CHERI is not just a hardware extension

Carl Shaw

Codasip

→ What is CHERI?

The term “CHERI” is (often confusingly) used to describe:

- The hardware ISA modifications
- The security model (based on capability-based addressing)
- The software programming language support

→ CHERI Task Group

The CHERI Task Group (TG) is working on the RISC-V hardware ISA changes, with the draft specification document now available from the CHERI TG’s repository [1]. These CHERI hardware RISC-V extensions efficiently implement the security model to provide the critical security protection measures of:

- Spatial and temporal memory safety
- Control flow integrity (CFI)
- Fine-grained compartmentalization

[1] https://github.com/riscv/riscv-cheri

→ Why is CHERI needed?

Memory safety is a critical security problem that has gained substantial visibility and there is strong demand for a practical solution that can be applied to legacy code.

~$10T
Worldwide cost estimate of cyberattacks per year
(and growing fast)

5X
Increase in firmware attacks reported by NIST between 2017 and 2021

~70%
Of CVEs reported by Microsoft, Chromium, and Ubuntu related to memory safety

CHERI also boosts the security and efficiency of newer software written in memory-safe languages by replacing software checks by robust hardware checks, securing software language virtual machines, and by protecting low-level memory accesses that cannot be protected by higher-level architectural constructs such as borrow-checking.

CFI prevents advanced attacks such as return-oriented programming.

Compartmentalization enables next-generation safe and secure software architectures based on the fundamental security principle of least privilege. It also protects ”hybrid” software today that mixes memory-safe and memory-unsafe languages (e.g. the common case of calling high-performance C libraries from Python).

→ CHERI Software

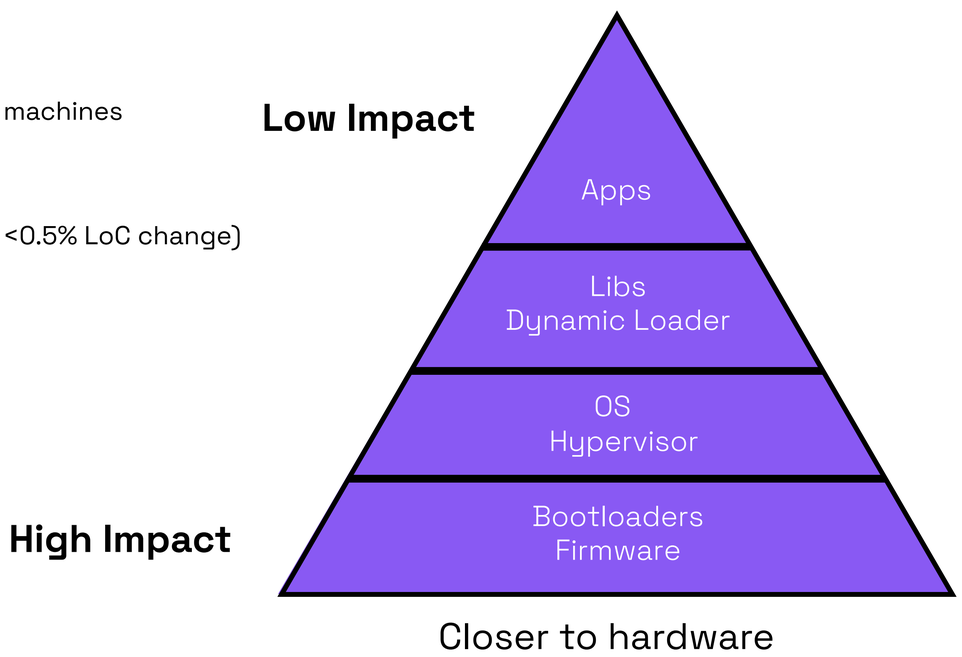
The CHERI ISA extensions are designed to be as minimal as possible and form a toolkit that software can use to build higher-level security constructs like temporal memory protection and strong compartmentalization. The CHERI approach is a strong example of hardware-software co-design.

The CHERI compilers (currently C/C++, with Rust planned) do most of the heavy lifting, but some modification to software may be necessary. As a rule of thumb, the lower level the code, the more modification is required. However, we have found that many of the modifications at the application level are code improvements and fixes.

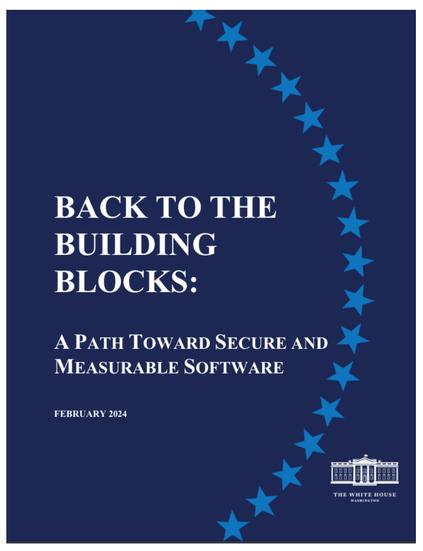
The software impact is highest in:

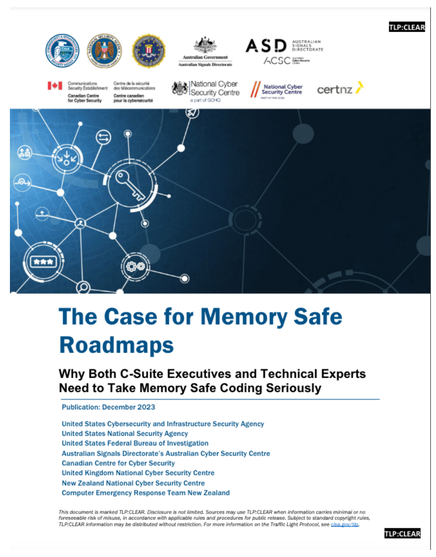
- Bootloaders and firmware
- Operating Systems (OS)
- Language libraries, runtimes and virtual machines
- Dynamic loaders
- Memory allocators

And lowest in application code (typically <0.5% LoC change)



→ Governments highlighting memory security





→ Supporting CHERI

For CHERI to be successfully adopted, it must have broad support in common software. We are starting by targeting lower-level software packages such as OS, tools and libraries, which are critical to ease the development of higher-level applications:

Software package	Current status
LLVM C/C++ compiler / toolchains	LLVM17 supported and now working on LLVM18. Bare-metal (Newlib) and BSD, Linux toolchains.
RISC-V Sail model	Preparing to upstream
QEMU	QEMU v6.0 supported. Currently working to increase version to support newer RISC-V extensions
CHERI Linux	In early development
CheriBSD	Currently adding support for latest draft specification
CheriFreeRTOS	Migrating support to latest draft specification
CHERI Zephyr RTOS	In early development
CHERI seL4	In development
CherIoT OS	In planning
CHERI Rust compiler	In planning

Many software packages enhanced with CHERI as part of the Arm Morello CHERI project will also be available and RISC-V support added where required, adding a considerable body of packages.

→ Impact of CHERI

CHERI has minimal hardware overhead, but what about software? We have found the impact of CHERI depends very much on the software being run and how CHERI is being used. For the following figures we assume memory and CFI protection only and no compartmentalization. First off, we can remove existing weak software-based security mechanisms that add code, memory and/or performance overhead : stack protector; ASLR; shadow stacks, etc. This can save up to 5% performance overhead. Adding CHERI support typically adds < 5% performance difference, hence any impact on performance is usually unnoticed but the security level is much higher.

Code size is increased with CHERI due to the extra instructions e.g. to set memory bounds and permissions. For example, comparison of a Linux kernel built with CHERI and one built without CHERI (but with KASLR and stack protection enabled) yielded an increase of 6% for the CHERI kernel.

There is also an impact in memory footprint as pointers and registers pushed to the stack are now double the size. However, the impact is very software dependent.

Find out more