Why You Should Care About CHERI

Ben Laurie, Laurence Tratt, Robert N. M. Watson

Introduction

CHERI is a hardware mechanism for fine-grained memory management that is portable to a wide range of CPUs. It has so far been implemented on MIPS-64, 64-bit Arm and 32/64-bit RISC-V (three different open source cores). CHERI *architectural capabilities*, a new hardware data type, can be used by the operating system, runtime, and compiler to constrain future execution.

There are two quite different ways to use CHERI - the first being a software compartmentalisation mechanism that can be used in a similar way to process isolation in existing operating systems and the second being a fine-grained in-process memory safety mechanism that may require modest source-code modifications. Of course, these can be combined, but their security and performance implications can be considered independently when it comes to deployment.

We discuss both use cases below.

Compartmentalisation

There are different ways to express compartmentalization using CHERI. One is an ambient mode, in which pointers remain 64 bits but are dereferenced relative to a small number of **capability registers**[1], typically one for code, one for data, and one for hardware exceptions . This can be combined with other CHERI features (notably function sentries, which allow low-privilege code to call high-privilege code in a safe way) to control when/how these capability registers are changed.

In this mode, CHERI acts like a very fine-grained, cheap MMU. This brings several advantages, the most obvious being more efficient use of memory, large gains in context-switching performance[2], reduced energy consumption and cheap, privilege-free creation of subprocesses.

In other words, you can have better, faster devices that don't drain battery and are also more secure.[3]

Efficient Use of Memory

Because capability registers use a clever encoding scheme, they can represent common memory sizes (notably “small” sub-page objects) precisely: memory does not have to be allocated in page

[1] Capability registers have bounds and permission bits and are 128 bits wide in a 64 bit architecture. All memory accesses are relative to capability registers.
[2] For some figures, here’s a 2016 paper. It shows a domain-crossing performance gain of up to a few hundred times, depending on buffer size. Unpublished results suggest we can expect even better gains.
[3] While it may seem like other hardware extensions such as the RISC-V pointer masking proposal offer similar security guarantees with lower hardware implementation complexity, the RISC-V proposal is actually significantly less flexible and incurs higher overheads. Switching between domains either requires involvement of a privileged domain, which can be costly, or alternatively making the base register writable by anyone, which negates the security benefits. Furthermore, accessing data outside of the current power-of-two-sized region requires a privilege escalation whereas on CHERI this can be achieved by explicitly delegating capabilities referencing data outside the compartment.

sized chunks, thus reducing overheads, particularly for the common case of small buffers or processes[4]. The encoding scheme is also capable of precisely representing common larger-than-page memory sizes (i.e. multiples of the page size) precisely, covering most practical use cases.

Context Switching Performance

If CHERI is used as the process isolation mechanism instead of the MMU - and the MMU is solely used for virtual memory - then context switches become a simple matter of changing the contents of the capability registers. This means that a context switch looks more like a function call than a traditional context switch. MMU-based context switches typically cost around 10,000 clocks once TLB misses and the like are factored in. CHERI-based context switches take around 50 clocks.

Furthermore, context switches require no privileged operations and so can be done directly between processes without requiring intervention from the kernel.

Benchmarks show that IPC typically gets a performance gain of at least 2 orders of magnitude as a result (in progress paper).

Reduced Energy Consumption

Because every pointer dereference is explicitly linked to its bounds, MMU lookups are no longer required, which means far less power wasted in preventing illegal memory accesses.

Cheap, Privilege-free, Lightweight Process Creation

In CHERI the capability registers provide memory bounds. Reduction of the power of a capability register (i.e. tightening bounds or reducing permissions) is an unprivileged operation and hence a process can, if it desired, carve up its own memory space without reference to the kernel or any other privileged entity.

Memory Provenance

Because the only operations permitted on capability registers reduce their power[5] (either by disabling permissions or tightening memory bounds) the provenance of memory regions is assured. That is, if two disjoint regions of memory are set up and neither contains a capability to the other, or to any other memory region that contains such a capability, then they will **never** be able to access each other.

How To Use This Mode

Probably the best way to use this is for the operating system to use CHERI for process isolation instead of an MMU or other mechanism. To get the performance benefits, the MMU should only be

---

[4] We anticipate cheap context switches leading to lots of small processes as this is one of the easiest ways to mitigate various common software security problems.

[5] Slightly annoyingly, the Arm implementation allows EL3 to create capabilities out of thin air, which does break this model for EL3. On the other hand, if you can't trust EL3, you're in trouble already. This is to allow efficient swapping of capabilities to storage - when they are retrieved they can be directly instantiated rather than having to laboriously rederive them from an existing capability. This annoyance is pretty minimal but if it didn't exist it would permit EL3 to voluntarily reduce its own power.

used for swap and the like - all processes should have the same memory map (more or less - differences can be useful and may not have a large impact on performance depending on exactly what is going on).

Using unmodified software will only get performance and granularity benefits. However, the increased performance of IPC also enables much finer-grained compartmentalisation - or more existing compartments (e.g. Chrome could run more sandboxes at the same time than is currently feasible) - this, of course, requires software engineering to achieve.

Alternatively, instead of the operating system directly using CHERI, it can instead be used as mentioned above: to create lightweight processes within an OS provided process. This may not yield all of the performance benefits but if used instead of process-based compartmentalisation for code that already works that way, some improvements should be made without loss of security.

Memory Safety

Since this is extensively covered elsewhere, I'll be brief.

In this mode, all pointers become "fat" - that is, they include bounds, permissions, and a tag as well as the pointer itself. This mitigates many common memory safety issues, the most obvious being buffer overflows. It also prevents the "invention" of pointers - that is, it is no longer possible to simply calculate a pointer and start using it, instead it must be derived, according to the CHERI rules, which prevent derivation of pointers with **more** access, which makes harder for attackers even if they manage to get arbitrary code execution (and, of course, that is also harder). Unlike other deployed C/C++ memory-safety techniques in software and hardware, CHERI memory protection is deterministic, and is not based on secrets that may be leaked or brute forced.

The downside of this mode is that it requires doubling pointer size which leads to reduced performance, though how much is very dependent on what the code does. As a good rule of thumb, code with high dynamic pointer density in its dynamic memory access pattern, which was likely affected negatively by a transition from 32-bit to 64-bit, may also see performance impact with CHERI memory protection. However, because of the 64-bit transition, those runtimes typically already employ pointer compression techniques (e.g., 37-bit pointers stored in 32-bit words) that can continue to be deployed (with new security tradeoffs, as capabilities would not be used for those specific pointers).

Also, some assumptions programmers tend to make about the compatibility of pointers and integers are false in this model and may require code changes (although such conversions are not portable anyway and are best avoided). In selected systems code and architecture-aware portions of language runtimes -- i.e., within the implementations of memory allocators, just-in-time compilers (JITs), etc -- this can be as high as 1% LoC change. However, a recent study showed that adapting an X11-based open-source desktop stack more typically required around 0.036% LoC change -- mostly for minor C hygiene correcting actual C bugs that don't manifest on current hardware or compilers. Almost no C++ code required any change. Additionally, the majority of these required changes were pointed out by compiler diagnostics and/or sanitizer instrumentation.

The upside is that many memory safety problems become fail stop instead of being exploitable vulnerabilities.

How To Use This Mode

This requires compilers to emit CHERI code and hence needs an updated toolchain. Compilation in “capability mode” then introduces some incompatibilities with code as it is written in practice which may lead to compilation or runtime errors that need to be tracked down and fixed. However, experience says these are fairly rare and generally quite easily fixed.

Can I use memory safety with CHERI compartmentalization?

Yes. And there are performance gains that can be realized best when code does use capabilities for at least selected, if not all, pointers, as capability-based pointers can safely reference memory shared between compartments. Most compartmentalization case studies to date have combined both techniques. Compartmentalization will also sometimes address the question of how to minimize the potential denial-of-service arising from attempts to exploit memory safety violations, by allowing stronger isolation of smaller software modules, such as image-processing libraries.

What Should You Do About This?

- Evaluate
    - There is a comprehensive list of CHERI resources.
- Tell Arm you are interested in CHERI.
- Encourage the RISC-V consortium to adopt CHERI-RISC-V.

Related

- Microsoft Security Response Center “Security Analysis of CHERI ISA”. Note that this predates Cambridge’s work on temporal safety. Notable quote: “Estimating conservatively, the current implementation of CHERI on the CheriBSD project would deterministically mitigate a large subset of spatial safety vulnerabilities which account for almost half of the vulnerabilities reported to MSRC in 2019”. Note that this is an analysis of the memory safety mode only, not the compartmentalisation mode.
- Arm Morello one year on - includes architecture, instruction set and a simulator of the chipset.
- Cambridge CHERI software stack on Morello.
- Assessing the Viability of an Open-Source CHERI Desktop Software Ecosystem.

Open challenges (very much non-exhaustive!)

- We don’t currently know how to define what extra security benefits we get by running things under CHERI: “minimal porting” is one thing; “adapt to take full advantage of CHERI” quite another. There are many shades of grey in between and we lack both terminology (to aid discussion) and an understanding of the meaningful points in the design space.
- Pure capabilities vs. hybrid: in some cases the latter can be easier to reason about than the former due to the DDC register’s semi-global access restrictions. When is it more appropriate to use pure capabilities or hybrid?
- What are good CHERI idioms? We only have limited examples to draw upon (see e.g. CHERI misidioms).

- When adapting language runtimes for the pure capability mode, which already face challenging tradeoffs with respect to pointer size, what is the security and performance impact of continuing to use integer pointers (e.g. for language heaps)?
- What are the performance consequences of different approaches to using CHERI?