities Limited. *Assessing the Viability of an Open Source CHERI Desktop Software Ecosystem*. (2021)

29

There's a whole lecture's worth of material about porting C/C++ programs to CHERI, but generally, higher in the stack means significantly less work.
By design, CHERI is also broadly compatible with modern C-based software stacks, so most of its obligations are felt at lower levels of the software stack.
We can quantify this by noting that the impact, measured in source LoC changes, diminishes as we move away from support and runtime layers, with many KDE applications requiring no modifications for CHERI at all once QT and libraries had been adapted.

Entering Reality: KDE on CHERI-RISC-V over VNC



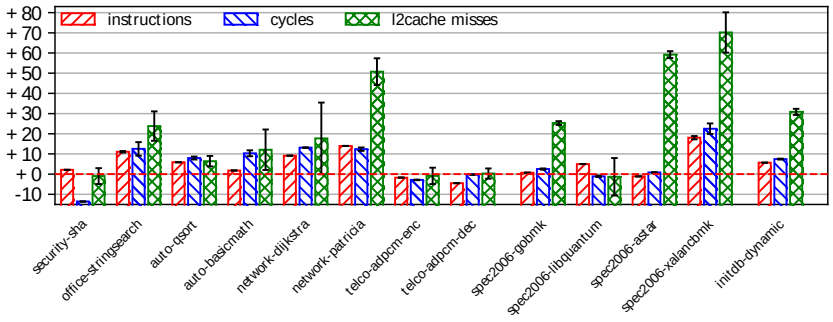
Capabilities Limited. *Assessing the Viability of an Open Source CHERI Desktop Software Ecosystem*. (2021)

30

Everybody likes screenshots, right? So, here's KDE and some of its applications running fully CHERIfied on RISC-V, in qemu over VNC.

This all runs on Morello, too, and Morello has a GPU and drivers, so, it will, Real Soon Now™, be a viable workstation! (Just a little bit more bring-up to do.)

Performance Overhead Measurements



- As of ASPLOS'19, on earlier CPU in FPGA:
  - 0 - ~10% cycle (= wall clock) overheads in most cases
- L2 cache misses increase for pointer-heavy workloads from increased pointer size
- Many caveats in both directions with these numbers
  - Morello will give us better understanding; work in progress!

Davis et al. *CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment*. (2019) 31

(@24m00)

The fact that all of that works is, I think, fairly exciting, but it turns out that there's much more to be gained from CHERI. An active area of research is exploring how CHERI can be used to *compartmentalize* software, confining the impacts of *arbitrarily bad behavior* in one compartment to just that part.

Research: Building a CODEC Sandbox

- Can *build* confined pieces of software with access to only particular resources
  - Without a (transitive) capability to a given resource, no way to access it! (Even if address known!)
- Sandboxing CODECs (`gzip`, `libpng`, …) is an attractive idea! If *all* we give some CODEC code is…

Resource	Permissions
CODEC code (& constants)	Read, Execute
Input buffer(s)	Read-only
Output buffer(s)	Write-only
Ephemeral stack / scratch region	Read, Write
Return pointer	Execute only?

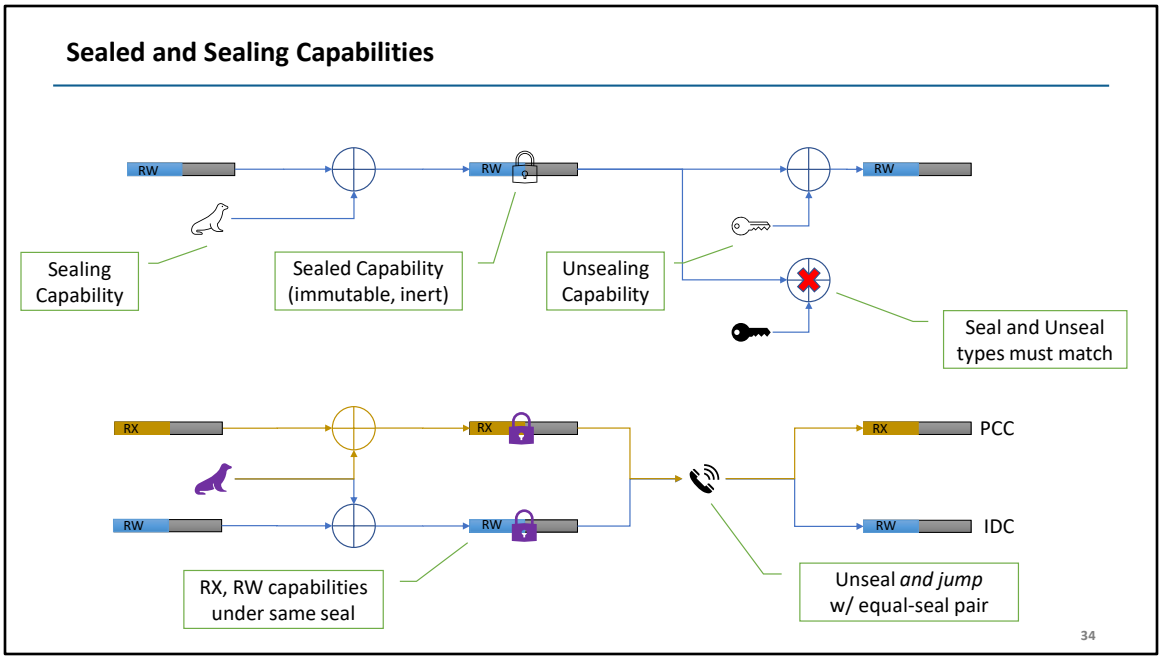
  - … then even a fully compromised CODEC has very limited consequence on the broader program!
- Entering sandbox is easy; getting back out might be tricky?

33

Capabilities let us build confined pieces of software, with guaranteed-at-construction limits on the resources that can be accessed. If there is no path in the capability graph from the register file to a given resource, then that resource is inaccessible, even if we know its address.

1. We can envision a kind of capability-based sandboxing mechanism, say, for CODECs like gzib and friends. If the only resources they have are [as per table], then even arbitrary code vulnerabilities here are very weak: controlled input gives rise to controlled output.

2. On the other hand, it's pretty easy to *get into* a sandbox like that, but how do you get back out to a *more permissive* context? That is, how do you prove that you're allowed back out?



One answer, it turns out, is to enrich CHERI with *additional kinds of capabilities*. We'll consider two, here: *sealed* capabilities as well as *sealing* (and unsealing) capabilities. At last, we'll come to talk about some of that other metadata inside the capability structure I alluded to way back in the intro.

1. A CHERI capability can be combined with a *sealing* capability to produce a *sealed* capability. These are immutable (try, and you'll clear the tag) and *inert*, in that they do not authorize other operations.

2. You can only pass around, or drop, a sealed capability until it is combined with an *unsealing* capability to reveal the original input, which can now be used as it was.

3. The sealing and unsealing capabilities must match: if they don't, unsealing fails.

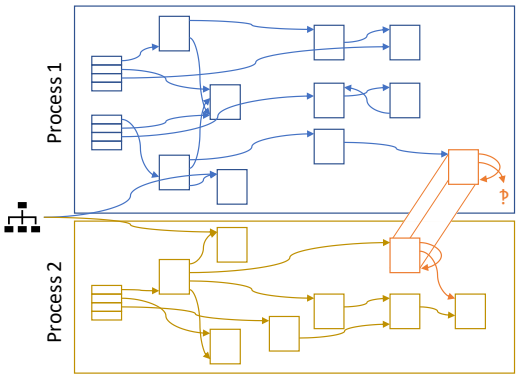
   These kinds of capabilities are useful to represent resources that we want software to be able to *reference* but not *directly use* without executing code that has access to the appropriate unsealer.

4. Expanding on that mechanism, we can also do something interesting if we

construct *two* sealed capabilities with the same seal.

5. We can *invoke* the sealed pair, handing them both to an instruction, which will unseal and jump. You can think of this in an OOP way: the sealed capabilities represent an object's *data* and a *method* that we want to call on that object. Or we could use this to exit a sandbox in a very continuation-passing style: the data is the outer context's continuation's data, and the method is the continuation's code.

Research: Collocation: Multiple Processes In One Address Space



- MMU-based isolation
  - Programs in separate address spaces
- IPC by context switch
  - Data copy by *kernel* (write/read on pipe)
  - TLB switching costs
    - Flush (time, power) or ASIDs (area, power)
- Selective sharing through *shared pages*
  - Pointers *to* shared memory: fine
  - Pointers *in* shared memory: … carefully
  - Pointers *from* shared memory: WTF???

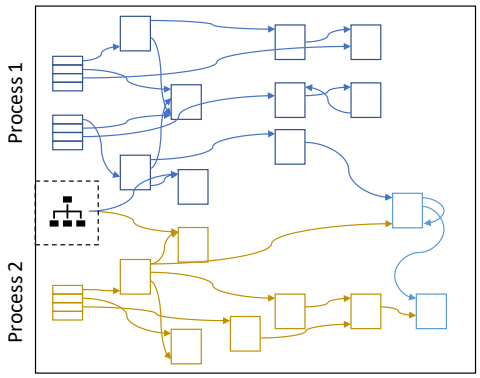
35

Another thing we can revisit with CHERI is the need for process isolation. Traditionally, processes live in different address spaces, and we use the MMU to isolate them.

1. If we want to do IPC, we context switch between the two…
2. And probably have the kernel copy some data for us.
3. This incurs TLB switching costs, in time, power, and/or silicon area.

4. We could also establish shared memory pages between the two address spaces, but there's something funny here.
5. Pointers *to* the shared region are fine…
6. Pointers *in* the shared region are dubious, but can work if we're careful…
7. But pointers *from* the shared region are probably nonsense.

Research: Collocation: Multiple Processes In One Address Space



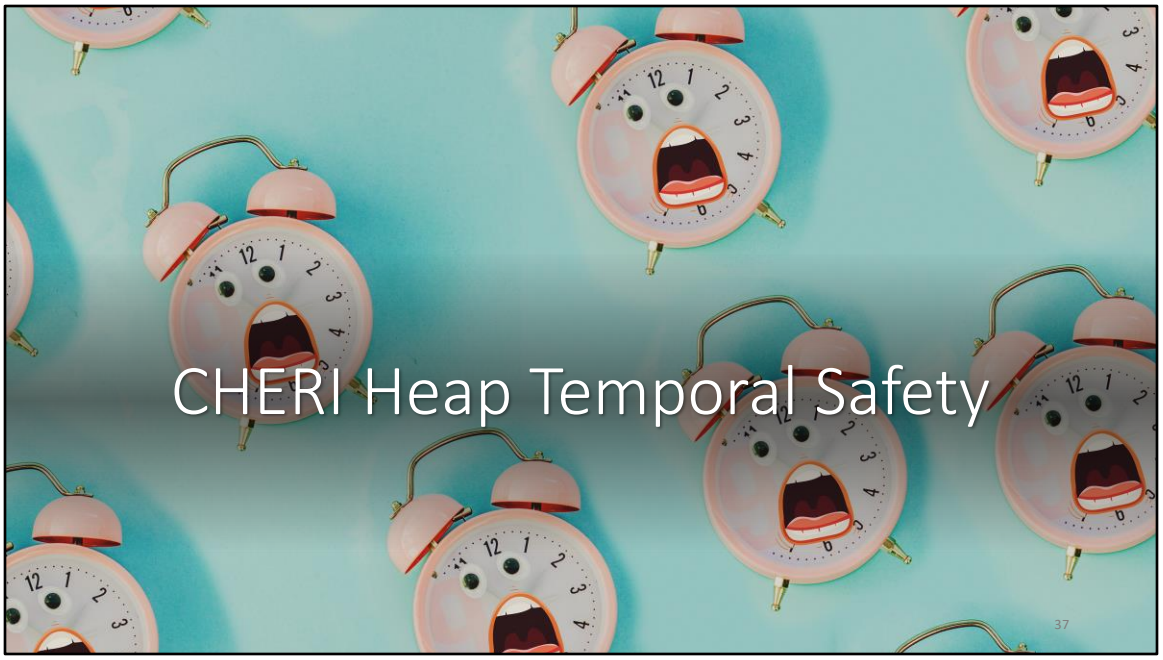
- *Collocated Processes*
  - Many programs in *one* address space
  - Isolation maintained with CHERI
- IPC by sealed capabilities
  - Copy on call through "trusted switcher"
  - Kernel-bypass for IPC!
- Really fast sharing: pass capability across IPC
  - No *interpretation* risk from shared pointers

36

CHERI lets us tear down the MMU-based walls between processes, so that we can run many processes in a single address space!
Isolation is maintained by the capability system: you can't access what you can't point at *with a capability*.

1. In this model we can do IPC through sealed capabilities, and, if we want copy semantics, we can use a trusted switcher that does that copying before completing the call. This is *kernel-bypass IPC*, with user threads directly crossing the traditional process boundary!

2. We get really fast sharing in this model, if we just pass a capability across that IPC layer.
3. And there's no risk of mis-understanding an address in the shared region; shared capabilities are just… capabilities.



(@31m00)

I'd like to very quickly touch on the part of the CHERI project I'm most directly involved with: investigating the use of CHERI for *temporal* safety.

Use After Free?

What about use-after-free?

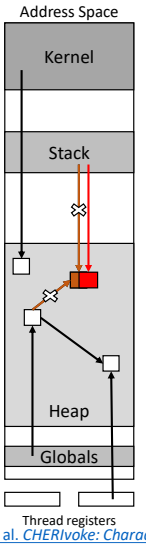
```
char *p = malloc(1024);
free(p);
char *q = malloc(1024); // == p (!)

strcpy(p, “oh no”);
```

38

Another way of phrasing temporal safety is “What about UAF?” It’s a reasonable question. After all, even having gone through and made pointers into capabilities at runtime, it’s still possible to use a capability after free-ing it.

Capability Revocation



- Focused on *heap* temporal safety
  - More complex lifetimes than stack objects, resist static approaches
- Heap pointers end up in globals, stacks, registers, kernel heap, …
- Risk: retain references to `free()` object, overlap new allocation
- Eliminate "use-after-reallocation" by *revoking* dead references
  - UAF still possible, but accesses old object
- "Dual" of garbage collection: (lazily) enforce `free()`

Xia et al. *CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety*. (MICRO 2019)

39

Focusing on temporal safety of *heap* allocation, since heap objects have a wider variety of, and more complicated, lifecycles than stack objects. That's not to say that stack temporal safety is trivial, just momentarily out of scope.

As part of those complex life cycles, pointers into the heap tend to spread: into other heap objects, globals, onto the stack, and even into the kernel heap (for example, as part of asynchronous I/O).

1. That means that there's a risk that the application inadvertently retains a reference to a free object, which then comes to overlap a new allocation. This is, of course, undefined behavior in C, but that doesn't mean it doesn't happen.

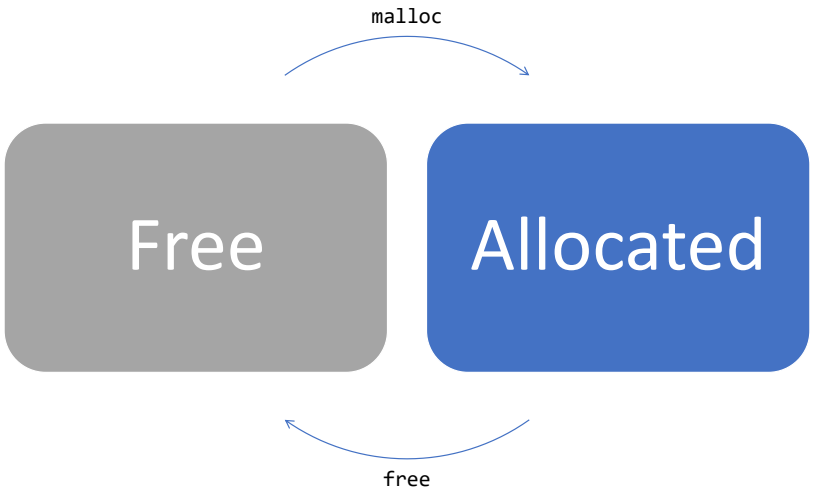
   This opens the possibility of "use after reallocation" wherein a stale reference is dereferenced and accesses or corrupts the new object, often either exposing data or corrupting data structures.
   ("Use after free", before addresses are repurposed, is typically less of a concern.)

2. The basic approach we're considering is to *revoke* these stale references, and be

sure that we have eliminated them all, before reusing address space.

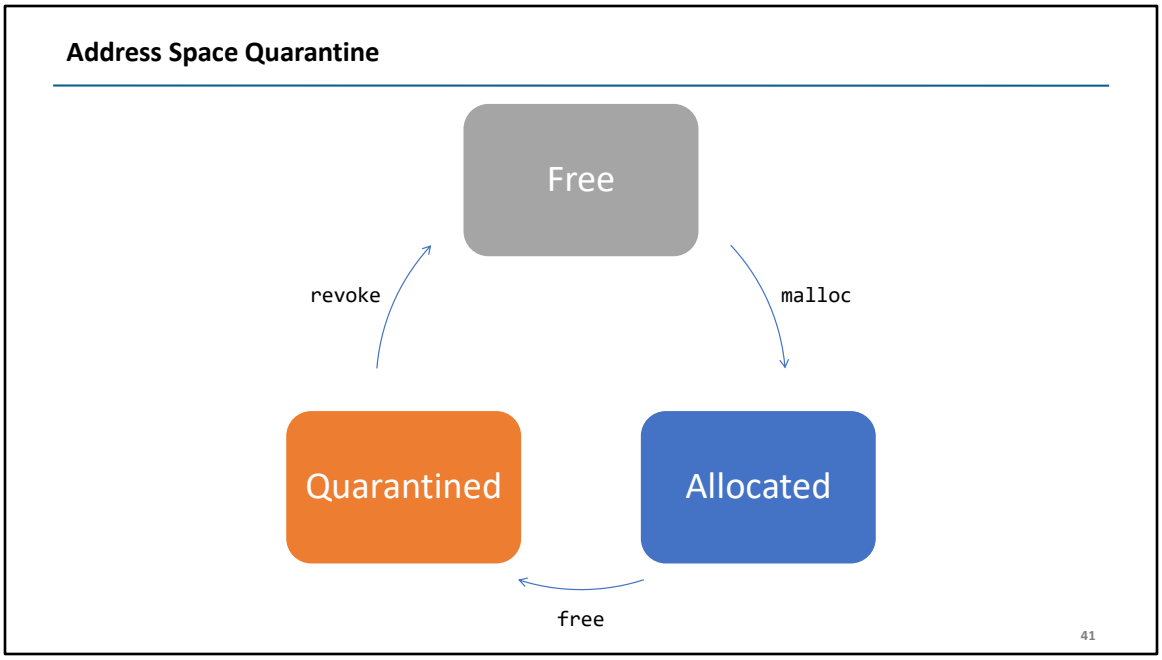
3. This is the dual of garbage collection: we *destroy* references to enforce free(), rather than extend object lifetime until there are no references.

Address Space Quarantine



40

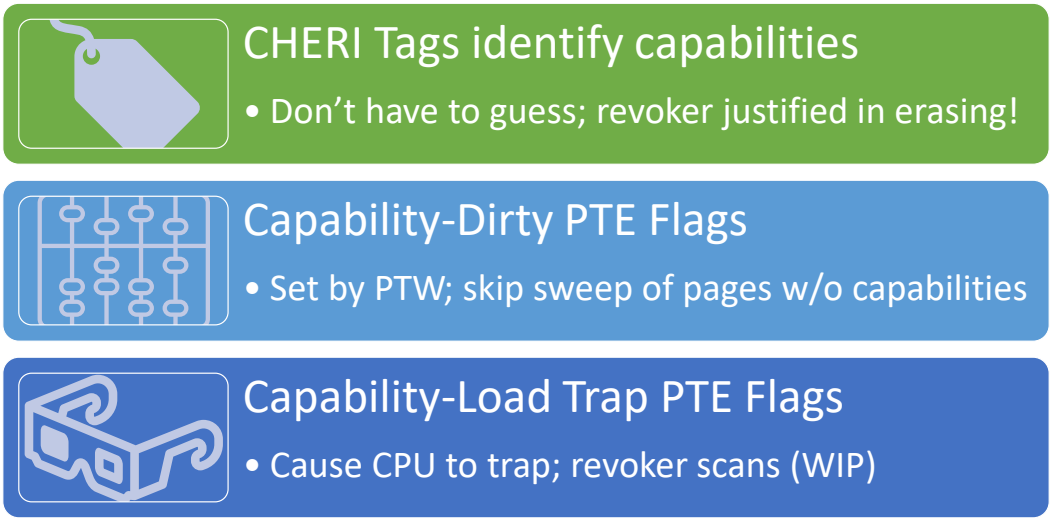
To pull this off, we're going to expand the usual view of heap memory, in which things are either free or allocated and just bounce back and forth…



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 system, we allow quarantine to accumulate for a while and make each revocation pass process a *batch* of quarantined address space at once.

Architectural Acceleration for Revocation



Filardo et al. *Cornucopia: Temporal Safety for CHERI Heaps*. (Oakland 2020)

42

This turns out to be feasible for CHERI *because* it is a capability architecture. We don't have to guess whether words are pointers to objects or just suspicious numbers, and since we know with certainty, we are justified in erasing capabilities.

Beyond merely being possible, it turns out we can add just a little bit of architecture to speed things up significantly: we can have the CPU assist us in tracking which pages have capabilities on them (so we don't need to sweep the ones that don't), and we can avoid stopping the world by arranging for the CPU to trap when it tries to load a capability through a page we haven't yet scanned.

Revocation Performance?

- Cornucopia, SPEC CPU2006, no cap-load traps, revoke @ 33% heap in quarantine:

	Geomean	Worst case
Revocation on 2nd core	**2.4%**	7.9%
Single core	5.2%	21.2%

- Using load traps (WIP):
  - Lowers overheads by ~10%
  - Significantly improves (nearly eliminates) application pause times
- Additional software and architecture tricks up our sleeves to tamp down on costs

(@35m30)

The last thing I'd like to touch on today is: is CHERI in competition with safe languages like Rust? If you know Betteridge's law of headlines, you already know the answer is "no".

With thanks to David Chisnall for his help with this section.

Safe Architecture vs. Safe Languages?

"OK, yes, everything's on fire, but …"

… it's all C's fault!

Safe languages solve all these problems!

Why do we need CHERI?

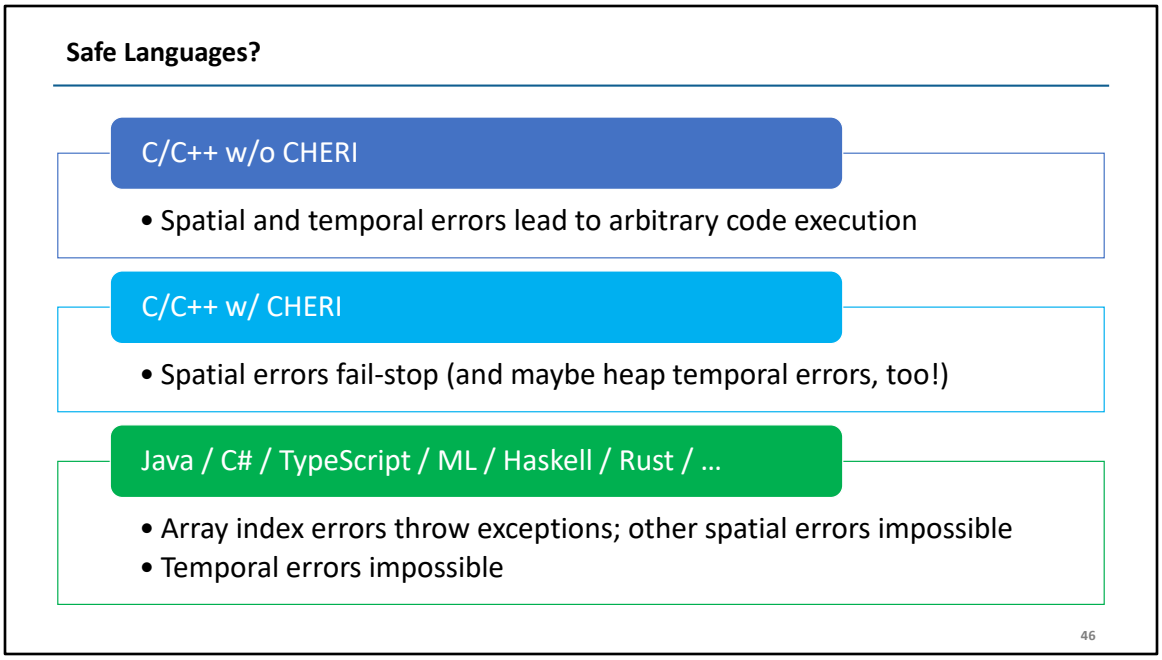
… it's all the architecture's fault.

CHERI fixes that!

Why do we need safe languages?

45

Depending on which side people think they're on, this supposed competition is phrased in one of two ways.

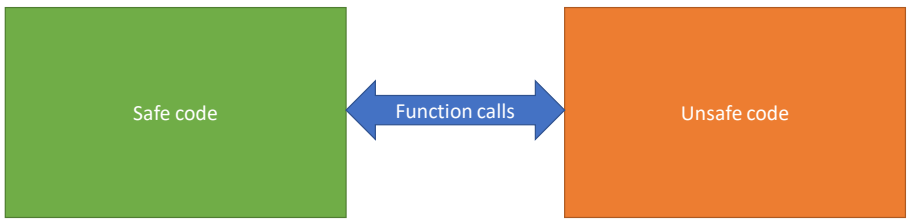


Why might people think there's competition between the two efforts? Let's look in a little more detail.

Rewrite Everything to be Safe?

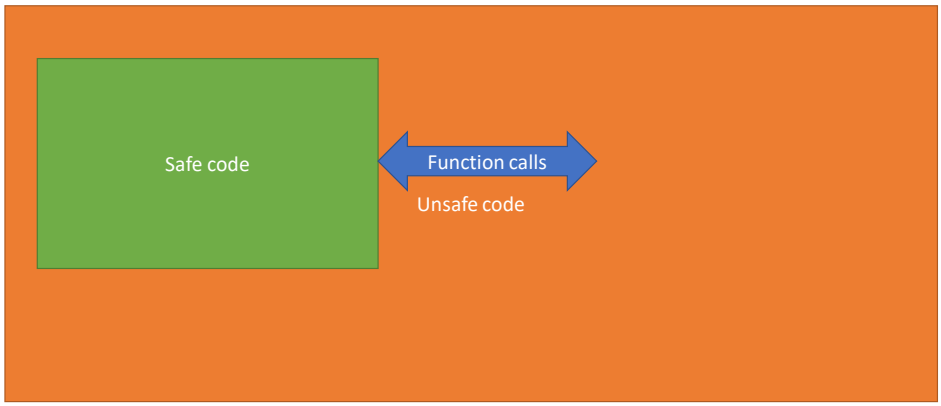
- OpenHub.net estimates ~10B LoC of C, ~3B LoC of C++ just in the open world.
  - That probably works out to $130G - $1.3T to rewrite everything.
- TCB code is *intrinsically unsafe* (sit below safe language abstraction)
  - Memory managers, garbage collector, context switcher, …
- Different safe language runtimes likely view each other as *unsafe*!
- Rewrite *parts* of programs?

A two-worlds abstraction



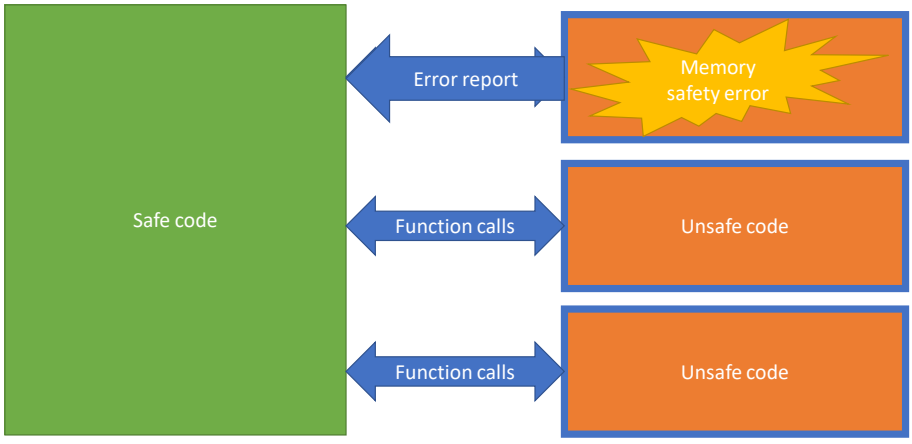
When we think about rewriting part of a program in a safe language, we have a two-worlds mental model: a safe world for the new safe code, which communicates with an unsafe world with some well-defined interfaces.

A two-worlds abstraction



Unfortunately, in the real world, the safe code is inside the unsafe world and any memory safety bugs in the unsafe code can violate all of the invariants that the safe-language code depends on.

A safe many-worlds abstraction



Chisnall et al. CHERI JNI: Sinking the Java security model into the C. (ASPLOS 2017)

CHERI sandboxes provide us with a mechanism to confine memory safety errors to instances of unsafe code. We can catch CHERI's architectural traps and turn them into error reports or exceptions for the safe language, which can then gracefully recover, because the error cannot have corrupted the safe world.

CHERI + (Unsafe) Rust

- Recently, Rust community has been fretting about semantics of unsafe Rust.
  - Compiler transformations threatening correctness
- Recent proposal to use CHERI-like "strict provenance" semantics!
  - No integer-to-pointer casts, trivially "NPVI" semantics
  - Distinguish `usize` from `uaddr` from `uptr`?
  - Integers must be *recombined* with pointers: address from integer but *provenance* from pointer
- Unsafe strict provenance Rust code should be less unsafe on CHERI!

Aria Beingessner. *Rust's Unsafe Pointer Types Need An Overhaul*. (2022) Tracking Issue for strict_provenance on GitHub

51

I'd just like to give a shout out to the Rust community where, for not entirely unrelated reasons, there is already a bit of a move towards a very compatible story.

CHERI Summary

- CHERI enriches CPUs to have tagged capabilities with architecturally-enforced invariants
  - Addresses many root causes of long-standing security vulnerabilities
  - Promising new compartmentalization mechanisms
- Looks quite real: FPGA RISC-V & Arm Morello SoC, LLVM, CheriBSD, Qt, KDE, …
- If you want to know more, please do get in touch:
  - http://www.cheri-cpu.org/ for (much) more reading material, slack, email lists, &c
- Play along at home, too; almost everything is FLOSS:
  - https://github.com/CTSRD-CHERI/cheripedia/wiki/Getting-Started a how-to
  - https://github.com/ctsrd-cheri/cheribuild one-stop-shop cross-build system
  - https://github.com/CTSRD-CHERI/cheri-exercises hands-on introductory exercises

52

(@40m00)

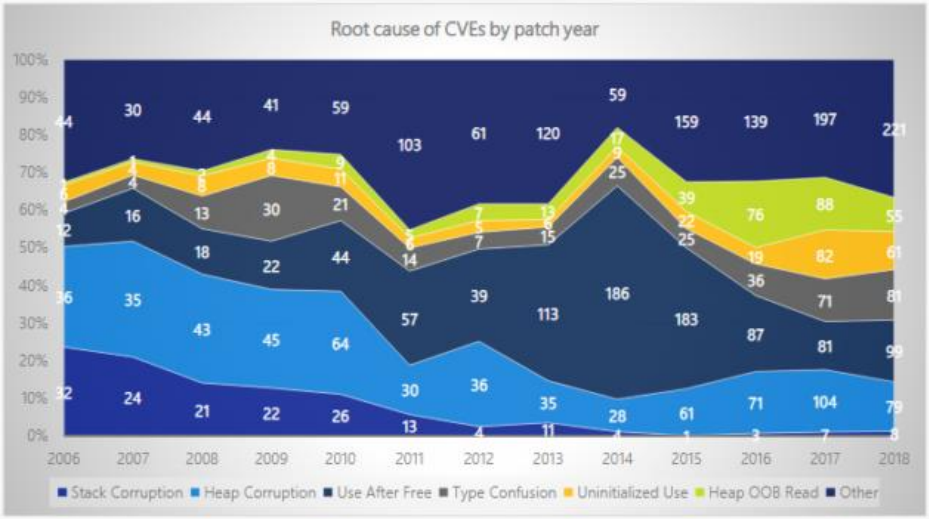
Additional Reading

Links on bottom left of many slides to relevant material. In addition,

- Davis et al. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. (Extended report, 2019).
- Esswood. CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor. (2021).
- Watson et al. Balancing Disruption and Deployability in the CHERI Instruction-Set Architecture. (2017).
- Watson et al. CHERI Instruction-Set Architecture (Version 8). (2020).

- Levy. Capability-Based Computer Systems. (1984).

CVEs and High Severity Bugs from (Lack of) Memory Safety

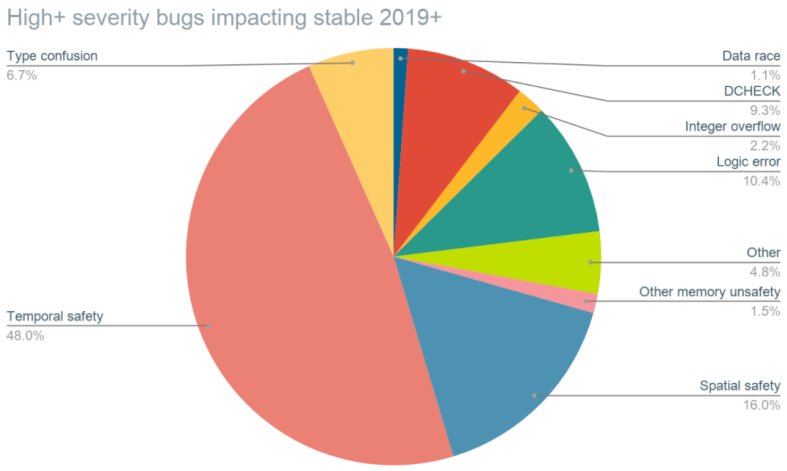


Matt Miller. *Trends, challenge, and shifts in software vulnerability mitigation.* (BlueHatIL 2019)

54

Sources: Matt Miller BlueHat 2019

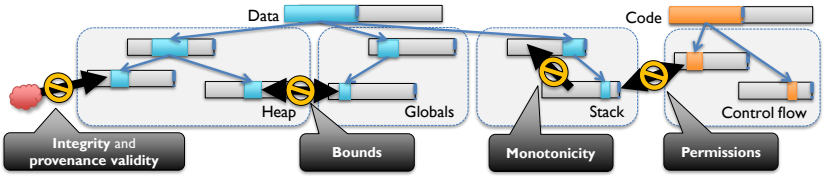
CVEs and High Severity Bugs from (Lack of) Memory Safety



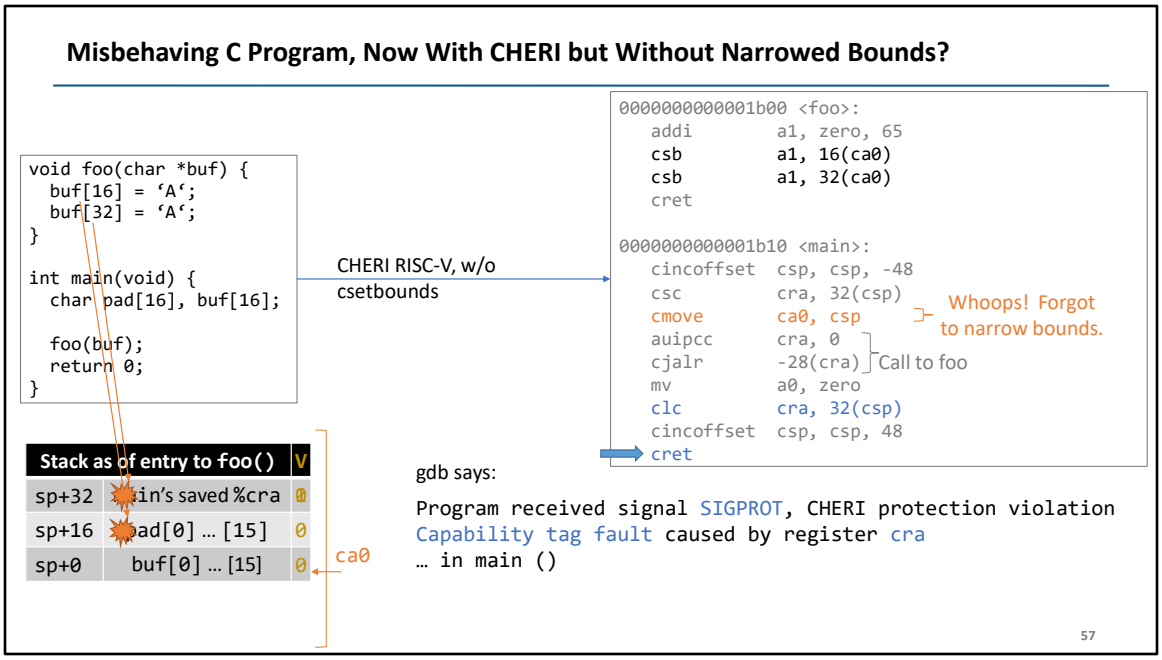
Google. *Safer Usage of C++.* (2021)

55

CHERI enforces protection semantics for pointers



- **Integrity** and **provenance validity** ensure that valid pointers are derived from other valid pointers via valid transformations; **invalid pointers cannot be used**
  - Valid pointers, once removed, cannot be reintroduced solely unless rederived from other valid pointers
  - E.g., Received network data cannot be interpreted as a code/data pointer – even previously leaked pointers
- **Bounds** prevent pointers from being manipulated to access the wrong object
  - Bounds can be minimized by software – e.g., stack allocator, heap allocator, linker
- **Monotonicity** prevents pointer privilege escalation – e.g., broadening bounds
- **Permissions** limit unintended use of pointers; e.g., W^X for pointers
- These primitives not only allow us to implement **strong spatial and temporal memory protection**, but also higher-level policies such as **scalable software compartmentalization**

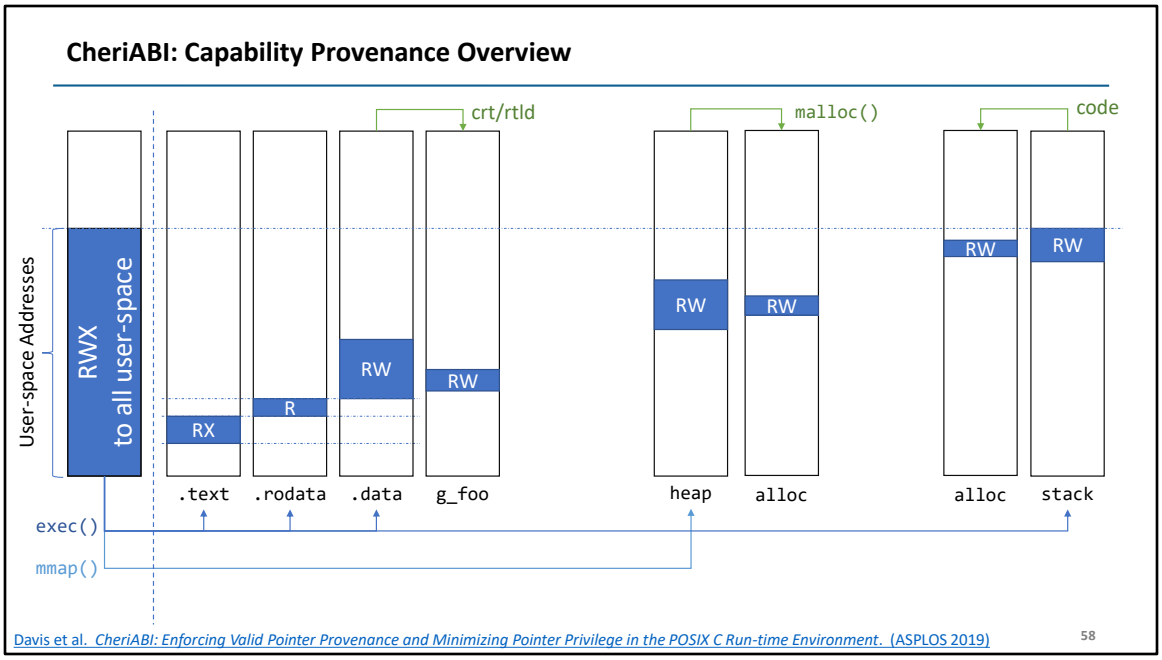


What happens if we forget to narrow the stack capability? Then…

1. The stores now happen without trapping

2. The store over the saved return address *clears the valid bit*

3. The program still crashes, just later, when main goes to return.
   When we reload the saved return address, it won't be a valid capability.
   When we try to jump to that, the processor will trap.



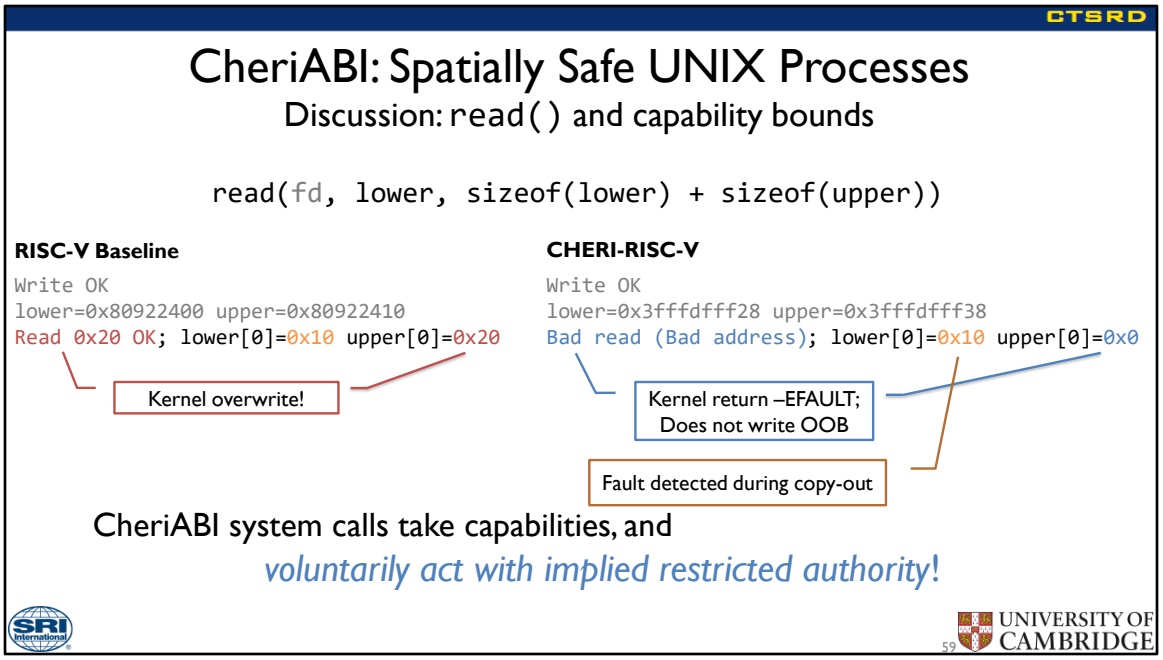
So, where does a process get its very first capability?
User capabilities originate from the kernel, which holds a capability to all of userspace with full permissions.
On exec(), the kernel constructs subset capabilities for the program sections (text, rodata, data and bss) and the initial stack.
It uses the .text capability to jump to the entrypoint (so that capability ends up in the program counter) and installs the initial stack capability in the stack capability register; .rodata and .data put somewhere agreed upon.

1. As the program runs, it can refine its own capabilities, as we saw it do with an on-stack array allocation.

2. If the program calls malloc(), malloc() calls mmap() and retrieves a capability to the new pages…
3. and then derives subset capabilities for each allocation.

4. We can even have the startup code or program loader narrow the capabilities used to access globals inside .data or .bss, so that the program can't access out of bounds there, either.

(The details here are shockingly involved, but I think it's important to note that this isn't a hole in CHERI's defenses.)



What happened when we asked the kernel to write out of bounds?

1. On the baseline architecture, the kernel had no way of knowing what our bounds were, so it took our word for the length of the structure.

2. With CHERI, the system call passed *a capability* to the kernel, and the kernel *used that capability* when writing to user memory, rather than using its own elevated authority.

3. [as it says]

4. In fact, we can see that the kernel only noticed the discrepancy between capability length and request length after it had started copying data to the user program. *nix-like kernels generally have well-defined points where they copy data in from or out to the user program; making these use capabilities is enough to catch these kinds of misbehaviors of userspace, and relatively little of the read() path needs to be altered from what was already there.

Sealed Entry Capability Flavors

- CHERI also defines some flavors of “sentry” (“sealed entry”) capabilities which unseal in jumps.
- Single capability, becomes PCC when unsealed
  - useful for function entry, return addresses
  - PCC-relative data inaccessible to caller
- Pointer to PCC, becomes IDC when unsealed, PCC loaded from target
  - PCC- and IDC-relative data inaccessible to caller
  - “pointer to intrusive v-table”
- Pointer to pair, PCC and IDC loaded
  - PCC- and IDC-relative data inaccessible to caller
  - “proxy for method and instance”

CheriBSD Code Changes

Area	Files total	Files modified	% files	LoC total	LoC changed	% LoC
Kernel	11,861	896	7.6	6,095k	6,961	**0.18**
• Core	7,867	705	9.0	3,195k	5,787	**0.18**
• Drivers	3,994	191	4.8	2,900k	1,174	**0.04**
Userspace	16,968	649	3.8	5,393k	2,149	**0.04**
• Runtimes (excl. libc++)	1,493	233	15.6	207k	989	**0.48**
• libc++	227	17	7.5	114k	133	**0.12**
• Programs and libraries	15,475	416	2.7	5,186k	1,160	**0.02**

Notes:

- Numbers from cloc counting modified files and lines for identifiable C, C++, and assembly files
- Kernel includes changes to be a hybrid program and most changes to be a pure-capability program
  - Also includes most of support for CHERI-MIPS, CHERI-RISC-V, Morello
  - Count includes partial support for 32 and 64-bit FreeBSD and Linux binaries.
  - 67 files and 25k LoC added to core in addition to modifications
  - Most generated code excluded, some existing code could likely be generated

DSbD Consortium Update. (2021/05)

61

Clang/LLVM/LLD Code Changes

Area	Files total	Files modified	% Files	LoC total	LoC changed	% LoC
LLVM	4220	44	1.0	1656k	217	**0.013**
Clang*	1593	30	1.9	911k	190	**0.021**
LLD	249	5	2.0	67.8k	30	**0.044**
Total	6062	79	1.3	2365k	432	**0.018**

Notes:

- Changes predominantly (u)intptr_t vs size_t/ptrdiff_t confusion, static_asserts about struct sizes/layouts no longer true with 128-bit pointers, and a few instances of using uint64_t for pointers
- Able to compile and link a pure-capability C hello world natively on CHERI-RISC-V
- (*) One outstanding known issue in the frontend prevents compiling a C++ hello world
  - Implementation and header files in question only total an additional 193 lines, or 0.021%, as a worst-case upper bound
- Just over half the Clang changes (99 LoC) are for its bytecode-based C++ constexpr interpreter

62

(From CL-internal presentation)

WebKit - JSC Code Changes

Area	Files total	Files modified	% Files	LoC total	LoC changed	% LoC
JSC-C	3368	148	4.4	550k	2217	**0.40**
JSC-JIT	3368	339	10.1	550k	7581	**1.38**

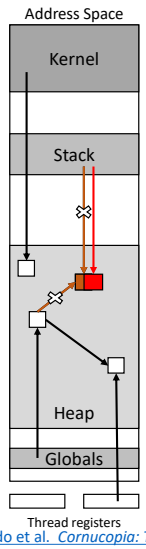
Notes:

- JSC-C is a port of the C-language JavaScriptCore interpreter backend
- JSC-JIT includes support for a meta-assembly language interpreter and JIT compiler
- Runs SunSpider JavaScript benchmarks to completion
- Language runtimes represent worst-case in compatibility for CHERI
  - Porting assembly interpreter and JIT compiler requires targeting new encodings
- Changes reported here did not target diff minimization
  - Prioritized debugging and multiple configurations (including integer offsets into bounded JS heap) for performance and security evaluation
  - Some changes may not be required with modern CHERI compiler

DSbD Consortium Update. (2021/05)

63

Cornucopia: Heap Temporal Safety Atop CHERI
Address Space Quarantine, Revocation



- Focused on *heap* temporal safety
  - More complex lifetimes than stack objects, resists static approaches
- Heap pointers end up in globals, stacks, registers, kernel heap, …
- Risk: retain references to `free()` object, overlap new allocation
- Eliminate "use-after-reallocation" by *revoking* dead references
  - UAF still possible, but accesses old object
- Hold address space in *quarantine* to amortize sweep cost
  - Quarantine state held *out of band*
- "Dual" of garbage collection: (lazily) enforce `free()`

Filardo et al. *Cornucopia: Temporal Safety for CHERI Heaps*. (Oakland 2020)

64

We've been considering temporal safety of *heap* allocation, since heap objects have a wider variety of, and more complicated, life cycles than stack objects. That's not to say that stack temporal safety is trivial, just momentarily out of scope.

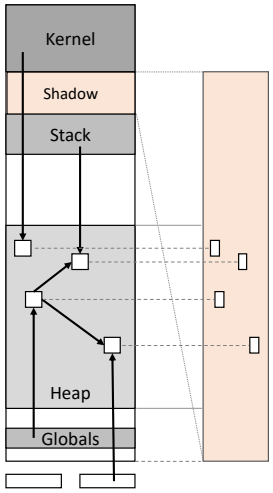
As part of those complex life cycles, pointers into the heap tend to spread: into other globals, onto the stack, and even into the kernel heap (for example, as part of asynchronous I/O).
That means that there's a risk that the application inadvertently retains a reference to a free object, which then comes to overlap a new allocation.
This is, of course, undefined behavior in C, but that doesn't mean it doesn't happen.

This opens the possibility of "use after reallocation" wherein a stale reference is dereferenced and accesses or corrupts the new object, often either exposing data or corrupting data structures.
("Use after free", before addresses are repurposed, is typically less of a concern.)
The basic approach we're considering is to *revoke* these stale references, and be sure that we have eliminated them all, before reusing address space.

Revocation is global and involves testing every capability in the process, so we

quarantine

Cornucopia's Kernel Revocation Service



- Kernel offers revocation *service* to user programs
- Exposes "shadow bitmap"
  - Encodes live/free state of memory, 1 bit per 16 bytes
- Deletes capabilities *to* addresses with set bits
  - Promises to inspect itself as well as program memory
- Thread-safe & mostly concurrent implementation

Filardo et al. *Cornucopia: Temporal Safety for CHERI Heaps*. (Oakland 2020) 65

We have an implementation of this idea in CheriBSD, which we call Cornucopia.

The *kernel* offers a revocation service.
It builds on CHERI's typed view of memory (and registers); CHERI precisely captures pointer/non-pointer distinction via tags.
It relies on CHERI's spatial safety, and the bounds set by the allocator, to map capabilities to shadow bits.

1. Exposes a "shadow bitmap", a kernel-provided mapping of anonymous memory within the process's address space.

2. A system call requests that the kernel apply the shadow bitmap to the address space, deleting all capabilities pointing into freed memory, including those held by the kernel.

3. Cornucopia is a thread-safe and mostly concurrent implementation; subsequent work leans further into concurrency.

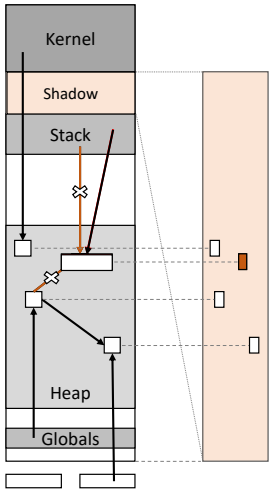
---

1. Allocators mmap() their heap and can ask the kernel for access to the corresponding region of the shadow.

2. 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 double-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.

Cornucopia's User-space: Quarantine & Revocation Batches



- Application `free()`-s object, might retain references. Risk: allocator creates new, overlapping object.
- Instead: quarantine space, set bits in shadow.
- Eventually, ask kernel to sweep to revoke access.
  - Sweep phase of M&S collector, w/ `free()` doing the marking.
- Now safe to re-issue memory.

66

Heap temporal safety violation contains: application free()-s but retains pointer and allocator creates overlapping object.
Cornucopia gives allocators a mechanism to know when there are no more references to an address, and so reuse is safe.

In more detail, on free, the allocator marks the shadow of the freed object and holds the address space in quarantine, to amortize the cost of sweeping revocation.

1. When quarantine fills, allocator calls revoker
2. Which deletes capabilities to addresses with marked shadows
3. After revocation finishes, it is safe to reuse address space
4. The allocator has to clear the shadow before reissuing memory, lest the revoker strike again

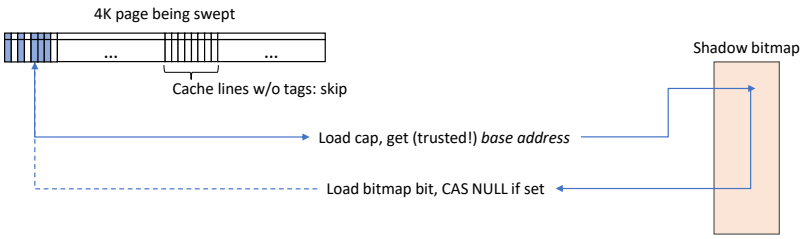
---

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 double-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.

Cornucopia Architecture
Per-Page Sweep in More Detail



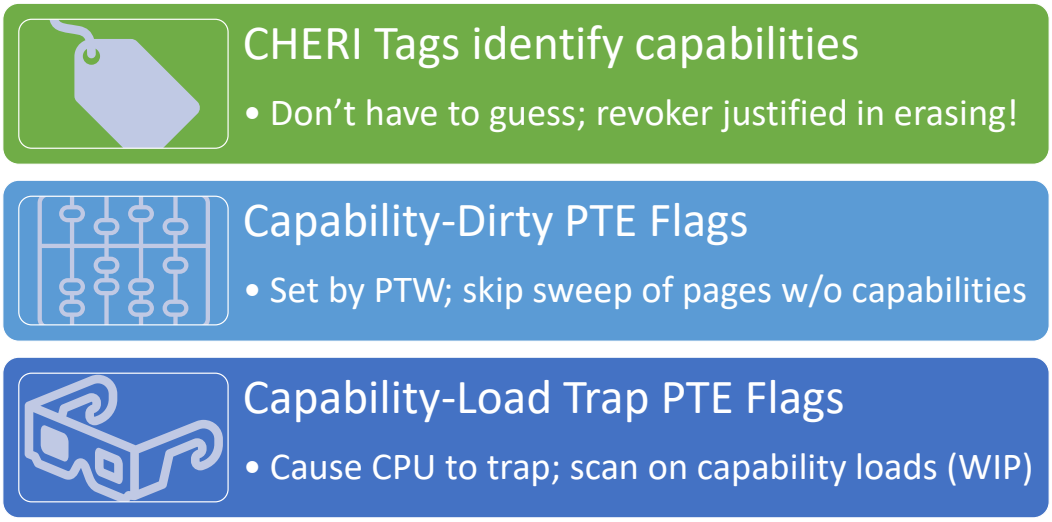
Filardo et al. *Cornucopia: Temporal Safety for CHERI Heaps*. (Oakland 2020)

67

For each capability, probe the shadow bitmap based on the *trustworthy* capability base value. (Recall: monotonicity ensures that base *must be in bounds* of original allocation.)

4) Have added CLoadTags instruction to coherently fetch CHERI tags w/o fetching data; lets us skip entire lines w/o capabilities.

Architectural Acceleration for Revocation



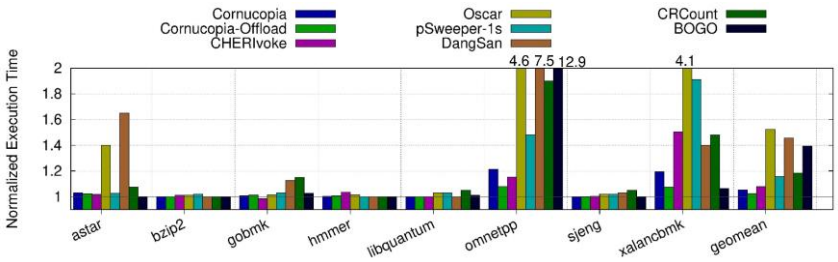
Filardo et al. *Cornucopia: Temporal Safety for CHERI Heaps*. (Oakland 2020)

68

This turns out to be feasible for CHERI *because* it is a capability architecture. We don't have to guess whether words are pointers to objects or just suspicious numbers, and since we know with certainty, we are justified in erasing capabilities.

Beyond merely being possible, it turns out we can add just a little bit of architecture to speed things up significantly: we can have the CPU assist us in tracking which pages have capabilities on them (so we don't need to sweep the ones that don't), and we can avoid stopping the world by arranging for the CPU to trap when it tries to load a capability through a page we haven't yet scanned.

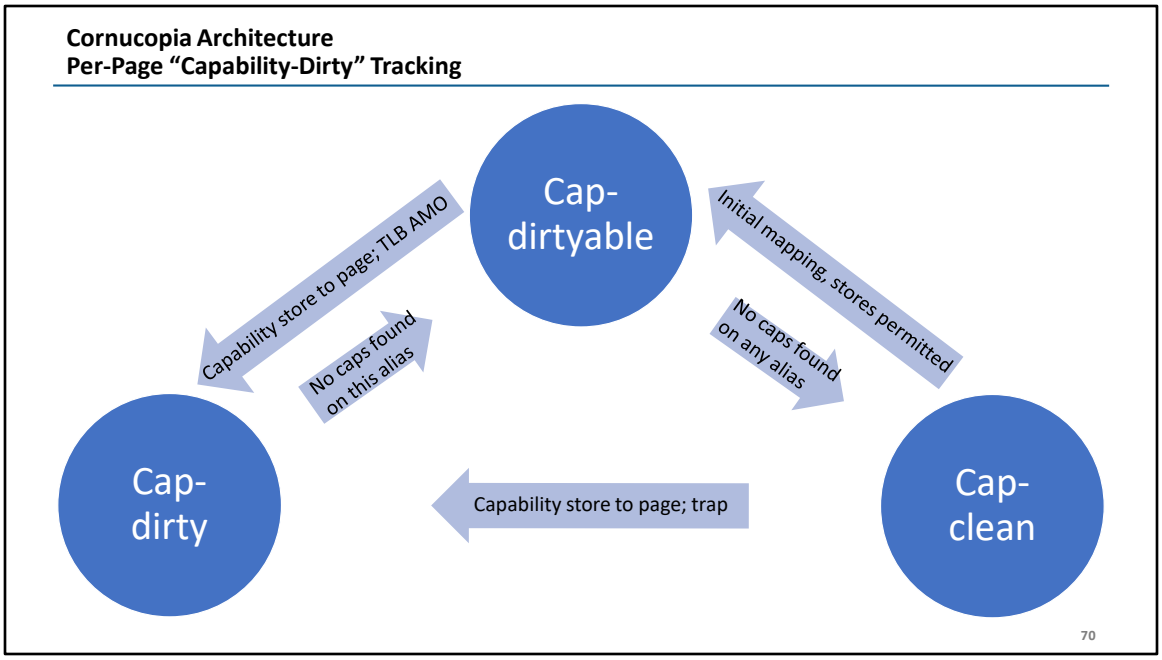
Cornucopia Evaluation: SPEC Cycle Overheads



- Baseline is "spatial safety" assumed (for us, CHERI)
- Initiate revocation at 25% heap in quarantine

- Worst case is omnetpp: 21.2% sequential, 7.9% concurrent
- Geomean: 5.2% sequential, 2.4% concurrent

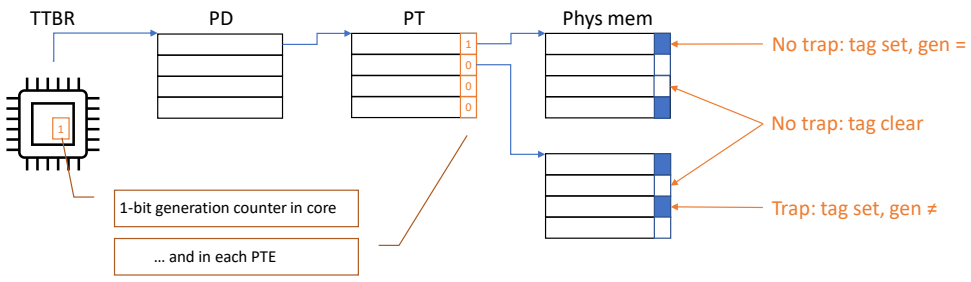
Filardo et al. *Cornucopia: Temporal Safety for CHERI Heaps*. (Oakland 2020)



OK, but how do we know which pages to sweep? Kernel tracks bits of AS that are forbidden to hold capabilities, but within permissive regions, many pages still do not hold capabilities. The revoker should ignore pages that are known to contain no capabilities.

We extend CHERI's page table bits to help track the spread of capabilities, adding a notion of "capdirty", a low-overhead "card-marking" store barrier scheme (though without generational considerations).
PTEs dictate the behavior of cap stores: forbid, permit, or permit subject to automatic, atomic marking.

Cornucopia Architecture
Per-Page Capability Load Generations

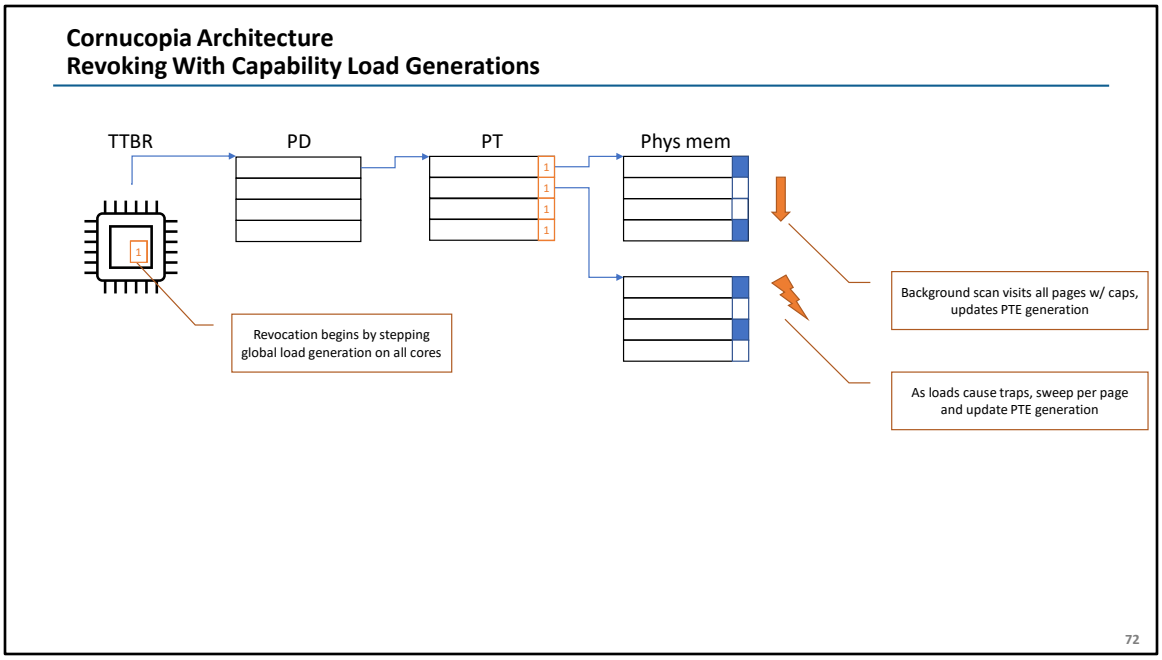


Loads trap if (loaded CHERI tag set) and (core gen ≠ source page PTE gen)

71

First Cornucopia design, published at IEEE S&P 2020, demonstrated viability but suffered from large pause times. As we want to sweep responsively as the application accesses memory (as well as completely, to allow reuse of memory), we have subsequently introduced an architectural notion of *capability load generations*: a per-PTE and per-core bit that must match on capability loads.

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.



Steady state is all generation bits equal

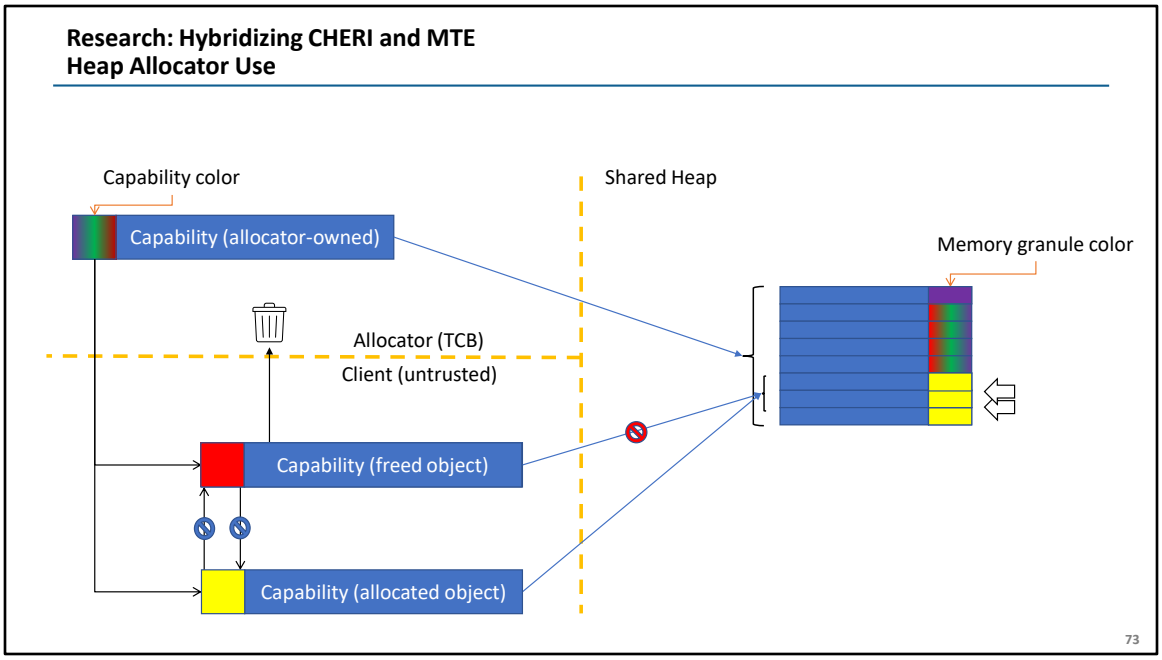
1) Revocation begins by incrementing the in-core generation. Now *all capabilities* is considered untested.

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)



MTE augments each pointer and memory granule with a "color". In CHERI, color bits are part of the capability metadata, and so protected against corruption.

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

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

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

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

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

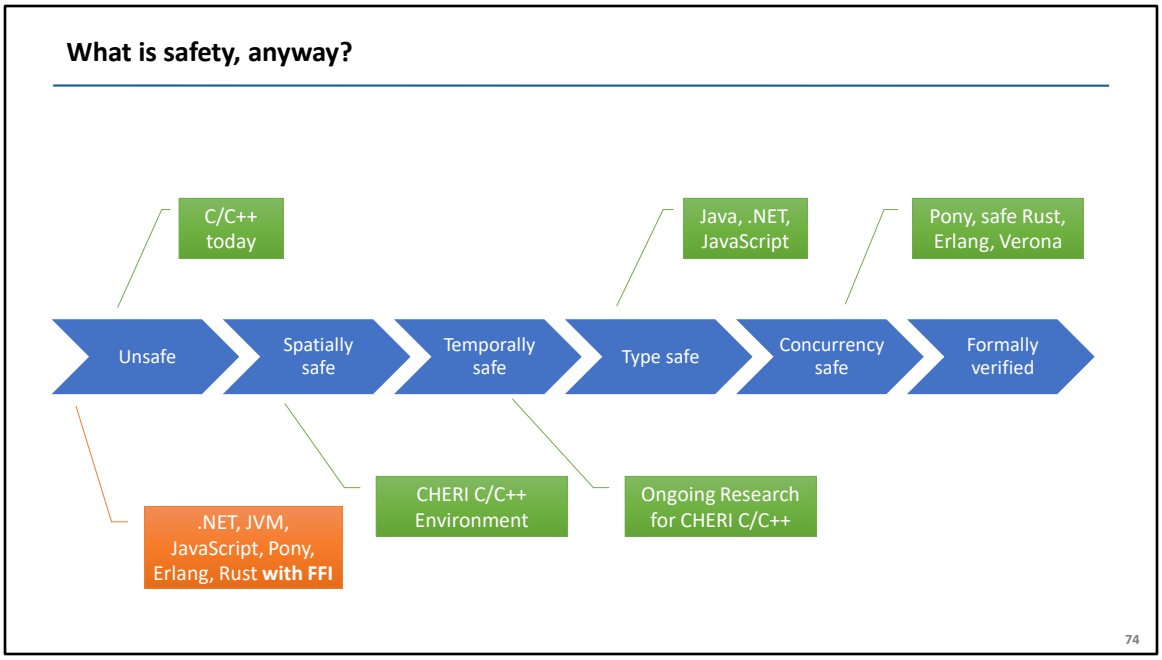
6) Re-allocation proceeds as last time, with the allocator constructing a new capability of the right color for the client. (Any in-band metadata cleared before return.)

7) Clients cannot change the color of their capabilities, nor can they recolor 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 colors mismatch?), but should hide latency of fetch for color comparison.

- A purpose-built atomic compare-and-decrement-color 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



"Language safety" is a spectrum, from completely unsafe (that is, everything is the programmer's responsibility not to break) to fully concurrency safe and formally verified.

1. Very nearly all C today has its pointers lowered to integers as we saw back at the very beginning, landing us firmly in the unsafe bin; maybe it's C's fault and we should rewrite the world in something (anything) else.

2. Candidate languages all bring different things to the table, but notably they all tend to have data representations and runtime systems that (claim to) rule out the kinds of problems we're considering.

3. Some languages go much further than we've discussed, even.

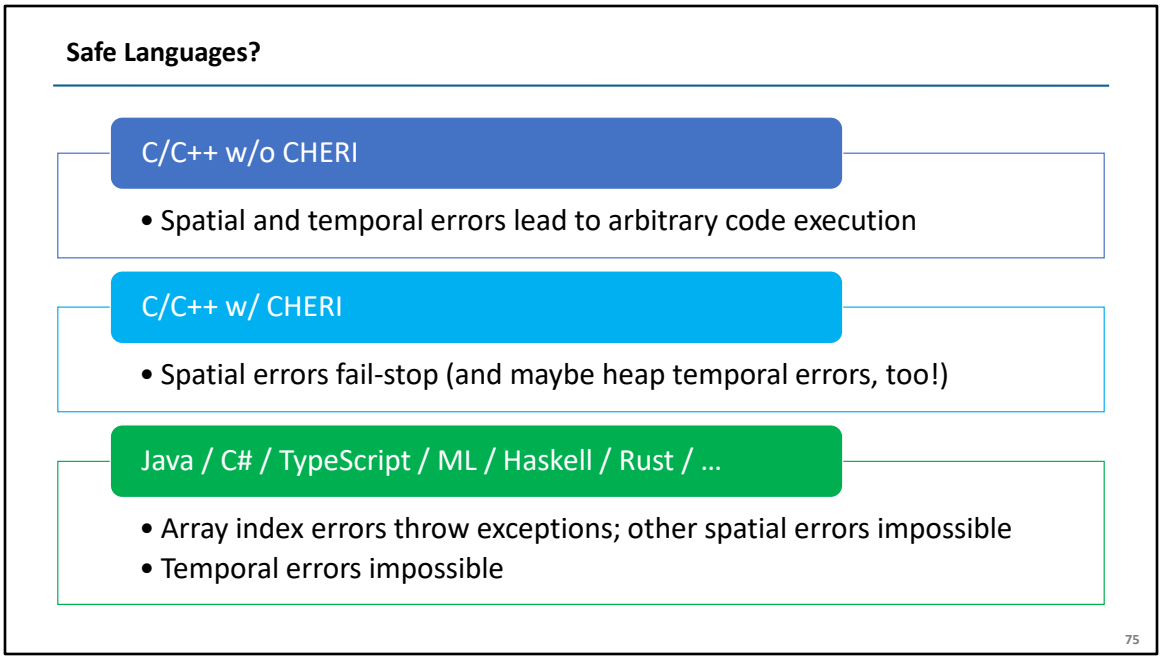
4. There are, however, two small problems with that proposal:
    1. There's *a lot* of C/C++ out there, some of which we might even want to keep using (like optimized libraries). We could introduce FFI bridges for such things, but now we've shot a hole through our runtime system and its defenses.

2. The runtimes for these languages can be *enormous*! They're very often some of the trickiest parts (e.g., garbage collector) and are written in C, and by humans, at that!

1. CHERI aims to give us, at a minimum, a *spatially safe* C/C++ runtime environment. This is the bare minimum for safe composition of software!

2. Ongoing work is working to compositionally and efficiently provide additional safety for our C/C++ environment.

OK, so what?



It's important to talk about what "safety" means.

Safe Architecture *and* Safe Languages?

- On modern architectures, safe languages *cannot safely interoperate*
  - With unsafe languages or among each other
- CHERI brings a new common denominator, especially useful at *boundary*
  - 1-bit tag system distinguishes references from data, prevents reference forgery
  - Spatial safety prevents cross-environment damage (lang A cannot access lang B's heap)
  - Sealing allows safe exchange of references w/ callbacks for invariant enforcement
- "CHERI JNI" work demonstrated extending Java object model into native code
  - C code *had to* call back to the VM to manipulate objects; forced to play by the JVM's rules!

Chisnall et al. CHERI JNI: Sinking the Java security model into the C. (ASPLOS 2017)