CTSRD

CRASH-worthy
Trustworthy
Systems
Research and
Development

CHERI: Reinventing Computer Architecture for Security

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

Jonathan Anderson, John Baldwin, Hadrien Barrel, Ruslan Bukin, David Chisnall, Nirav Dave, Brooks Davis, Lawrence Esswood, Khilan Gudka, Alexandre Joannou, Robert Kovacsics, Ben Laurie, A. Theo Markettos, J. Edward Maste, Alfredo Mazzinghi, Alan Mujumdar, **Prashanth Mundkur**, Steven J. Murdoch, Edward Napierala, Robert Norton-Wright, Philip Paeps, Alex Richardson, Michael Roe, Colin Rothwell, Hassen Saidi, Peter Sewell, Stacey Son, Andrew Turner, Munraj Vadera, Jonathan Woodruff, Hongyan Xia, and Bjoern A. Zeeb

University of Cambridge SRI International

Summer School on Formal Techniques, Menlo College

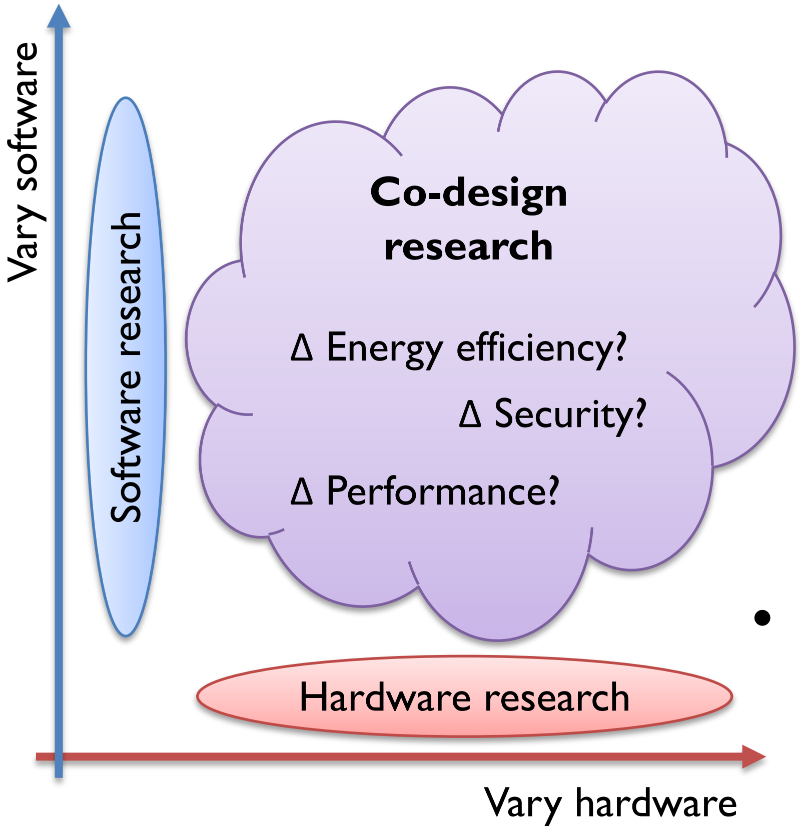
23 May 2019

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 contracts FA8750-10-C-0237 ('CTSRD') and FA8750-11-C-0249 ('MRC2'). 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.

SRI International

UNIVERSITY OF CAMBRIDGE

Hardware-software co-design



- There are many deterrents to co-design:
  - Design space is much larger
  - Vastly more work to realistically evaluate
  - Long transition times to industrial practice
  - Non-overlapping areas of expertise
  - Differing implementation cycles / timelines
  - Skeptical academic/industrial views
- But potential for **enormous rewards**
  - **New computer architecture** enables new workloads, deployments, and use cases
  - Disrupts current fundamental tradeoff spaces
  - E.g., GPUs, BigLittle, NVM, etc.

DARPA – CRASH

If you could revise the fundamental principles of computer-system design to improve security…

…what would you change?

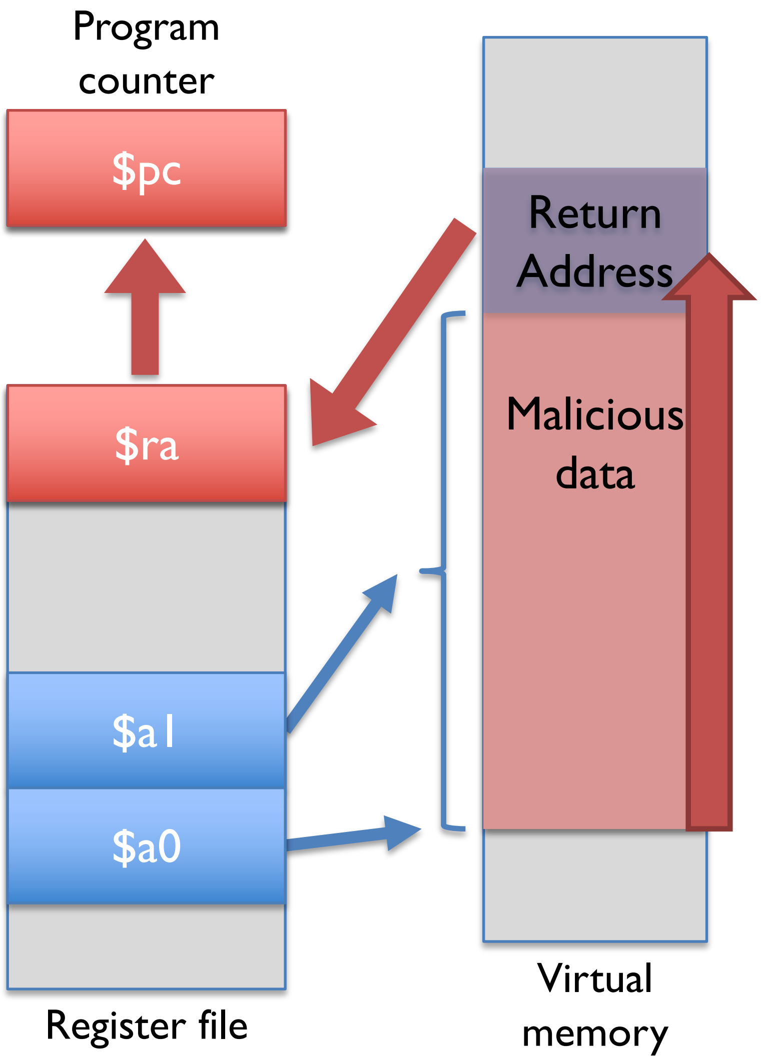
Principle of least privilege

Every program and every privileged user of the system should operate using the **least amount of privilege necessary** to complete the job.

Saltzer 1974 - CACM 17(7)
Saltzer and Schroeder 1975 - Proc. IEEE 63(9)
Needham 1972 - AFIPS 41(1)
…

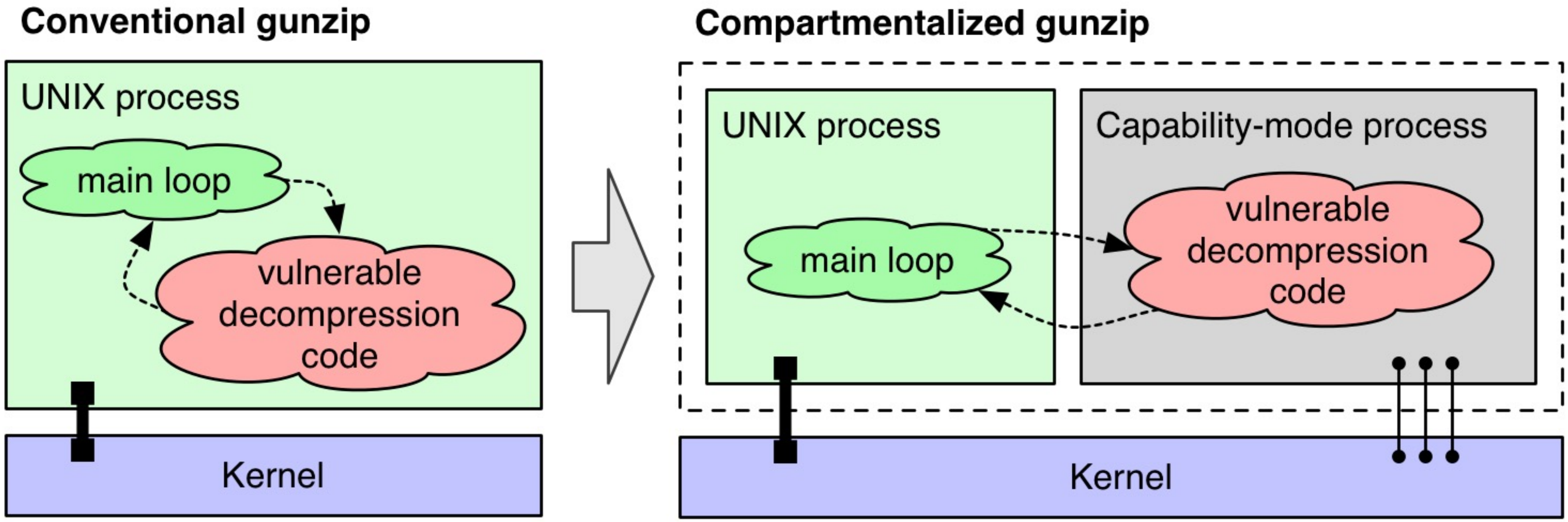
Architectural least privilege

- Classical buffer-overflow attack
  - Buggy code overruns a buffer, overwrites on-stack return address
  - Overwritten return address is loaded and jumped to, corrupting control flow
- These privileges were not required by the C language – so why grant them:
  - Ability to overrun the buffer?
  - Ability to corrupt or inject a code pointer?
  - Ability to execute data as code?
- Limiting privilege doesn't fix bugs – but does provide **vulnerability mitigation**
- Current ISAs do not enable **efficient, fine-grained privilege reduction**



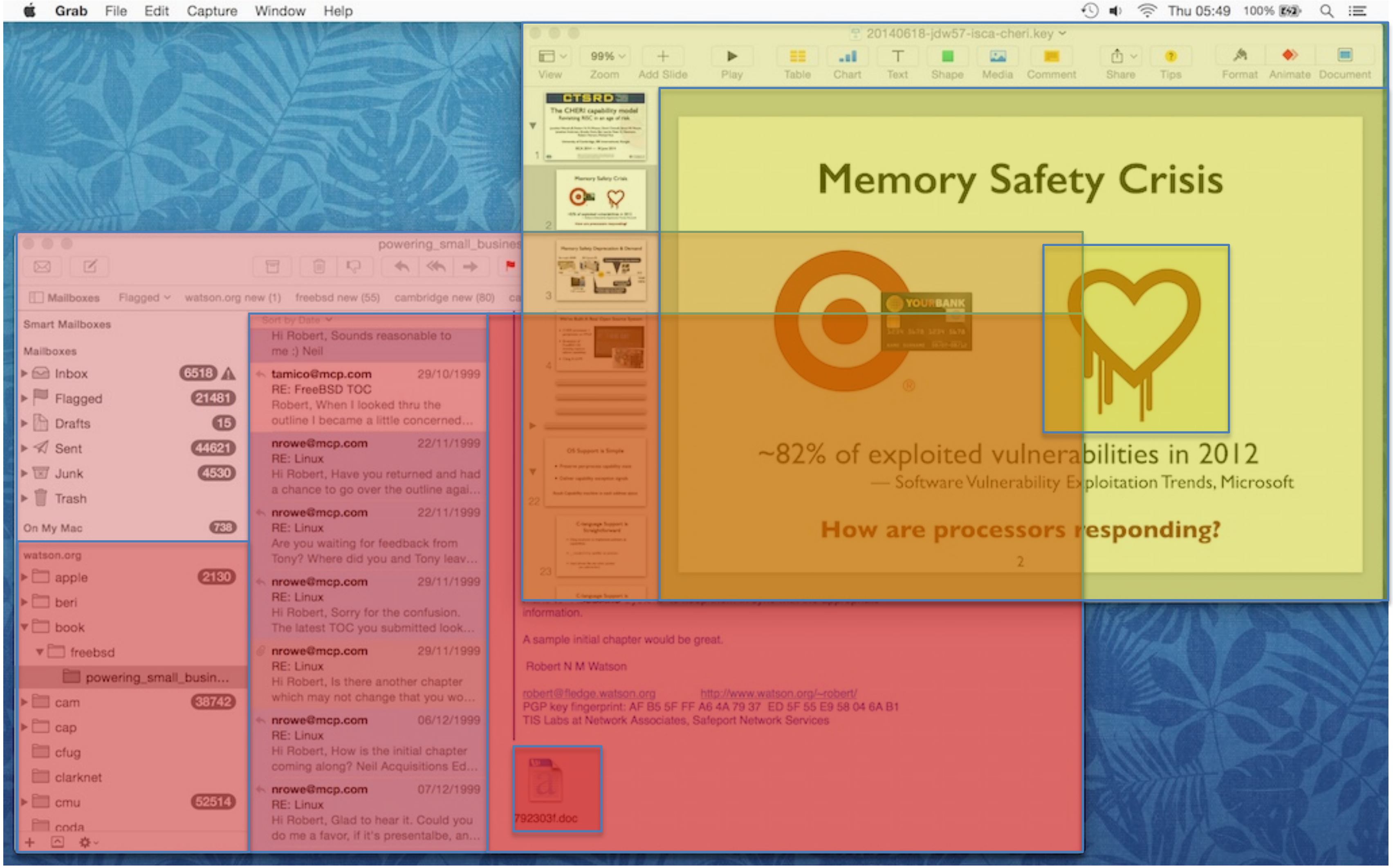
Application-level least privilege (1)

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/exploits**

Application-level least privilege (2)



Hi Robert, Sounds reasonable to me :) Neil

tamico@mcp.com 29/10/1999
RE: FreeBSD TOC
Robert, When I looked thru the outline I became a little concerned...

nrowe@mcp.com 22/11/1999
RE: Linux
Hi Robert, Have you returned and had a chance to go over the outline agai...

nrowe@mcp.com 22/11/1999
RE: Linux
Are you waiting for feedback from Tony? Where did you and Tony leav...

nrowe@mcp.com 29/11/1999
RE: Linux
Hi Robert, Sorry for the confusion. The latest TOC you submitted look...

nrowe@mcp.com 29/11/1999
RE: Linux
Hi Robert, Is there another chapter which may not change that you wo...

nrowe@mcp.com 06/12/1999
RE: Linux
Hi Robert, How is the initial chapter coming along? Neil Acquisitions Ed...

nrowe@mcp.com 07/12/1999
RE: Linux
Hi Robert, Glad to hear it. Could you do me a favor, if it's presentalbe, an...

information.

A sample initial chapter would be great.

Robert N M Watson

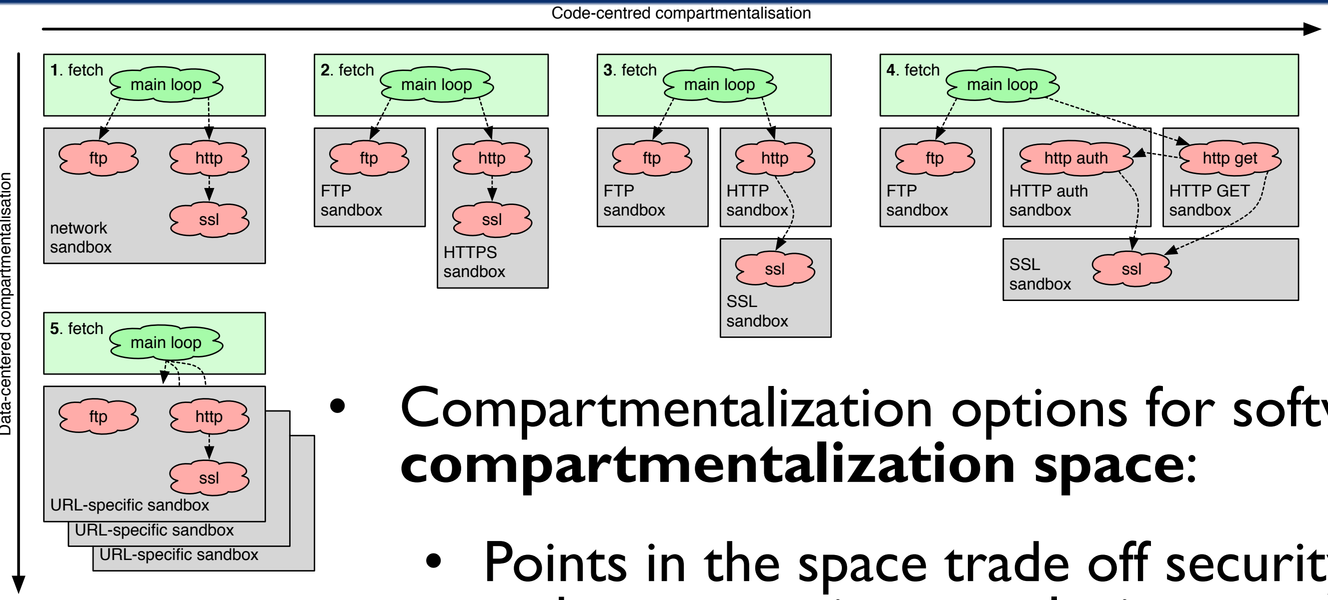
robert@fledge.watson.org http://www.watson.org/~robert/
PGP key fingerprint: AF B5 5F FF A6 4A 79 37 ED 5F 55 E9 58 04 6A B1
TIS Labs at Network Associates, Safeport Network Services

Memory Safety Crisis

~82% of exploited vulnerabilities in 2012
— Software Vulnerability Exploitation Trends, Microsoft

How are processors responding?

2



- Compartmentalization options for software describe a **compartmentalization space**:
  - Points in the space trade off security against performance and programming complexity
- Increasing **compartmentalization granularity** better approximates the principle of least privilege …
- … but **Memory Management Unit (MMU)**-based architectures do not scale to many compartments (processes):
  - Poor spatial protection granularity
  - Limited simultaneous-process scalability
  - Multi-address-space programming model

CHERI PROTECTION MODEL

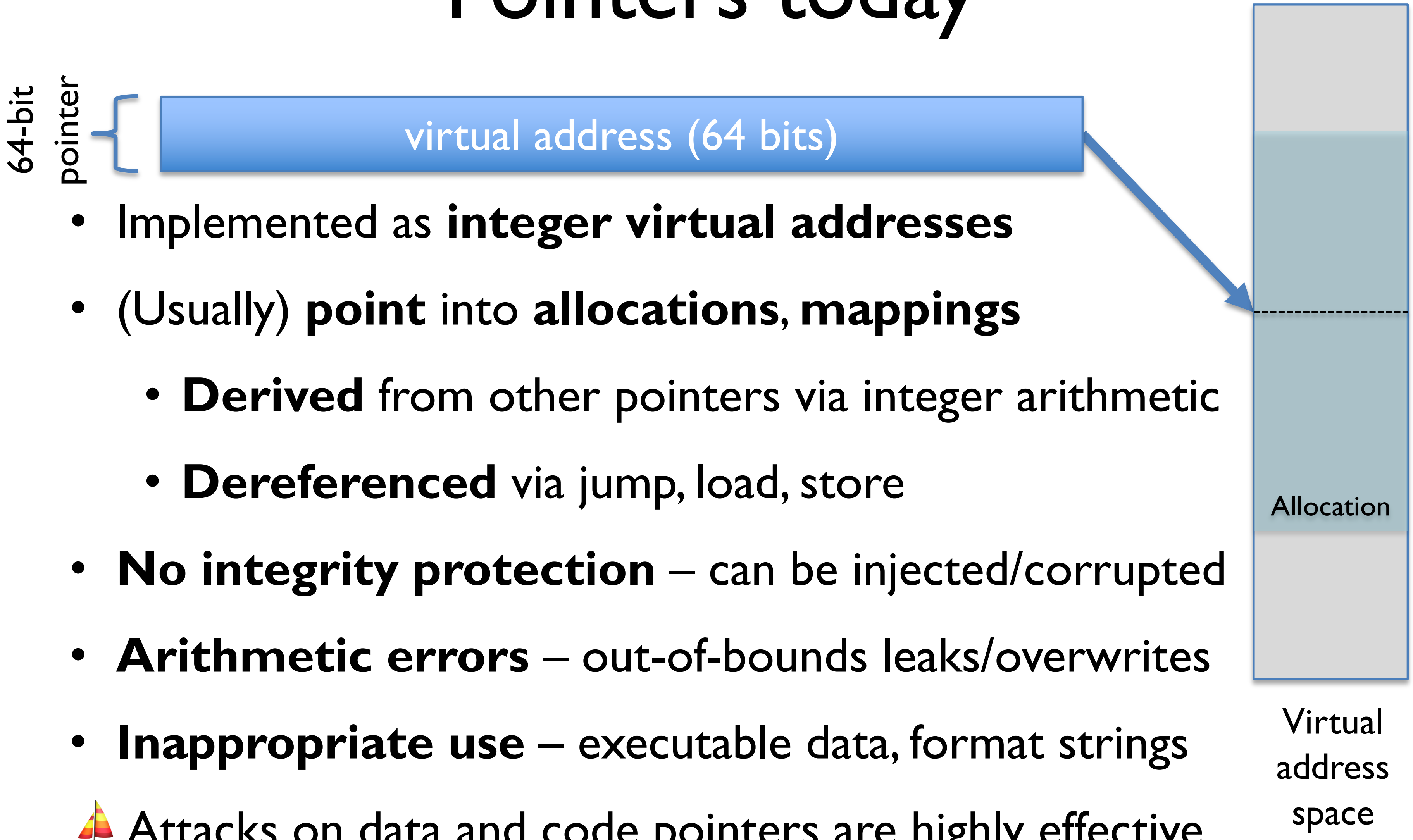
CHERI software protection goals

- **C/C++-language TCBs:** kernels, runtimes, browsers, …
- **Granular spatial protection, pointer protection**
  - Buffer overflows, control-flow attacks (ROP, JOP), …
- **Foundations for temporal safety**
  - Mitigate memory re-use attacks
  - E.g., through accurate C-language garbage collection
- **Higher-level language safety**
  - E.g., mitigate C++ COOP attacks
- **Scalable in-process compartmentalization**
  - Facilitate exploit-independent mitigation techniques

CHERI architectural goals

- **De-conflate virtualization and protection**
  - **Memory Management Units (MMUs)** protect by **location**
  - **CHERI** protects **references** to code and data: **pointers**
- **Architectural mechanism** directed by **software policy**
  - **Language-based properties**
    (e.g., C/C++ compiler, linkers, OS model, runtime)
  - **New software abstractions**
    (e.g., confined objects for compartmentalization)
- **Hybrid capability-system model**
  - **Capability systems** target the **principle of least privilege**
    (more on capabilities in a moment)
  - **Hybrid capability systems** compose cleanly w/current designs
- **Low overhead** for **fine-grained memory protection**
- **Significant performance gain** for **compartmentalization**

Pointers today



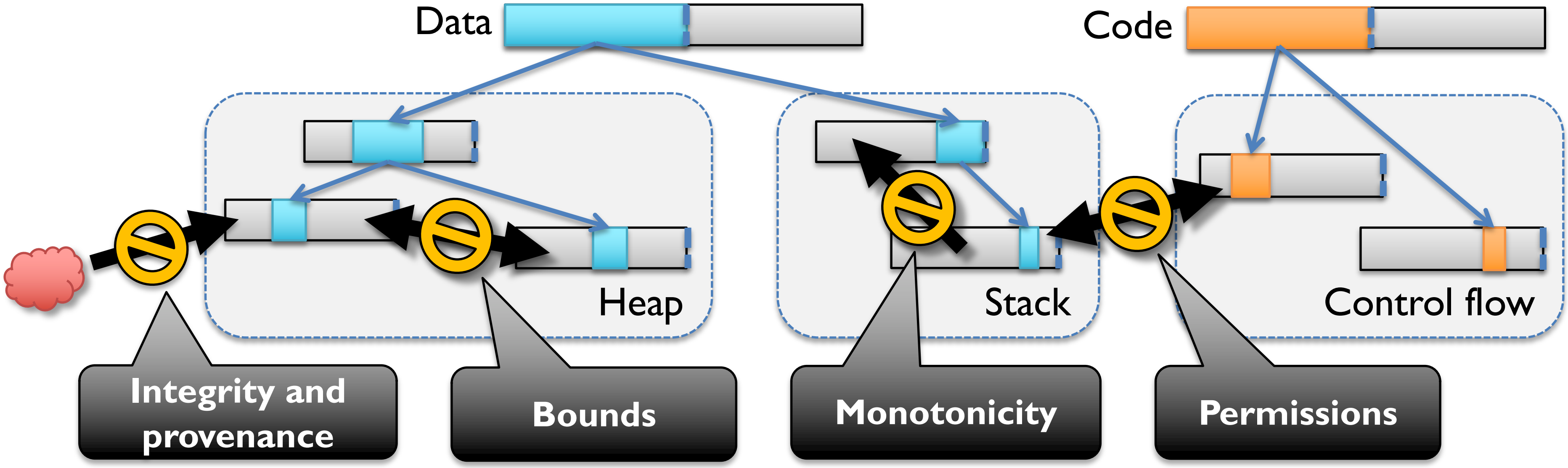
- Implemented as **integer virtual addresses**
- (Usually) **point** into **allocations**, **mappings**
  - **Derived** from other pointers via integer arithmetic
  - **Dereferenced** via jump, load, store
- **No integrity protection** – can be injected/corrupted
- **Arithmetic errors** – out-of-bounds leaks/overwrites
- **Inappropriate use** – executable data, format strings

⛵ Attacks on data and code pointers are highly effective achieving **arbitrary code execution**

CHERI protection model

- **RISC hybrid-capability architecture** supporting fine-grained, **pointer-based memory protection**:
  - **pointer integrity** (e.g., no pointer corruption)
  - **pointer provenance validity** (e.g., no pointer injection)
  - **bounds checking** (e.g., no buffer overflows)
  - **permission checking** (e.g., W^X for pointers)
  - **monotonicity** (e.g., no pointer privilege escalation / improper re-use)
  - **encapsulation** (e.g., protect software objects)

Architectural protection model for pointers



Valid userspace pointer set – provenance rules control dereference

- Valid pointers are derived from valid pointers via valid transformations
- E.g., Received network data cannot be interpreted as a code pointer

Pointer privilege reduction – capabilities allow pointers to carry specific privileges, which can be minimized with OS, compiler, and linker support:

- E.g., Pointers cannot be manipulated to access other heap or stack data

Foundation for **memory protection, software compartmentalization**

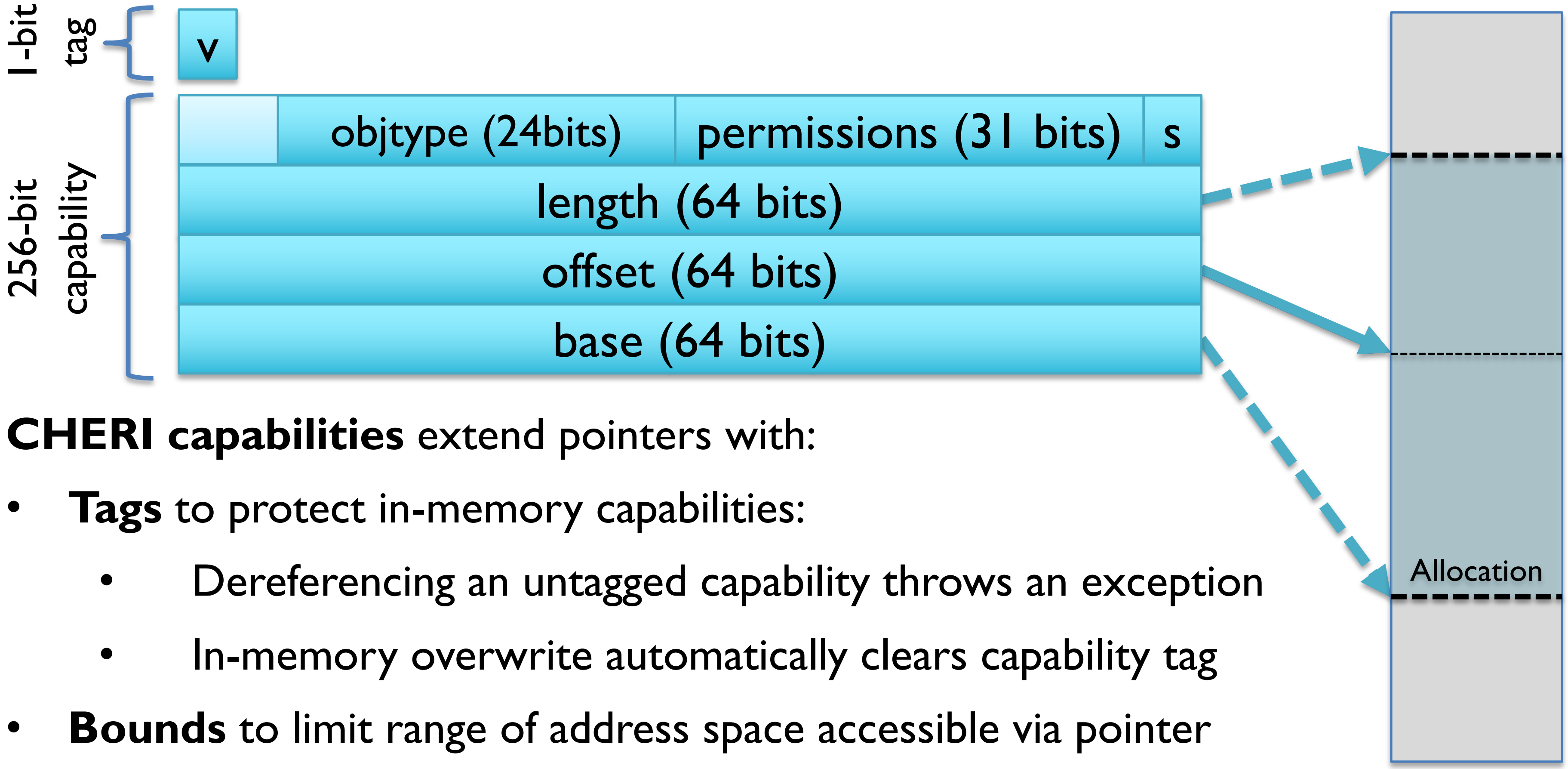
Approved for public release; distribution is unlimited. This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 ('CTSRD') and FA8750-11-C-0249 ('MRC2'). The views, opinions, and/or findings contained 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.

CHERI-MIPS INSTRUCTION-SET ARCHITECTURE (ISA)

CHERI architectural approach

- **RISC ISA extensions** that avoid microcode, table lookups, exceptions:
  - **MMUs** control the **implementation** of virtual addresses
  - **CHERI** protects **references** to virtual addresses (pointers)
- **Pointers** can be implemented via **architectural capabilities**
  - **Capabilities: unforgeable, delegable tokens of authority**
  - **Tagged memory** protects capability **integrity**, **provenance** in DRAM
  - **Pointer metadata**, including **bounds** and **permissions**, limit use
  - **Guarded manipulation** implements **capability monotonicity**
  - **Sealing** provides **immutable, software-defined capabilities**
  - **Exception model** allows controlled escape from constrained contexts
- **256-bit architectural model** – 64-bit addresses, 2x 64-bit bounds, etc.
  - Efficient **128-bit architectural, microarchitecture implementation**

256-bit architectural capabilities

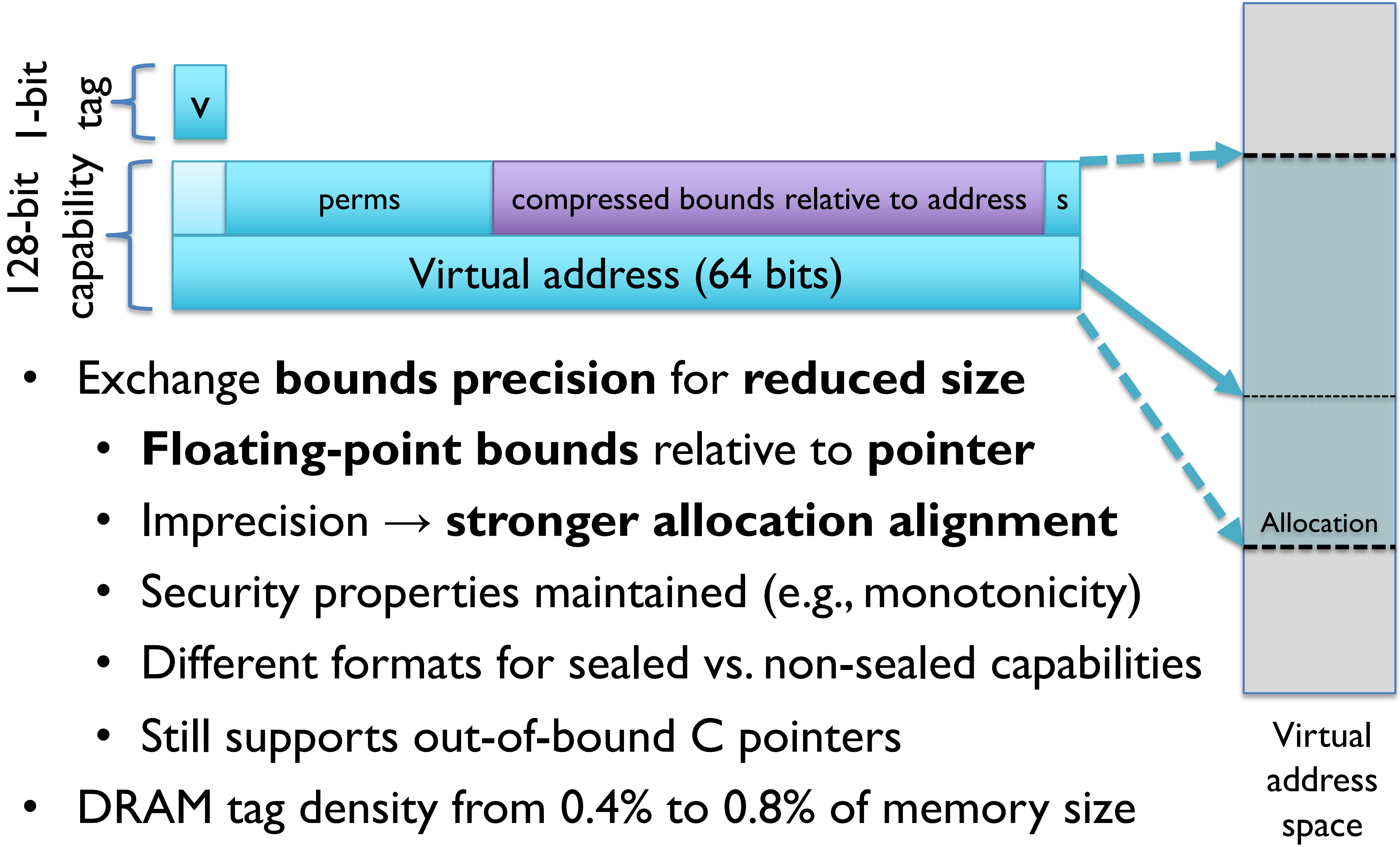


CHERI capabilities extend pointers with:

- **Tags** to protect in-memory capabilities:
  - Dereferencing an untagged capability throws an exception
  - In-memory overwrite automatically clears capability tag
- **Bounds** to limit range of address space accessible via pointer
- **Permissions to** limit operations – e.g., load, store, instruction fetch
- **Sealing** for **encapsulation**: **immutable**, **non-dereferenceable**
- **Guarded manipulation** enforces **monotonic rights non-increase**

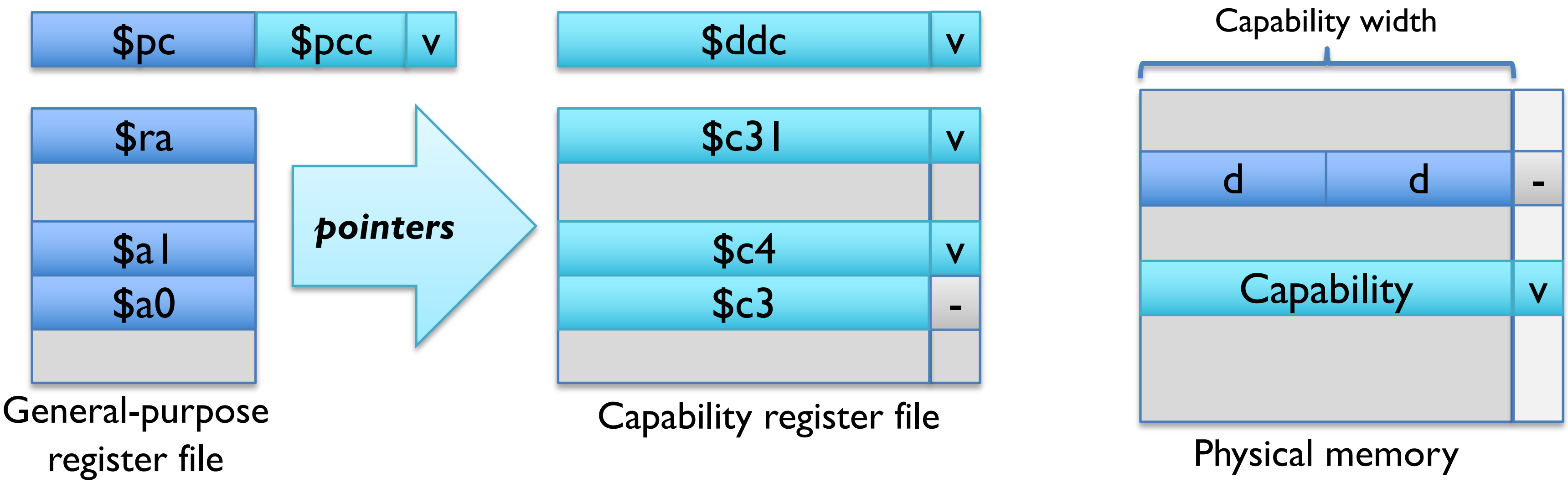
Virtual address space

128-bit micro-architectural capabilities



- Exchange **bounds precision** for **reduced size**
  - **Floating-point bounds** relative to **pointer**
  - Imprecision → **stronger allocation alignment**
  - Security properties maintained (e.g., monotonicity)
  - Different formats for sealed vs. non-sealed capabilities
  - Still supports out-of-bound C pointers
- DRAM tag density from 0.4% to 0.8% of memory size
- Full prototype with full software stack on FPGA

Mapping CHERI into 64-bit MIPS



- **Capability register file** holds in-use capabilities (code and data pointers)
- **Tagged memory** protects capability-sized and -aligned words in DRAM
- **Program-counter capability** ($pcc) constrains program counter ($pc)
- **Default data capability** ($ddc) constrains legacy MIPS loads/stores
- **System control registers** are also extended – e.g., $epc→$epcc, TLB

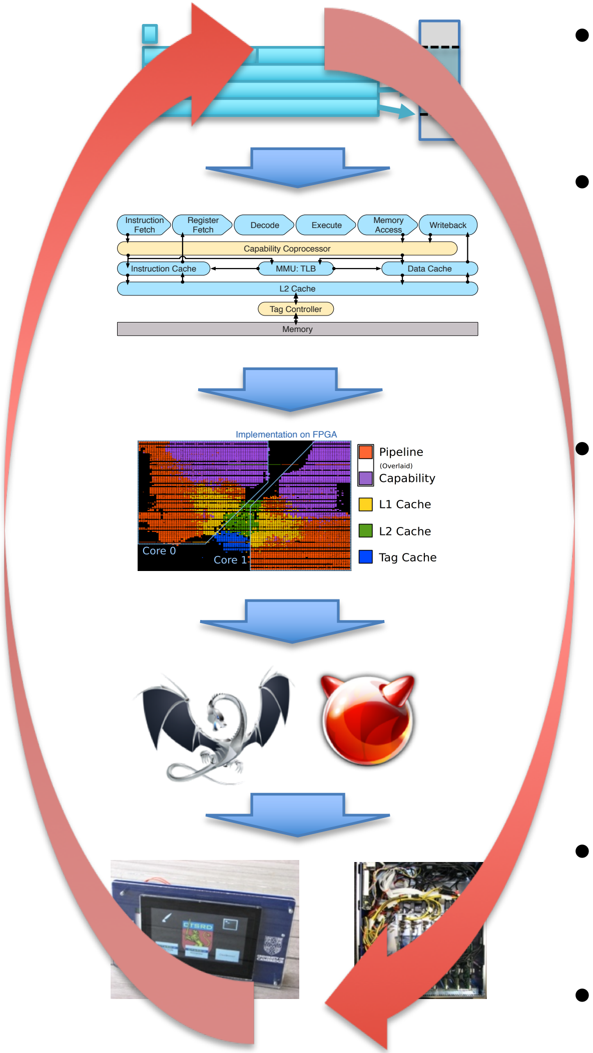
Virtual memory **and** capabilities

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

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

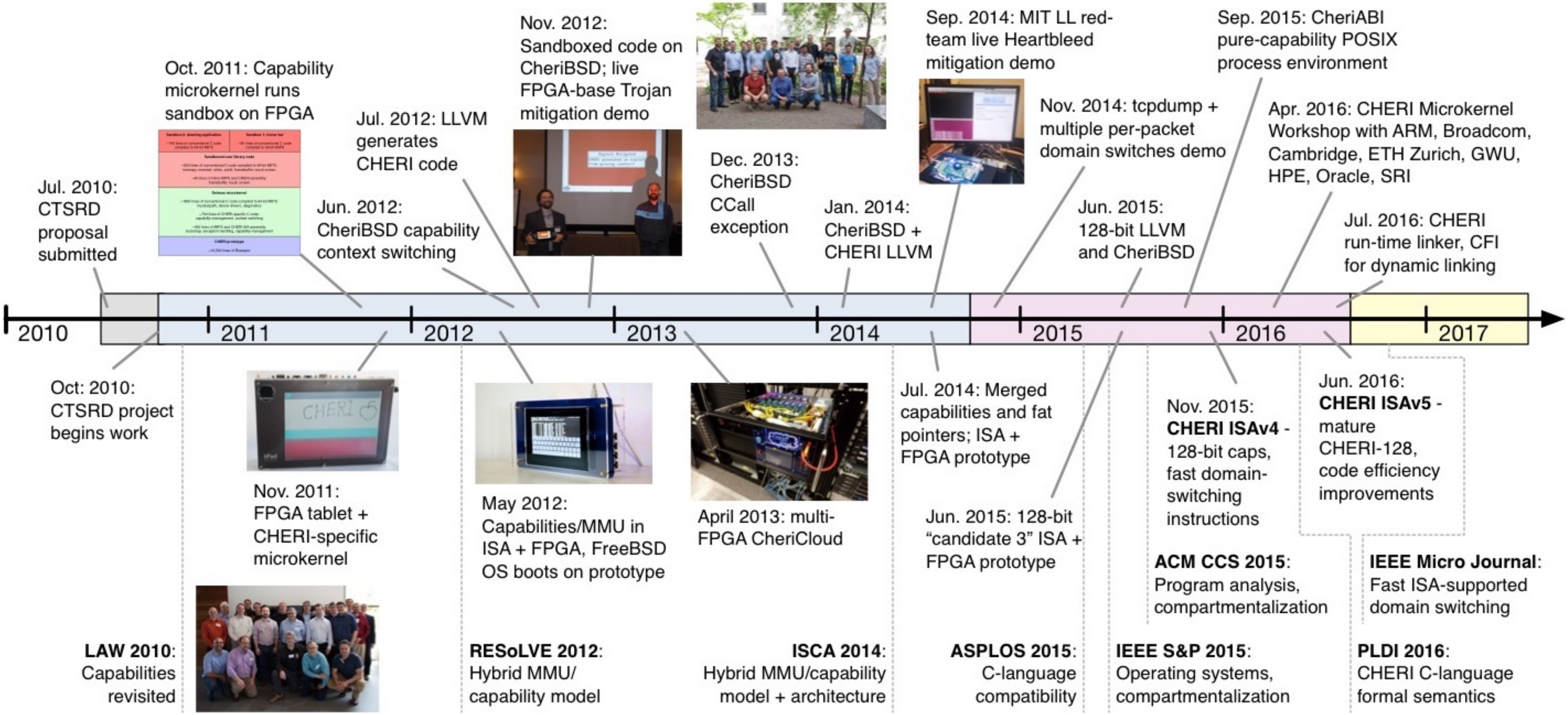
HARDWARE-SOFTWARE CO-DESIGN FOR CHERI

Hardware-software co-design



- **CHERI protection model** protects OS, C, linker, application structures and abstractions
- **CHERI-MIPS ISA** extends the 64-bit MIPS ISA
  - L3 + Sail MIPS + CHERI ISA **formal models**
  - Qemu-CHERI fast **ISA emulator**
- Bluespec SystemVerilog (BSV) pipelined, multicore **BERI MIPS + CHERI processor prototype**
  - Simple but realistic microarchitecture
  - C → Cycle-accurate software simulator
  - Verilog → Field Programmable Gate Array (FPGA) @100MHz
- **CHERI software stack**: FreeBSD, Clang/LLVM, application corpus – OpenSSH, Postgres, nginx, …
- **Evaluation**: Performance, security, compatibility, …

CHERI R&D Timeline



Jul. 2010: CTSRD proposal submitted

Oct: 2010: CTSRD project begins work

LAW 2010: Capabilities revisited

Oct. 2011: Capability microkernel runs sandbox on FPGA

Nov. 2011: FPGA tablet + CHERI-specific microkernel

May 2012: Capabilities/MMU in ISA + FPGA, FreeBSD OS boots on prototype

RESoLVE 2012: Hybrid MMU/ capability model

Jun. 2012: CheriBSD capability context switching

Jul. 2012: LLVM generates CHERI code

Nov. 2012: Sandboxed code on CheriBSD; live FPGA-base Trojan mitigation demo

April 2013: multi-FPGA CheriCloud

Dec. 2013: CheriBSD CCall exception

Jan. 2014: CheriBSD + CHERI LLVM

ISCA 2014: Hybrid MMU/capability model + architecture

Jul. 2014: Merged capabilities and fat pointers; ISA + FPGA prototype

Sep. 2014: MIT LL red-team live Heartbleed mitigation demo

Nov. 2014: tcpdump + multiple per-packet domain switches demo

ASPLOS 2015: C-language compatibility

Jun. 2015: 128-bit LLVM and CheriBSD

Jun. 2015: 128-bit “candidate 3” ISA + FPGA prototype

IEEE S&P 2015: Operating systems, compartmentalization

Sep. 2015: CheriABI pure-capability POSIX process environment

Nov. 2015: **CHERI ISAv4** - 128-bit caps, fast domain-switching instructions

ACM CCS 2015: Program analysis, compartmentalization

Apr. 2016: CHERI Microkernel Workshop with ARM, Broadcom, Cambridge, ETH Zurich, GWU, HPE, Oracle, SRI

Jun. 2016: **CHERI ISAv5** - mature CHERI-128, code efficiency improvements

PLDI 2016: CHERI C-language formal semantics

Jul. 2016: CHERI run-time linker, CFI for dynamic linking

IEEE Micro Journal: Fast ISA-supported domain switching

CHERI ISA refinement (+reinvention)

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 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 CHERI-x86 and CHERI-RISC-V sketches Exception-free domain transition

RISC + MMU + capabilities

Compartmentalization

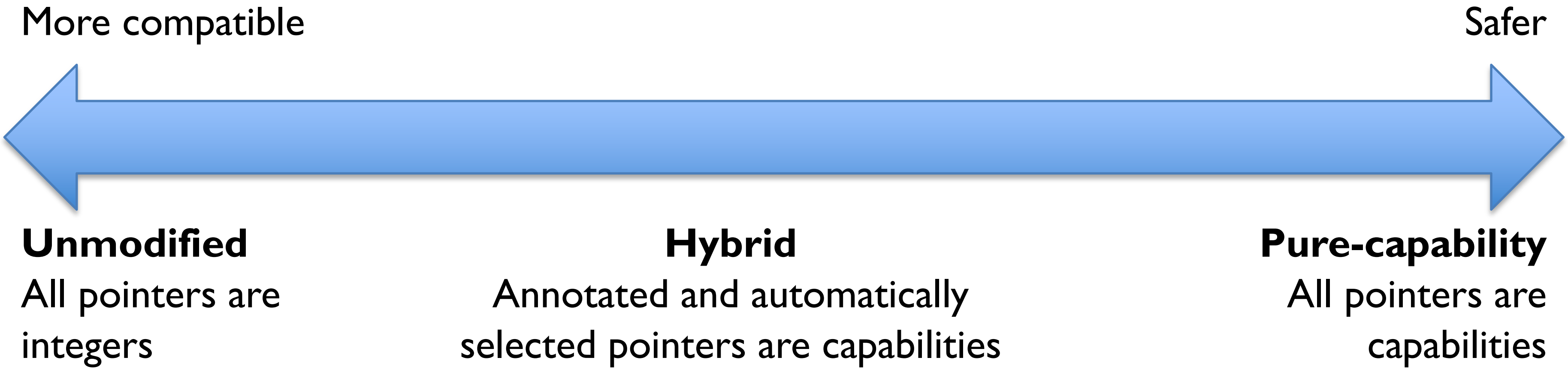
In-kernel use

C + capabilities

128-bit, code efficiency

CHERI SOFTWARE

CHERI software models



More compatible

Safer

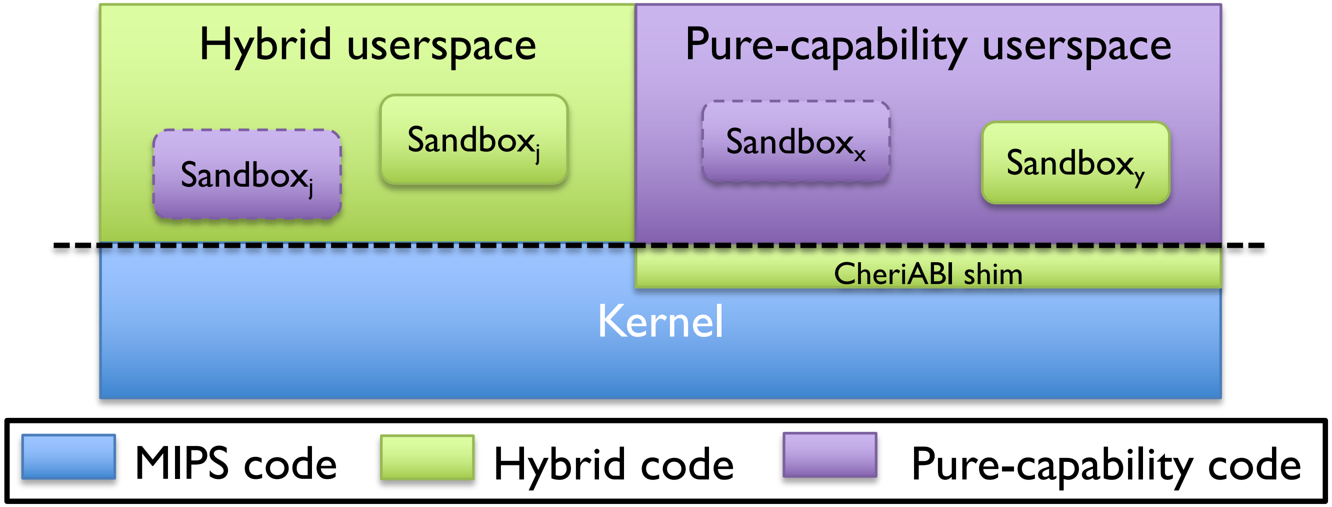
Unmodified
All pointers are integers

Hybrid
Annotated and automatically selected pointers are capabilities

Pure-capability
All pointers are capabilities

- **Source and binary compatibility**: **C-language idioms,** multiple **ABIs**
  - **Unmodified code**: Existing n64 code runs without modification
  - **Hybrid code**: E.g., used in return addresses, for annotated data/code pointers, for specific types, stack pointers, etc. n64-interoperable.
  - **Pure-capability code**: Ubiquitous data- and data-pointer protection. Non-n64-interoperable due to changed pointer size.
- **CHERI Clang/LLVM compiler prototype** generates code for all three

Multiple process ABIs



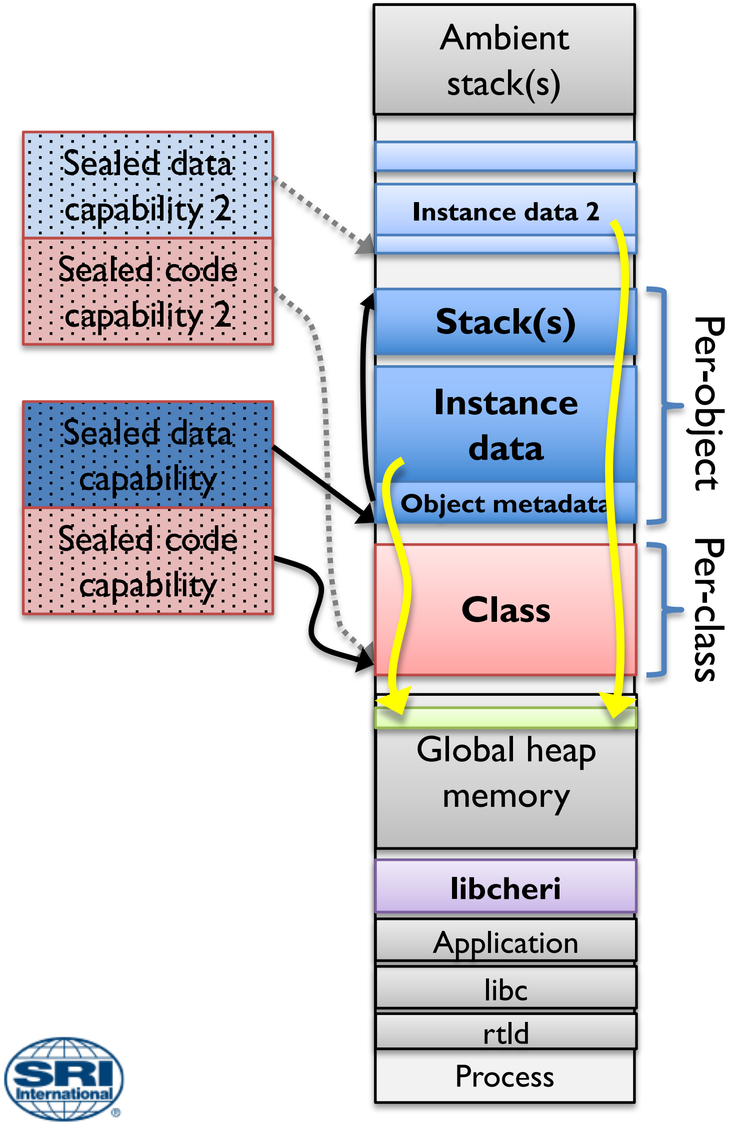
- **64-bit MIPS ABI:** n64-compatible hybrid code execution
    - Many pointers are integers (including system-call arguments)
    - Pure-capability code supported in sandboxes
- **CheriABI:** Strong pure-capability code throughout
    - All pointers are capabilities (including system-call arguments)
    - Hybrid-capability code supported in sandboxes

CheriABI: "pure-capability" processes

- Userspace compiled for **ubiquitous pointer protection**
  - Goal: OpenSSH (etc) without buffer overflows, ROP, JOP, …
- **Ensure valid provenance**, **minimize privilege** for pointers
  - Where does (or should) every pointer come from?
  - What bounds and permissions should each pointer have?
- Grand tour of the OS, process model, and toolchain:
  - execve() mappings, ELF auxiliary arguments, signals, …
  - Compile-time and run-time linker for code, globals
  - System calls accepting, returning, and stash pointers
  - Stack, heap, and application-specific allocators
- Trading off **privilege minimization** vs. **API conformance**

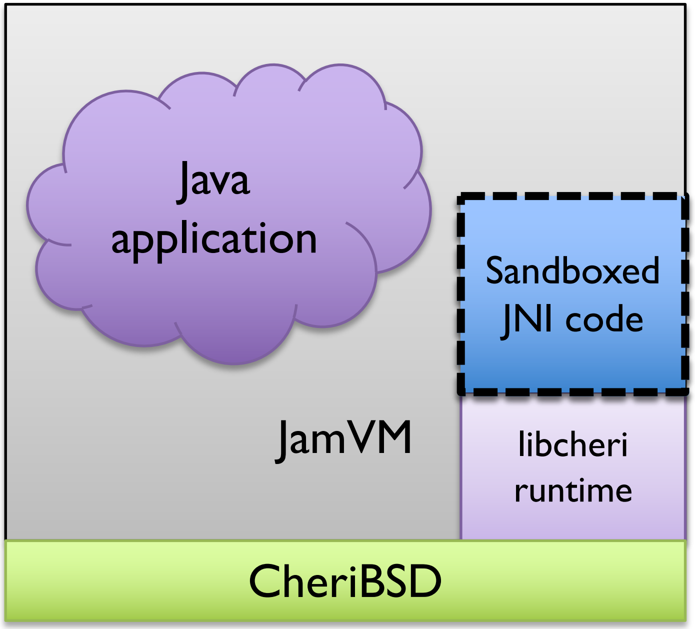
CHERI COMPARTMENTALIZATION

In-process compartmentalization using CHERI (sketch)



- **In-process protection domains**
  - Thread registers describe rights of running code
  - I.e., transitive closure of reachable capabilities
- **Userspace object-capability model**
  - libcheri loads and run-time links **classes**
  - Instantiates **confined objects** w/limited rights
  - **Sealed capabilities** enforce encapsulation
  - **Shared code and data** within address space
- **Fast and robust domain transition**
  - Controlled **non-monotonic transformations** of thread capability registers
- **Efficient object and memory sharing**
  - Delegate capabilities across invocation, return
- Paper at IEEE SSP 2015 (“Oakland”)

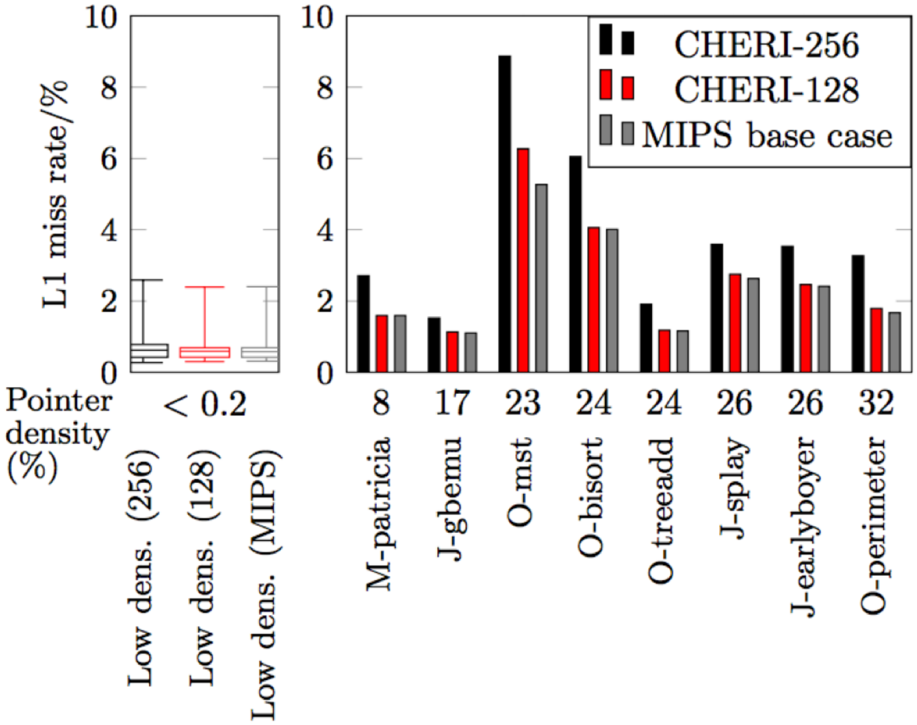
CHERI-JNI: Protecting Java from JNI



- **Java Native Interface (JNI)** allows Java programs to use native code for performance, code portability, functionality
  - Often fragile; sometimes overtly insecure
- Impose Java **memory-safety and security models** on JNI code
  - Full memory safety for native code
  - Limit JNI access to JVM state
  - Allow safe copy-free JNI access to Java buffers
  - Enforce Java security model on access to Java objects and system services (e.g., files, sockets)
- Prototyped using JamVM on CHERI-MIPS
- Paper at ASPLOS 2017

PERFORMANCE

Memory-protection performance



L1 D-Cache miss rate for CHERI-256, CHERI-128, and MIPS

Overhead tracks in-memory pointer density (e.g., increased memory use)

Metric: **D-Cache miss-rate change** from pointer-size growth?

Left: **Low pointer-density** benchmarks from MiBench

Right: **High pointer-density** benchmarks

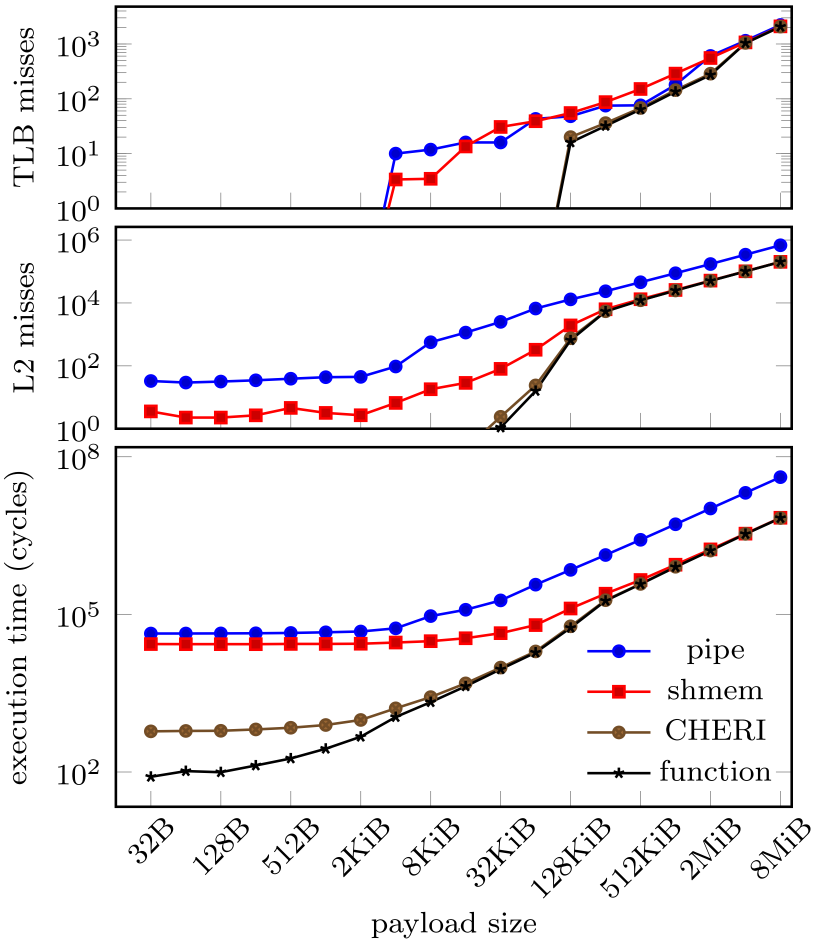
M - MiBench

O - Olden

J - Octane JavaScript

1%-2% increase in L1 D-Cache miss rate for 128-bit capabilities for most practical workloads

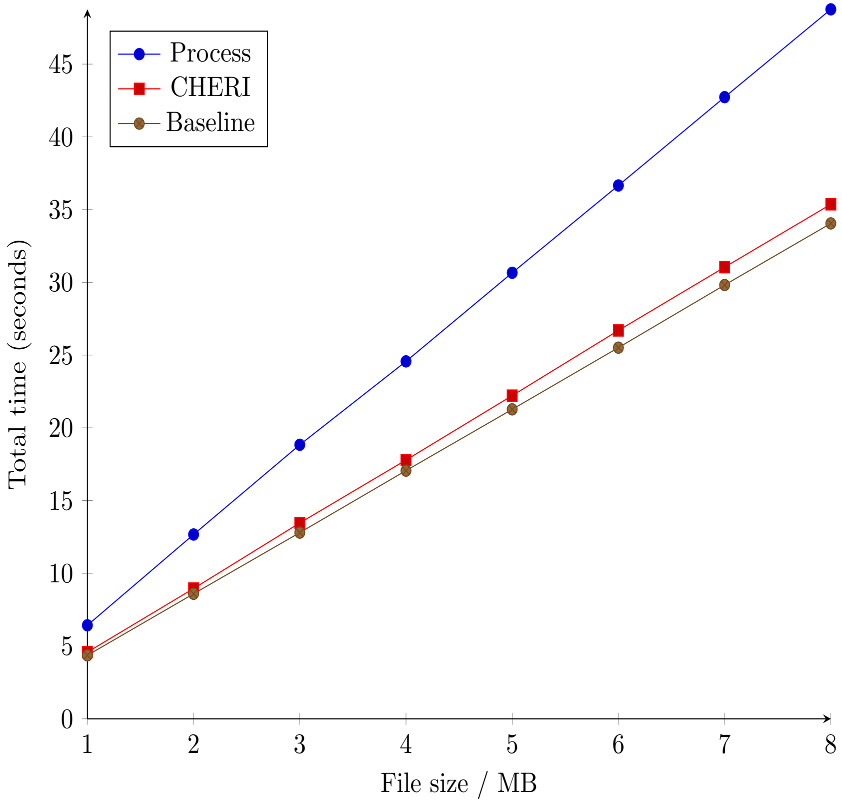
Domain switching



Comparison of domain-crossing methods – absolute cycle cost (log-log)

- **Function-call performance semantics** with low fixed overhead
- **Metrics**: L2 cache miss rate, TLB miss rates, execution time as workload footprint approaches limits
- Fixed cost for CCall/CReturn
  - No overhead to delegate memory or object capabilities
  - Much faster than IPC for frequent, small messages (<512K) common in compartmentalized programs
- Shared memory access scales with **in-process access** rather than **MMU-based sharing**

Memory-safe, compartmentalized gzip



Compression time in seconds for compartmentalized gzip

- **zlib** library compartmentalization
- Best **cut point** for security and reusability is a **shared-memory API**
  - Extremely awkward for MMU-based compartmentalization…
  - … but simple pointer delegation with CHERI compartmentalization
- MMU process-based sandboxing w/o memory safety
  - **40%-43%** wall-clock overhead
- CHERI object-based sandboxing with full memory safety
  - **3%-5%** wall-clock overhead (measured on 256-bit CHERI)

WHERE NEXT?

Ongoing research

- Quantitatively motivated ISA optimization and code generation
- Implications for more complex microarchitectures (e.g., superscalar)
- Tagged memory: tag cache vs. native support in DRAM
- Complete tool-chain: linker, debugger
- C++ compilation to CHERI (+COOP)
- OS support, larger application corpus
- CHERI and ISO C/POSIX APIs
- Map sandbox frameworks into CHERI
- CHERI-specific (MMU-free) microkernel
- CHERI for safe native-code interfaces (e.g., for Java's JNI)
- CHERI as a safe inter-language substrate
- Efficient C-language garbage collection
- CHERI and managed languages
- Formal proofs of ISA properties
- Formal proofs of software properties
- Verifying hardware implementations
- Interactions with persistent memory
- Dynamic tracing of CHERI provenance
- MMU-free HW designs for "IoT"

Conclusion

- **CHERI** is a **RISC hybrid capability-system architecture**
  - Iterative **hardware-software co-design** over 7 years
  - Novel convergence of MMU and capability-based approaches
  - Fine-grained, pointer-oriented protection for code and data
  - Strong, real-world C-language support with low overhead
  - Scalable, fine-grained intra-process compartmentalization
- Substantial vulnerability-mitigation benefit
  - Validated against large, real-world software corpora
- Publications include: ISCA 2014, ASPLOS 2015, IEEE SSP 2015, ACM CCS 2015, PLDI 2016, IEEE Micro 2016; ASPLOS 2017, …
- Open-source hardware and software; publication specifications

https://www.cheri-cpu.org/

Q&A

CHERI papers

ISCA 2014: Fine-grained, in-address-space memory protection hybridizing MMU, capability model

ASPLOS 2015: Explore and refine C-language compatibility; converge capabilities and fat pointers

Oakland 2015: Efficient, capability-based hardware-software compartmentalization within processes

ACM CCS 2015: Compartmentalization modeling

PLDI 2016: C-language semantics + CHERI extension (w/REMS)

IEEE Micro Journal September/October 2016: Hardware assistance for efficient domain switching

ASPLOS 2017: CHERI reinforcement for Java JNI

(Various other submissions, in-flight papers)

CHERI technical reports

Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (CHERI ISAv5)

- UCAM-CL-TR-891 – June 2016
- Mature 128-bit capabilities, code generation efficiency improvements, detailed exploration of language/ISA linkage
- **CHERI ISAv6** in May 2017 w/privileged code support

Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide

- UCAM-CL-TR-877 – November 2015
- C language, compiler, OS internals

- Multiple technical reports on the BERI prototyping platform

BACKUP SLIDES

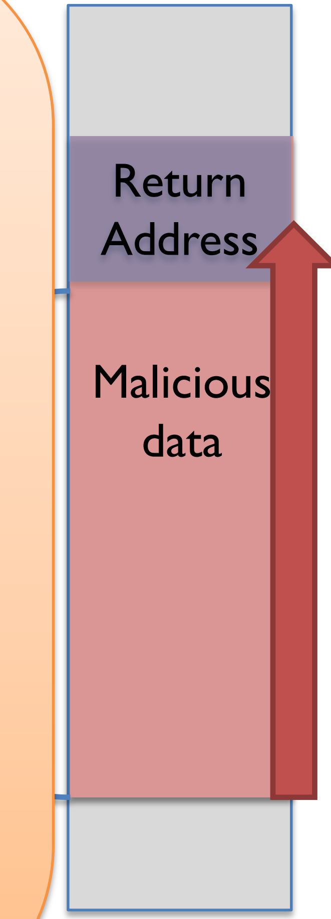
Architectural support for least privilege

CHERI memory protection:

- Eliminates out-of-bounds accesses
- Prevents injected data use as a code or data pointer
- Disallows jumping to data pointers
- Protects code pointers to limit code reuse attacks
- Mitigates as-yet undiscovered exploit techniques and supply-chain attacks through scalable compartmentalization
- Supports managed-language runtimes (e.g., accurate C garbage collection, safe native-code interfaces for Java)
- …

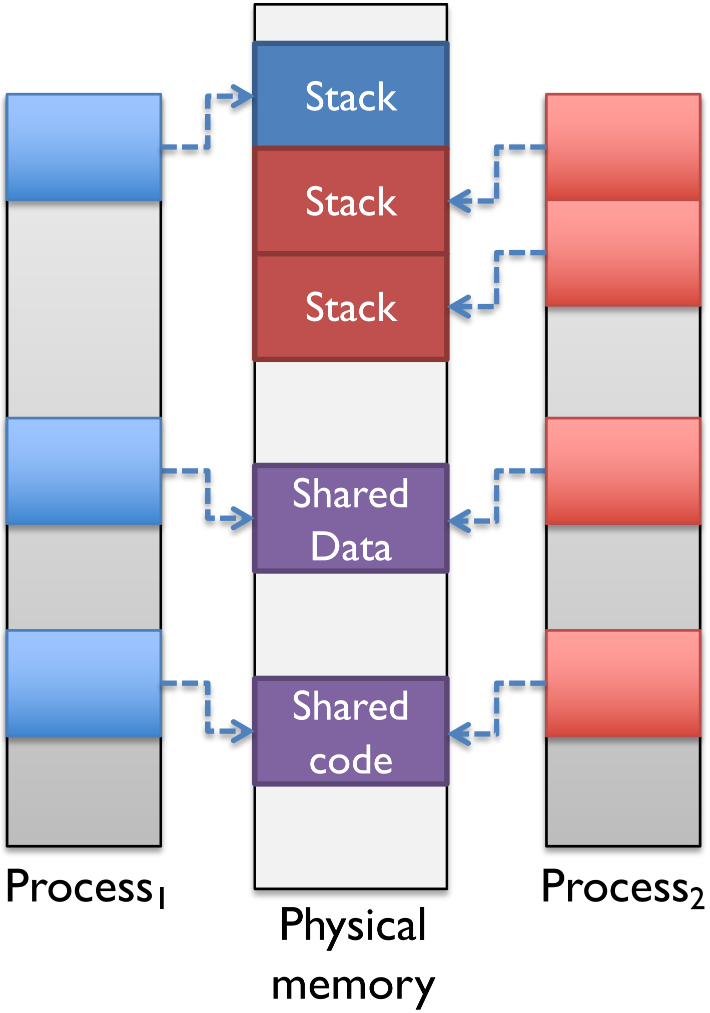
While:

- Retaining current programming languages and models
- Supporting incremental deployment in software stack



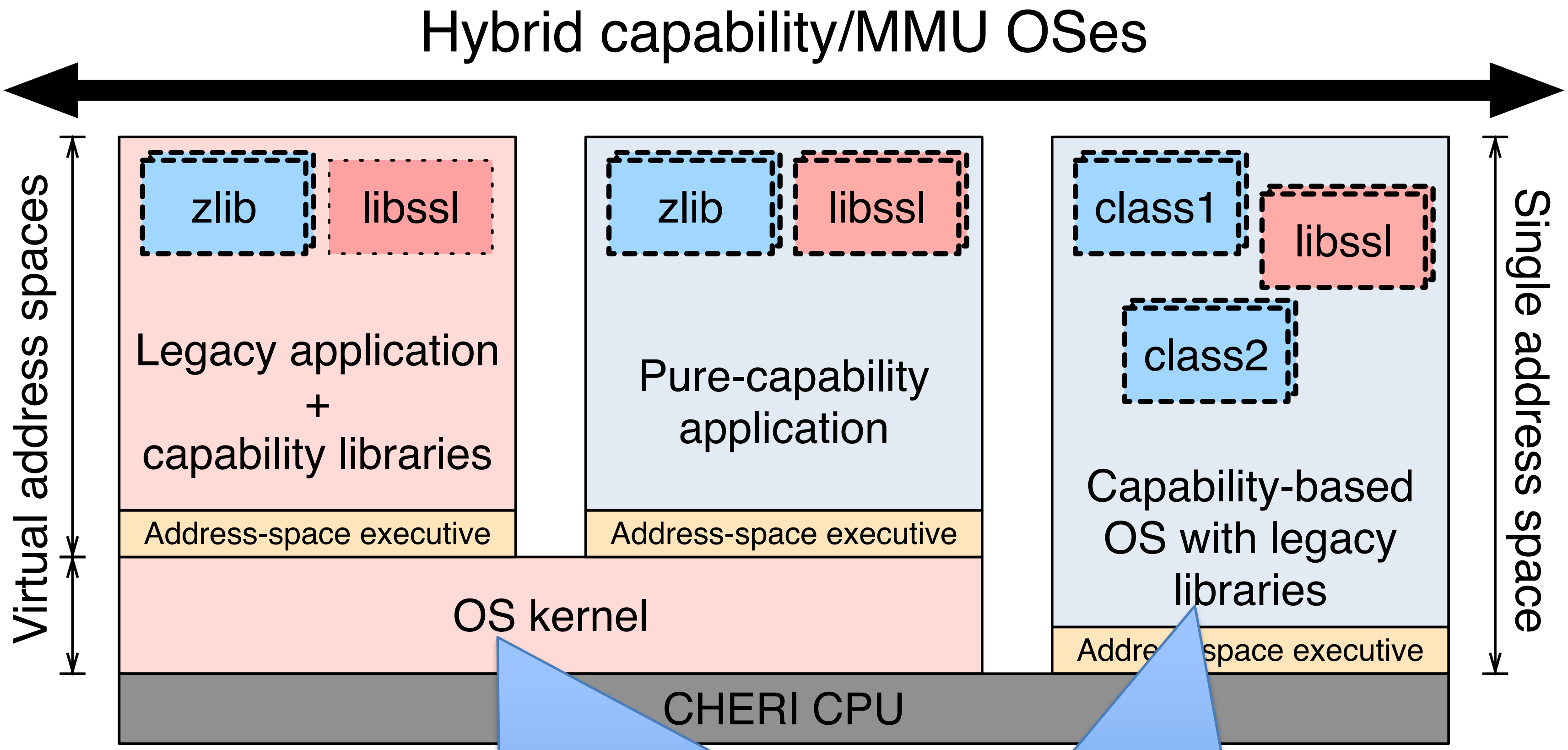
Virtual memory

Reminder: MMU process model



- Coarse-grained process isolation
  - Inter-program robustness
  - Bridged by kernel services (e.g., IPC); OS access control limits global rights
- Memory Management Unit (MMU)
  - **Page tables** control per-process **virtual-to-physical mappings**
- Powerful tool for application isolation
- **Inefficient** and **hard-to-program** for compartmentalization
  - Process-model costs: page tables, dynamic linkage, globals, heaps, …
  - High explicit domain-switch costs
  - Implied overhead growth on page table, cache, TLB as sharing grows

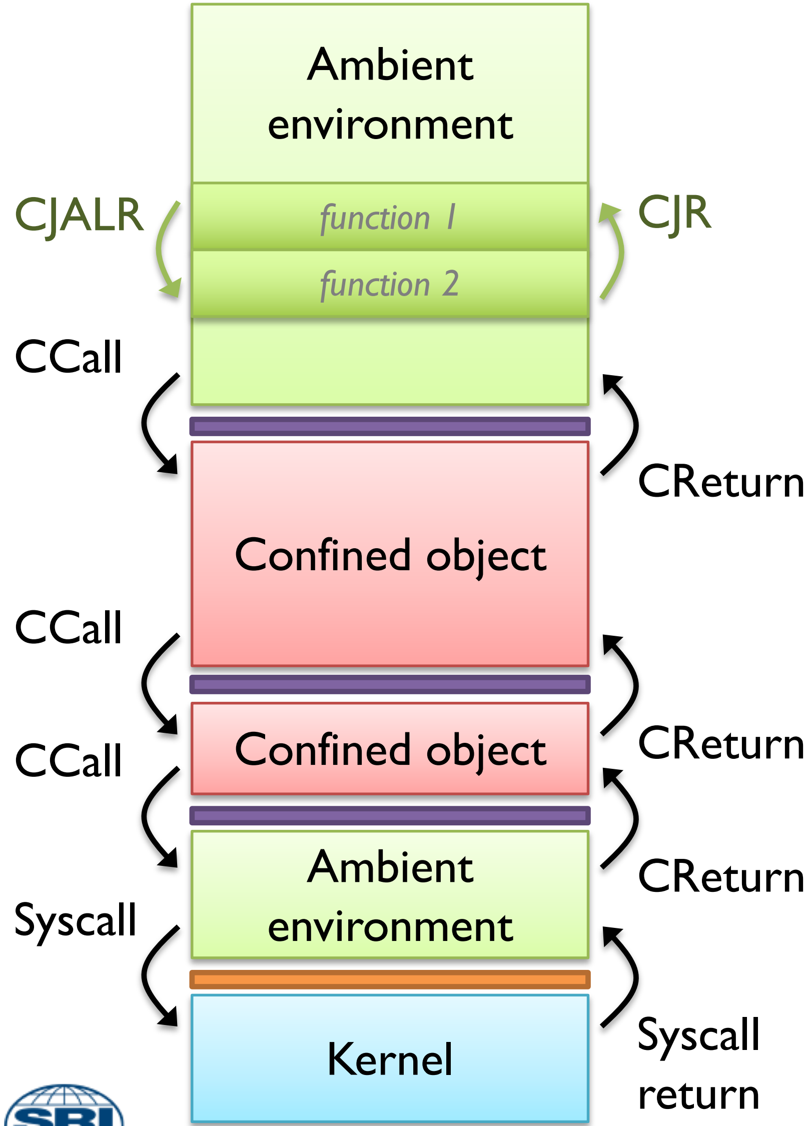
Software deployment models



Hybrid MMU-capability models: protection and compartmentalization within virtual address spaces

Single-address-space systems are possible but still experimental

Object-capability invocation



- **Mutual trust**: robust function calls
  - CHERI-aware CJALR and CJR instructions
  - Destination + return address are capabilities
  - Shared stack, globals, …
- **Mutual distrust**: Object-capability invocation
  - CCall/CReturn instructions w/exceptions
  - Independent stacks, globals, …
- Per-thread **trusted stack** links object stacks
  - Reliable call-return semantics
  - Reliable recovery on uncaught exception
- Classes permissions control system calls
  - Similar to Java JNI: "system classes"