

Introduction to CHERI

ASPLOS'22 Edition

Welcome

- Who are we?



Dr. Nathaniel “Wes” Filardo
MSR Cambridge



Prof. Robert Watson
University of Cambridge



Konrad Witaszczyk
University of Cambridge



Dr. Jonathan Woodruff
University of Cambridge

- Who are you?

Why Are We Here?

- Familiarize ASPLOS attendees with CHERI architecture and CHERI C
 - Large-scale tech-transfer w/ UKRI's DSbD & Arm Morello chip
 - Significant revision of commodity *abstract machine*
- Bonus takeaway: a working SDK for continued experimentation

Prerequisites

Software stack

If you want to work through the exercises with us, you should have read the Introduction chapter of the [CHERI Exercises and Missions book](#) ().

Most importantly, you should have:

- [Obtained CHERI Compilers and Simulators](#),
 - We recommended to use the CHERI Software Release with Docker;
 - You could also spend several hours with the CHERI-RISC-V flavored DIY option.
- Read the [instructions for cross-compilation](#), and
- Tested that you can [compile and run programs from the book](#) ()

We also recommend to join #workshop-asplos22 at <https://cheri-cpu.slack.com/> to discuss the workshop with others.

Approximate Schedule (UTC+1)

09:00 – 10:00	Preliminaries & CHERI Overview (Prof. Robert Watson & nwf)
10:00 – 10:30	Exercise: CHERI Pointer Integrity (nwf)
10:30 – 11:00	Exercise: CHERI Stack Spatial Safety (nwf)
11:00 – 11:30	Break
11:30 – 12:00	Exercise: CHERI C and Sub-objects (Konrad)
12:00 – 12:30	Exercise: Spatially Safe Heap (nwf)
12:30 – 13:00	Exercise: CHERI C Adaptation (Konrad)
13:00 – 14:00	Lunch
14:00 – 14:30	Microarchitectural Implications of CHERI (Dr. Jon Woodruff)
14:30 – 15:00	Exercise: The CheriABI *nix System Call Interface (nwf)
15:00 – 15:30	Heap Temporal Safety (nwf)
15:30 – 16:00	Break
16:00 – 16:30	Exercise: CHERI C Heap Adaptation (Prof. Robert Watson)
16:30 – 17:00	Scalable Software Compartmentalization (Prof. Robert Watson)
17:00 – 17:30	Open Q&A and Wrap Up

From CHERI to Morello

Architectural Support for Memory Protection and Software Compartmentalization

Robert N. M. Watson, Simon W. Moore, Peter Sewell, Peter G. Neumann

Hesham Almatary, Jonathan Anderson, Alasdair Armstrong, Peter Blandford-Baker, John Baldwin, Hadrien Barrel, Thomas Bauereiss, Ruslan Bukin, David Chisnall, Jessica Clarke, Nirav Dave, Brooks Davis, Lawrence Esswood, Nathaniel W. Filardo, Franz Fuchs, Dapeng Gao, Khilan Gudka, Brett Gutstein, Alexandre Joannou, Mark Johnston, Robert Kovacsics, Ben Laurie, A. Theo Markettos, J. Edward Maste, Alfredo Mazzinghi, Alan Mujumdar, Prashanth Mundkur, Steven J. Murdoch, Edward Napierala, George Neville-Neil, Robert Norton-Wright, Philip Paeps, Lucian Paul-Trifu, Allison Randal, Ivan Ribeiro, Alex Richardson, Michael Roe, Colin Rothwell, Peter Rugg, Hassen Saidi, Peter Sewell, Thomas Sewell, Stacey Son, Domagoj Stolfa, Andrew Turner, Munraj Vadera, Konrad Witaszczyk, Jonathan Woodruff, Hongyan Xia, and Bjoern A. Zeeb

University of Cambridge and SRI International
ASPLOS 2022 – CHERI tutorial – 27 February 2022

Approved for public release; distribution is unlimited. This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this article/presentation are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.



Approved for public release; distribution is unlimited.

This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 (“CTSRD”), with additional support from FA8750-11-C-0249 (“MRC2”), HR0011-18-C-0016 (“ECATS”), and FA8650-18-C-7809 (“CIFV”) as part of the DARPA CRASH, MRC, and SSITH research programs. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

This work was supported in part by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 105694.

We also acknowledge the EPSRC REMS Programme Grant (EP/K008528/1), the ERC ELVER Advanced Grant (789108), the Isaac Newton Trust, the UK Higher Education Innovation Fund (HEIF), Thales E-Security, Microsoft Research Cambridge, Arm Limited, Google, Google DeepMind, HP Enterprise, and the Gates Cambridge Trust.

CHERI introduction

- **CHERI is a new processor technology that mitigates software vulnerabilities**
 - Developed by the University of Cambridge and SRI International starting in 2010, supported by DARPA
 - Arm collaboration from 2014
 - Arm Morello CPU, SoC, and board announced 2019, with support from UKRI; shipping as of Jan 2022
- Today's talk:
 - What is CHERI, how does it work, and is it any good?
 - What is a Morello board, and what can I do with one?

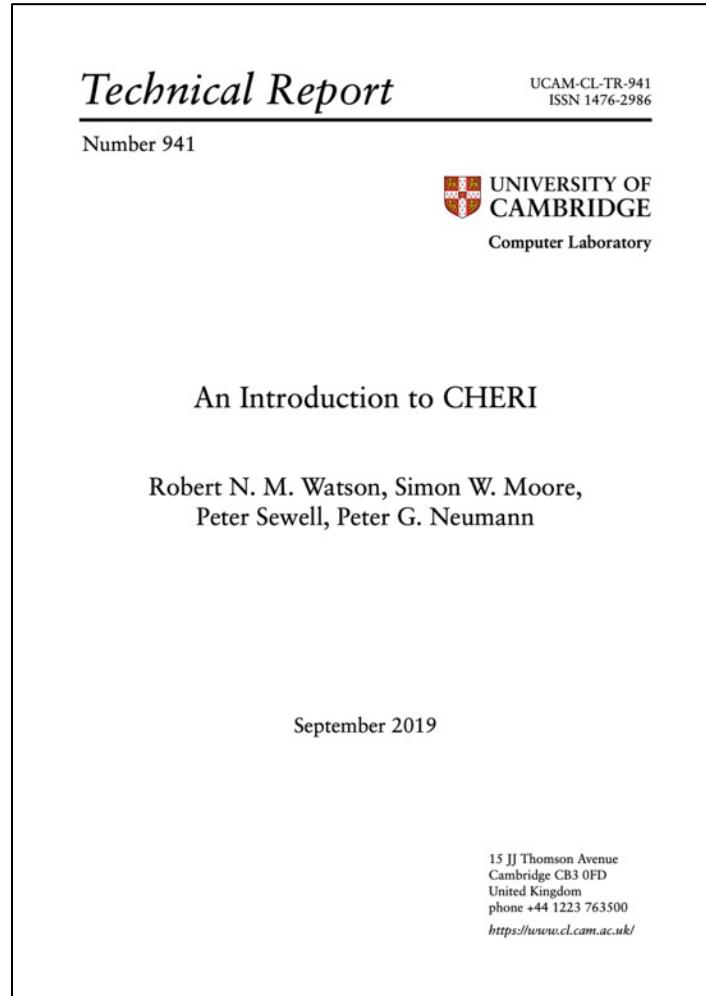


An early experimental FPGA-based CHERI tablet prototype running the CheriBSD operating system and applications, Cambridge, 2013.



High-performance Arm Morello chip able to run a full CHERI software stack, Cambridge, 2022

An Introduction to CHERI



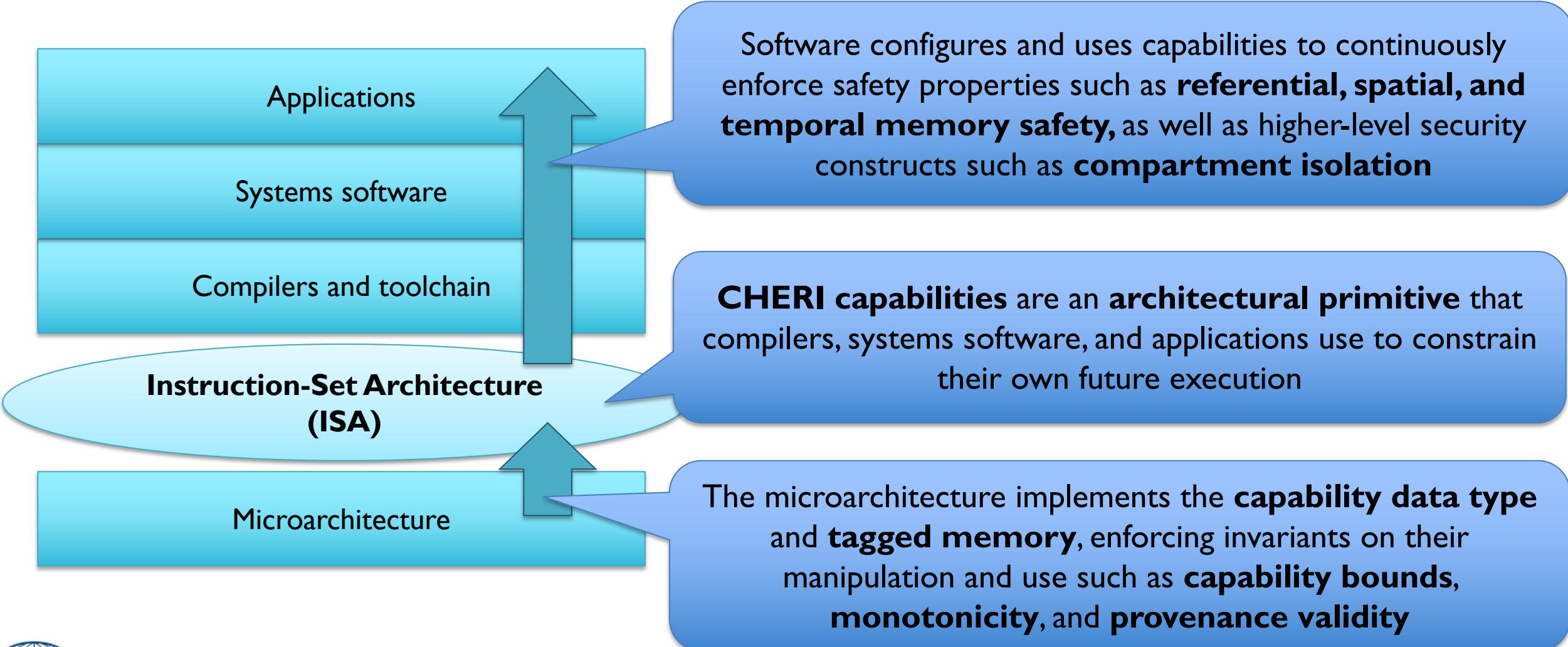
- Watson, et al. **An Introduction to CHERI**, UCAM-CL-TR-941, September 2019
 - Architectural capabilities and the CHERI ISA
 - CHERI microarchitecture
 - ISA formal modeling and proof
 - Software construction with CHERI
 - Language and compiler extensions
 - OS extensions
 - Application-level adaptations

Predates public announcement of Morello

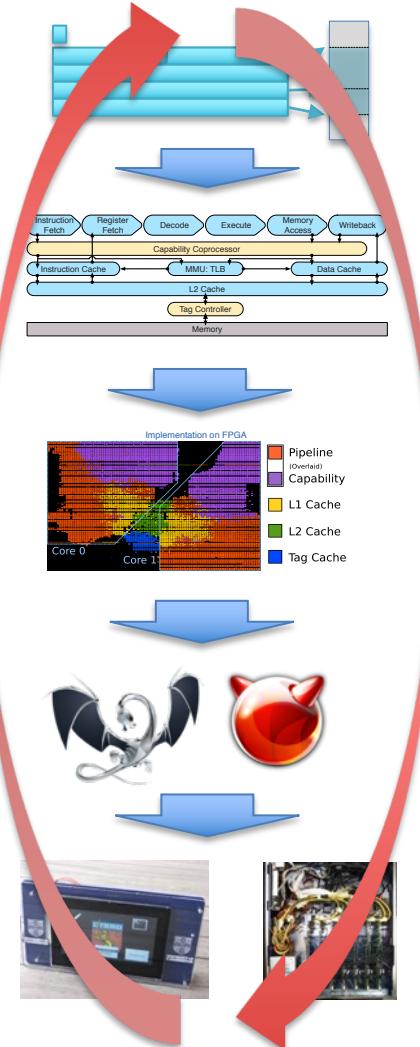
What is CHERI?

- CHERI is a processor **architectural protection model**
 - Composes a **capability-system model** with hardware and software
 - Adds new security primitives to Instruction-Set Architectures (ISAs)
 - Implemented by microarchitectural extensions to the CPU and SoC
 - Enables new security behavior in software
- CHERI mitigates vulnerabilities in **C/C++ Trusted Computing Bases**
 - Hypervisors, operating systems, language runtimes, browsers,
 - **Fine-grained memory protection** deterministically closes many arbitrary code execution attacks, and directly impedes common exploit-chain tools
 - **Scalable compartmentalization** mitigates many vulnerability classes .. even unknown future classes .. by extending the idea of software sandboxing

Processor primitives for software security

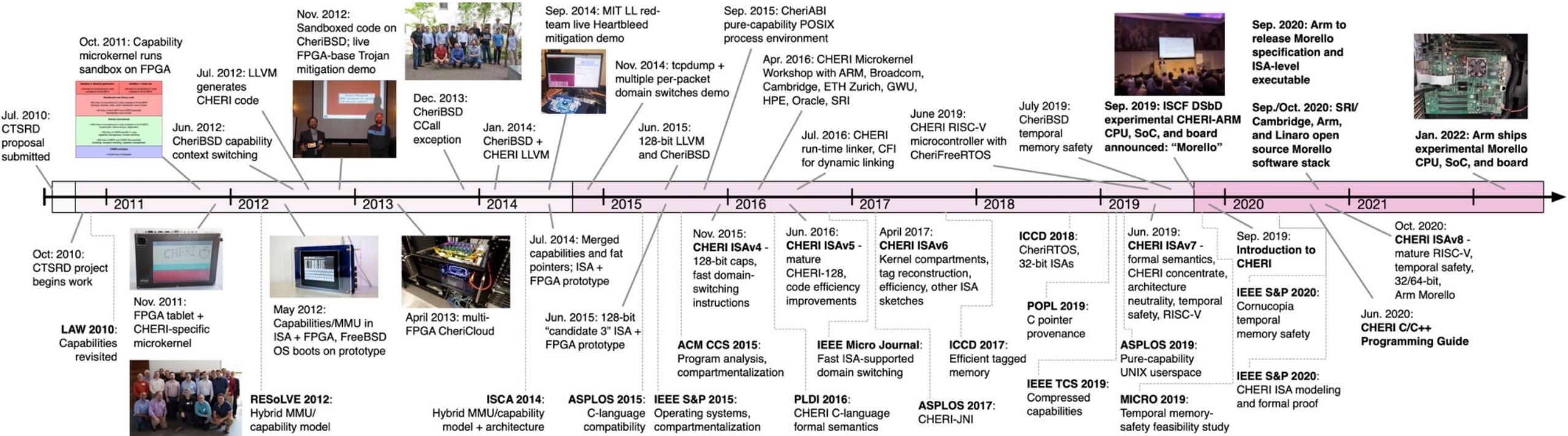


Hardware-software-semantics co-design



- CHERI abstract protection model; concrete ISA instantiations in ~~64-bit MIPS, 32/64-bit RISC-V, 64-bit ARMv8-A~~
- Formal ISA models, QEMU-CHERI, and multiple FPGA prototypes
- Formal proofs that ISA security properties are met, automatic test general from formal model
- CHERI Clang/LLVM/LLD, CheriBSD, C/C++-language applications
- Repeated iteration to improve {performance, security, compatibility, ..}

CHERI research and development timeline



Years 1-2: Research platform, prototype architecture

Years 2-4: Hybrid C/OS model, compartment model

Years 4-7: Efficiency, CheriABI/C/C++/linker, ARMv8-A

Years 8-11: RISC-V, temporal safety, formal proof

CHERI ISA refinement over 10 years

Technical Report

UCAM-CL-TR-951
ISSN 1476-2986

Number 951



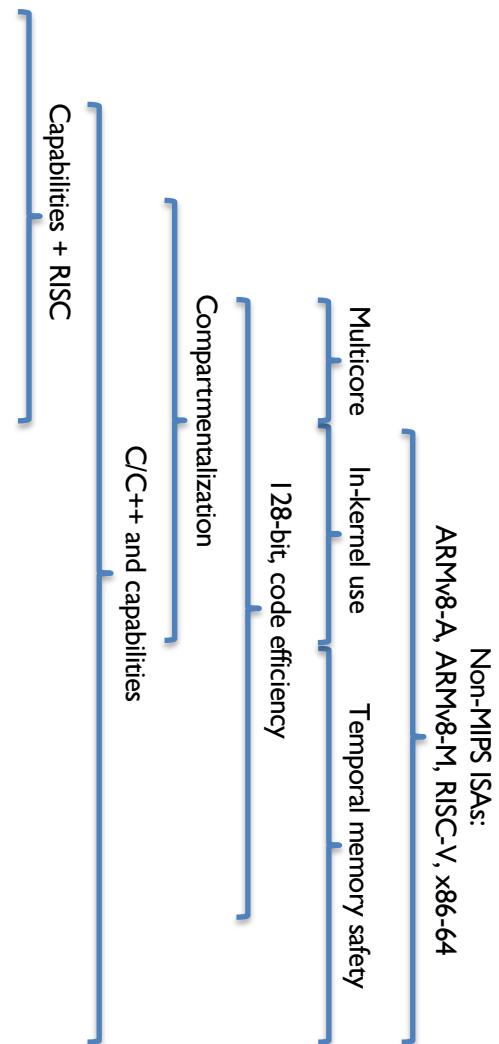
Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)

Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff,
Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin,
Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis,
Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite,
Alexandre Joannou, Ben Laurie, A. Theodore Markettos,
Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis,
Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell,
Stacey Son, Hongyan Xia

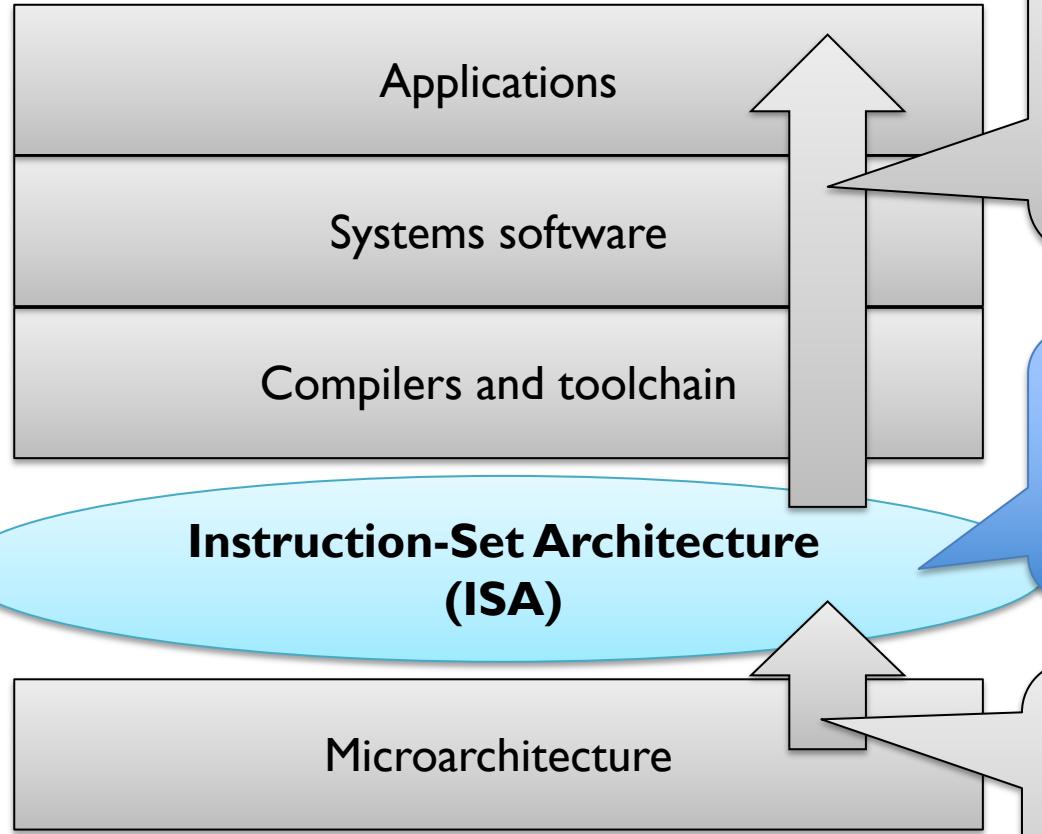
October 2020

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500
<https://www.cl.cam.ac.uk/>

Year	Version	Description
2010-2012	ISAv1	RISC capability-system model w/64-bit MIPS Capability registers, tagged memory Guarded manipulation of registers
2012	ISAv2	Extended tagging to capability registers Capability-aware exception handling Boots an MMU-based OS with CHERI support
2014	ISAv3	Fat pointers + capabilities, compiler support Instructions to optimize hybrid code Sealed capabilities, CCall/CReturn
2015	ISAv4	MMU-CHERI integration (TLB permissions) ISA support for compressed 128-bit capabilities HW-accelerated domain switching Multicore instructions: full suite of LL/SC variants
2016	ISAv5	CHERI-128 compressed capability model Improved generated code efficiency Initial in-kernel privilege limitations
2017	ISAv6	Mature kernel privilege limitations Further generated code efficiency Architectural portability: CHERI-x86, CHERI-RISC-V sketches Exception-free domain transition
2019	ISAv7	Architectural performance optimization for C++ applications Microarchitectural side-channel resistance features Architecture-neutral CHERI protection model All instruction pseudocode from a formal model CHERI Concentrate capability compression Improved C-language support, dynamic linking, sentry capabilities Elaborated CHERI-RISC-V ISA 64-bit capabilities for 32-bit architectures Accelerated tag operations for temporal memory safety
2020	ISAv8	MMU temporal memory-safety assist; e.g., capability dirty bit Optimizations for sentry capabilities CHERI-RISC-V privileged support, general maturity Further C-language semantics improvements



Architectural primitives for software security

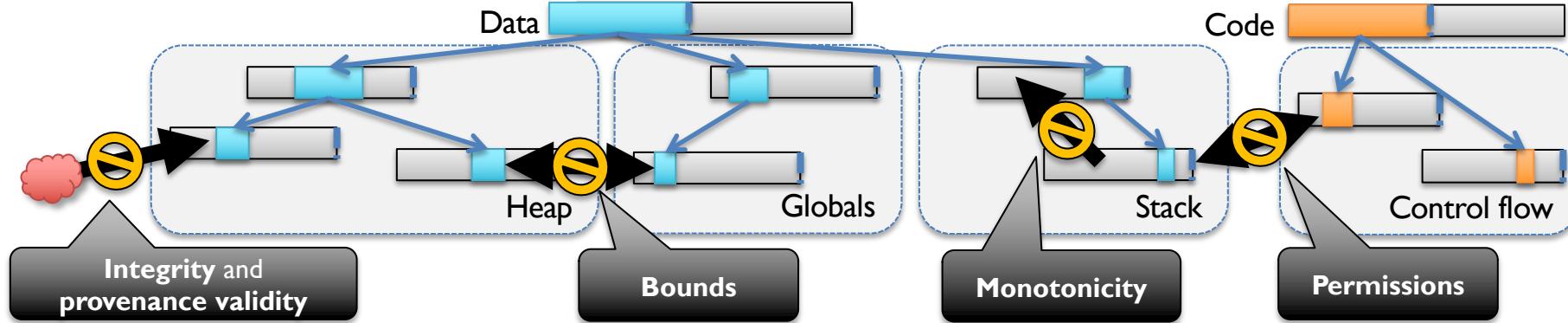


Software configures and uses capabilities to continuously enforce safety properties such as **referential, spatial, and temporal memory safety**, as well as higher-level security constructs such as **compartment isolation**

CHERI capabilities are an **architectural primitive** that compilers, systems software, and applications use to constrain their own future execution

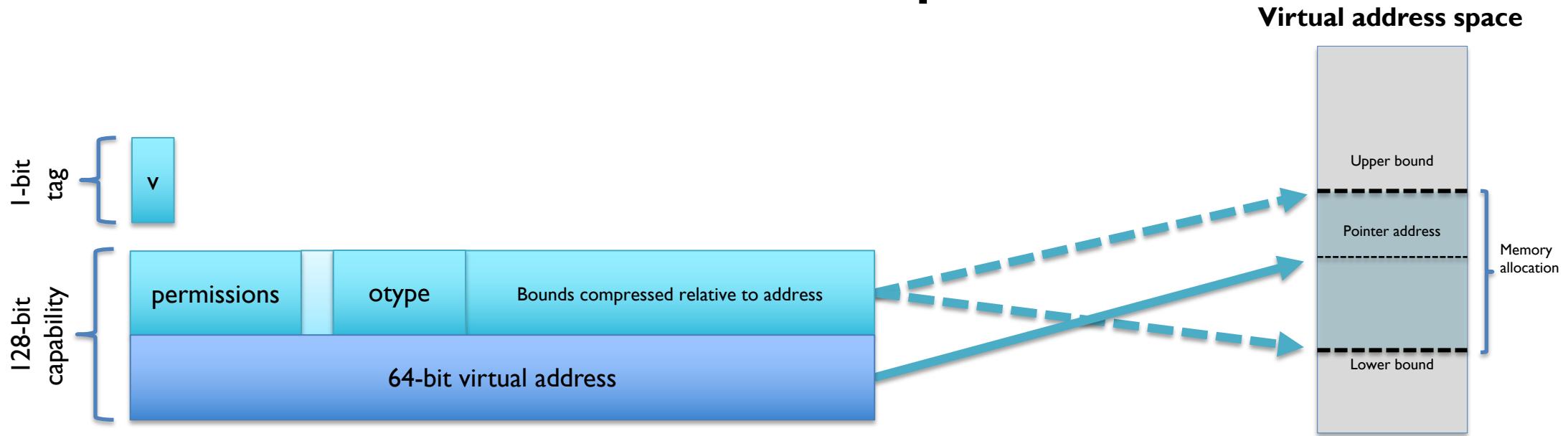
The microarchitecture implements the **capability data type** and **tagged memory**, enforcing invariants on their manipulation and use such as **capability bounds, monotonicity, and provenance validity**

CHERI enforces protection semantics for pointers



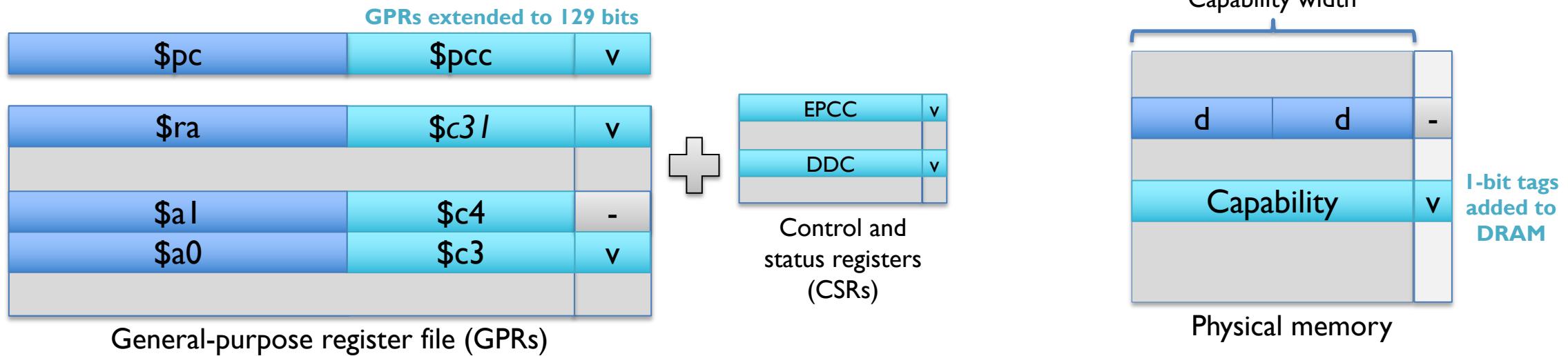
- **Integrity and provenance validity** ensure that valid pointers are derived from other valid pointers via valid transformations; **invalid pointers cannot be used**
 - Valid pointers, once removed, cannot be reintroduced solely unless rederived from other valid pointers
 - E.g., Received network data 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**

CHERI 128-bit capabilities



- **Capabilities extend integer memory addresses**
- **Metadata** (bounds, permissions, ...) control how it may be used
- **Tags** protect capability integrity/derivation in registers + memory

Merged capability register file + tagged memory

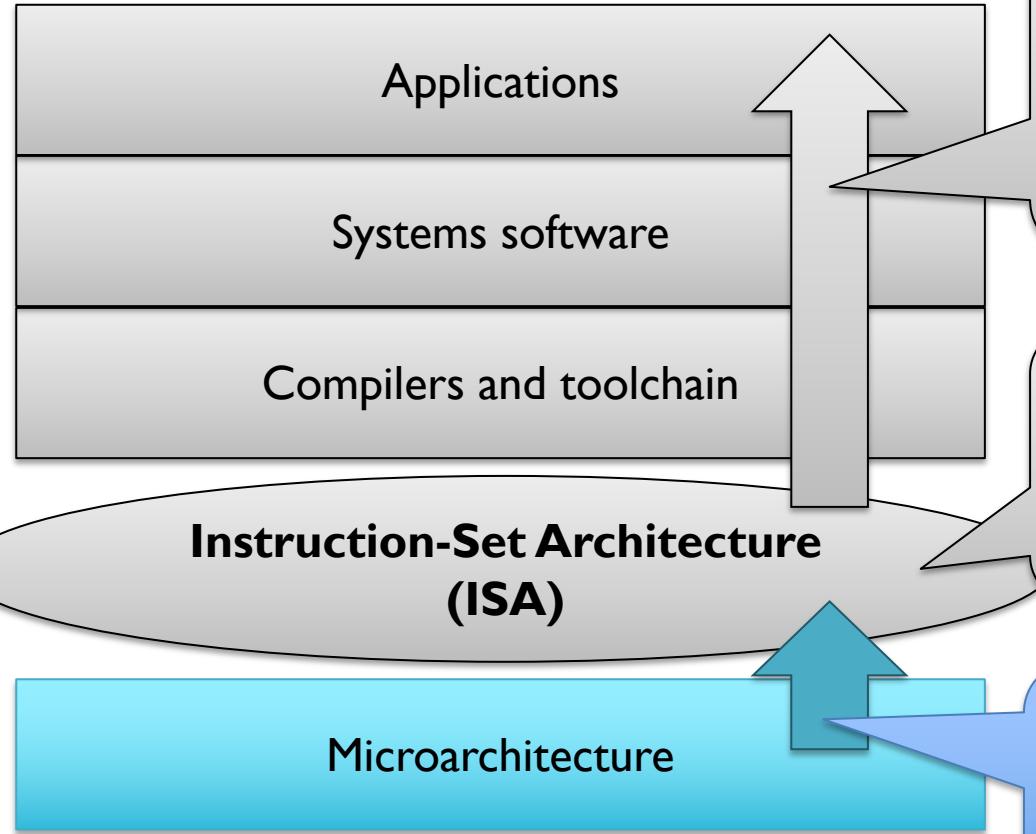


- **64-bit general-purpose registers (GPRs)** are extended with **64 bits of metadata** and a **1-bit validity tag**
- **Program counter (PC)** is extended to be the **program-counter capability (\$PCC)**
- **Default data capability (\$DDC)** constrains legacy integer-relative ISA load and store instructions
- **Tagged memory** protects capability-sized and -aligned words in DRAM by adding a **1-bit validity tag**
- **Various system mechanisms** are extended (e.g., capability-instruction enable control register, new TLB/PTE permission bits, exception code extensions, saved exception stack pointers and vectors become capabilities, etc.)

CHERI-RISC-V formal ISA model

- CHERI RISC-V ISA model extends RISC-V formal ISA specification, in Sail
- Sail RISC-V ISA specification developed by UCam + SRI
 - Selected as official RISC-V spec by the Foundation
 - Sail is a custom first-order imperative language for expressing ISA specifications, usable by engineers but with static type checking of bitvector lengths etc.
 - The Sail spec is inlined in versions of the unprivileged and privileged RISC-V manuals
 - Sail auto-generates a C emulator, theorem-prover definitions, and SMT definitions
 - Machinery for configuring model WRT YAML from compliance group
 - Readable, precise definition of ISA behavior, usable as test oracle for testing hardware against and for software bring-up, and providing prover definitions if you want more rigorous reasoning
- Paper on earlier CHERI-MIPS L3 modelling and proof work at IEEE SSP 2020
- Most recently completed monotonicity proofs for the Arm Morello architecture

Architectural primitives for software security



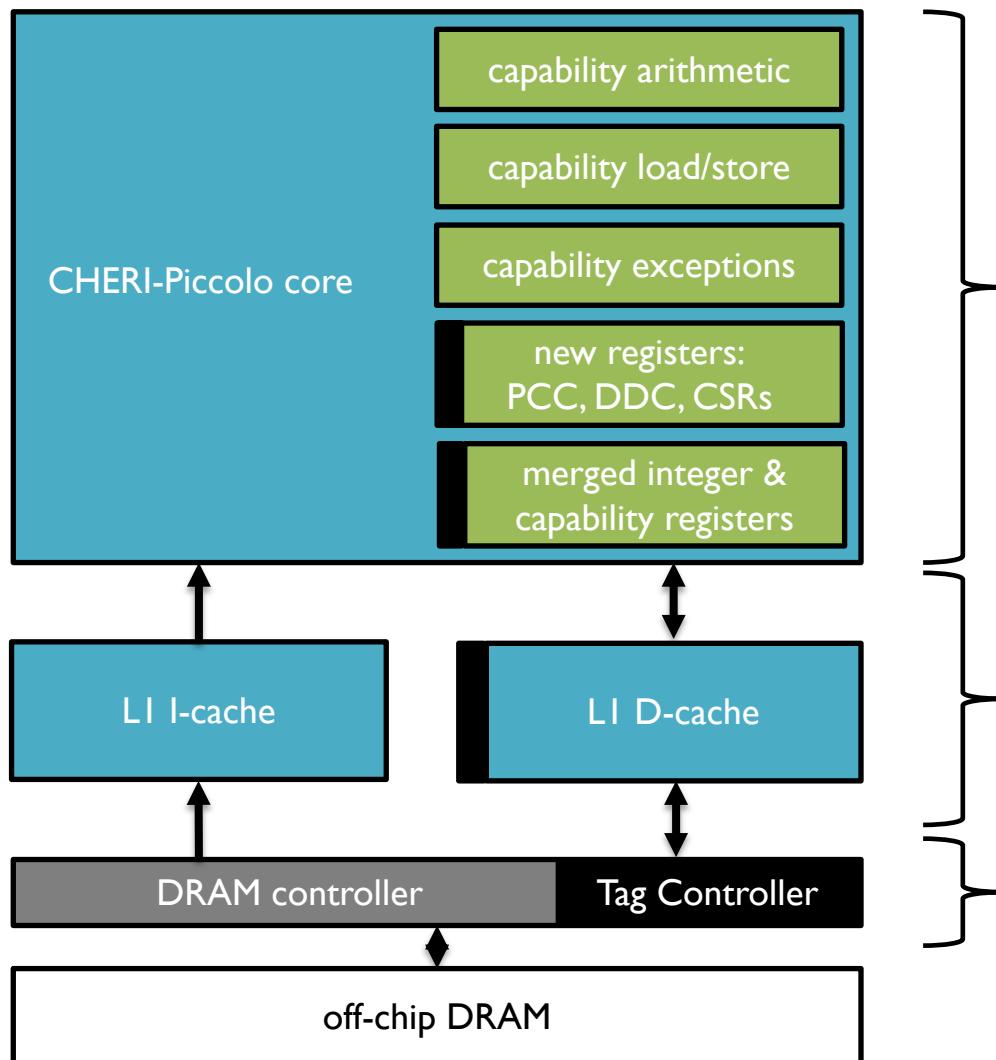
Software configures and uses capabilities to continuously enforce safety properties such as **referential, spatial, and temporal memory safety**, as well as higher-level security constructs such as **compartment isolation**

CHERI capabilities are an **architectural primitive** that compilers, systems software, and applications use to constrain their own future execution

The microarchitecture implements the **capability data type** and **tagged memory**, enforcing invariants on their manipulation and use such as **capability bounds, monotonicity, and provenance validity**

Example microarchitecture: CHERI-Piccolo microcontroller

= tag storage



Changes to the Piccolo core (RISC-V 3-stage pipeline):

- capability arithmetic
- capability load/store operations with bounds checking
- extended exception model
- PC becomes a capability (PCC)
- default data capability (DDC)
- new control/status registers
- merged integer & capability register file

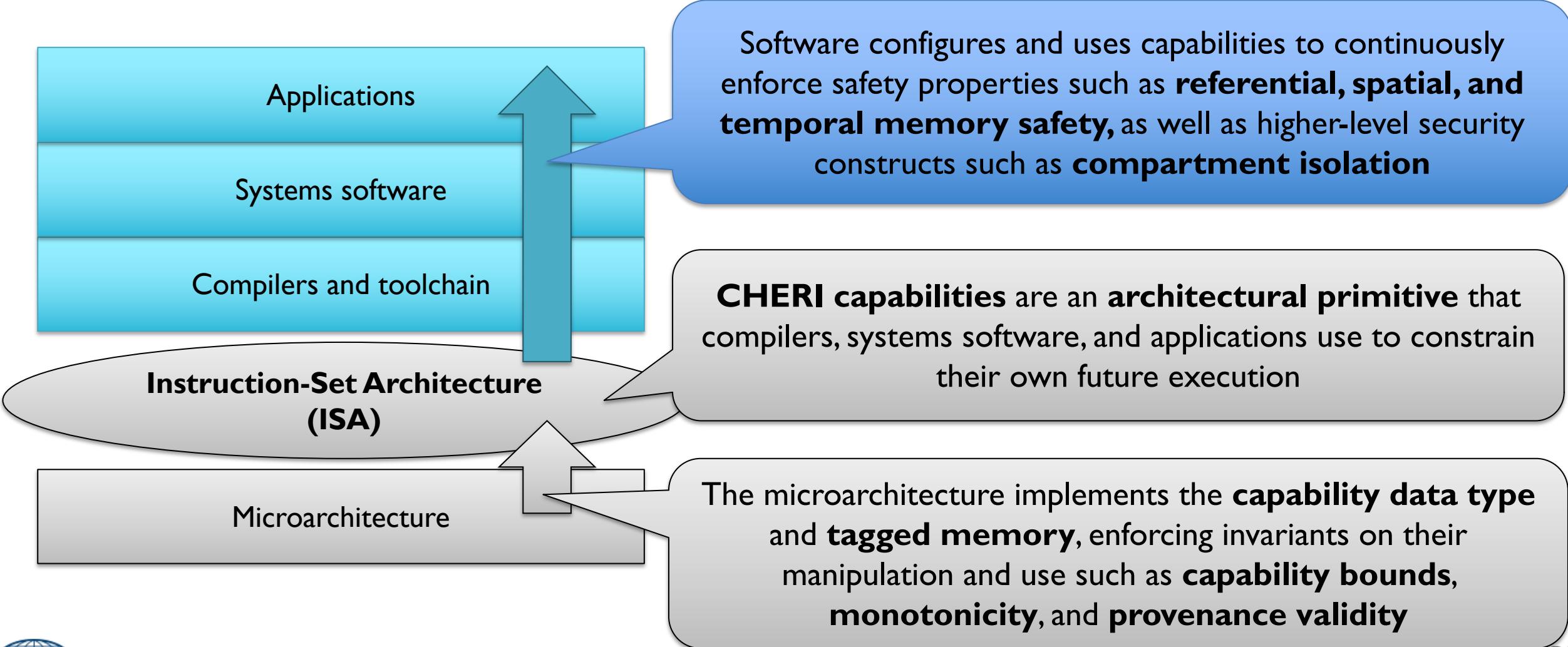
Memory subsystem:

- AXI user-field added to transport tag bits & data width doubled
- caches extended to include tags

DRAM changes:

- New tag controller uses a hierarchical tag table to efficiently store tag bits backed by top of DRAM

Architectural primitives for software security



Two key applications of the CHERI primitives

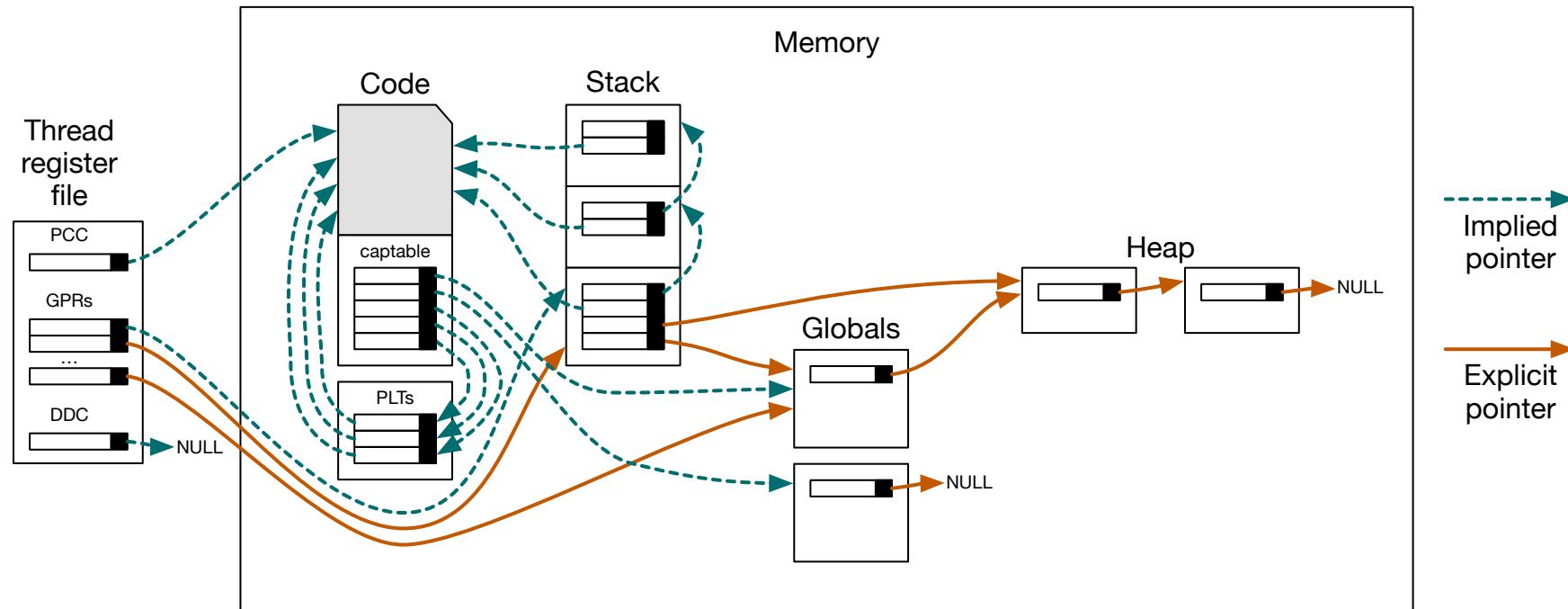
I. Efficient, fine-grained memory protection for C/C++

- Strong source-level compatibility, but requires recompilation
- Deterministic and secret-free referential, spatial, and temporal memory safety
- Retrospective studies estimate $\frac{2}{3}$ of memory-safety vulnerabilities mitigated
- Generally modest overhead (0%-5%, some pointer-dense workloads higher)

2. Scalable software compartmentalization

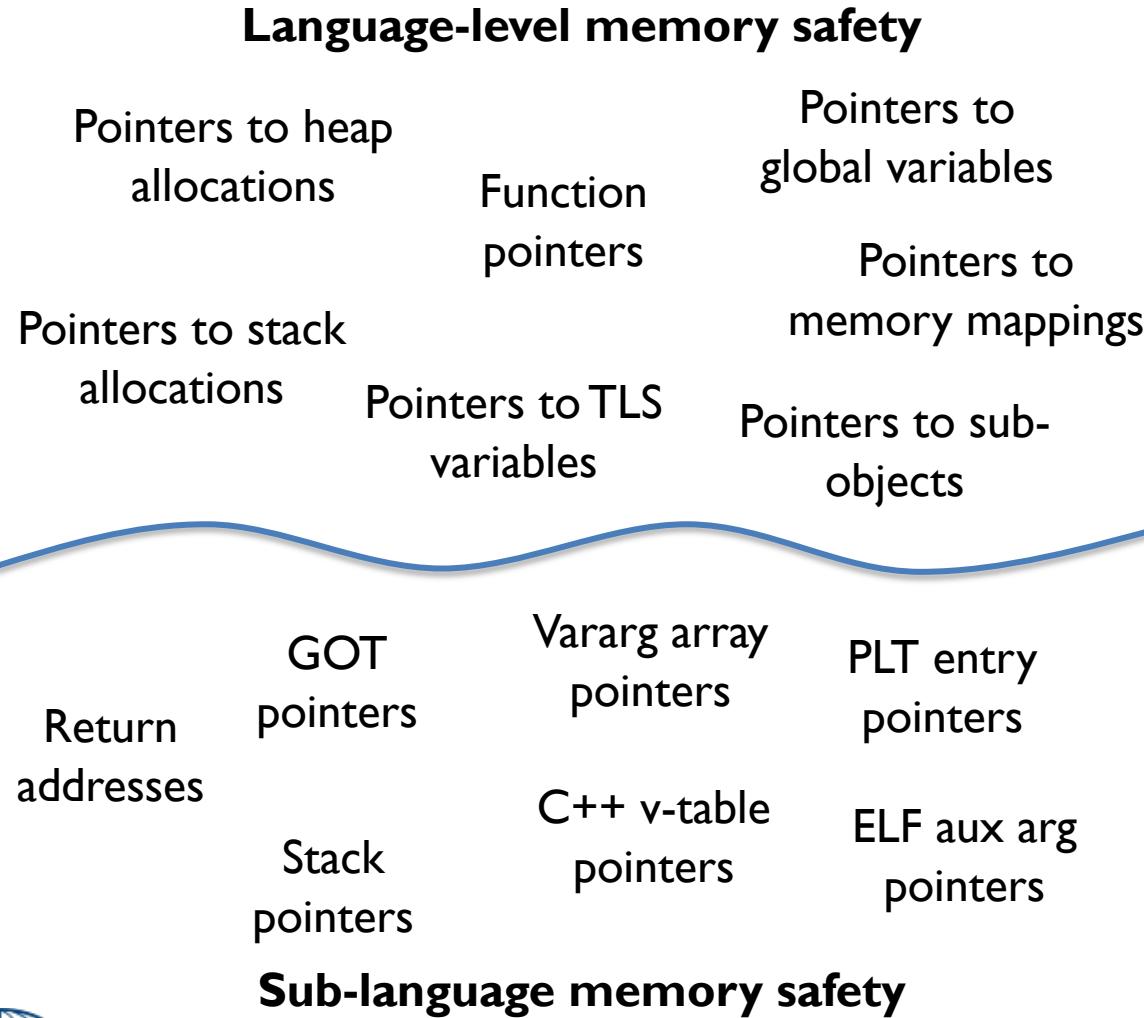
- Multiple software operational models from objects to processes
- Increases exploit chain length: Attackers must find and exploit more vulnerabilities
- Orders-of-magnitude performance improvement over MMU-based techniques (<90% reduction in IPC overhead in early FPGA-based benchmarks)

CHERI-based pure-capability process memory



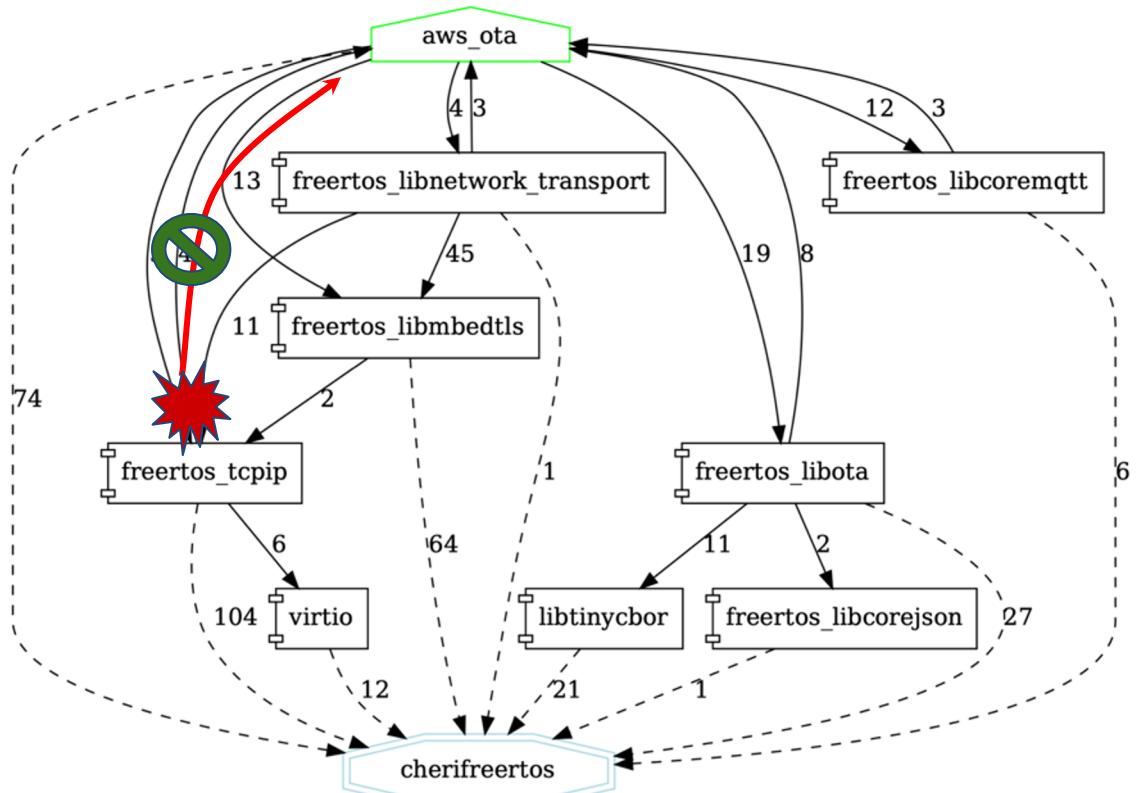
- Capabilities are substituted for integer addresses throughout the address space
- Bounds and permissions are minimized by software including the kernel, runtime linker, memory allocator, and compiler-generated code
- Hardware permits fetch, load, and store only through granted capabilities
- Tags ensure integrity and provenance validity of all pointers

Memory protection for the language and the language runtime



- Capabilities are refined by the kernel, run-time linker, compiler-generated code, heap allocator, ...
- Protection mechanisms:
 - Referential memory safety
 - Spatial memory safety + privilege minimization
 - Temporal memory safety
- Applied **automatically** at two levels:
 - **Language-level pointers** point explicitly at stack and heap allocations, global variables, ...
 - **Sub-language pointers** used to implement control flow, linkage, etc.
- Sub-language protection mitigates bugs in the language runtime and generated code, as well as attacks that cannot be mitigated by higher-level memory safety
 - (e.g., union type confusion)

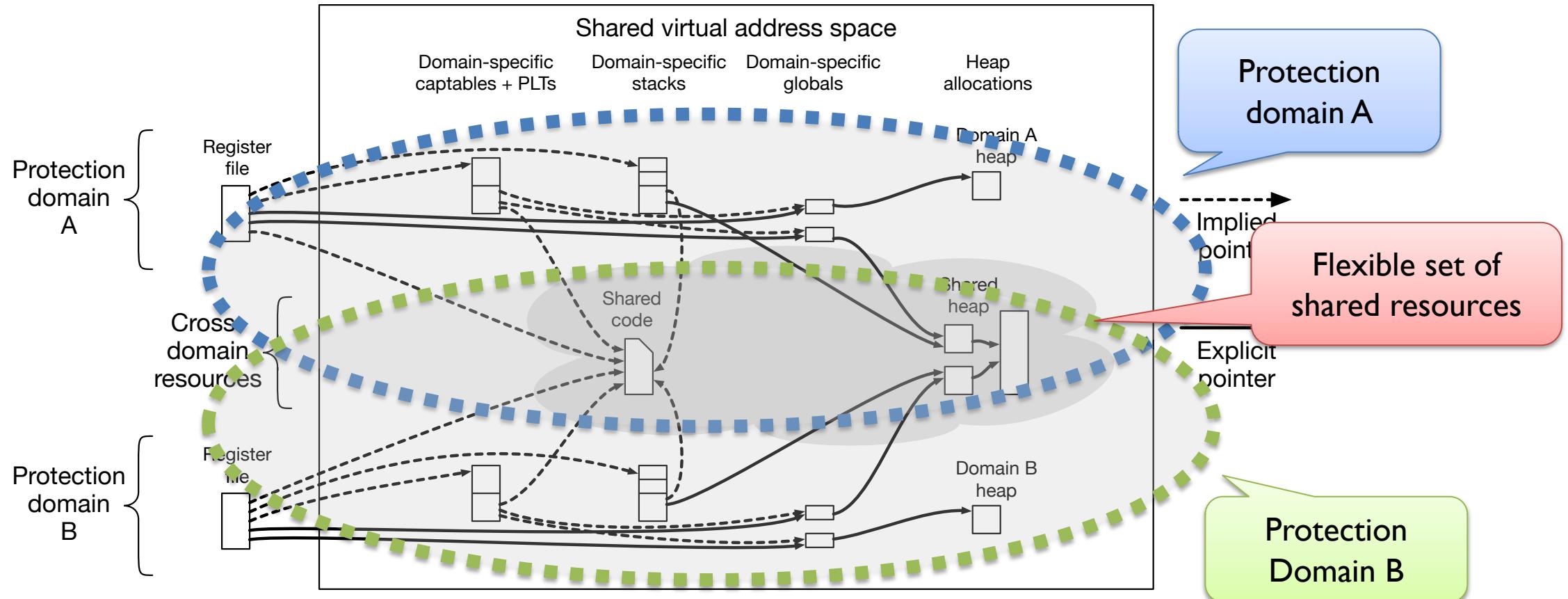
What is software compartmentalization?



CheriFreeRTOS components and the application execute in compartments. CHERI contains an attack within TCP/IP compartment, which access neither flash nor the internals of the software update (OTA) compartment.

- Fine-grained decomposition of a larger software system into **isolated modules** to constrain the impact of faults or attacks
- Goals is to **minimize privileges yielded by a successful attack, and to limit further attack surfaces**
- Usefully thought about as a **graph of interconnected components**, where the attacker's goal is to compromise nodes of the graph providing a route from a point of entry to a specific target

CHERI-based compartmentalization



- Isolated compartments can be created using closed graphs of capabilities, combined with a constrained non-monotonic domain-transition mechanism

Compartmentalization scalability

- CHERI dramatically improves **compartmentalization scalability**
 - More compartments
 - More frequent and faster domain transitions
 - Faster shared memory between compartments
 - Many potential use cases – e.g., sandbox processing of each image in a web browser, processing each message in a mail application
 - Unlike memory protection, software compartmentalization requires **careful software refactoring** to support strong encapsulation, and affects the software operational model
- } Early benchmarks show a 1-to-2 order of magnitude performance inter-compartment communication improvement compared to conventional designs

Microsoft security analysis of CHERI C/C++

SECURITY ANALYSIS OF CHERI ISA

Nicolas Joly, Saif ElSherei, Saar Amar – Microsoft Security Response Center (MSRC)

INTRODUCTION AND SCOPE

The CHERI ISA extension provides memory-protection features which allow historically memory-unsafe programming languages such as C and C++ to be adapted to provide strong, compatible, and efficient protection against many currently widely exploited vulnerabilities.

CHERI requires addressing memory through unforgeable, bounded references called capabilities. These capabilities are 128-bit extensions of traditional 64-bit pointers which embed protection metadata for how the pointer can be dereferenced. A separate tag table is maintained to distinguish each capability word of physical memory from non-capability data to enforce unforgeability.

In this document, we evaluate attacks against the pure-capability mode of CHERI since non-capability code in CHERI's hybrid mode could be attacked as-is today. The CHERI system assessed for this research is the CheriBSD operating system running under QEMU as it is the largest CHERI adapted software available today.

CHERI also provides hardware features for application compartmentalization^[15]. In this document, we will review only the memory safety guarantees, and show concrete examples of exploitation primitives and techniques for various classes of vulnerabilities.

SUMMARY

CHERI's ISA is not yet stabilized. We reviewed the current revision 7, but some of the protections such as executable pointer sealing is still experimental and likely subject to future change.

The CHERI protections applied to a codebase are also highly dependent on compiler configuration, with stricter configurations requiring more refactoring and qualification testing (highly security-critical code can opt into more guarantees), with the strict sub-allocation bounds behavior being the most likely high friction to enable. Examples of the protections that can be configured include:

- Pure-capability vs hybrid mode
- Chosen heap allocator's resilience
- Sub-allocation bounds compilation flag
- Linkage model (PC-relative, PLT, and per-function .capturable)
- Extensions for additional protections on execute capabilities
- Extensions for temporal safety

However, even with enabling all the strictest protections, it is possible that the cost of making existing code CHERI compatible will be less than the cost of rewriting the code in a memory safe language, though this remains to be demonstrated.

We conservatively assessed the percentage of vulnerabilities reported to the Microsoft Security Response Center (MSRC) in 2019 and found that approximately 31% would no longer pose a risk to customers and therefore would not require addressing through a security update on a CHERI system based on the default configuration of the CheriBSD operating system. If we also assume that automatic initialization of stack variables ([InitAll](#)) and of heap allocations (e.g. [pool_zeroing](#)) is present, the total number of vulnerabilities deterministically mitigated exceeds 43%. With additional features such as [Cornucopia](#) that help prevent temporal safety issues such as use after free, and assuming that it would cover 80% of all the UAFs, the number of deterministically mitigated vulnerabilities would be at least 67%. There is additional work that needs to be done to protect the stack and add fined grained CFI, but this combination means CHERI looks very promising in its early stages.

1 | Page

Microsoft Security Response Center (MSRC)

- Microsoft Security Research Center (MSRC) study analyzed all 2019 Microsoft critical memory-safety security vulnerabilities
 - Metric: “Poses a risk to customers → requires a software update”
 - **Vulnerability mitigated if no security update required**
- Blog post and 42-page report
 - Concrete vulnerability analysis for spatial safety
 - Abstract analysis of the impact of temporal safety
 - Red teaming of specific artifacts to gain experience
- CHERI, “in its current state, and combined with other mitigations, it would have **deterministically mitigated at least two thirds of all those issues**”

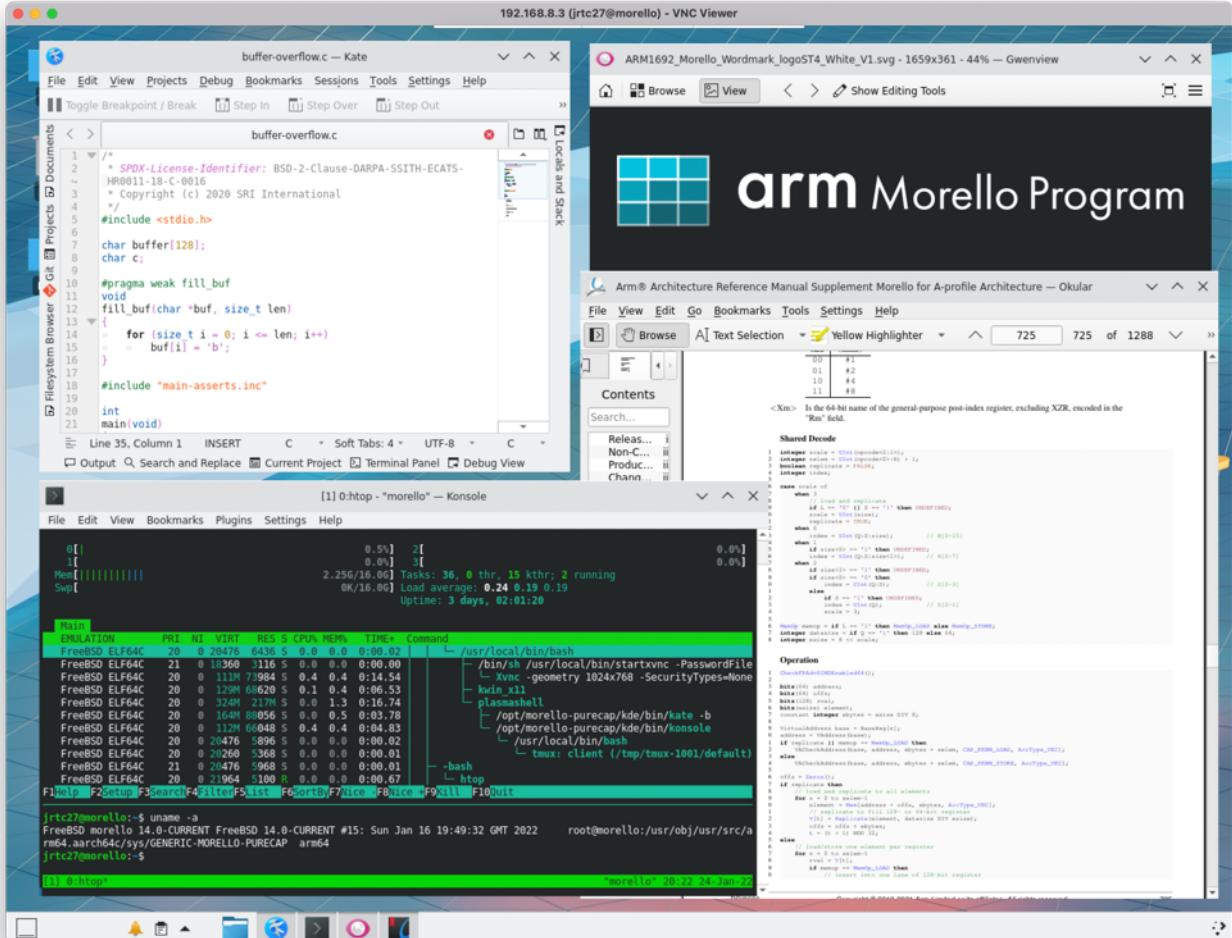
3-month CHERI Desktop UKRI pilot study

InnovateUK-funded project at Capabilities Limited to assess the viability of a CHERI/Morello open-source desktop software stack (on QEMU model):

- Selected slice of open-source desktop stack: X11, Qt, KDE, applications
- Implemented CHERI C/C++ referential and spatial memory protection
- Whiteboarded possible software compartmentalizations
- Evaluated software change as %LoC changed
- Evaluated security via 5-year retrospective vulnerability analysis

<http://www.capabilitieslimited.co.uk/pdfs/20210917-capltd-cheri-desktop-report-version1-FINAL.pdf>

CHERI desktop ecosystem study: Key outcomes



Developed:

- **6 million lines of C/C++ code** compiled for memory safety; modest dynamic testing
- **Three compartmentalization case studies in Qt/KDE**

Evaluation results:

- **0.026% LoC modification rate** across full corpus for memory safety
- **73.8% mitigation rate** across full corpus, using memory safety and compartmentalization

CHERI TRANSITION

Morello and CHERI-RISC-V

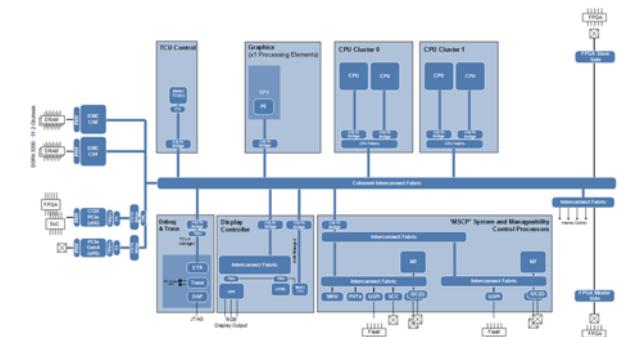
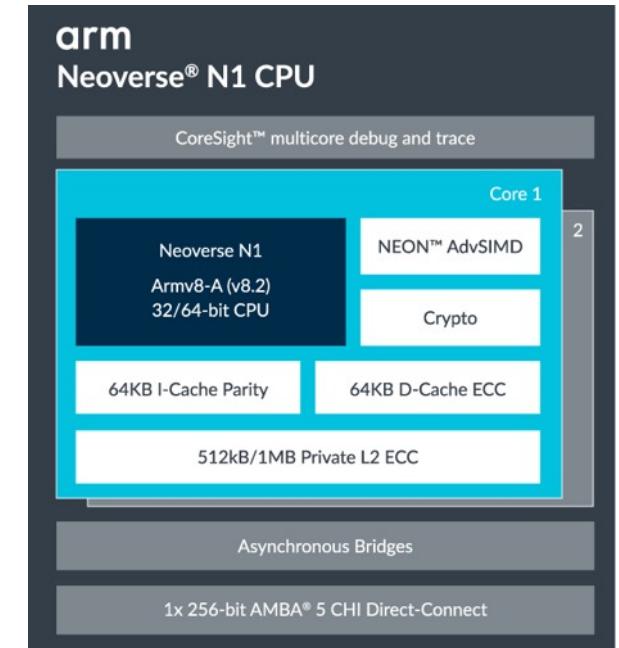
- We are pursuing two CHERI adaptations to post-MIPS ISAs:
 - 2014 Joint with Arm, an experimental adaptation of 64-bit ARMv8-A **Arm Morello** multicore SoC, development board, etc.
(announced Oct. 2019; experimental SoC shipped 2022)
 - 2017 An experimental adaptation of 32/64-bit RISC-V
(open-source research processors on FPGA)
- Complete elaborations of the full hardware-software stack for each ISA:
 - All aspects of the architectures (e.g., ARMv8-A VM features, etc.)
 - Formal models + proofs, hardware implementations, compilers, OSes
- Potential for transition through both paths

CHERI-ARM research since 2014

- Since 2014, in collaboration with Arm, we have been pursuing joint research to experimentally incorporate CHERI into ARMv8-A:
 - Develop CHERI as an architecture-neutral and portable protection model implemented in multiple concrete architectures
 - Refine and extend the CHERI architecture – e.g., capability compression, tagging, μarch, domain transition, and temporal safety
 - Apply concept of architecture neutrality to the CHERI-enabled software stack, including compiler, OS, and applications
 - Expand software: large-scale application experiments, OS use, debuggers, ...
 - Extend work in formal modeling and proofs to an industrial-scale architecture
- Solve arising practical {hardware, software, ...} problems as part of the research
- Build evidence, demonstrations, SW templates to support potential CHERI adoption

ISCF: Digital Security by Design (UKRI)

- 5-year **Digital Security by Design** UKRI program: £70M UK gov. funding, £117M UK industrial match, to create CHERI-ARM demonstrator SoC + board with proven ISA
- Leap supply-chain gap that makes adopting new architecture difficult – in particular, validation of concepts in microarchitecture, architecture, and software “at scale”
- Support industrial and academic R&D (EPSRC, ESRC, InnovateUK)
- Baseline CPU is Neoverse N1; reuses existing SoC/board designs
- Collaborative review distillation of CHERI ISAv8; experimental additions relating to temporal safety, compartmentalization
- Science designed allowed: Multiple architectural + microarchitectural design choices for software-based evaluation
- 2020 emulation models; **January 2022 Morello board shipped!**



arm

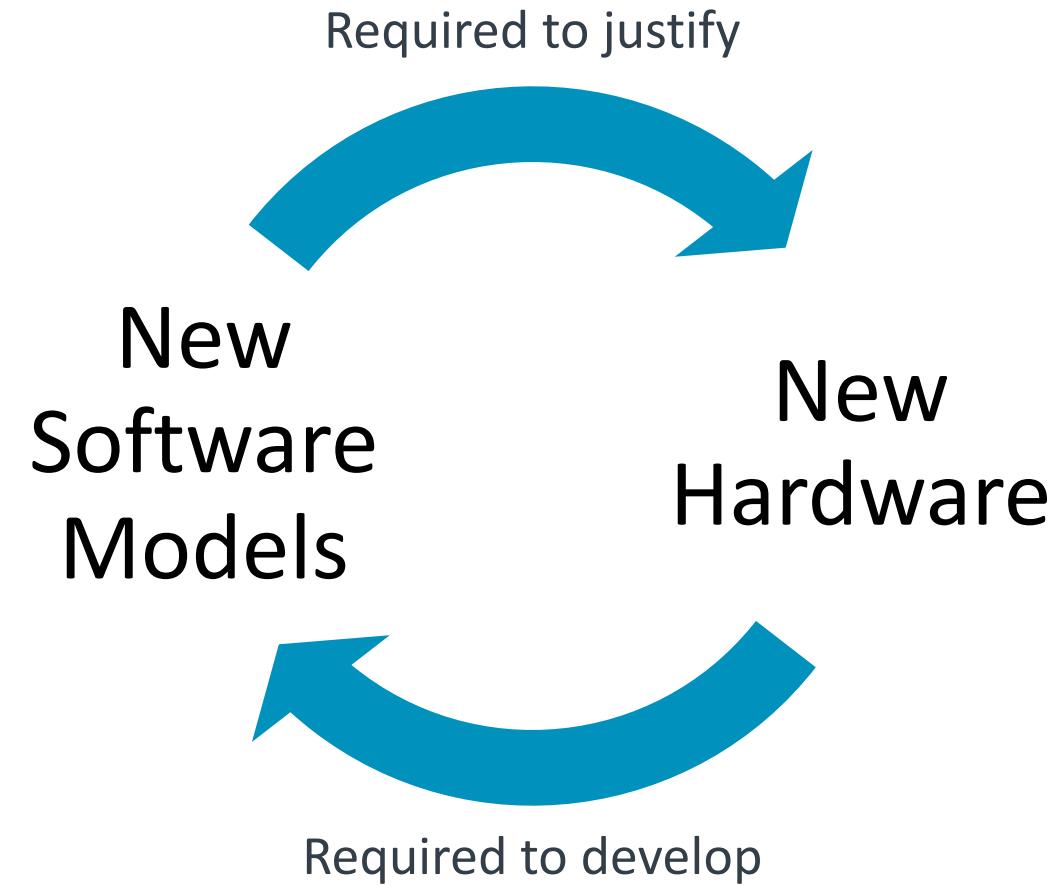
Digital Security by Design

Richard Grisenthwaite

SVP Chief Architect and Fellow

Richard.Grisenthwaite@arm.com

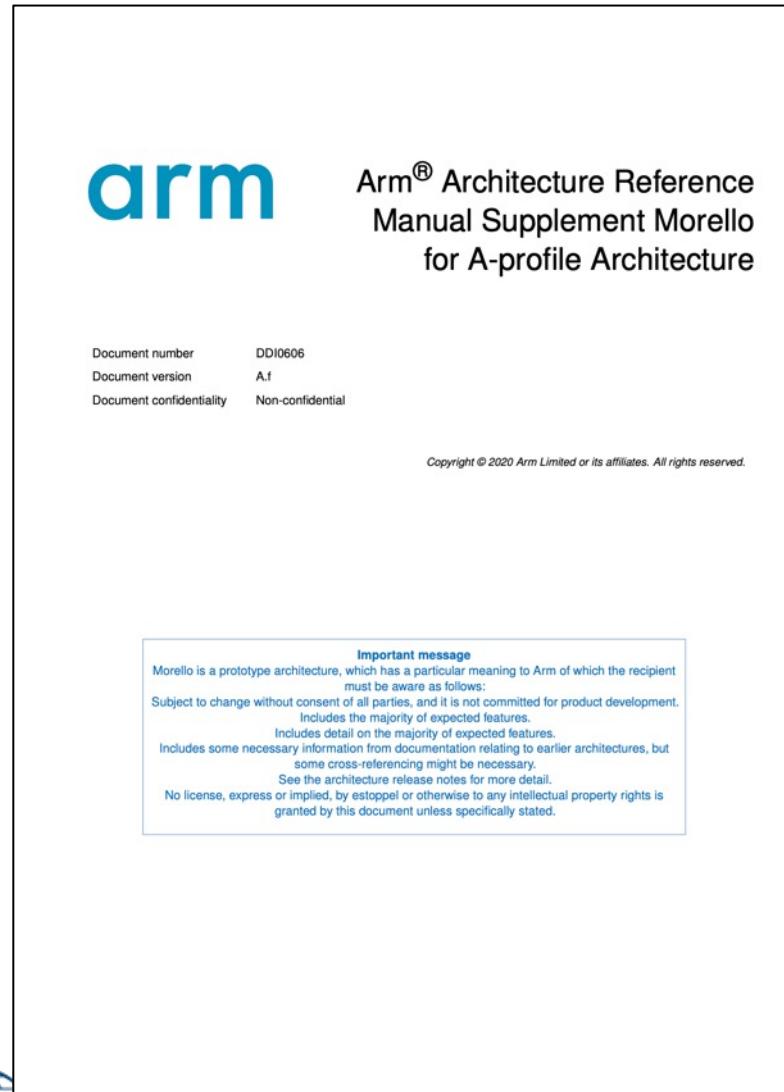
Challenges with creating substantially new architecture



IP Position

- Today's CPU architectures have largely the same basic functionality
 - "Similar but different" approaches to most aspects of system architecture
 - Small scale optimisations exist
- This position very beneficial for the porting of system software
 - Anything that fundamentally changes the system software architecture is likely to be ignored
- Arm believes that this reality needs to continue with capabilities
 - Implication is that we'd like the world's leading architectures to adopt capabilities
 - The Digital Security by Design program

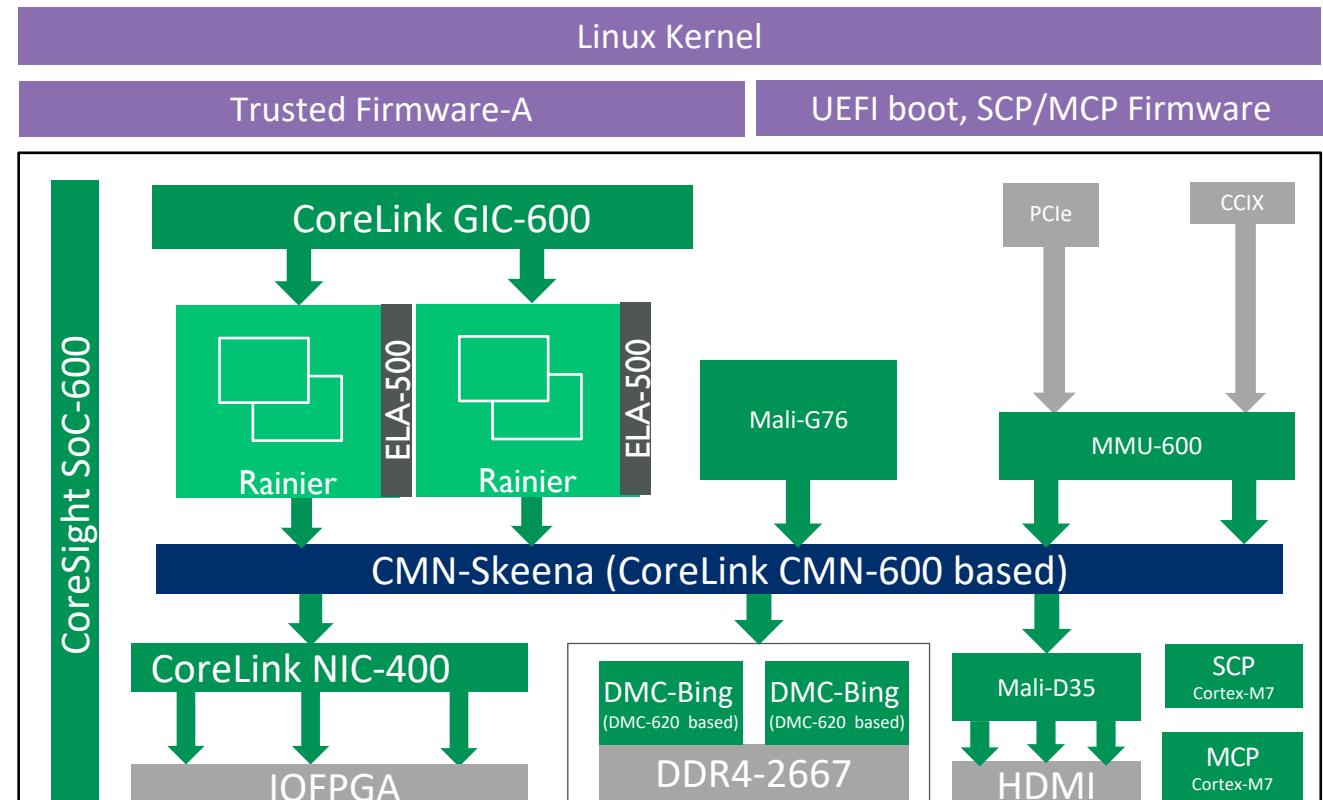
Arm Morello specification



- Experimental application of CHERI ISAv8 to ARMv8-A
- Much richer base ISA .. Much longer spec - 2,155 pages excluding additional material!
- Describes ISA as implemented in Arm Morello FVP and processor/SoC
- Includes recent features such as sentry and load-side barrier support

Morello Board: Capability Hardware Prototype Platform

- Silicon implementation of a Capability Hardware CPU Instruction Set Architecture
 - Implements Morello Profile for A-class Prototype Architecture
 - Two clusters each of two Rainier CPUs
 - Interconnect and Memory Controller support for tagged memory
 - Two channel DDR4 DRAM interface
 - PCIe Gen3 and Gen4 x16 interface
 - CCIX (Cache Coherent Interconnect for Accelerators) interface
 - Mid-range GPU, display processor and HDMI output
 - On standard uATX form factor board



Legend:

- Supporting Arm system IP: GIC-600 (Generic Interrupt Controller), MMU-600 (IO MMU), Dynamic Memory Controller derived from DMC-620, SoC-600 (SoC Debug and Trace), Coherent Mesh Network derived from CMN-600, NIC-400 (Non-coherent interconnect)
- Supporting 3rd party system IP/hardware: PCIe/CCIX Root Complex (PHY and controller), DDR4/3 PHY, DDR4 memory, IO FPGA
- Open-source software stack

CHERI prototype software stack on Morello

- **Complete open-source software stack** from bare metal up: compilers, toolchain, debuggers, hypervisor, OS, applications – all demonstrating CHERI
 - Rich CHERI feature use, but fundamentally incremental/hybridized deployment
 - Aim: Mature and highly useful research and development platform for Morello

Open-source application suite (KDE, X11, WebKit, Python, OpenSSH, nginx, PostgreSQL ...)

CheriBSD/Morello (funded by DARPA and UKRI)

- FreeBSD kernel + userspace, application stack
- Kernel spatial and referential memory protection
- Userspace spatial, referential, and temporal memory protection
- Co-process compartmentalization
- Intra-process compartmentalization
- Morello-enabled bhyve Type-2 hypervisor
- ARMv8-A 64-bit binary compatibility for legacy binaries

Android (Arm)
(Morello only)

Linux (Arm)
(Morello only)

CHERI-extended Google Hafnium hypervisor

CHERI Clang/LLVM compiler suite, LLD, LLDB, GDB

Baseline CHERI
Clang/LLVM from
SRI/Cambridge;
Morello
adaptation by
Arm + Linaro

UK EPSRC DSbD research program 2020-2023

EPSRC Competition

- £10M Research funding
 - £7M from ISCF/DSbD
 - £3m from DCMS
- The EPSRC call covered 3 areas:
 - Capability enabled hardware proof and software verification
 - Impact on system software and libraries
 - Future implications of capability enabled Hardware
- Projects starting July-Oct

Selected Projects

AppControl: Enforcing Application Behaviour through Type-Based Constraints
Dr Wim Vanderbauwhede (University of Glasgow)

CapableVMs – Capable Virtual Machines
Dr Laurence Tratt (King's College London) & Dr Jeremy Singer (University of Glasgow)

CAPcelerate: Capabilities for Heterogeneous Accelerators
Dr Timothy Jones (University of Cambridge)

CapC: Capability C semantics, tools and reasoning
Dr Mark Batty (University of Kent)

CAP-TEE: Capability Architectures for Trusted Execution
Dr David Oswald (University of Birmingham)

CHaOS: CHERI for Hypervisors and Operating Systems
Dr Robert Watson (University of Cambridge)

CloudCAP: Capability-based Isolation for Cloud-Native Applications
Prof Peter Pietzuch (Imperial College London)

HD-Sec: Holistic Design of Secure Systems on Capability Hardware
Professor Michael Butler (University of Southampton)

SCorCH: Secure Code for Capability Hardware
Dr Giles Reger (The University of Manchester)
Prof Daniel Kroening (University of Oxford)



- 9 EPSRC projects funded across 10 UK universities
- Several InnovateUK industrial projects supporting exploration, evaluation, demonstration

Some potential software research areas

- **Clean-slate OSes and languages**

Current research has focused on incremental CHERI adoption within current software and languages. How would we design new OSes, languages, etc., assuming CHERI as an ISA baseline?

- **Compilers, language runtimes, and JITs**

How can we mitigate the performance overheads of more pointer-dense executions, such as with language runtimes? Are vulnerabilities in code generated by compilers and JIT susceptible to mitigation using CHERI? How does CHERI break or potentially improve current compiler analyses and optimization?

- **Further C/C++ protections with CHERI**

We have focused on spatial, referential, and temporal memory safety for C/C++. But the CHERI primitives could assist with data-oriented protections, garbage collection, type checking, etc. Could these improve security, and at what performance cost?

- **Safe and managed languages**

Languages such as Java, Rust, C#, OCaml, etc., offer strong safety properties, but frequently depend on C/C++ runtimes and FFI-linked native code. Can CHERI provide stronger foundations for higher-level language stacks?

- **Virtualization**

Can memory protection usefully harden hypervisors? Can we compartmentalize hypervisors? Can CHERI offer a better mechanism for virtualizing code than an MMU?

- **Debuggers and tracing**

Debugging/tracing tools rely on high levels of privilege to operate. How can we reduce their privilege to mitigate vulnerabilities in these tools? With stronger architectural semantics, is new dynamic analysis possible?

- **Software compartmentalization tools**

Granular software compartmentalization offers vulnerability mitigation through privilege reduction and strong encapsulation. How should current applications be refactored, and new applications be designed, to accomplish maintainable and more secure software?

- **Security evaluation and adversarial research**

What is the impact of CHERI on known vulnerabilities and attack techniques? How does a CHERI-aware attacker change their behavior? Could formal models and proofs support stronger security arguments for CHERI?

Conclusion

CHERI architectural primitives require rich HW and SW evaluation:

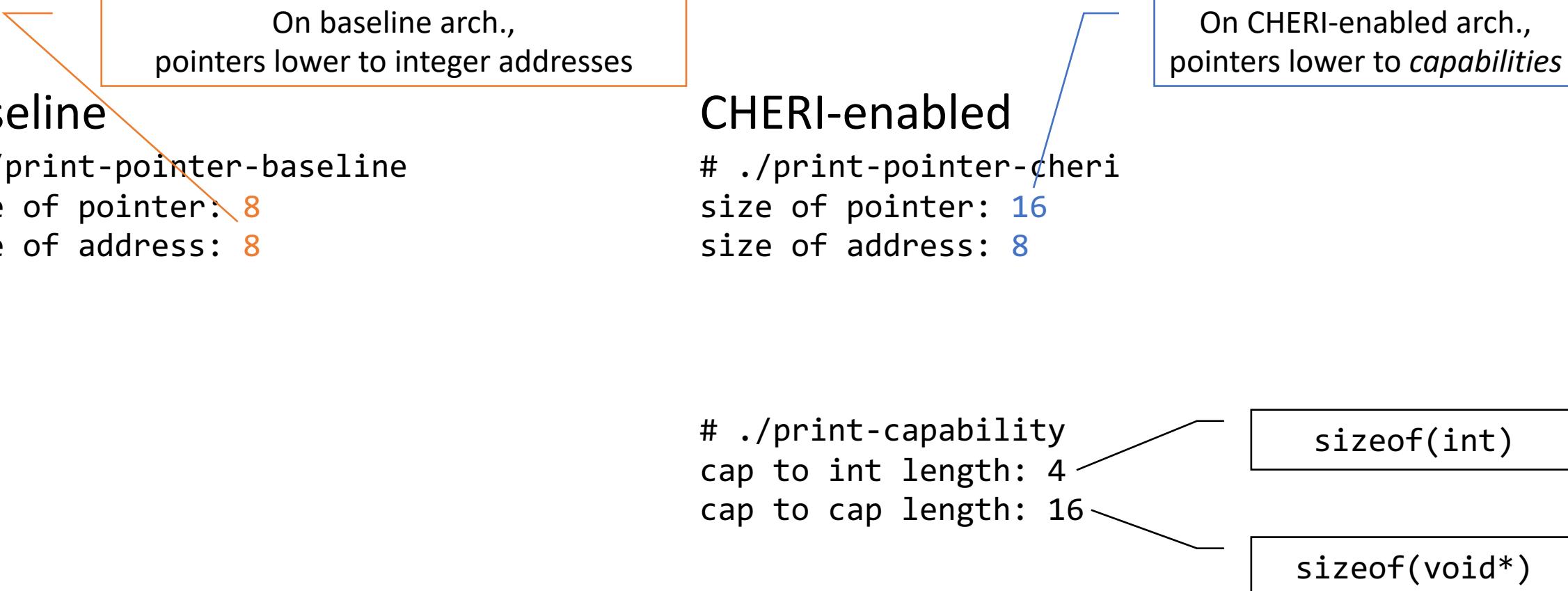
- CHERI C/C++ offers strong protection from memory-related attacks
- CHERI compartmentalization has much higher performance than MMU-based techniques
- Arm Morello integrates CHERI protection into an experimental implementation in an industrial quality implementation – which we are eager to validate at scale!

Where to learn more:

<http://www.cheri-cpu.org/>

- Watson, et al. **An Introduction to CHERI**, Technical Report UCAM-CL-TR-941, Computer Laboratory, September 2019.
- Watson, et al. **Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)**, UCAM-CL-TR-951, October 2020.
- Watson, et al. **CHERI C/C++ Programming Guide**, UCAM-CL-TR-947, June 2020.

Explaining Output From Prerequisite Exercise

	On baseline arch., pointers lower to integer addresses	On CHERI-enabled arch., pointers lower to <i>capabilities</i>
Baseline	<pre># ./print-pointer-baseline size of pointer: 8 size of address: 8</pre>	
CHERI-enabled		<pre># ./print-pointer-cheri size of pointer: 16 size of address: 8</pre> <pre># ./print-capability cap to int length: 4 cap to cap length: 16</pre> 

Exercise: CHERI Pointer Integrity

Introduction

- Spatial safety *depends upon* reference/pointer integrity
 - Pointers must come from other pointers.
 - Can't *forgo* a reference (and associated bounds) arbitrarily
- CHERI *tags* prevent confusion between *data per se* and *capabilities*

 §2.2 Demonstrate CHERI Tag Protection !

(& keep  §1.7 Cheatsheet for the CHERI Software Release to hand)

Exercise: CHERI Pointer Integrity

Discussion: Contrasting Baseline and CHERI

- Exercise program constructed “the same” pointer two different ways:
 - By *setting the last byte* of a pointer in memory
 - By *transforming the address* of a language-level pointer

RISC-V Baseline

`q=0x80ec2000 (0xb7 into buf)`

`*q=b7`

`r=0x80ec2000 (0xb7)`

`*r=b7`

CHERI-RISC-V

`q=0x3ffffdffe00 [rwRW,0x3ffffdff71-0x3ffffdff70] (0x8f into buf)`

`*q=8f`

`r=0x3ffffdffe00 [rwRW,0x3ffffdff71-0x3ffffdff70] (invalid) (0x8f)`

`In-address space security exception`

Exercise: CHERI Pointer Integrity

Discussion: gdb

Program received signal **SIGPROT**, CHERI protection violation

Capability tag fault caused by register **cs1**.

main () at ./src/exercises/cheri-tags/corrupt-pointer.c:45

Thread 1 (LWP 100057 of process 1231):

#0 main () at ./src/exercises/cheri-tags/corrupt-pointer.c:45

(gdb) p \$_siginfo

\$1 = {si_signo = **34**, si_errno = 0, si_code = **2**, ...
 _reason = {_fault = {si_trapno = 28, si_capreg = **9**}}, ...}

AKA PROT_CHERI_TAG

Exercise: CHERI Pointer Integrity

Discussion: Investigating the Opcodes

Ptr	Op	Baseline	CHERI
r	Store	sb zero, 0(sp)	<code>csb zero, 32(csp)</code>
	Load	ld s0, 0(sp)	<code>clc cs1, 32(csp)</code>
q	Load	ld a0, 0(sp)	<code>clc ca0, 32(csp)</code>
	Extract		<code>cgetaddr a1, ca0</code>
	Mask	andi s0, a0, -256	<code>andi a1, a1, -256</code>
	Update		<code>csetaddr cs1, ca0, a1</code>

- CHERI *data* instructions *clear tags*, *capability* instructions *preserve*
 - Tagged capabilities have *provenance* that is *exclusively capability instructions*
 - Initial tagged quantities provided in registers at boot!



Exercise: Stack Data Buffer Overflow

Introduction

- CHERI offers architectural mechanisms for *spatially-safe* C.
 - What does that mean? How does it work in practice?

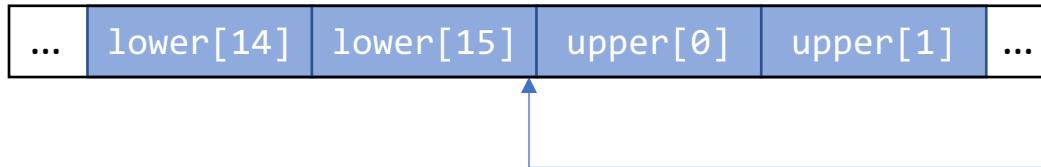


§2.3 Exercise an inter-stack-object buffer overflow

Exercise: Stack Data Buffer Overflow

Introduction: Stack Layout

- Straightforward buffer overflow:



- Write OOB to lower & damage upper
- C rules this *undefined behavior*

§2.3 !

```
void write_buf(char *buf, size_t ix)
{
    buf[ix] = 'b';
}

int main(void)
{
    char upper[0x10];
    char lower[0x10];

    printf("upper = %p, lower = %p, diff = %zx\n",
           upper, lower, (size_t)(upper - lower));

    /* Assert that these get placed how we expect */
    assert((ptraddr_t)upper
           == (ptraddr_t)&lower[sizeof(lower)]);

    upper[0] = 'a';
    printf("upper[0] = %c\n", upper[0]);

    → write_buf(lower, sizeof(lower));

    printf("upper[0] = %c\n", upper[0]);
    return 0;
}
```

Exercise: Stack Data Buffer Overflow

Discussion: So, what happened?

RISC-V Baseline

```
# ./buffer-overflow-stack-baseline
upper = 0x80d879d0, lower = 0x80d879c0, diff = 10
upper[0] = a
upper[0] = b
```

CHERI-RISC-V

```
# ./buffer-overflow-stack-cheri
upper = 0x3ffffdff50, lower = 0x3ffffdff40, diff = 10
upper[0] = a
In-address space security exception
```

- Baseline CPU wrote to “16th” position in `lower`, aliasing `upper`
- CHERI CPU trapped; kernel delivered fatal SIGPROT
- How did the CHERI CPU know to do that?

Exercise: Stack Data Buffer Overflow

Discussion: gdb

Program received signal SIGPROT, CHERI protection violation
Capability bounds fault caused by register ca0.

0x0000000000101ce8 in write_buf (buf=<optimized out>, ix=<optimized out>)
at ./buffer-overflow-stack.c:13

```
13           buf[ix] = 'b';
```

(gdb) disass

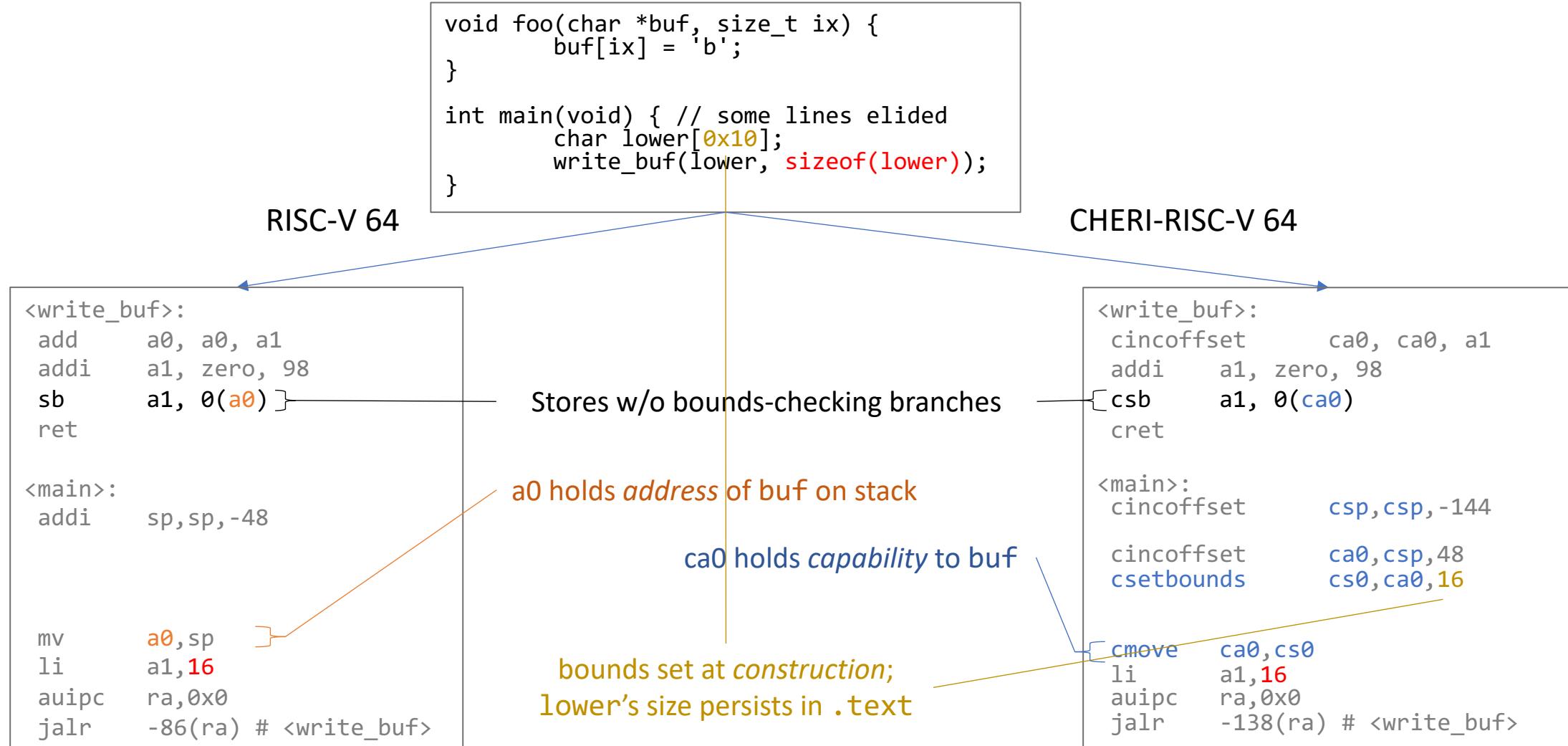
Dump of assembler code for function write_buf:

0x0000000000101ce0 <+0>:	cincoffset	ca0,ca0,a1
0x0000000000101ce4 <+4>:	li	a1,98
=> 0x0000000000101ce8 <+8>:	sb	a1,0(a0)
0x0000000000101cec <+12>:	ret	

End of assembler dump.

Exercise: Stack data buffer overflow

Discussion: Program .text



Morning Tea (11h00 – 11h30)

Exercise: Explore Sub-Object Bounds

Introduction

- CHERI C defaults to bounding to “allocations” or “objects”
 - Pointers into arrays and structures *inherit* bounds from container
- “Sub-object” overflows not stopped by default
- Compilation flags for sub-object bounds hardening
 - (And directives for fine-tuning in source in 2nd part of exercise; “extra credit”)

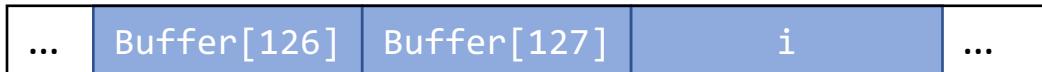


§2.4 Explore Subobject Bounds

Exercise: Explore Sub-Object Bounds

Introduction: Structure Layout

- Structure representation in memory:



- Fill loop risks buffer overflow
- C rules this, too, *undefined behavior*



```
struct buf {  
    char buffer[128];  
    int i;  
} b;  
  
#pragma weak fill_buf  
void  
fill_buf(char *buf, size_t len)  
{  
    for (size_t i = 0; i <= len; i++)  
        buf[i] = 'b';  
}
```

Exercise: Sub-object Overflow

Part 1 Discussion

RISC-V Baseline and CHERI!

```
b.i = c
```

```
b.i = b
```

```
Breakpoint 1,
```

```
fill_buf (buf=0x103e50 <b>  
[rwRW,0x103e50-0x103ed4] "",  
len=128)
```

Capability length:
132 bytes!

CHERI with Sub-object Hardening

```
b.i = c
```

In-address space security exception

```
Breakpoint 1,
```

```
fill_buf (buf=0x103e50 <b>  
[rwRW,0x103e50-0x103ed0] "",  
len=128)
```

Capability length:
128 bytes

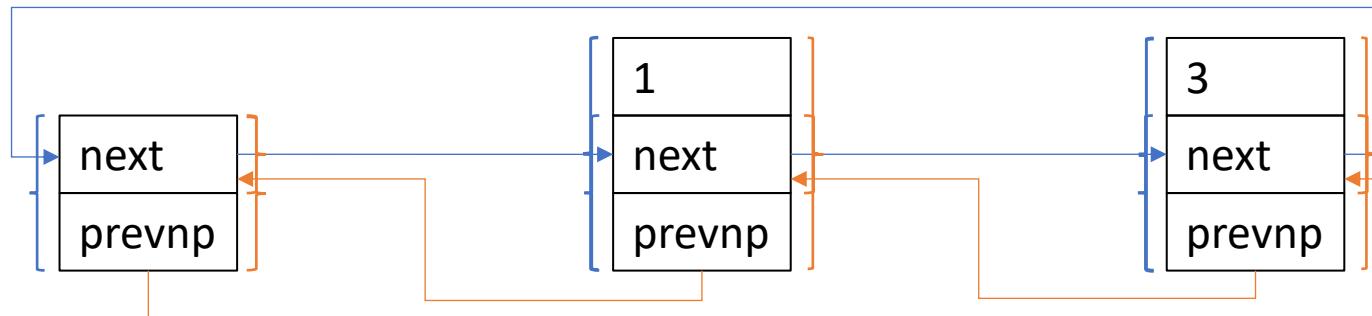
Exercise: Sub-object Overflow

Part 2 Discussion: Why isn't this the default?

- C spec defines `offsetof()` primitive and `char*` casts
- Software uses `containerof()` for intrusive data structures
 - Especially popular in “systems” and “runtime” code
- In general, incorrect to narrow bounds of pointers to sub-objects?

Exercise: Sub-object Overflow

Part 2 Discussion



- Without sub-object bounds narrowing, all caps include full structure
- Applying sub-object bounds *everywhere*:
 - Next pointers grant access to **whole intrusive list structure**,
 - Previous pointers only to **next pointers**
- Annotations can widen pointers as needed

Exercise: Sub-object Overflow

Part 2 Discussion: Could It be the Default in the Future?

- Counterpoint: `offsetof()` / `containerof()` not *that* common.
- Add static asserts to `containerof`, enforce sub-objects non-narrowing.
 - This is what you saw with `-DUSE_CDEFS_CONTAINEROF`

Exercise: Spatially Safe Heap

Introduction

- Heap memory (malloc) provided by library code
 - Responsible for handing out unused, unaliased address space upon request
 - For CHERI C runtime, must provide *spatial safety* too!
- What goes wrong when heap isn't spatially safe?



§2.5 Exercise heap overflows

Exercise: Spatially Safe Heap

Introduction

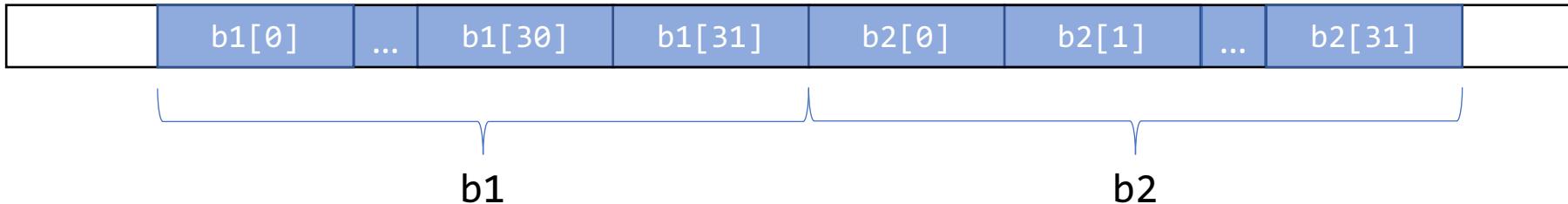
Explore two aspects of heap memory:

- *Preventing inter-allocation overflow* with CHERI bounds
- *Reaching allocator metadata* via the pointer argument to free despite bounds and monotonicity



Exercise: Spatially Safe Heap

Discussion for 0x20



RISC-V baseline

b1=0x83e82000 b2=0x83e82020 diff=20

Overflowing by 1

b2 begins: **ABBB**

Overflowing by 2

b2 begins: **AABB**

CHERI-RISC-V

sz=20, CRRL(sz)=20

b1=0x407c7000 [rwRW, 0x407c7000-0x407c7020]

b2=0x407c7020 [rwRW, 0x407c7020-0x407c7040]

diff=20

Overflowing by 1

In-address space security exception

Exercise: Spatially Safe Heap

Discussion: Compressed Bounds' Precision

- Heap allocations occasionally large
 - Exposes capability representation
- CHERI capability logically has...
 - Address (64 bits), base (64), limit (64)
 - Permissions, type, flags, ...
- “[CHERI Concentrate](#)”: 256 bits in 128
 - Base & Limit usually “near” Addr

Length \leq 4096: 1-byte alignment



Length \leq 8192: 8-byte alignment



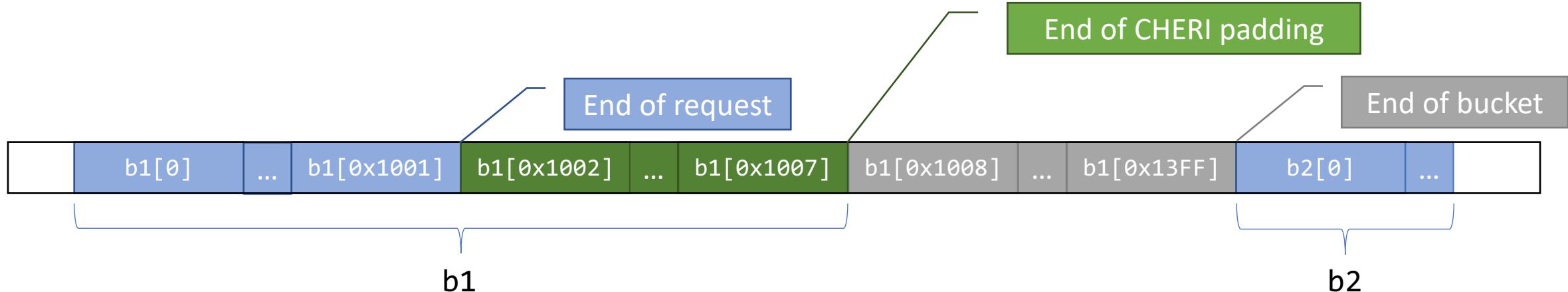
Length \leq 16384: 16-byte alignment



⋮

Exercise: Spatially Safe Heap

Discussion for 0x1001



RISC-V baseline

b1=0x840ec000 b2=0x840ed400 diff=1400

Overflowing by 1

b2 begins: BBBB

Overflowing by 401

b2 begins: AABB

CHERI-RISC-V

sz=1001, CRRL(sz)=1008

b1=0x407c7000 [rwRW, 0x407c7000-0x407c8008]

b2=0x407c8400 [rwRW, 0x407c8400-0x407c9408]

diff=1400

Overflowing by 1

b2 begins: BBBB

Overflowing by 401

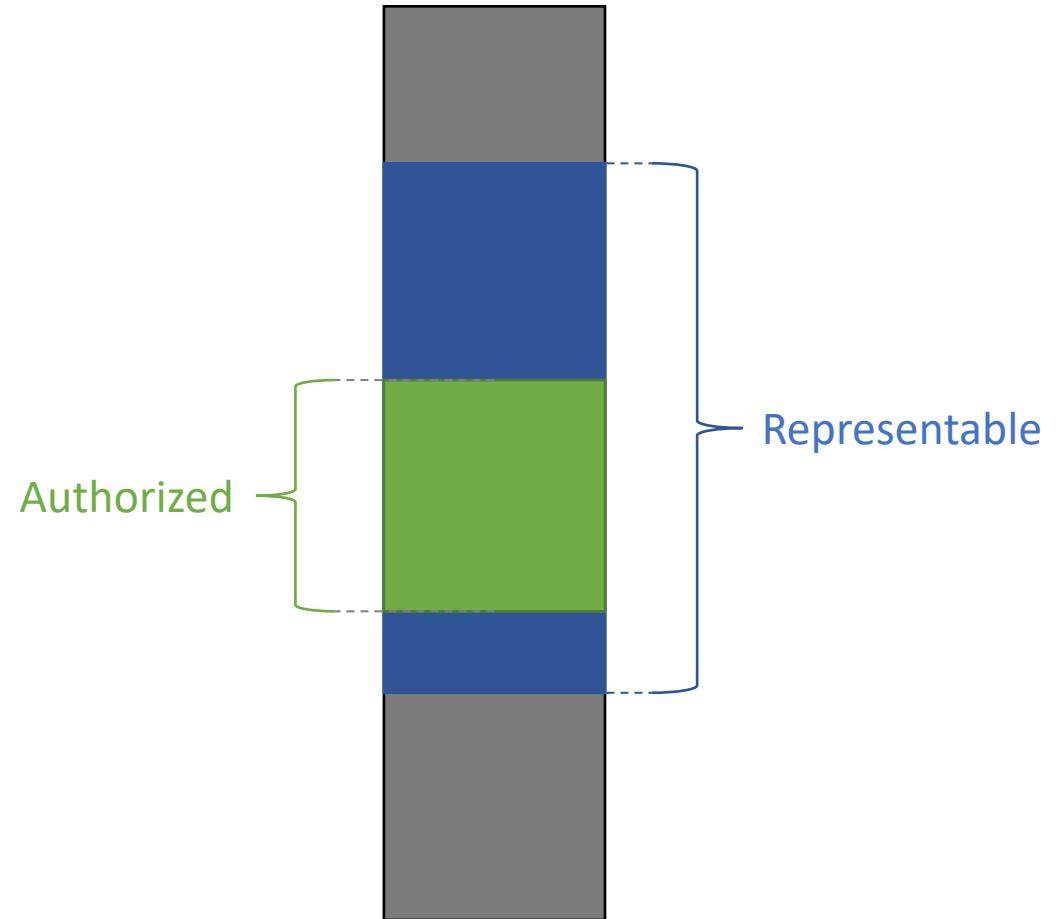
In-address space security exception

Exercise: Spatially Safe Heap

Discussion (bonus): Representable Regions

CHERI capabilities can be out of bounds, but not arbitrarily so:

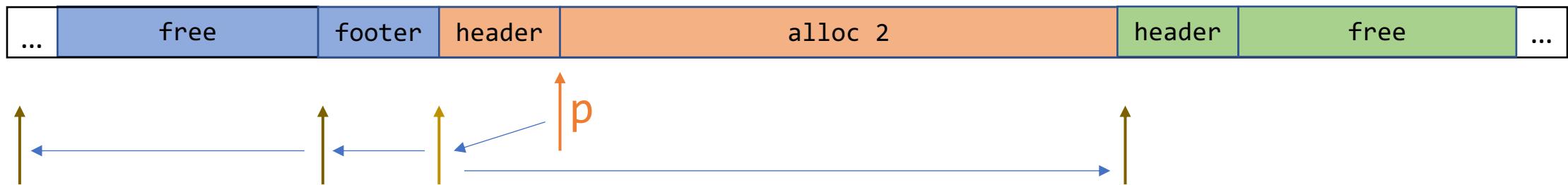
- *de facto* C takes pointers beyond bounds and brings them back
- Cap compression means OOB limited to *representable window*
- $>1/8^{\text{th}}$ object below, $>1/4^{\text{th}}$ above



Exercise: Spatially Safe Heap

Discussion (bonus): Non-monotonicity in `free()`

- Popular trick in mallocs: metadata next to data. `dlmalloc`:



- In baseline arch., `free(p)` “cheats” to find **header** & adjacent blocks.
 - If `malloc()` bounds returns, all these are OOB caps on CHERI!
- CHERI-aware heaps find heap metadata from *globals* (tree, table, ...)

Exercise: Adapt a C Program to CHERI C

Introduction: pointer provenance validity (1/3)

- CHERI C/C++ implement pointers using architectural capabilities;
- Only pointers implemented using valid capabilities can be dereferenced. Otherwise, a dereference would fire a hardware exception;
- An integer data type cast to a pointer data type results in a NULL-derived capability without a tag;
- However, there are data types that can hold pointer or integer values (e.g., `uintptr_t`).



[§4.2, §4.2.1, CHERI C/C++ Programming Guide](#)

Exercise: Adapt a C Program to CHERI C

Introduction: pointer provenance validity (2/3)

- In the CHERI memory protection model, capabilities are derived from a single other capability;
- In CHERI C/C++, a capability can be a result of a complex expression with multiple data types and casts;
- A variable that can hold a valid capability but should not be used as a source capability must be appropriately cast to indicate that (e.g., to an integer data type).



§4.2.3, CHERI C/C++ Programming Guide

Exercise: Adapt a C Program to CHERI C

Introduction: pointer provenance validity (3/3)

- Ideally, we would like to recompile source code for CheriABI and automatically gain security;
- Unfortunately, there is a lot of software that use incorrect data types to hold values that fit in them but have different semantics.

Exercise: Adapt a C Program to CHERI C

Introduction: CHERI compiler warnings and errors

- CHERI LLVM can identify capability-related issues and print warnings:
 - Loss of provenance (-Wcheri-capability-misuse);
 - Ambiguous provenance (-Wcheri-provenance);
 - Underaligned capabilities of packed structures (-Wcheri-capability-misuse);
 - Underaligned load of capability type (-Wcheri-inefficient).
- Sanitizers (available in the master branch, not in the Summer 2021 release):
 - A group of CHERI-specific sanitizers (-fsanitize=cheri);
 - Detect capabilities that become unrepresentable when significantly (>1) out-of-bounds (-fsanitize=cheri-unrepresentable);
 - In the future: detect capabilities that are created >1 out-of-bounds.



[§6, CHERI C/C++ Programming Guide](#)



[Assessing the Viability of an Open-Source CHERI Desktop Software Ecosystem](#)

Exercise: Adapt a C Program to CHERI C



§2.6 Adapt a C Program to CHERI C 🧑💻!

Exercise: Adapt a C Program to CHERI C

Discussion for -Wcheri-provenance

```
# ./cat-baseline /etc/hostid  
bb5fbb47-10ab-11ec-a609-f5a47707c223
```



```
# ./cat-cheri /etc/hostid  
cat-cheri: write(2) failed: Bad address
```

Exercise: Adapt a C Program to CHERI C

Discussion for -Wcheri-provenance

```
write(fd, (const void *)off + (uintptr_t)buf), nbytes)
```

```
Breakpoint 1, _write () at _write.S:4
```

```
(gdb) disassemble
```

```
Dump of assembler code for function _write:
```

```
=> 0x0000000040299130 <+0>:    li    t0,4
```

```
          0x0000000040299132 <+2>:    ecall
```

```
(gdb) info registers ca0 ca1 ca2
```

```
ca0      0x1    0x1
```

```
ca1      0x40810000  0x40810000
```

```
ca2      0x25   0x25
```

```
(gdb) ni 2
```

```
4     in _write.S
```

```
(gdb) info registers ca0 ct0
```

```
ca0      0xe    0xe
```

```
ct0      0x1    0x1
```

```
(gdb)
```

Exercise: Adapt a C Program to CHERI C

Discussion for -Wcheri-provenance

```
./src/exercises/adapt-c/cat/methods.c:70:43: warning: binary expression  
on capability types 'ptroff_t' (aka 'unsigned __intcap') and 'uintptr_t'  
(aka 'unsigned __intcap');
```

it is not clear which should be used as the source of provenance; currently provenance is inherited from the left-hand side [-Wcheri-provenance]

```
return (write(fildes, (const void *)(off + (uintptr_t)buf), nbyte));
```

~~~ ^ ~~~~~

# Exercise: Adapt a C Program to CHERI C

## Discussion for -Wcheri-provenance

```
write(fd, (const void *)(off + (uintptr_t)buf), nbytes)
```

```
Breakpoint 1, _write () at _write.S:4
```

```
(gdb) disassemble
```

```
Dump of assembler code for function _write:
```

```
=> 0x0000000040299130 <+0>: li t0,4
```

```
    0x0000000040299132 <+2>: ecall
```

```
(gdb) info registers ca0 ca1 ca2
```

```
ca0 0x1 0x1
```

```
ca1 0x40810000 0x40810000
```

```
ca2 0x25 0x25
```

```
(gdb) ni 2
```

```
4 in _write.S
```

```
(gdb) info registers ca0 ct0
```

```
ca0 0xe 0xe
```

```
ct0 0x1 0x1
```

```
(gdb)
```

```
write(fd, (const void *)(size_t off + (uintptr_t)buf), nbytes)
```

```
Breakpoint 1, _write () at _write.S:4
```

```
(gdb) disassemble
```

```
Dump of assembler code for function _write:
```

```
=> 0x0000000040299130 <+0>: li t0,4
```

```
    0x0000000040299132 <+2>: ecall
```

```
(gdb) info registers ca0 ca1 ca2
```

```
ca0 0x1 0x1
```

```
ca1 0xd17d000000180040000000040810000
```

```
0x40810000 [rwRW,0x40810000-0x40811000]
```

```
ca2 0x25 0x25
```

```
(gdb) ni 2
```

```
bb5fbb47-10ab-11ec-a609-f5a47707c223
```

```
4 in _write.S
```

```
(gdb) info registers ca0 ct0
```

```
ca0 0x25 0x25
```

```
ct0 0x0 0x0
```

```
(gdb)
```

# Exercise: Adapt a C Program to CHERI C

## Discussion for -Wcheri-capability-misuse

```
# ./cat-baseline -n /etc/hostid  
1 bb5fbb47-10ab-11ec-a609-f5a47707c223
```

```
# ./cat-cheri -n /etc/hostid  
In-address space security exception (core dumped)
```

# Exercise: Adapt a C Program to CHERI C

## Discussion for -Wcheri-capability-misuse

Program received signal SIGPROT, CHERI protection violation

Capability tag fault caused by register cs2.

verbose\_cat (file=<optimized out>) at  
./src/exercises/adapt-c/cat/methods.c:87

```
(gdb) info registers cs2
cs2      0x4037a400  0x4037a400
```

```
(gdb) disassemble $pcc,+4
Dump of assembler code from 0x103094 to 0x103098:
=> 0x0000000000103094 <do_cat+228>: lw    a0,16($2)
End of assembler dump.
```

```
(gdb) p fp
$1 = (FILE *) 0x4037a400
(gdb)
```

# Exercise: Adapt a C Program to CHERI C

## Discussion for -Wcheri-capability-misuse

./src/exercises/adapt-c/cat/methods.c:80:7: warning: cast from provenance-free integer type to pointer type will give pointer that can not be dereferenced [-Wcheri-capability-misuse]

```
fp = (FILE *)file;  
^
```

|                                                                                                         |                                                                                                              |
|---------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| <pre>static void<br/>verbose_cat(<b>long</b> file)<br/>{<br/>    (...)<br/>    fp = (FILE *)file;</pre> | <pre>static void<br/>verbose_cat(<b>uintptr_t</b> file)<br/>{<br/>    (...)<br/>    fp = (FILE *)file;</pre> |
|---------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|

# Exercise: Adapt a C Program to CHERI C

## Discussion for -Wcheri-capability-misuse

```
Program received signal SIGPROT, CHERI protection violation  
Capability tag fault caused by register cs2.  
verbose_cat (file=<optimized out>) at  
./src/exercises/adapt-c/cat/methods.c:87
```

```
(gdb) info registers cs2  
cs2      0x4037a400  0x4037a400
```

```
(gdb) disassemble $pcc,+4  
Dump of assembler code from 0x103094 to 0x103098:  
=> 0x0000000000103094 <do_cat+228>: lw    a0,16($2)  
End of assembler dump.
```

```
(gdb) p fp  
$1 = (FILE *) 0x4037a400  
(gdb)
```

```
Breakpoint 1, verbose_cat  
(file=320992984091701168938624228367068013568) at  
./src/exercises/adapt-c/cat/methods.c:87
```

```
(gdb) info registers cs2  
cs2      0xf17d000000b5a404000000004037a400  
0x4037a400 [rwRW,0x4037a400-0x4037c2d0]
```

```
(gdb) disassemble $pcc,+4  
Dump of assembler code from 0x103082 to 0x103086:  
=> 0x0000000000103082 <do_cat+226>: lw    a0,16($2)  
End of assembler dump.
```

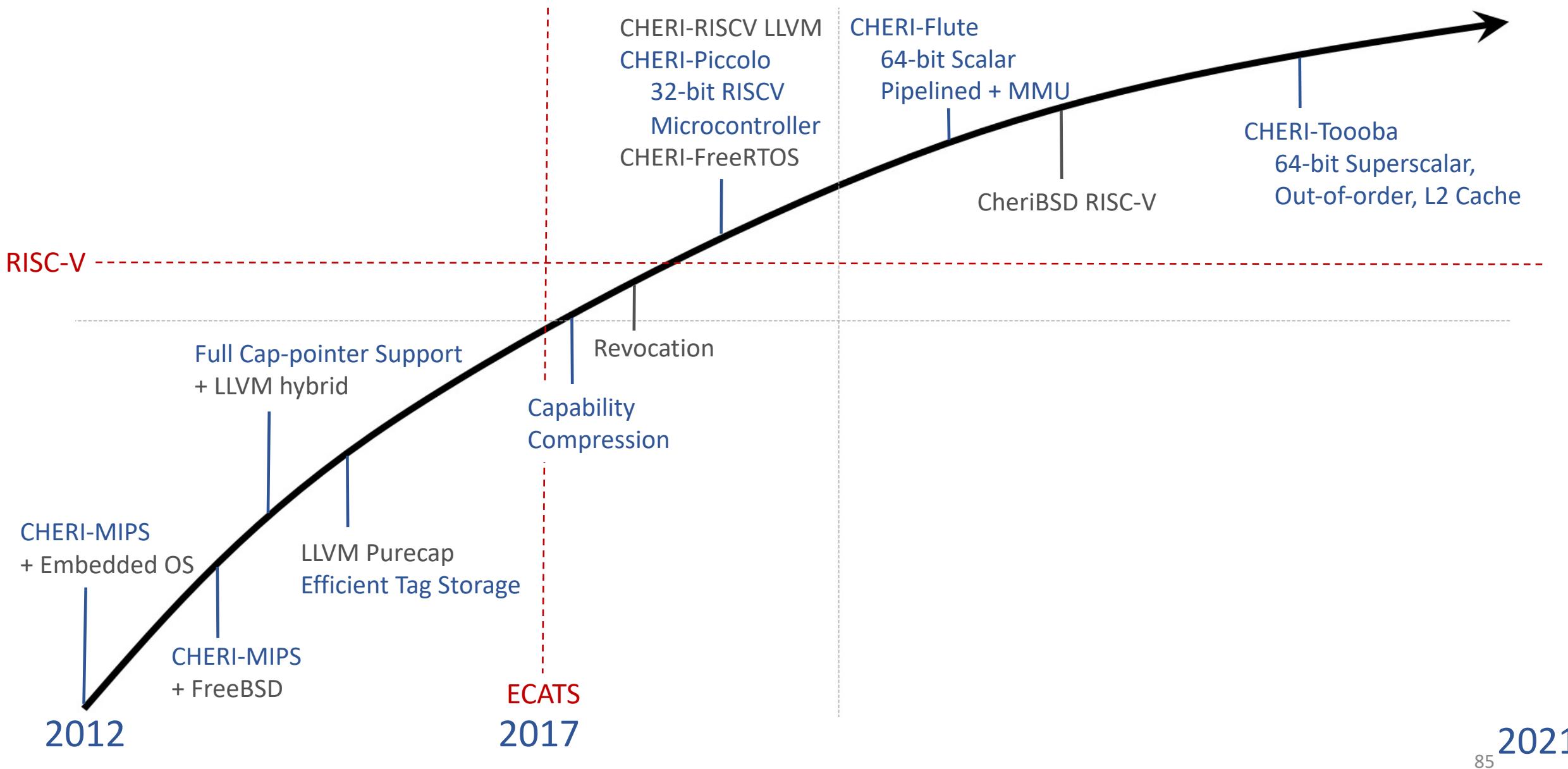
```
(gdb) p fp  
$1 = (FILE *) 0x4037a400 [rwRW,0x4037a400-0x4037c2d0]  
(gdb)
```

Lunch (13h00 – 14h00)

# Tooba CHERI-RISC-V

A superscalar, out-of-order, multi-core reference implementation.

# History of CHERI Architecture and Innovations



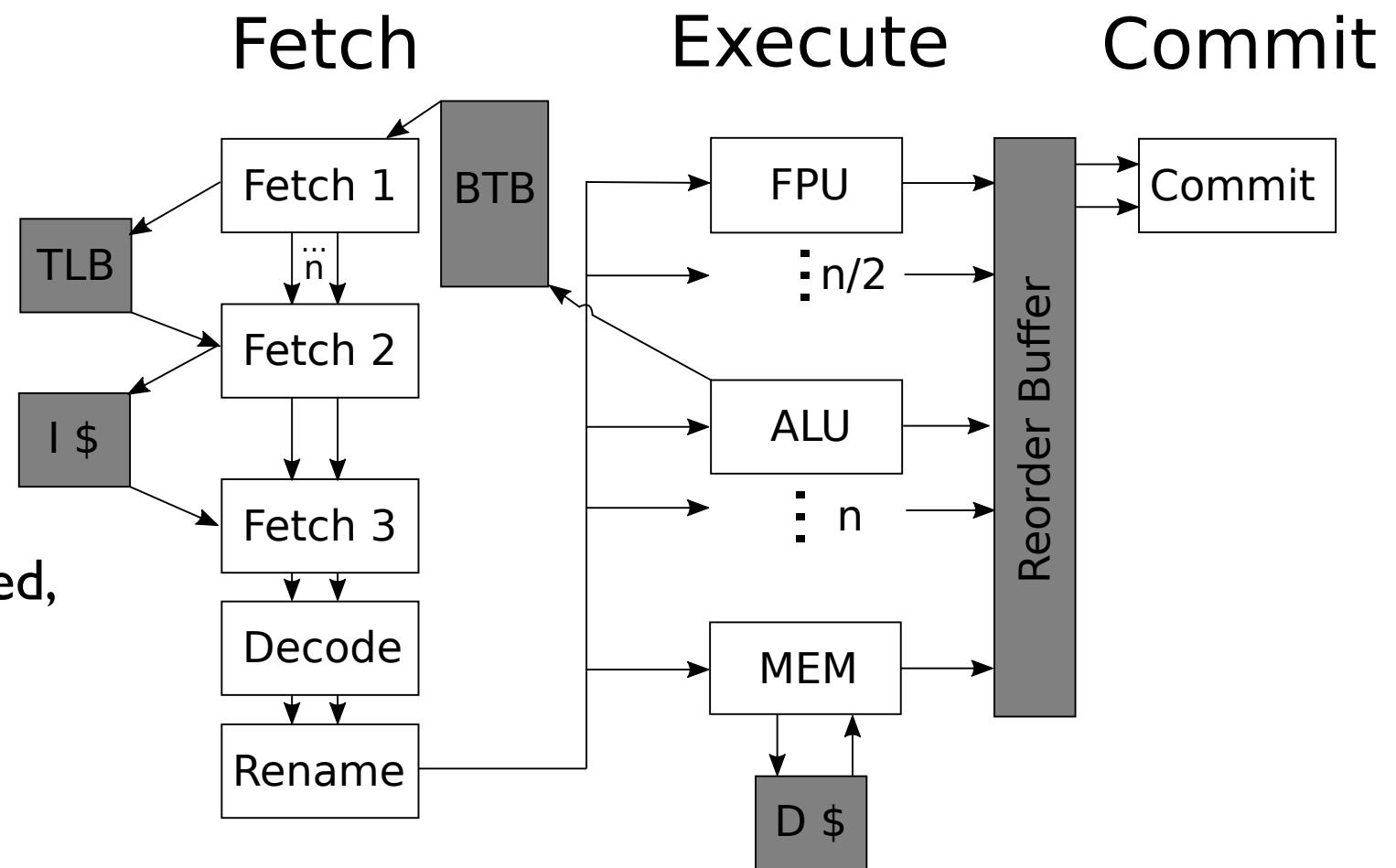
# CHERI RISC-V Implementations

|                     | <b>Piccolo</b>                                                                                                                                                                                               | <b>Flute</b>                                                                                                                             | <b>Tooba</b>                                                                                                                                                                                                                    |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Features            | <ul style="list-style-type: none"><li>• 32-bit</li><li>• 3-stage pipeline</li></ul>                                                                                                                          | <ul style="list-style-type: none"><li>• 64-bit</li><li>• MMU</li><li>• 5-stage pipeline</li></ul>                                        | <ul style="list-style-type: none"><li>• 64-bit</li><li>• Out-of-order, Superscalar</li><li>• L2 cache</li></ul>                                                                                                                 |
| CHERI challenges    | <ul style="list-style-type: none"><li>• Tag-capable SoC</li><li>• Tag controller interaction</li><li>• CHERI-RISCV ISA</li><li>• User-mode ISA verification</li><li>• 3-stage pipeline integration</li></ul> | <ul style="list-style-type: none"><li>• High clock speed</li><li>• Full system ISA for CHERI</li><li>• System ISA verification</li></ul> | <ul style="list-style-type: none"><li>• CHERI integration with vastly more sophisticated pipeline</li></ul>                                                                                                                     |
| New science enabled | <ul style="list-style-type: none"><li>• CHERI embedded OS structure (physical memory)</li><li>• CHERI logic in (short) embedded pipeline</li><li>• Unified register file code performance</li></ul>          | <ul style="list-style-type: none"><li>• Cross-platform CHERI OS structure</li></ul>                                                      | <ul style="list-style-type: none"><li>• CHERI performance overhead with memory reordering</li><li>• CHERI implementation overhead for "more real" core</li><li>• CHERI interaction with speculative execution attacks</li></ul> |

# RISCV-000 Pipeline (very simplified)

- Parameterisable design, with base configuration:

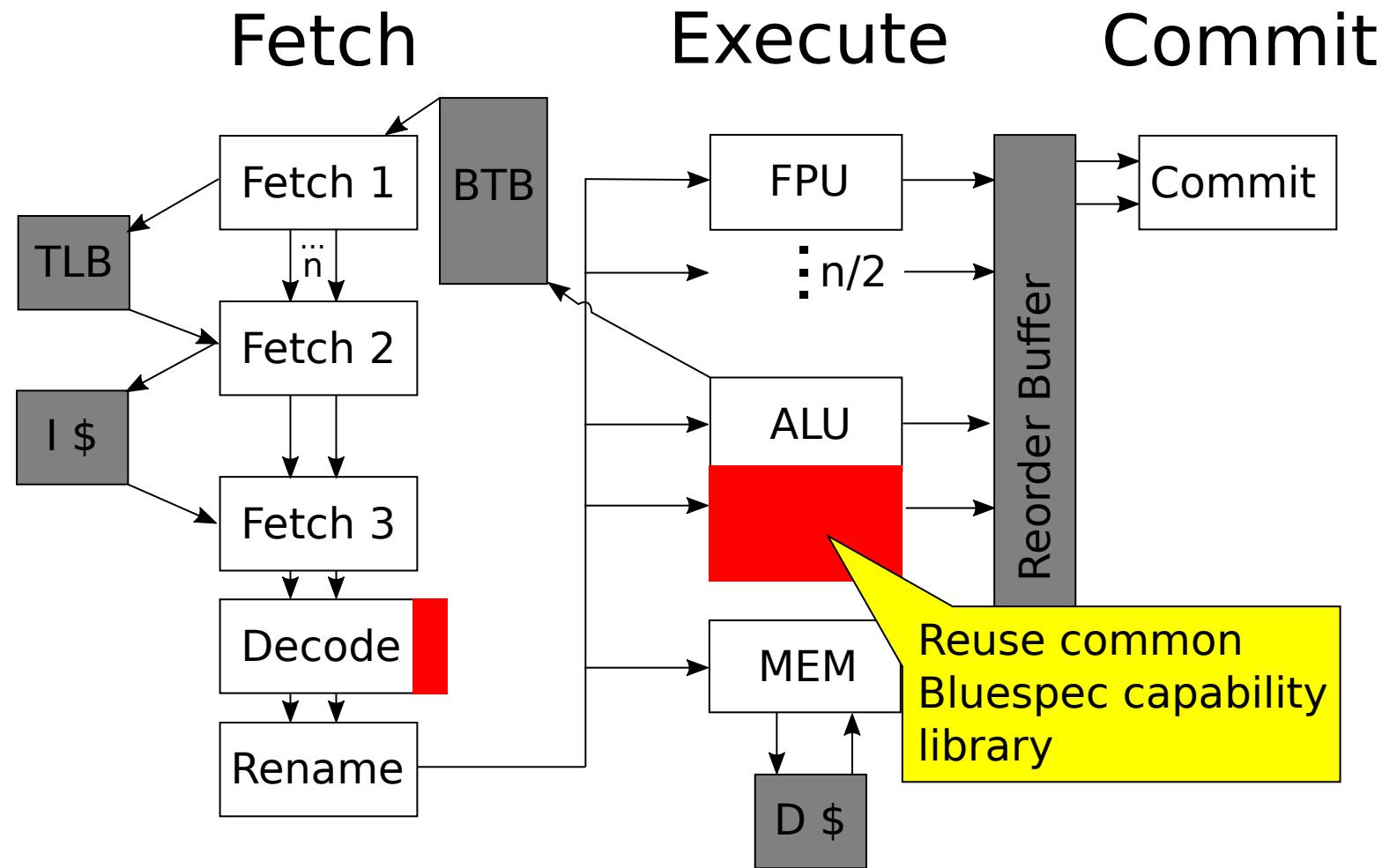
2-way superscalar ( $n=2$ )  
ROB: 64  
Reservation Stations: 16 each  
LD Queue: 24, ST Queue: 12  
Store Buffer: 4



- Pipeline cleanly expressed, easy to work with

# Extending Decode and ALU Pipes for CHERI

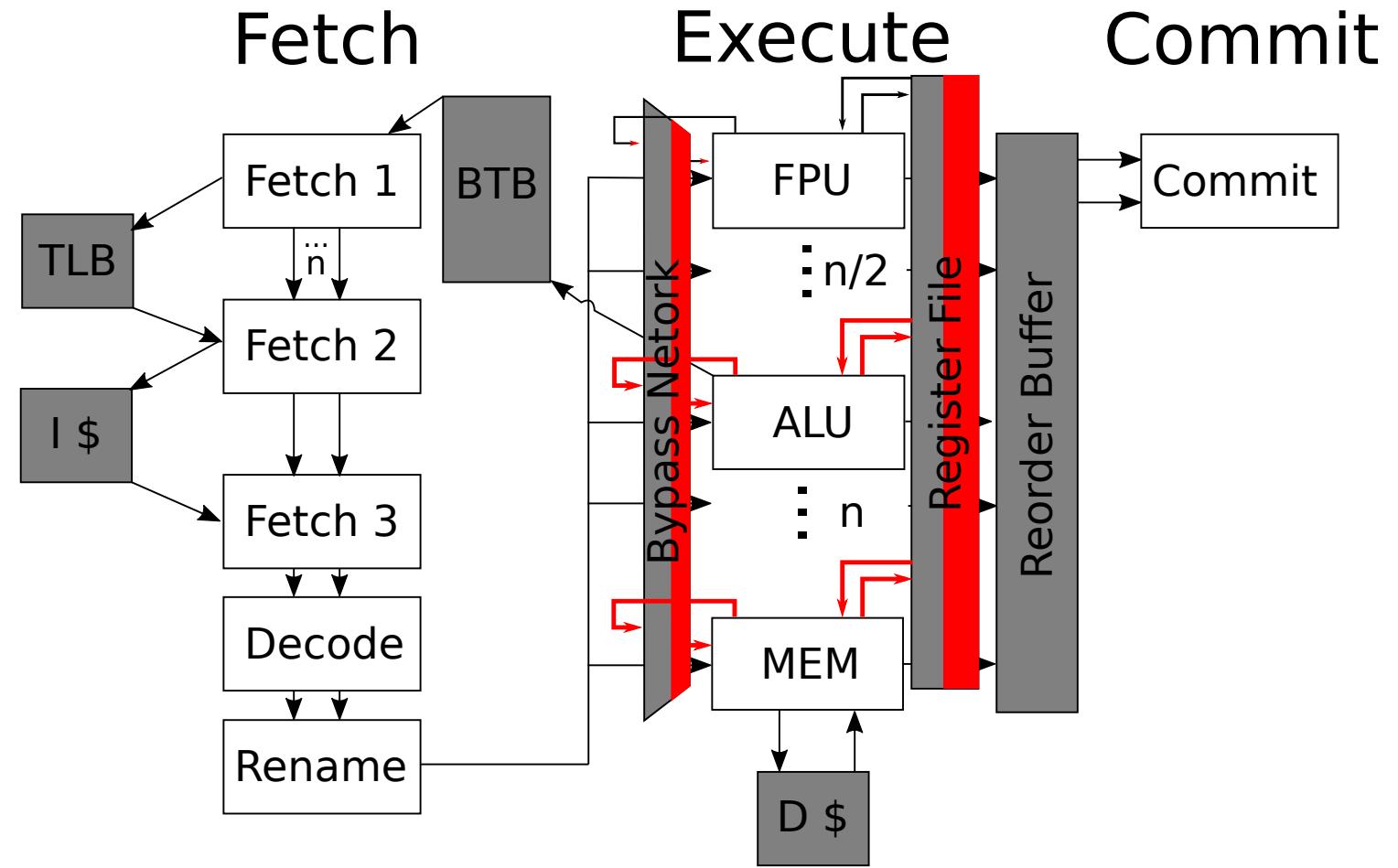
- Straightforward extension of Decode
- Add Bluespec Library functions to generic, replicated ALU pipeline module



# Register File -> CapReg, Bypasses -> CapPipe

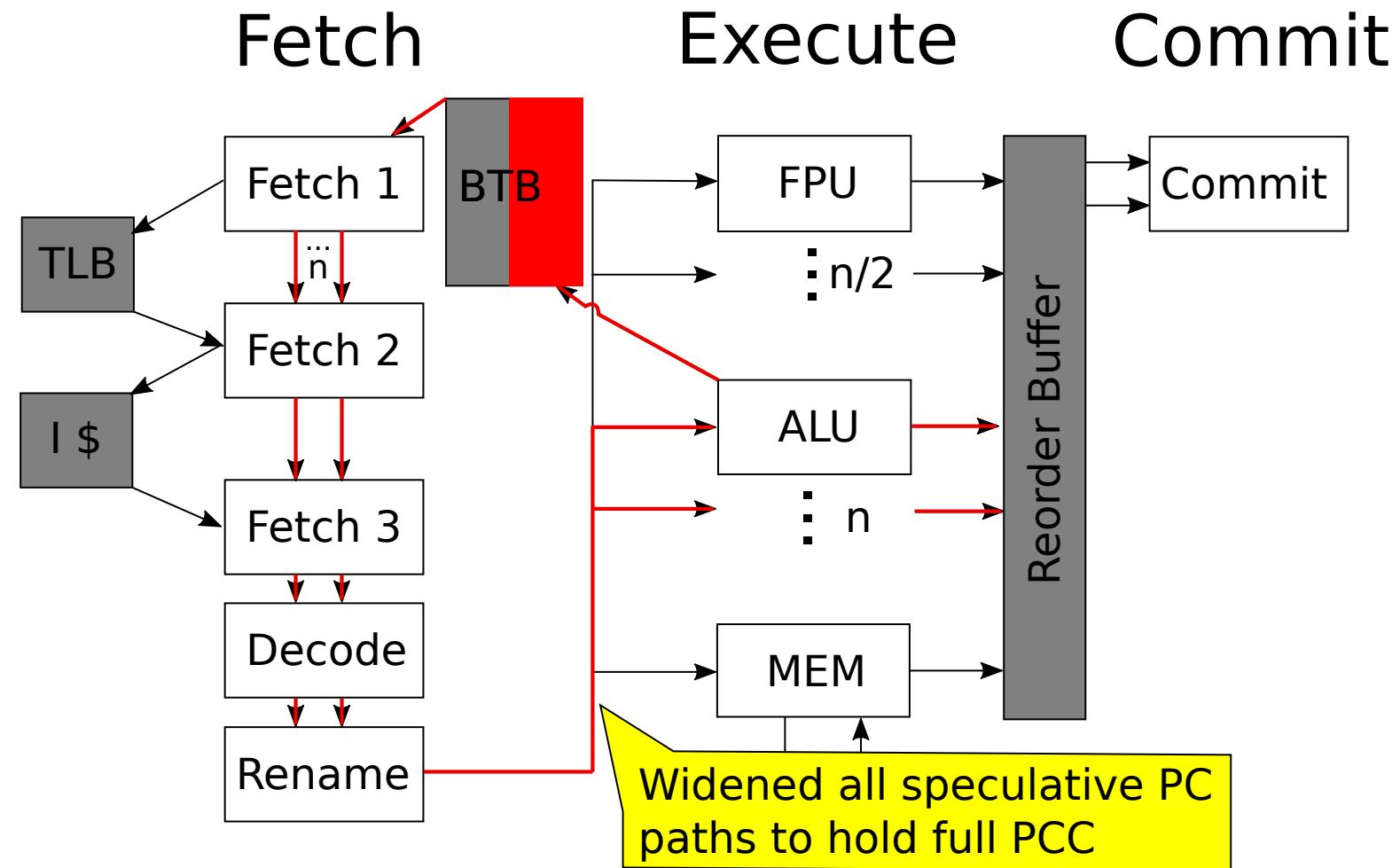
- Changed the data type of the register file, making it parameterizable, and adding a reset value parameter (nullCap)
- Change data type of the bypass network
- In both cases, changing the type somewhere and chasing the type errors...

Strong typing in Bluespec really helps here (very different experience than Verilog)



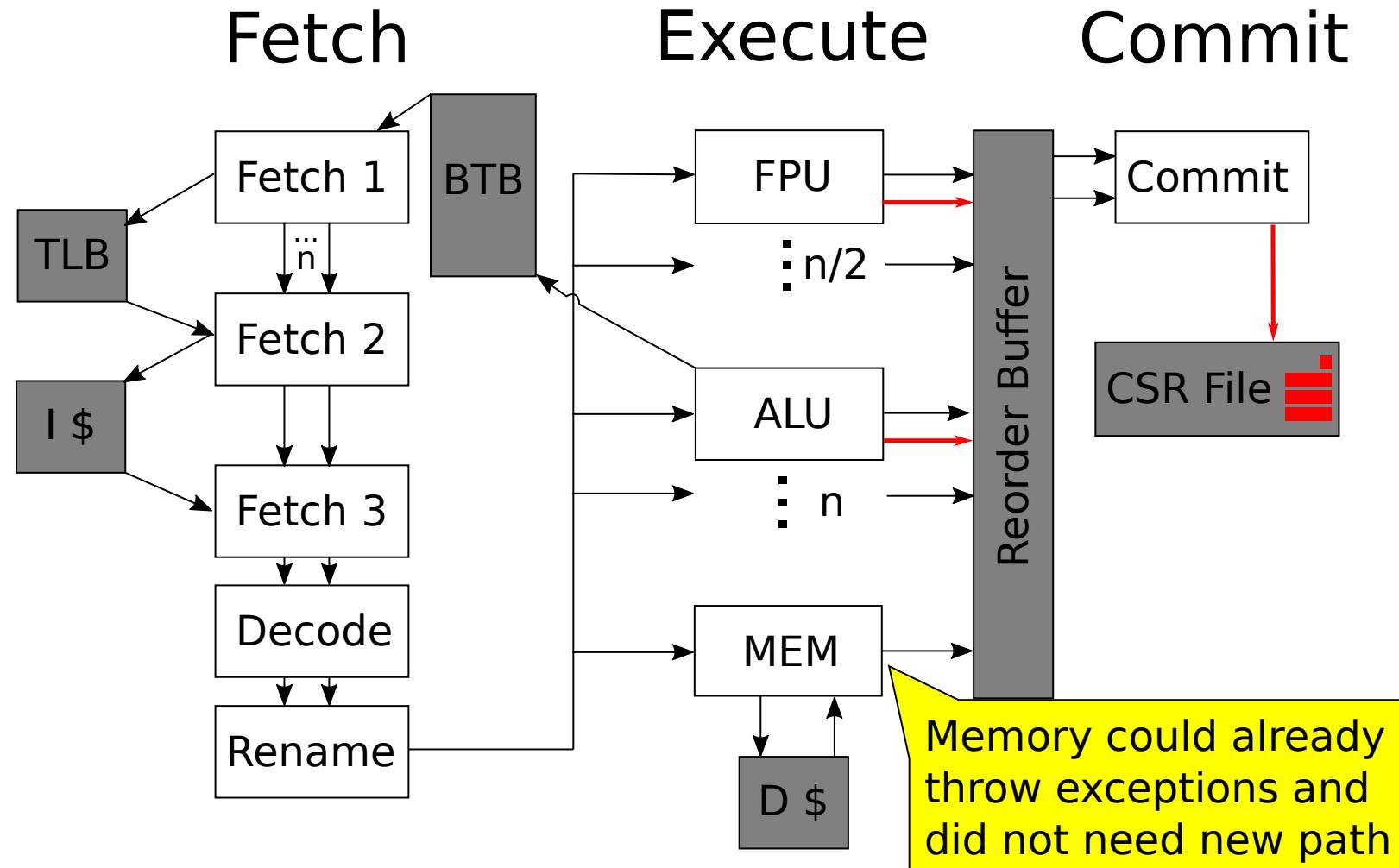
# Program Counter Capability Speculation

- Added PCC to all existing PC paths, which are highly speculated. The Branch Target Buffer now includes bounds, as well as all speculative targets and redirection paths.
- These decisions reduce friction with original design and provide equivalent performance to the baseline.



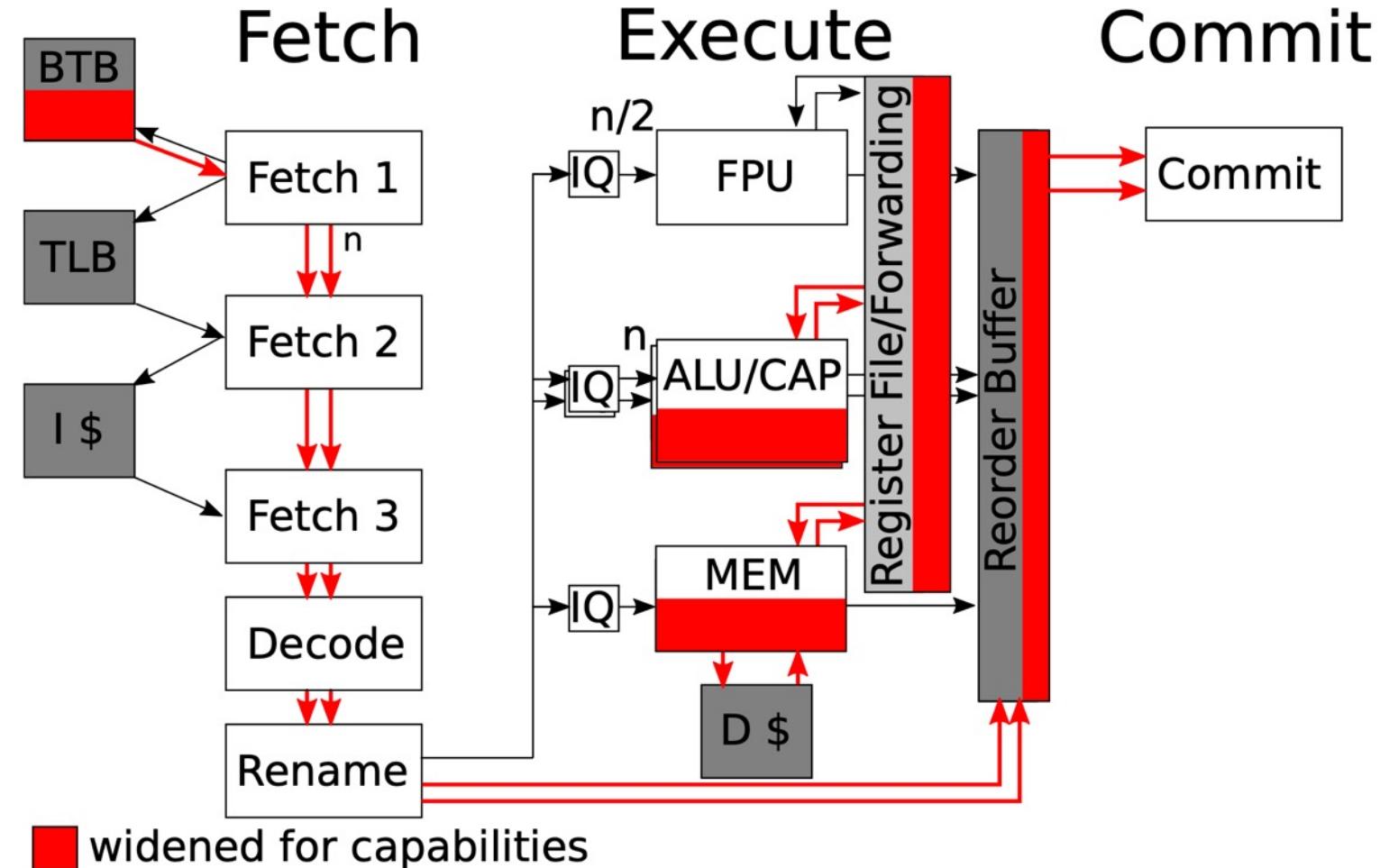
# New Exception Routing

- Added exception delivery to ALU pipes for capability exceptions
- Also added the Capability Exception Code to exception reporting paths everywhere
- Added new special capability registers for changing privilege level on exception and a register for the Capability Exception Code



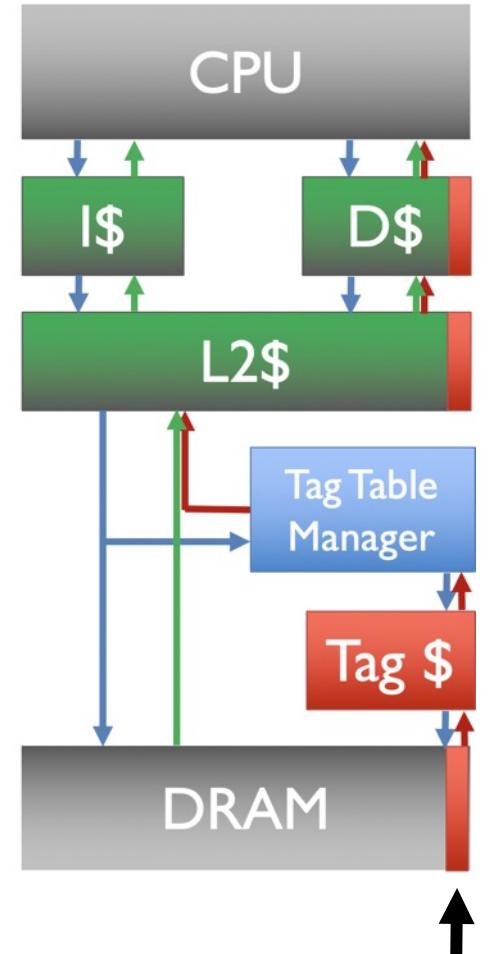
# CHERI-Toooba – Superscalar Out-of-Order 64b CHERI-RISC-V

- Parameterizable Issue/Commit Width
- All PCs extended to full capability: PCC
- All registers extended to 128-bits
- Memory paths (load/store queues) extended to 128-bits
- Some special registers extended to hold capabilities, and new capability registers added
- Tags added to all caches

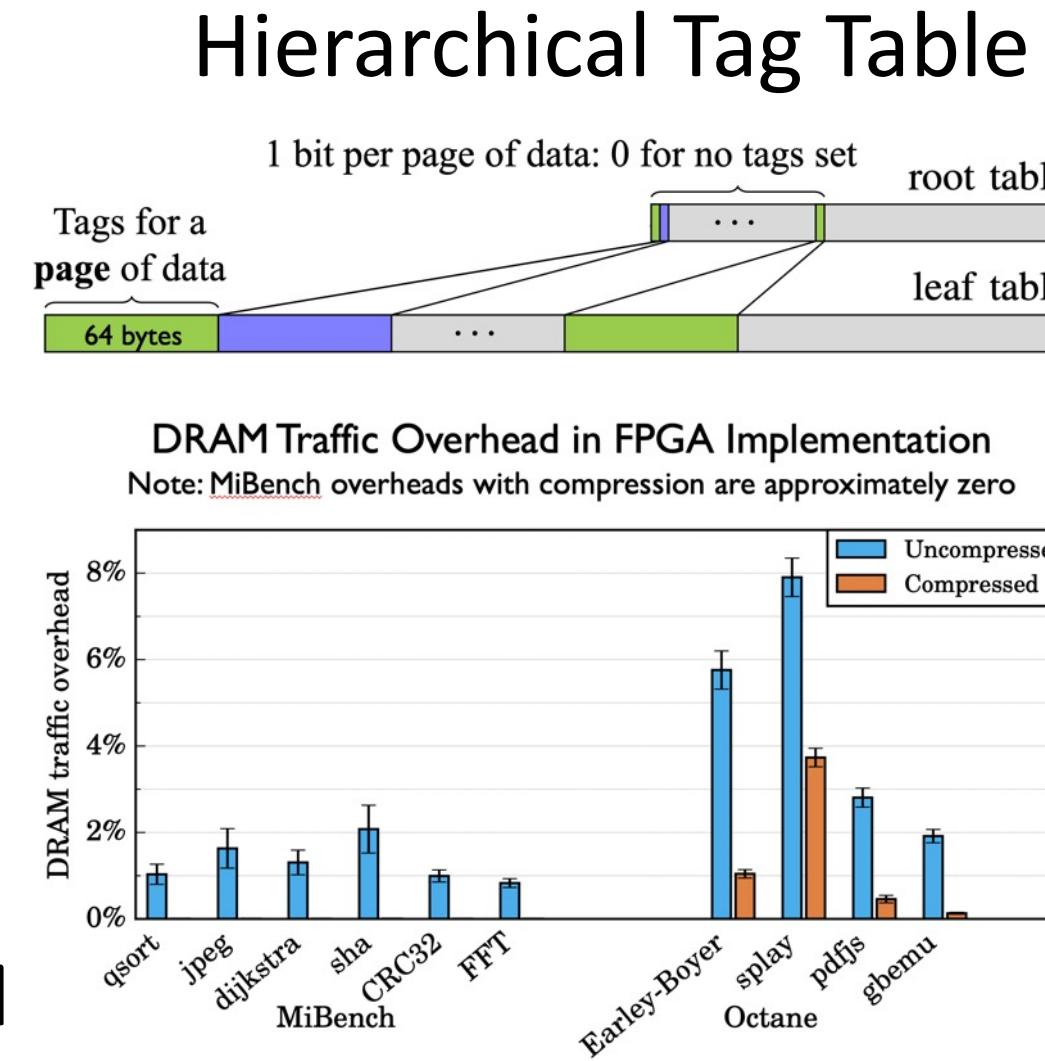


# Sample Optimization: Hierarchical Tag Controller

- Capabilities rendered unforgeable by hardware tags in registers and memory, requiring 129-bit hardware words.
- Where to store these tags in memory?
- Add a tag table in DRAM and emulate a memory of 129-bit words using a hardware tag table manager with a tag cache.



Tag Table in DRAM

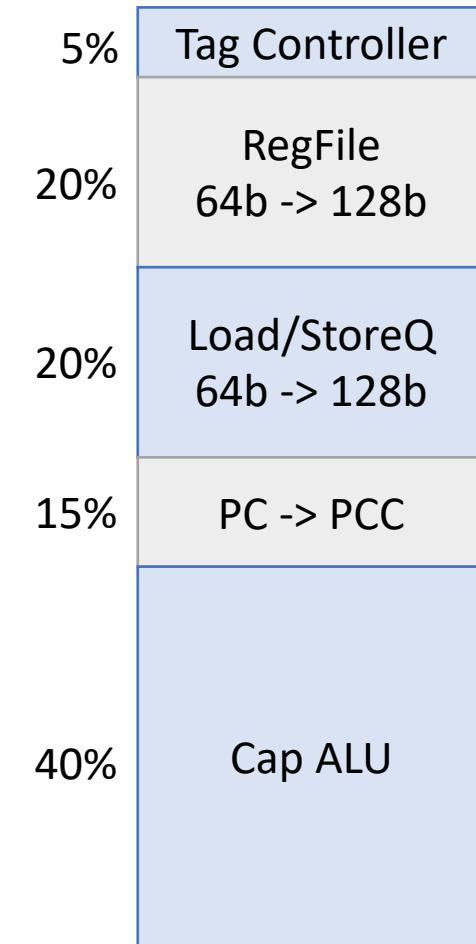


# CHERI Toooba - Area

## FPGA synthesis overview:

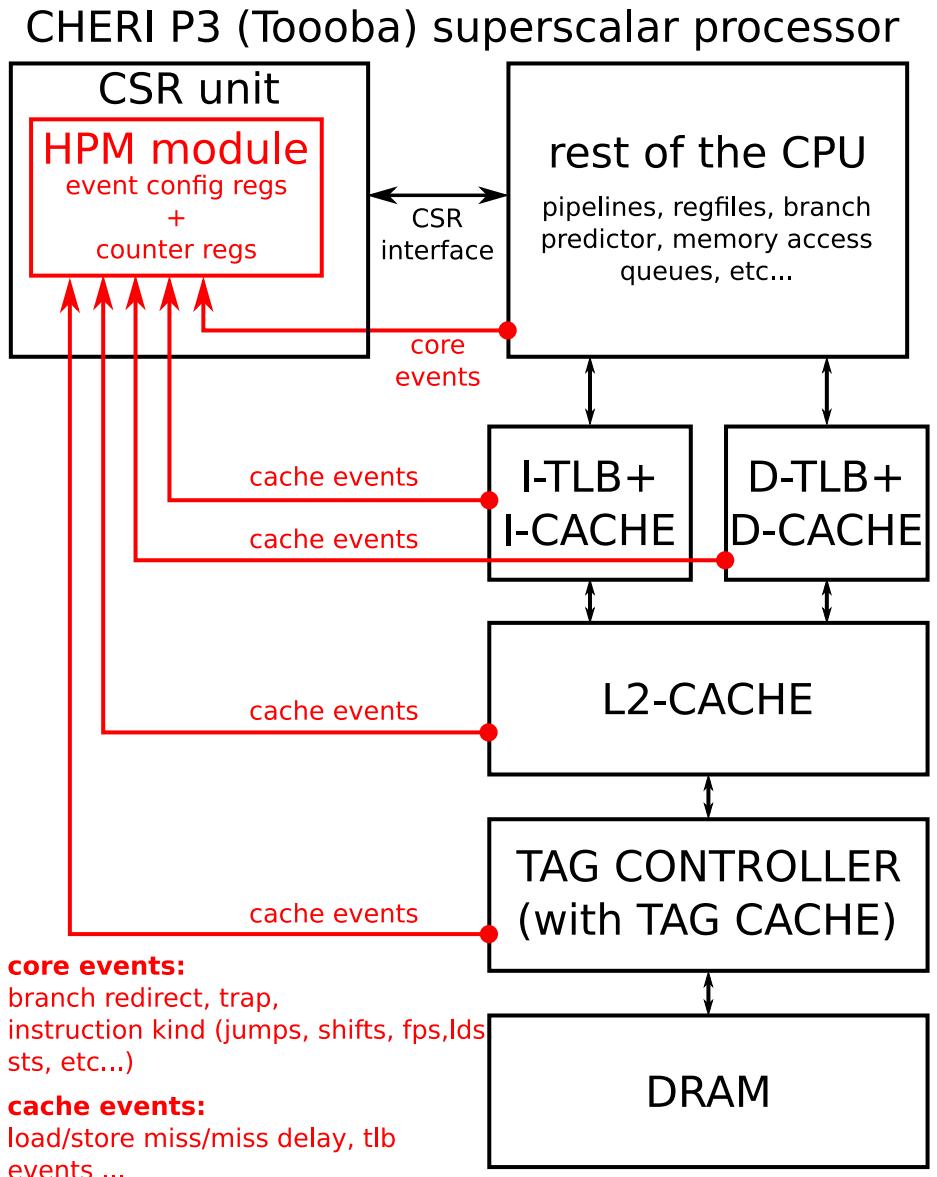
- 26% logic overhead (look up tables – LUTs)
- ~0% memory overhead (BRAM)

LUT overhead breakdown



# Hardware Performance Monitoring Framework to Measure and Understand Performance

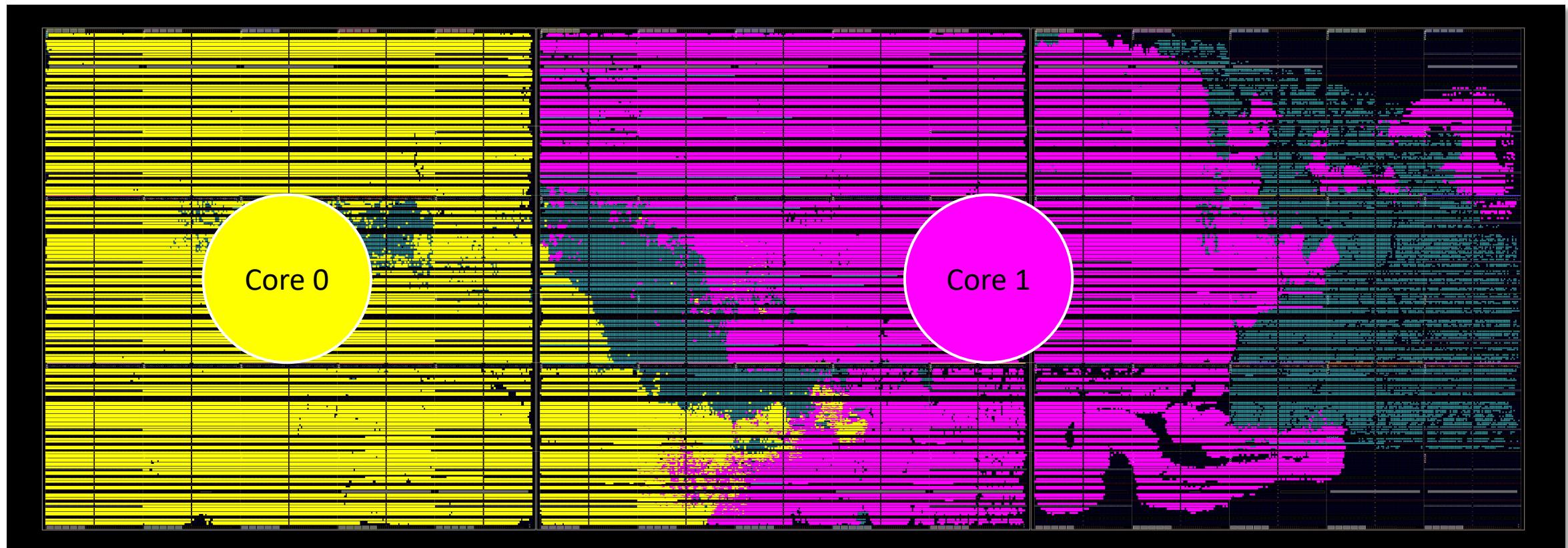
- Report a variety of architectural/microarchitectural events
  - Expose many existing RiscyOO events not previously exposed in the SSITH GFE P3
  - Support for > 1 increments within a cycle, useful for example when reporting retired instructions in a superscalar processor, or bulk reporting cycle latencies for memory accesses...
- Events exposed through the RISC-V specified Hardware Performance Monitoring (HPM) mechanism
  - Privileged mode counter configuration (currently, Berkley Bootloader/RISC-V Proxy Kernel support for a statically curated set of events)
  - User mode counter read (currently, CheriBSD libstatcounters support to use with benchmarks)
- HPM approach believed generalized enough to be suitable for upstream implementation



# CHERI-Toooba dual-core

- CHERI-Toooba dual-core successfully running FreeBSD and CheriBSD
- Needed for our work on temporal memory safety
- Allows revocation to proceed concurrently

```
# top -P
last pid: 49; load averages: 0.21, 0.22, 0.10 up 0+00:04:18 16:16:39
3 processes: 1 running, 2 sleeping
CPU 0: 5.1% user, 0.0% nice, 1.9% system, 15.3% interrupt, 77.7% idle
CPU 1: 7.9% user, 0.0% nice, 2.1% system, 0.9% interrupt, 89.2% idle
Mem: 3404K Active, 164K Inact, 43M Wired, 7564K Buf, 2885M Free
```



# CheriABI: Spatially Safe UNIX Processes

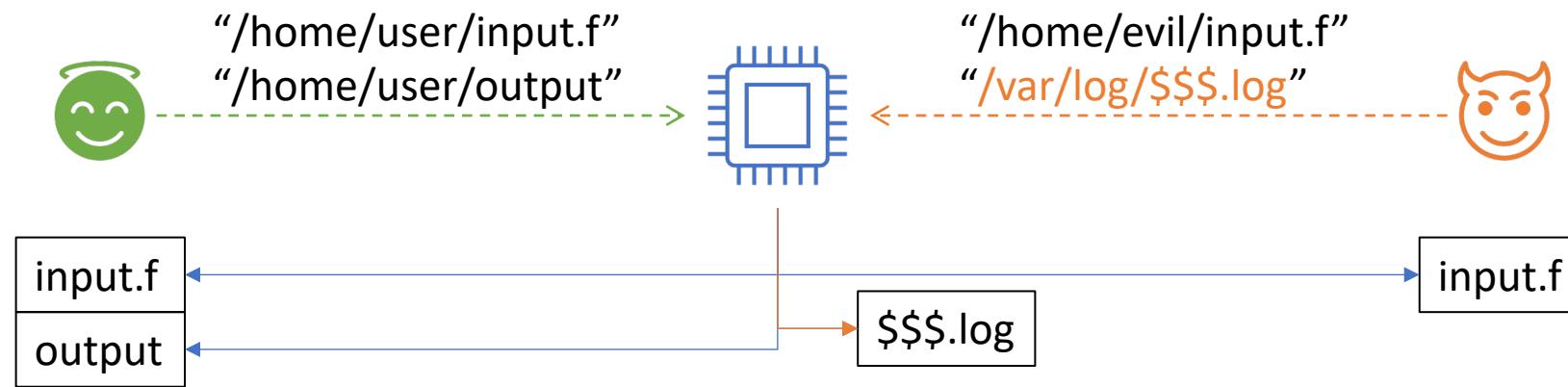
## Introduction

- Have pointedly been ignoring the *kernel*
  - By design, has access to whole program! Breaks spatial safety??
-  [§2.7 CheriABI Showcase](#)
  - Exercise has two short parts and a longer “extra credit” for the enthusiastic

# CheriABI: Spatially Safe UNIX Processes

## Introduction: Confused Deputies

- A confused deputy mistakenly uses *its own authority* when acting
- Hardy's example:



- Explore two examples here, with *the kernel* being the deputy

# CheriABI: Spatially Safe UNIX Processes

Introduction: Deputy 1: `read()` and capability bounds

- Consider “`read(fd, buf, len)`”
- If `len` is larger than `buf` target, this could overflow!
  - Kernel has access to all of process memory, has no reason to stop writing.

# CheriABI: Spatially Safe UNIX Processes

## Introduction: Deputy 2: `mmap()` and friends

- Processes want to add and remove pages from their address space
  - Kernel exposes system calls `mmap()`, `munmap()`, &c.
  - Baseline architectures: take *integers* to identify pages on which to act
- Risk: integers can be forged or corrupted
  - Capabilities carry virtual addresses; could completely change their meaning!



§2.7



!

# CheriABI: Spatially Safe UNIX Processes

Discussion: `read()` and capability bounds

`read(fd, lower, sizeof(lower) + sizeof(upper))`

## RISC-V Baseline

Write OK

lower=0x80922400 upper=0x80922410

Read 0x20 OK; lower[0]=0x10 upper[0]=0x20

Kernel overwrite!

## CHERI-RISC-V

Write OK

lower=0x3ffffdff28 upper=0x3ffffdff38

Bad read (Bad address); lower[0]=0x10 upper[0]=0x0

Kernel return -EFAULT;  
Does not write OOB

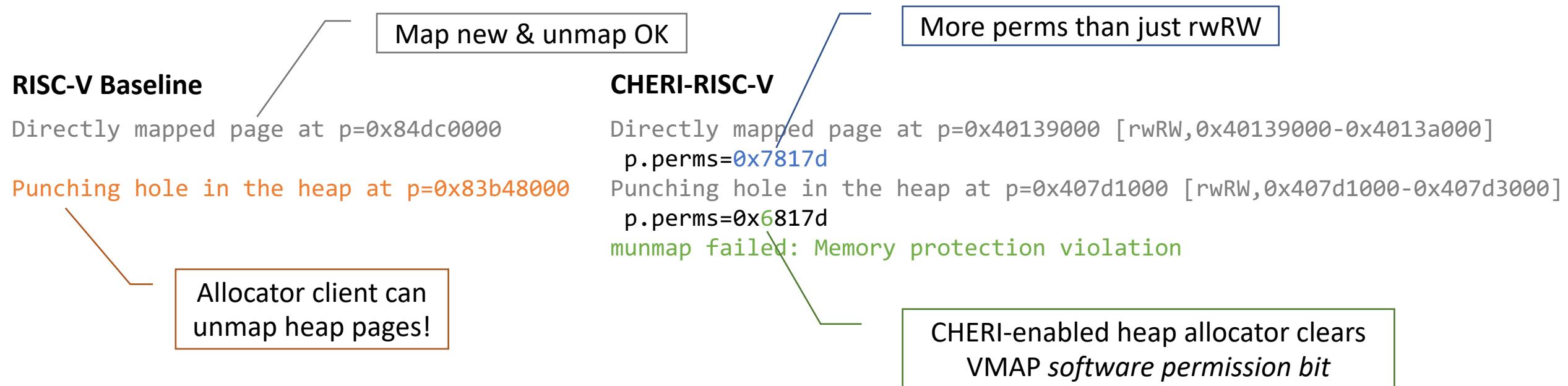
Fault detected during copy-out

CheriABI system calls take capabilities, and

*voluntarily act with implied restricted authority!*

# CheriABI: Spatially Safe UNIX Processes

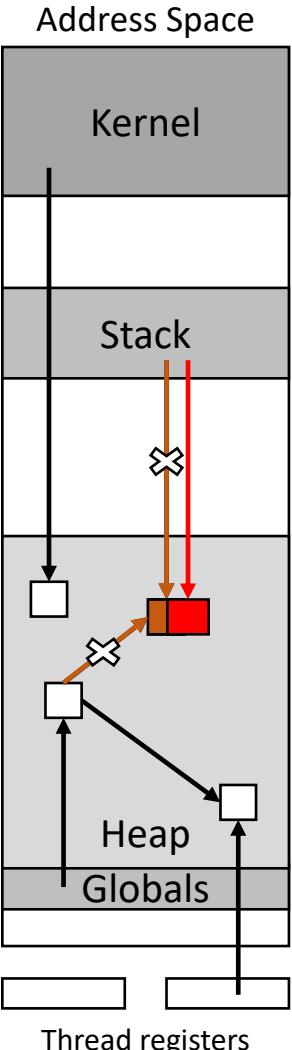
## Discussion: mmap() and friends



- CHERI exposes *software permission bits* uninterpreted by architecture
  - CheriBSD uses one of these to indicate *ownership* of address space, and `malloc()` clears this bit on returns

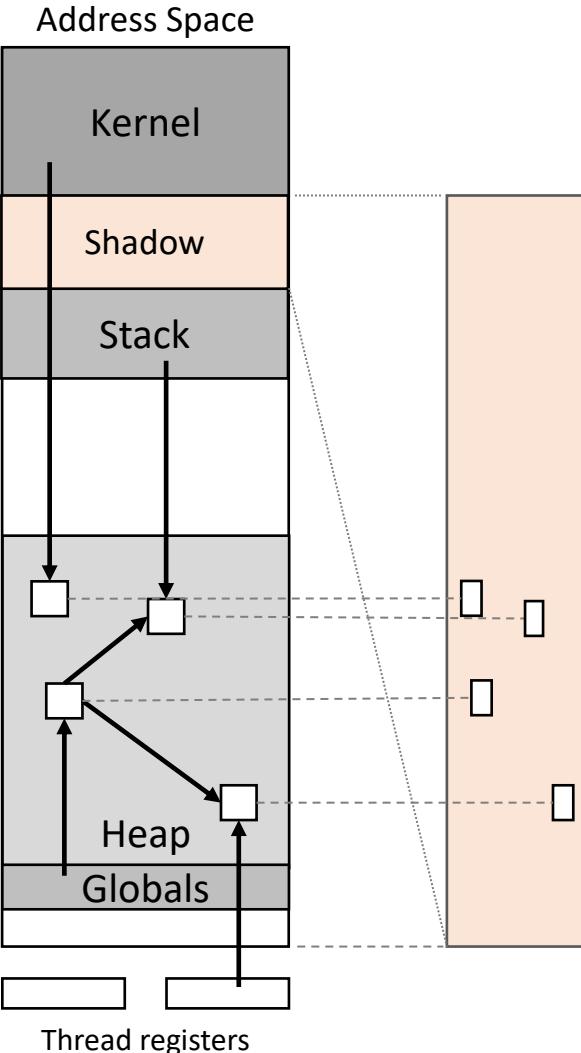
# Cornucopia: CHERI Heap Temporal Safety

## Address Space Quarantine, Revocation



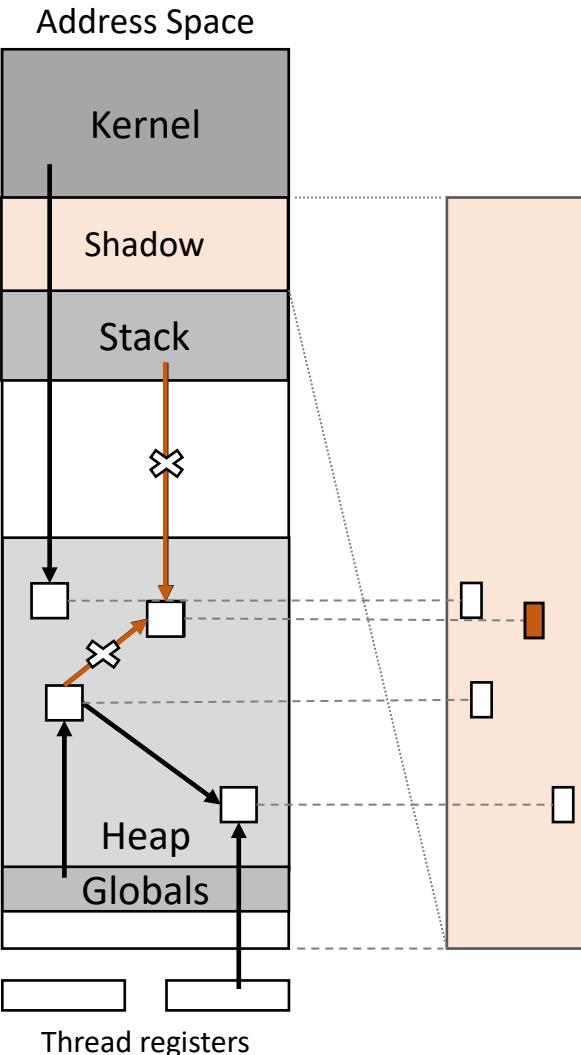
- Focused on *heap* temporal safety
  - More complex lifetimes than stack objects, resists static approaches
- Heap pointers end up in globals, stacks, registers, kernel heap, ...
- Risk: **retain references to free( ) object, overlap new allocation**
  - **Use After Reallocation**: use old reference to access new allocation
  - UAF-but-not-UAR less of a concern
- Eliminate UAR by *revoking* dead references
  - UAF left possible, but guaranteed to access old object
- “Dual” of garbage collection: (lazily) enforce free()

# Cornucopia: CHERI Heap Temporal Safety 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 user program
- Thread-safe & mostly concurrent implementation

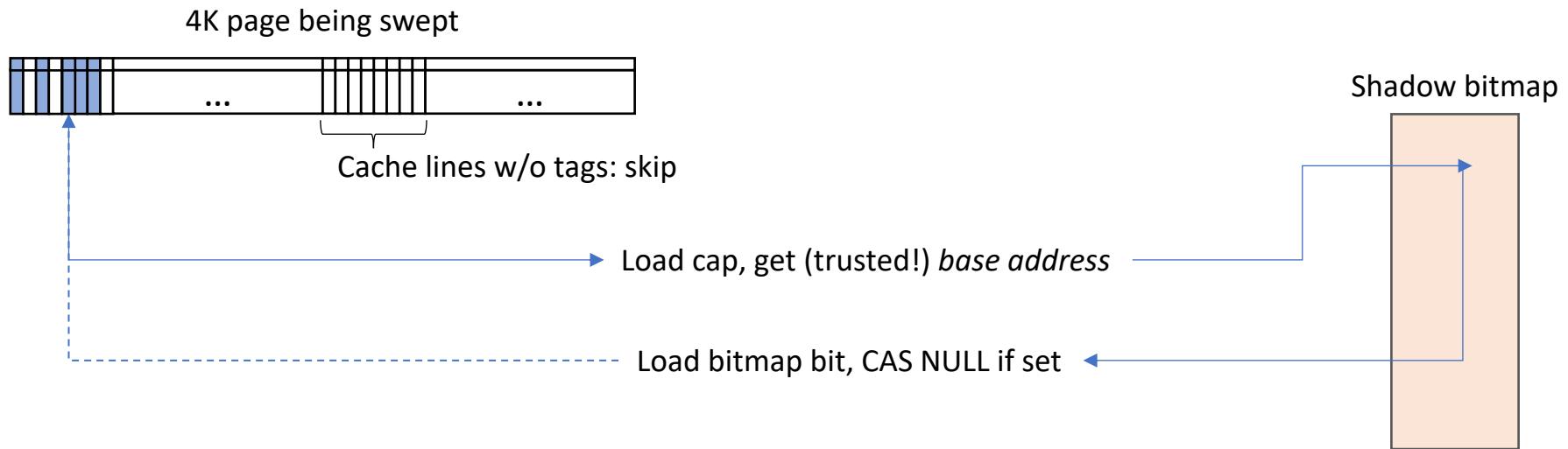
# Cornucopia: CHERI Heap Temporal Safety Quarantine & Batched Revocation



- On free, allocator...
  - marks shadow of object
  - holds address space in *quarantine*
- When quarantine fills, allocator invokes revoker service
  - Deletes all capabilities whose targets have marked shadows
- After revocation, safe to reuse address space
  - Allocator *clears* shadow, enqueues address space to free lists

# Cornucopia: CHERI Heap Temporal Safety

## Per-Page Sweep in More Detail



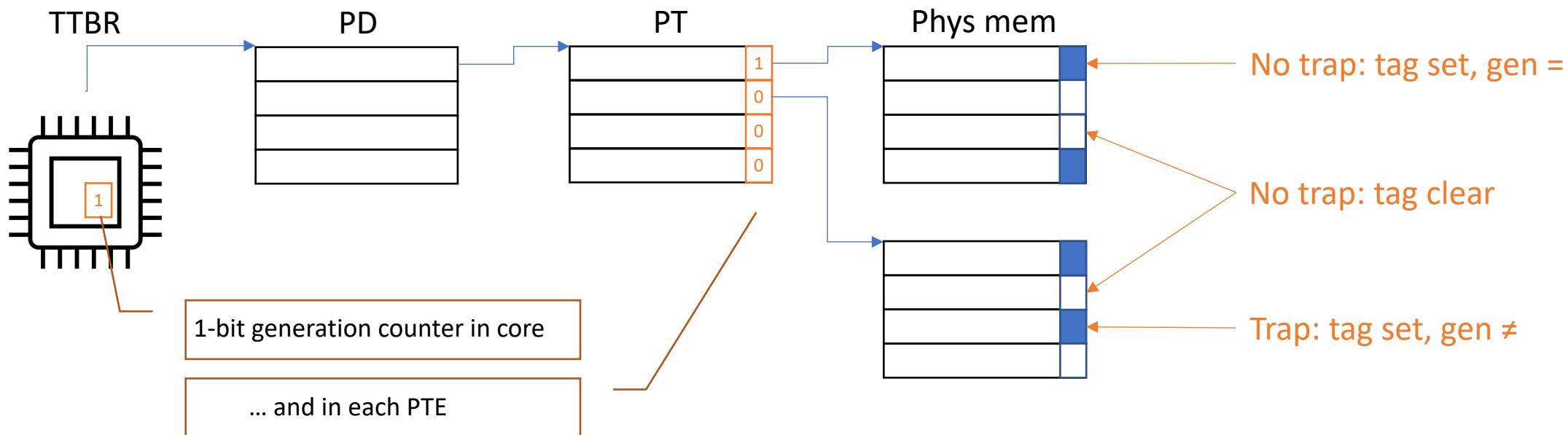
# Cornucopia Architecture

## Per-Page “Capability-Dirty” Tracking



# Cornucopia Architecture

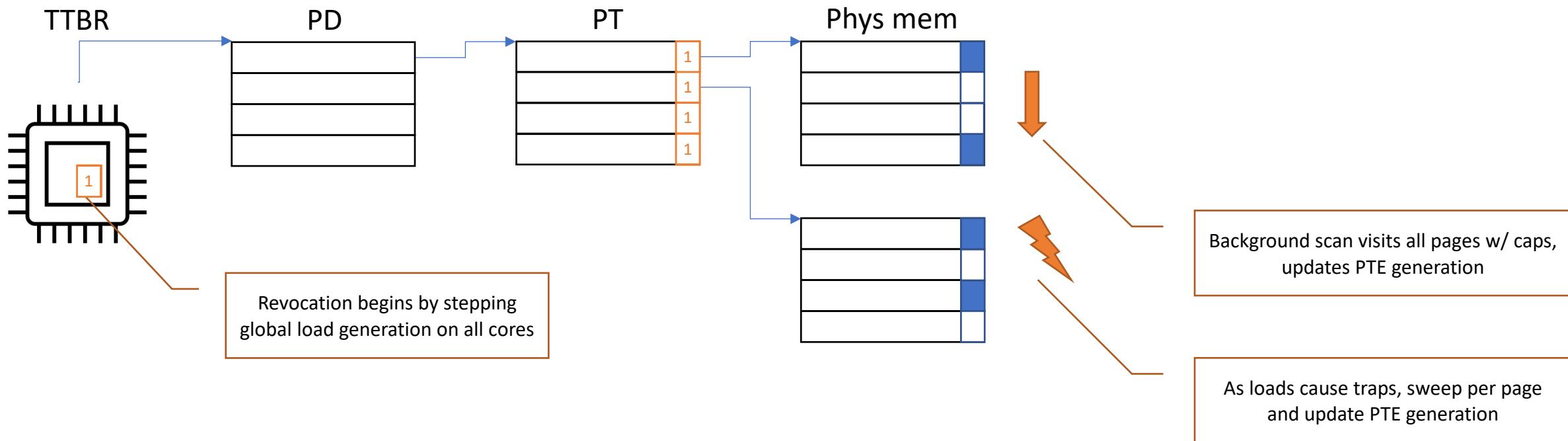
## Per-Page Capability Load Generations



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

# Cornucopia Architecture

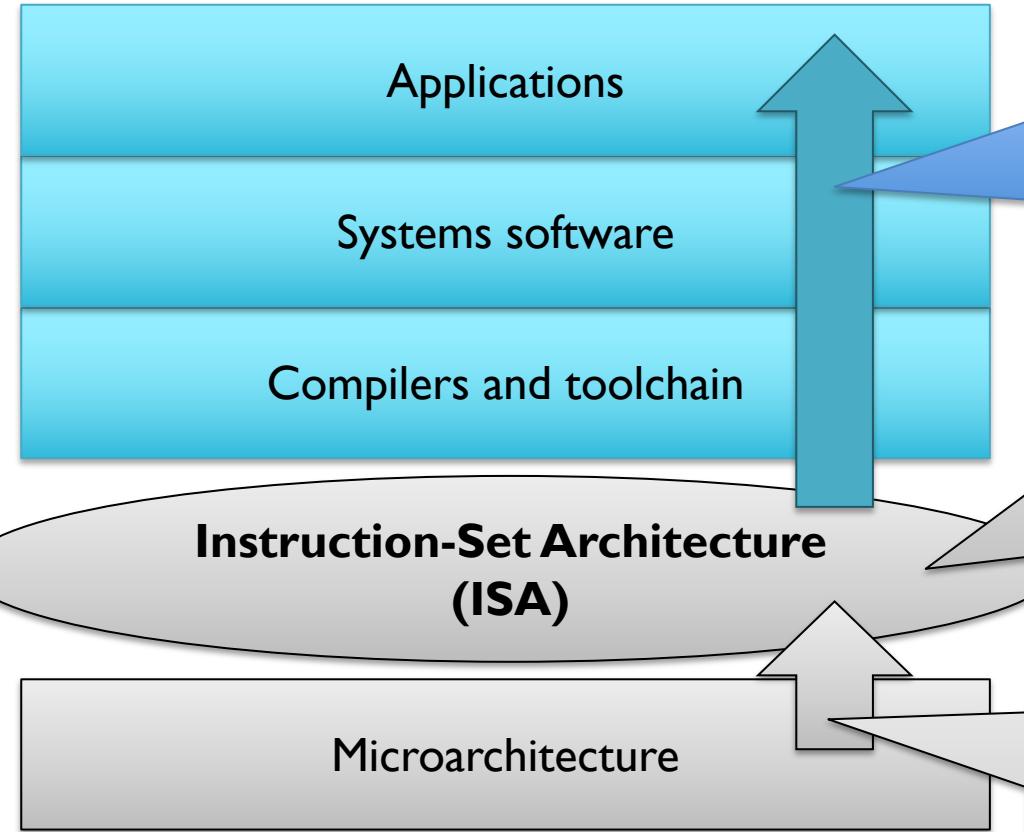
## Revoking With Capability Load Generations



Afternoon Tea (16h00 – 16h30)

# **COMPARTMENTALIZATION**

# Architectural primitives for software security



Software configures and uses capabilities to continuously enforce safety properties such as **referential, spatial, and temporal memory safety**, as well as higher-level security constructs such as **compartment isolation**

**CHERI capabilities** are an **architectural primitive** that compilers, systems software, and applications use to constrain their own future execution

The microarchitecture implements the **capability data type** and **tagged memory**, enforcing invariants on their manipulation and use such as **capability bounds, monotonicity, and provenance validity**

# Two key applications of the CHERI primitives

## I. Efficient, fine-grained memory protection for C/C++

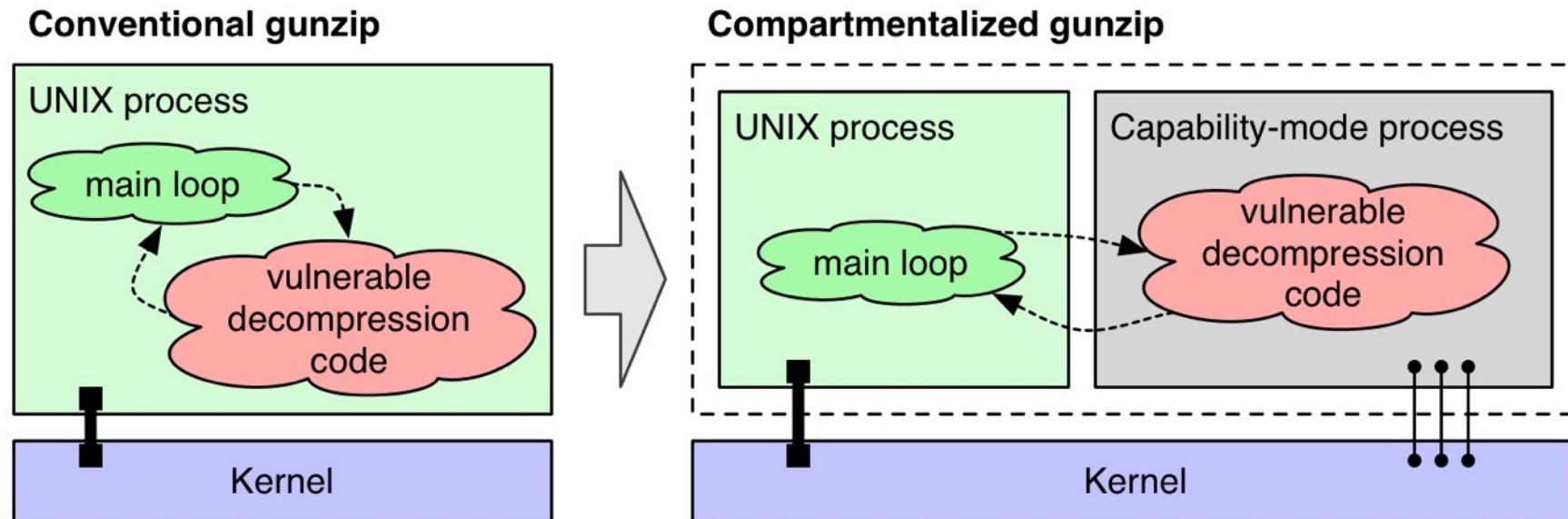
- Strong source-level compatibility, but requires recompilation
- Deterministic and secret-free referential, spatial, and temporal memory safety
- Retrospective studies estimate  $\frac{2}{3}$  of memory-safety vulnerabilities mitigated
- Generally modest overhead (0%-5%, some pointer-dense workloads higher)

## 2. Scalable software compartmentalization

- Multiple software operational models from objects to processes
- Increases exploit chain length: Attackers must find and exploit more vulnerabilities
- Orders-of-magnitude performance improvement over MMU-based techniques (<90% reduction in IPC overhead in early FPGA-based benchmarks)

# Application-level least privilege

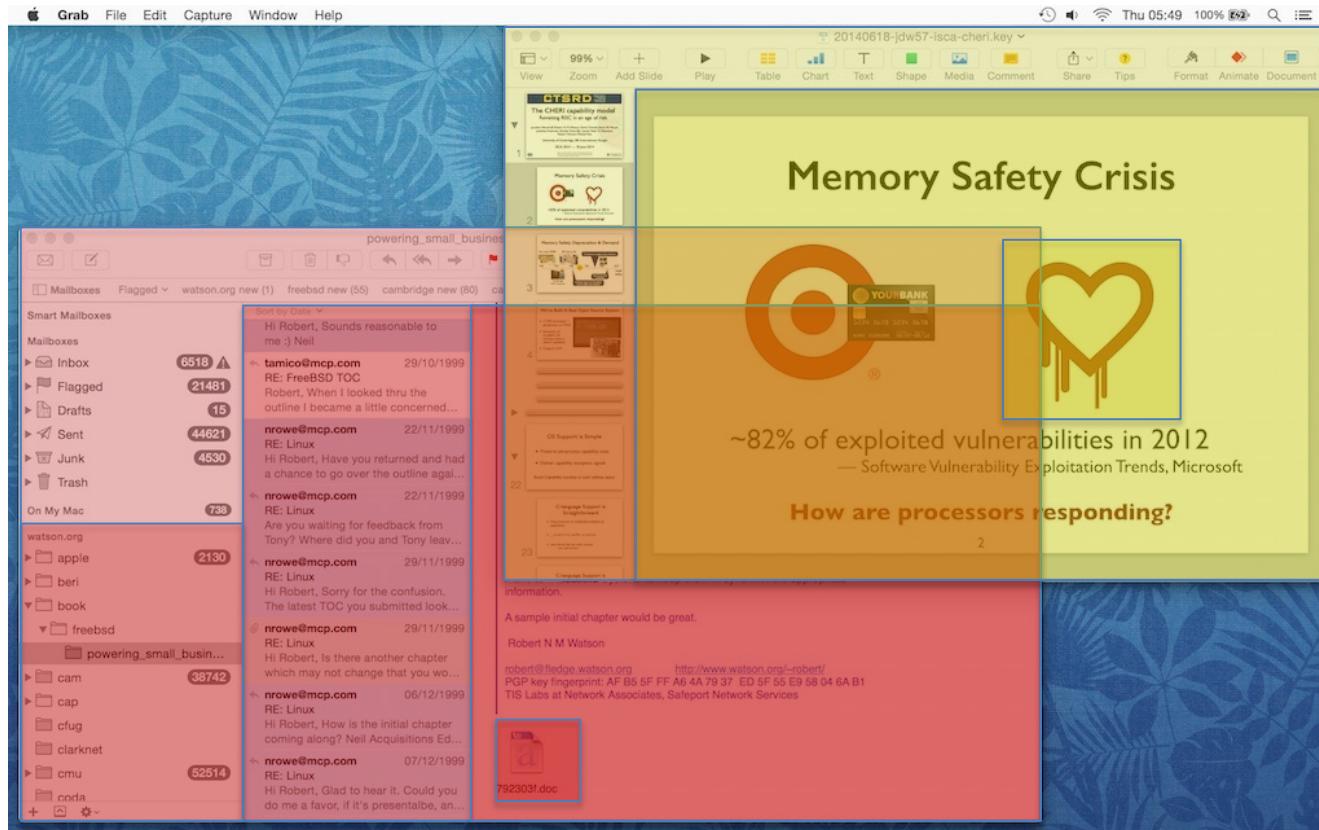
**Software compartmentalization** decomposes software into **isolated compartments** that are delegated **limited rights**



Able to mitigate not only unknown vulnerabilities, but also **as-yet undiscovered classes of vulnerabilities and exploits**

# Application-level least privilege

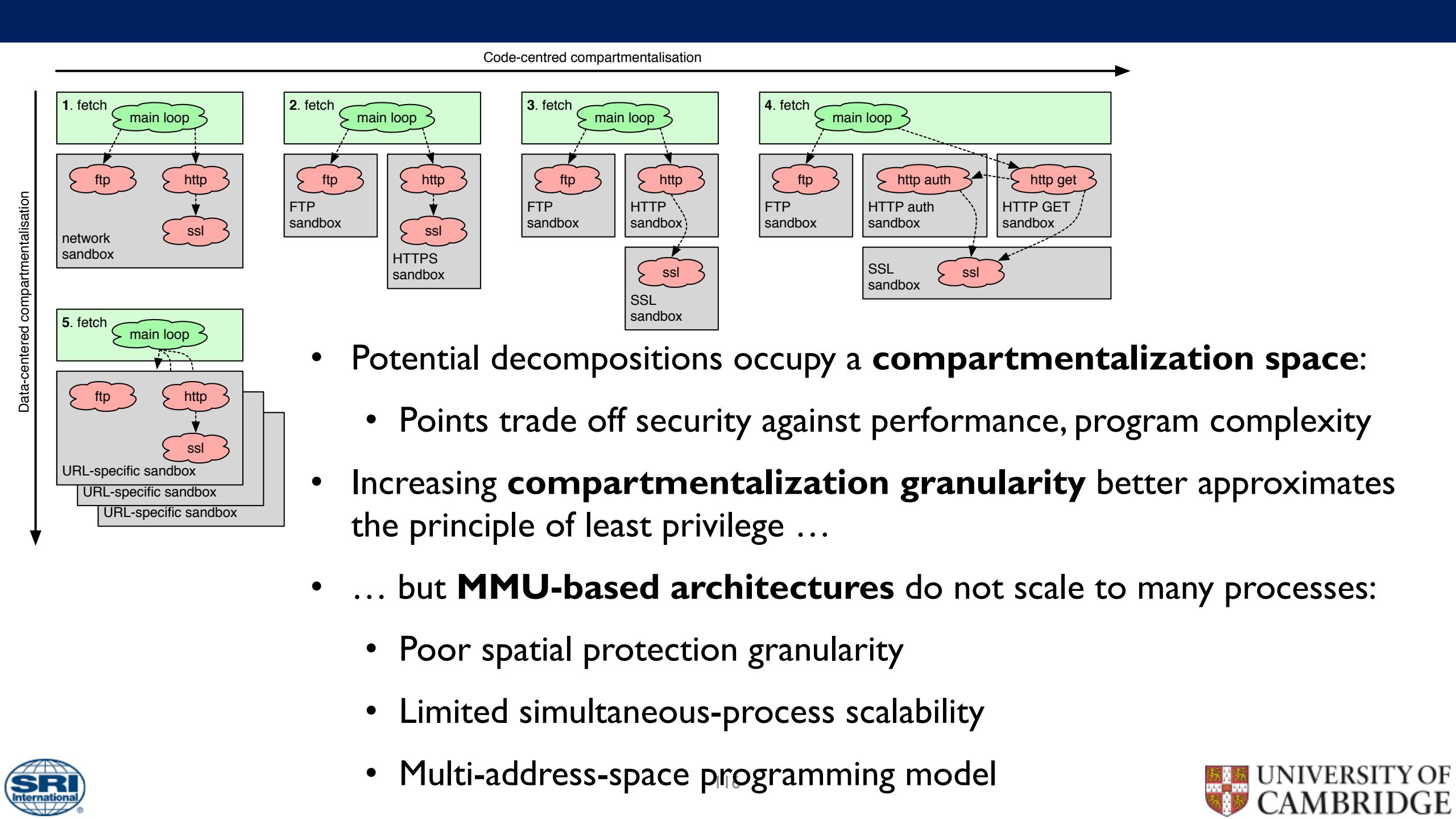
**Software compartmentalization** decomposes software into **isolated compartments** that are delegated **limited rights**



**Potential compartmentalization boundaries** matching reasonable user expectations for **least privilege** can be found in many user-facing apps.

E.g., a malicious email attachment should not be able to gain access to other attachments, messages, folders, accounts, or the system as a whole.

Able to mitigate not only **unknown vulnerabilities**, but also **as-yet undiscovered classes of vulnerabilities and exploits**

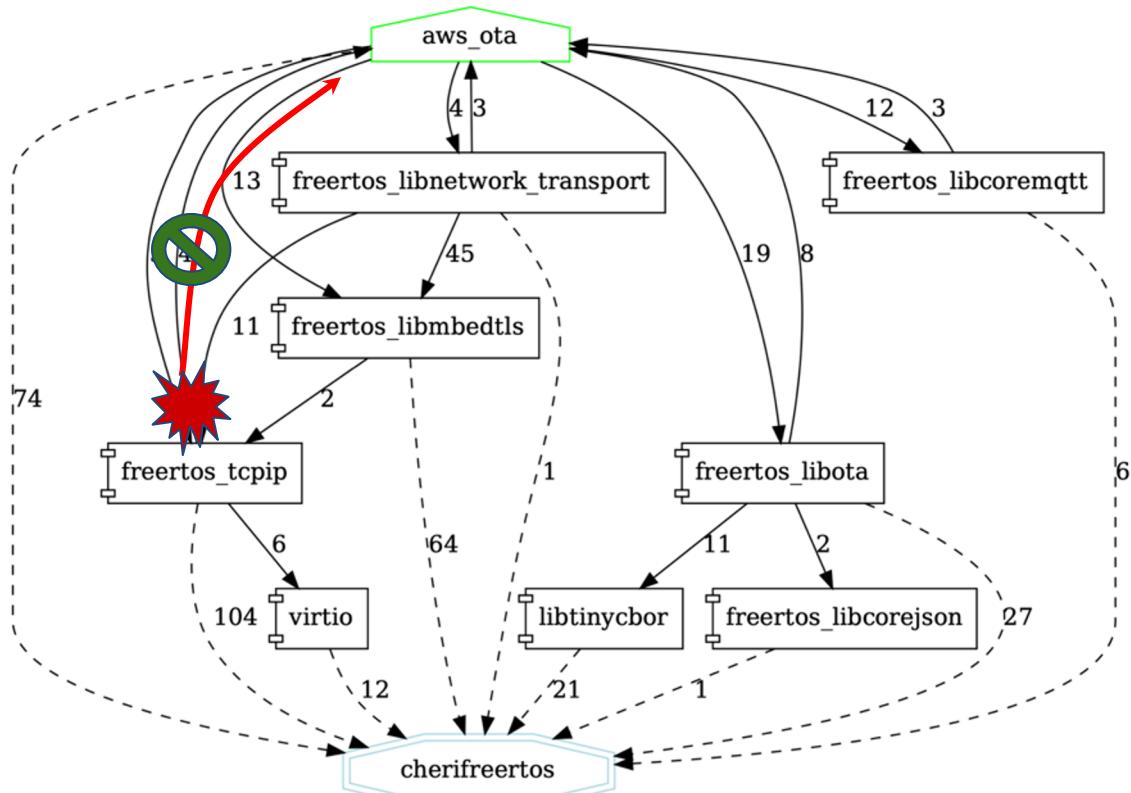


# Virtual memory and capabilities

|                                | Virtual Memory                                        | Capabilities                                       |
|--------------------------------|-------------------------------------------------------|----------------------------------------------------|
| <b>Protects</b>                | Virtual addresses and pages                           | References (pointers) to C code, data structures   |
| <b>Hardware</b>                | MMU, TLB, page-table walker                           | Capability registers, tagged memory                |
| <b>Costs</b>                   | TLB, page tables, page-table lookups, shoot-down IPIs | Per-pointer overhead, context switching            |
| <b>Compartment scalability</b> | Tens to hundreds                                      | Thousands or more                                  |
| <b>Domain crossing</b>         | IPC                                                   | In-address-space function calls or message passing |
| <b>Optimization goals</b>      | Isolation, full virtualization                        | Memory sharing, frequent domain transitions        |

CHERI hybridizes the two models: use the best combination for any given problem

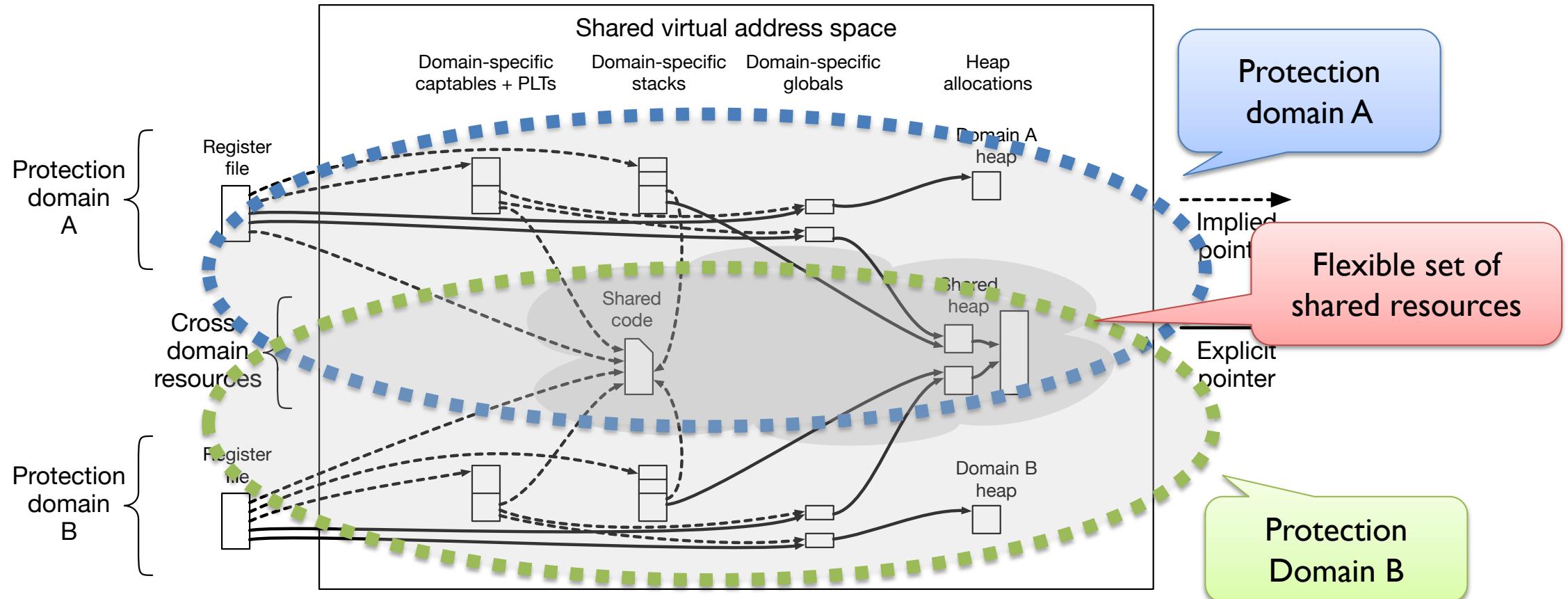
# What is software compartmentalization?



CheriFreeRTOS components and the application execute in compartments. CHERI contains an attack within TCP/IP compartment, which access neither flash nor the internals of the software update (OTA) compartment.

- Fine-grained decomposition of a larger software system into **isolated modules** to constrain the impact of faults or attacks
- Goals is to **minimize privileges yielded by a successful attack, and to limit further attack surfaces**
- Usefully thought about as a **graph of interconnected components**, where the attacker's goal is to compromise nodes of the graph providing a route from a point of entry to a specific target

# CHERI-based compartmentalization



- Isolated compartments can be created using closed graphs of capabilities, combined with a constrained non-monotonic domain-transition mechanism

# Compartmentalization scalability

- CHERI dramatically improves **compartmentalization scalability**
    - More compartments
    - More frequent and faster domain transitions
    - Faster shared memory between compartments
  - Many potential use cases – e.g., sandbox processing of each image in a web browser, processing each message in a mail application
  - Unlike memory protection, software compartmentalization requires **careful software refactoring** to support strong encapsulation, and affects the software operational model
- } Early benchmarks show a 1-to-2 order of magnitude performance inter-compartment communication improvement compared to conventional designs

# Operational models for CHERI compartmentalization

- An **architectural protection model** enabling new software behavior
- As with virtual memory, multiple **operational models** can be supported
  - E.g., with an MMU: Microkernels, processes, virtual machines, etc.
  - How are compartments created/destroyed? Function calls vs. message passing? Signaling, debugging, ...?
- We have explored multiple viable CHERI-based models to date, including:

|                                   |                                              |
|-----------------------------------|----------------------------------------------|
| <b>Isolated dynamic libraries</b> | Efficient but simple sandboxing in processes |
| <b>UNIX co-processes</b>          | Multiple processes share an address space    |
- Improved performance and new paradigms using CHERI primitives
- Both will be available in CheriBSD/Morello

# Proposed operational models: Isolated libraries and UNIX co-processes

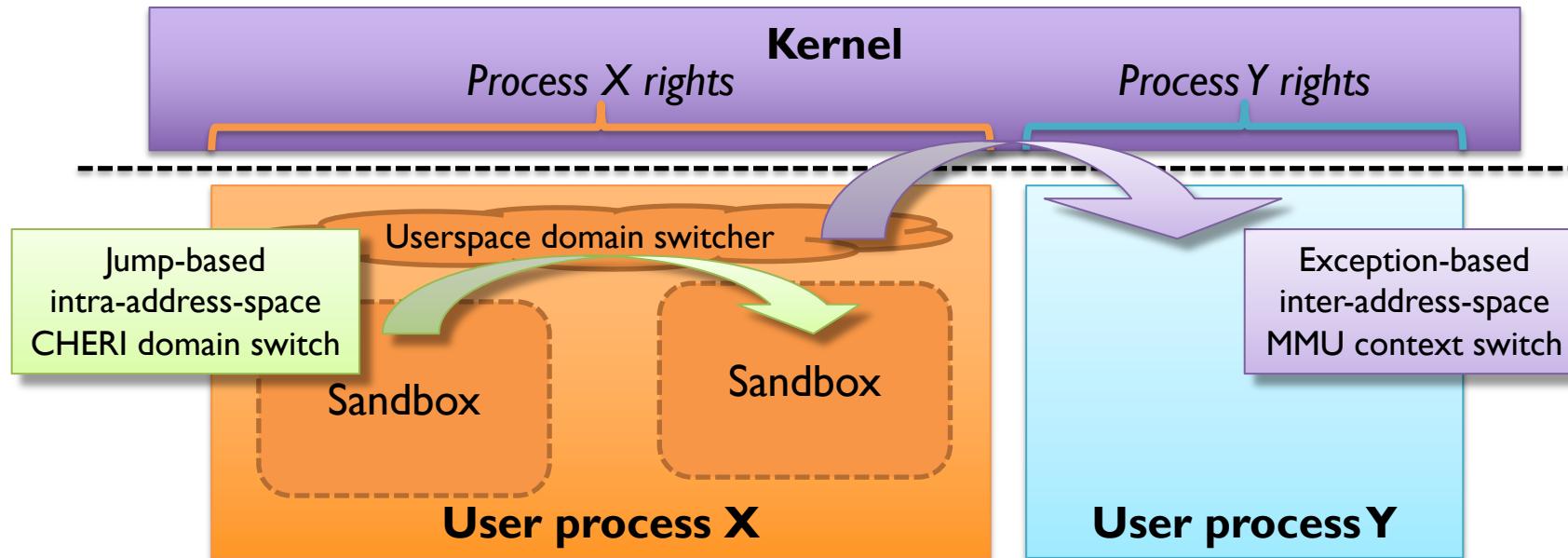
## Isolated dynamically linked libraries

- New API loads libraries into in-process sandboxes.
- Calling functions in isolated libraries performs a domain transition, with overheads comparable to function calls.
- Simple model eschews asynchrony, independent debugging, etc.

## UNIX co-processes

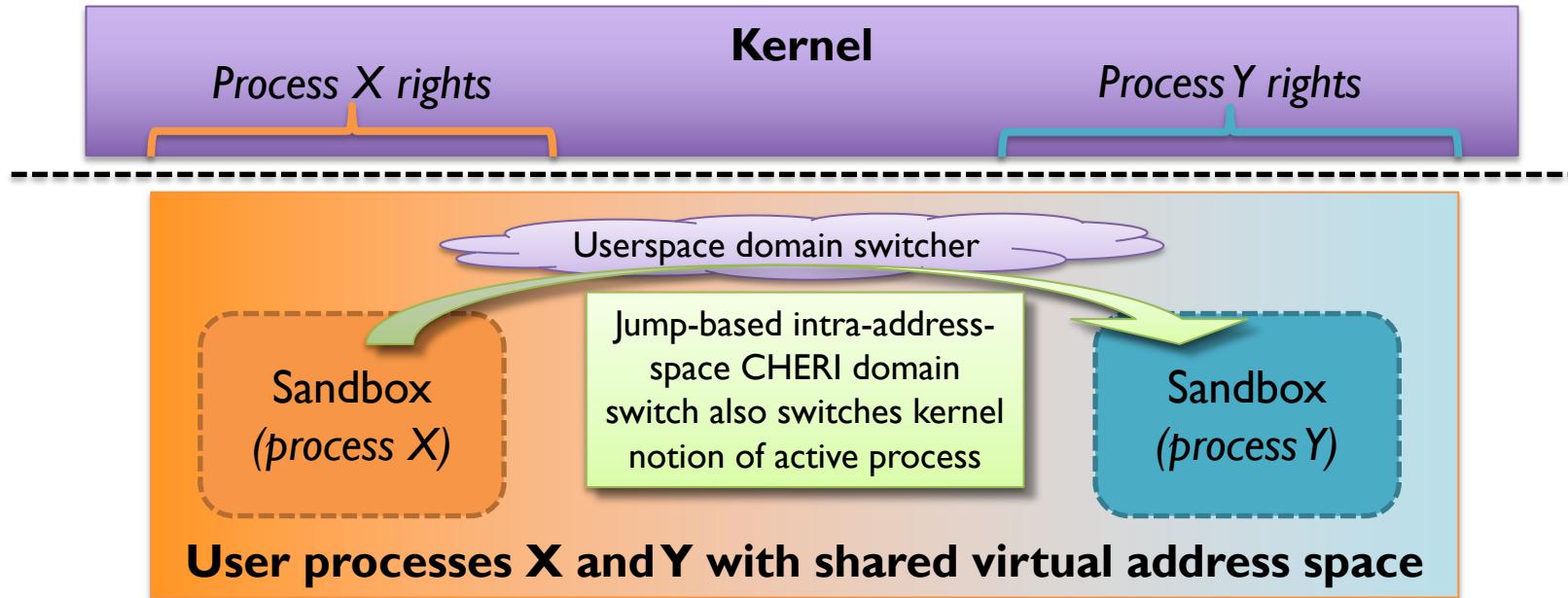
- Multiple processes share a single virtual address space, separated using independent CHERI capability graphs.
- CHERI capabilities enable efficient sharing, domain transition.
- Rich model associates UNIX process with each compartment.
- **Active area of research; early prototype available for co-processes**

# Example: Robust shared libraries



- User compartments exist **within individual UNIX processes** (“robust shared libraries”):
  - CHERI isolates compartments within each address spaces
  - Compartment switcher is itself a trusted userspace library
  - Compartments have strict subset of OS rights of the process
- Intra-process domain switches take **no architectural exceptions** and **do not enter the kernel**
- Multiple processes + IPC required if differing OS right sets needed

# Example: CHERI co-process model



- CHERI isolates **multiple processes** within a single virtual address space
  - Kernel-provided trusted compartment switcher runs in userspace (actually a microkernel)
  - CHERI-based inter-process memory sharing + domain switching
  - A compartment's OS rights correspond to the owning process
- Inter-process context switches take **no architectural exceptions** and **do not enter the kernel**
- CHERI can be pitched as **improving IPC performance** while **retaining a (largely) conventional process model**

# Co-Processes

- **Key insight:** With CheriABI, we can safely colocate multiple UNIX processes within the same virtual address space using CHERI capabilities
  - **Kernel constrains capabilities returned to user processes** such that they can only reach their own pages within address space
  - **Kernel selectively shares capabilities between processes** to facilitate fast IPC: shared PTEs, TLB entries
  - **Cheap exception-free inter-process context switch** using sealed capabilities, lazy kernel context switch
- Significantly reduce IPC overhead in compartmentalized/sandboxed systems with high process counts and frequent message passing

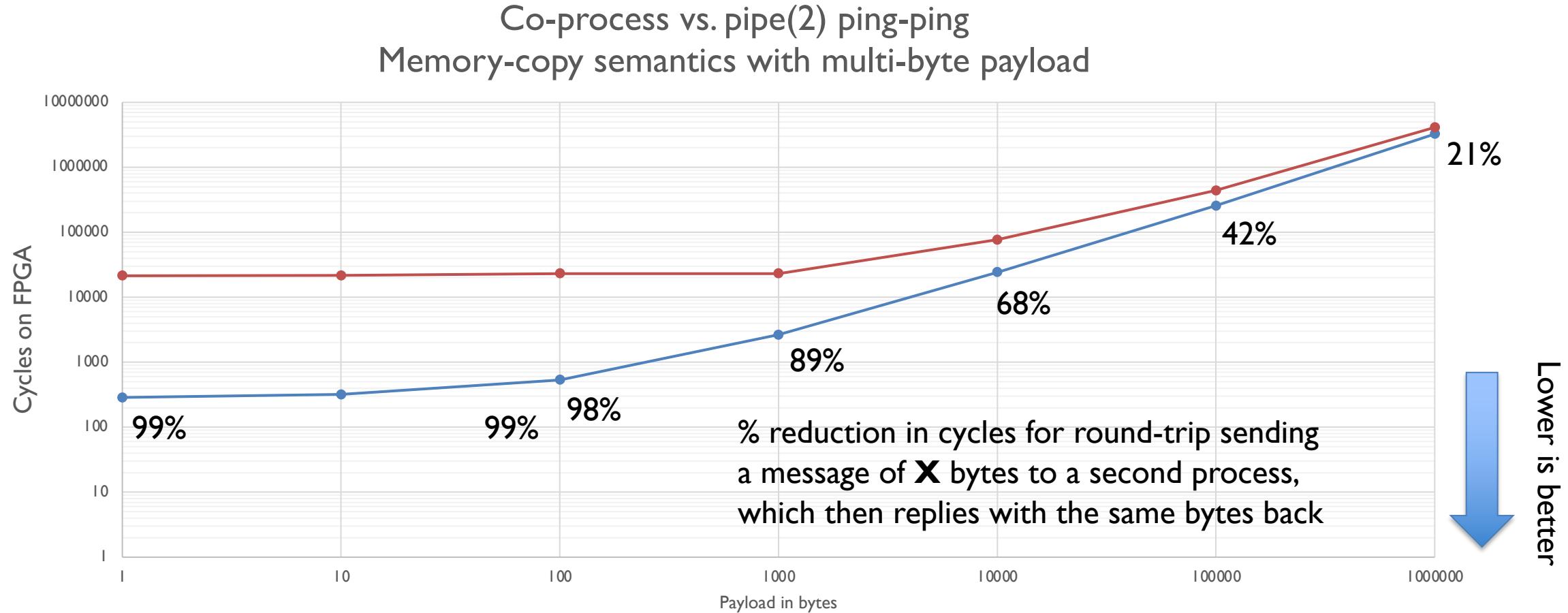
# Prototype status

- Sealed switcher service is exposed via vdso-like kernel page in userspace
- New APIs implemented by kernel and libc:
  - coexecve(2)** Create new process environment within current address space
  - cosetup(2)** Set up coprocess services;  
Register to invoke or accept calls
  - coregister(2)** Register named coprocess service
  - colookup(2)** Return sealed capabilities for  
named coprocess service
  - cocall(2)** Invoke coprocess service
  - coaccept(2)** Donate thread to receive invocations
- **coexecve(2)**, **cosetup(2)**, **coregister(2)**, and **colookup(2)** are system calls
- **cocall(2)** and **coaccept(2)** are libc symbols wrapping sealed-capability invocation of the kernel-provided switcher

# coping benchmark

- Simple IPC benchmark compares pipe(2) IPC and coprocess communication
  - Message-passing (copy) semantics (emulate pipes)
  - Send **X** bytes to service, which returns same **X** bytes
  - Measure overhead as **X** scales up (and down)
- For now, does not attempt to exploit pointer sharing between processes
  - only the switcher can access memory from both processes
- In the future, will explore pointer passing between processes in coprocess-aware RPC libraries to further reduce copies

# Early results: IPC ping-pong microbenchmark



**The fine print:** Cycles include IPC setup, amortized over 10,000 iterations of the IPC loop. Both processes use the pure-capability ABI using 256-bit capabilities. 272-entry TLB, 32K LI I-Cache, 32K LI D-Cache, 256K L2 Cache.

Co-process    pipe(2)

# Co-process next directions

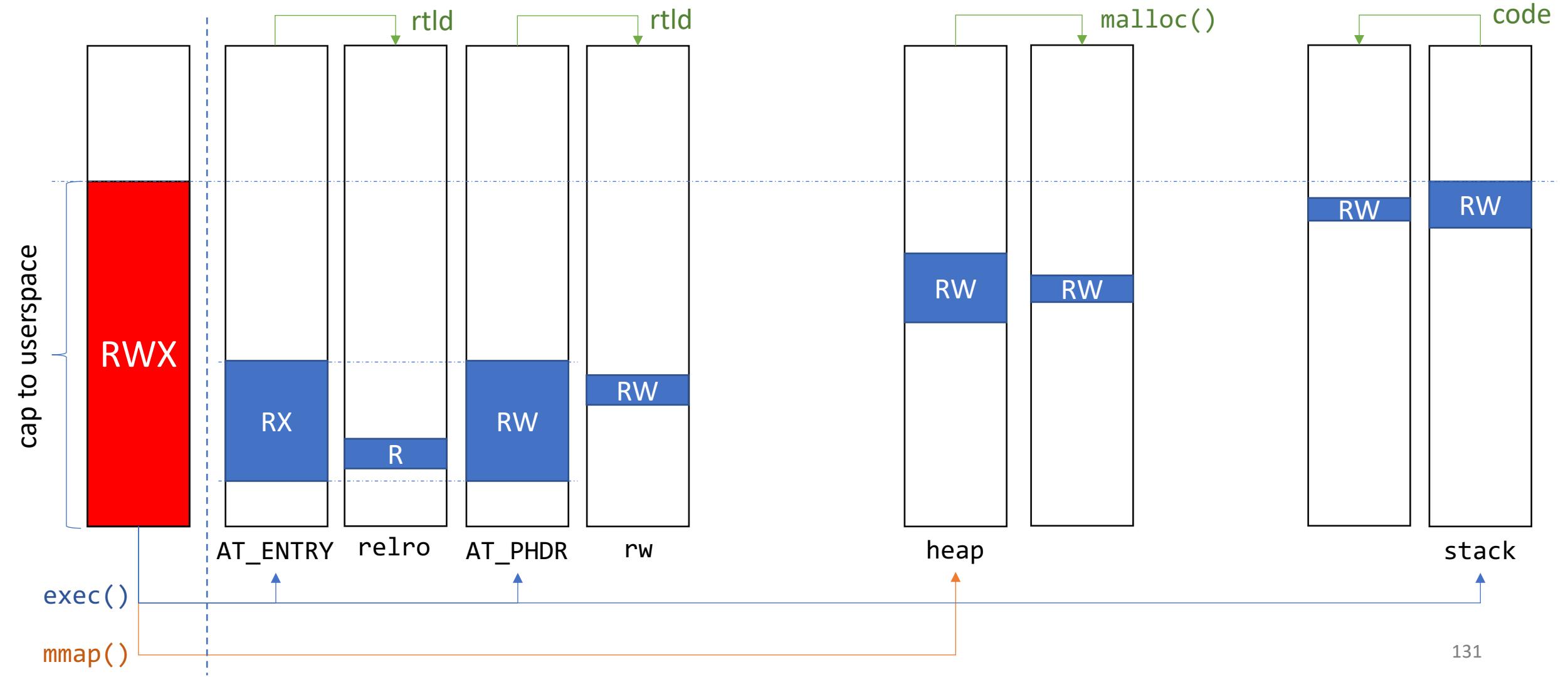
- Debugging, resource accounting, ..., while in a lazily switched process
- Enhancements to IPC APIs, namespaces, ...
- Introduce asynchronous **cosend(2)/corecv(2)** to complement synchronous **cocall(2)/coaccept(2)**
- Explore impact of pointer sharing on common RPC libraries and IPC-based applications: As knowledge goes up the stack, eliminate copies
- More mature benchmarking and evaluation
- Compare with optimized UNIX IPC: shm + semaphore on multicore
- **Experimentation on Arm Morello**

# Q&A

- Thanks for joining us. We hope you have enjoyed the exercises.
- Remaining time for open Q&A.
- Additional links for your clicking pleasure:
  - <https://cheri-cpu.org> – CHERI project at University of Cambridge
  - [Publications](#)
  - [cl-cheri-discuss mailing list](#)

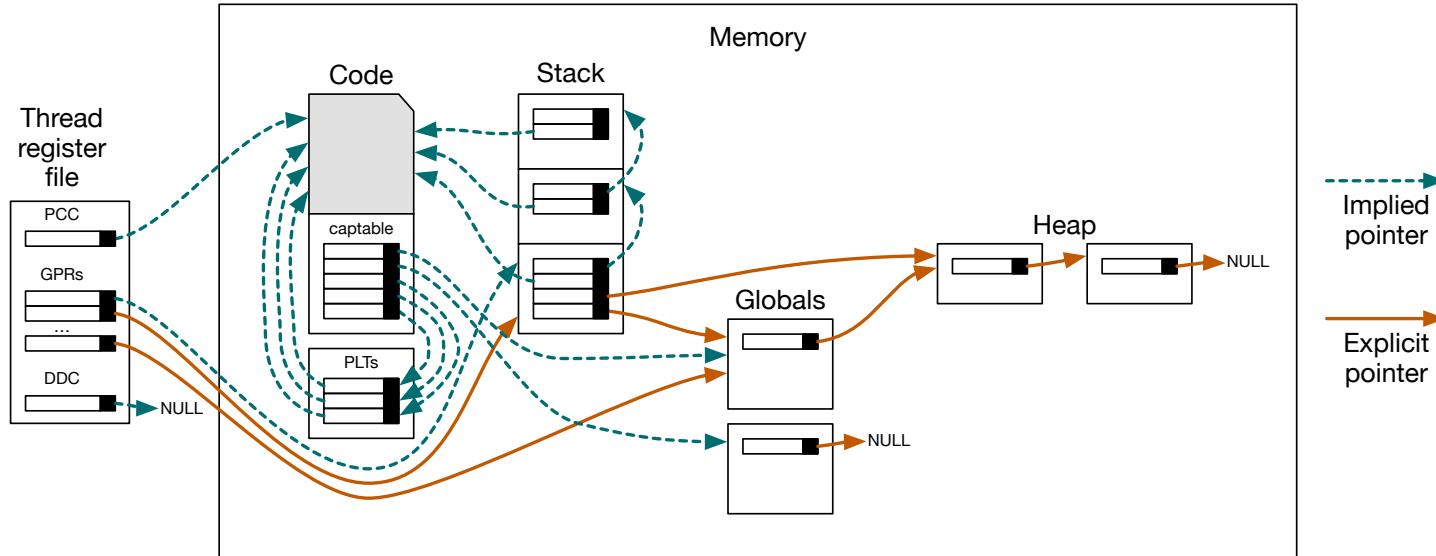
# CheriABI: Spatially Safe UNIX Processes

## Discussion (bonus): Process Construction



# CheriABI: Spatially Safe UNIX Processes

## Discussion (bonus): Capability Graph



- Pure-capability user programs speak capabilities across kernel syscall boundary
  - cap roots come from execve() and mmap() and friends
- Hardware checks all dereference operations
- Consequence: process is *confined to transitive closure of register file!*

# CHERI Tags in Cores and Caches

