CTSRD

CRASH-WORTHY TRUSTWORTHY SYSTEMS RESEARCH AND DEVELOPMENT

CHERI: A Hybrid Capability Architecture

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, Lucian Paul-Trifu, Alex Richardson, Michael Roe, Colin Rothwell, Hassen Saidi, Peter Sewell, Stacey Son, Domagoj Stolfa, Andrew Turner, Munraj Vadera, Jonathan Woodruff, Hongyan Xia, and Bjoern A. Zeeb

University of Cambridge, SRI International

MIT CSAIL - 9 November 2017

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.

SRI International

UNIVERSITY OF CAMBRIDGE

DARPA – CRASH

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

…what would you change?

Principle of least privilege

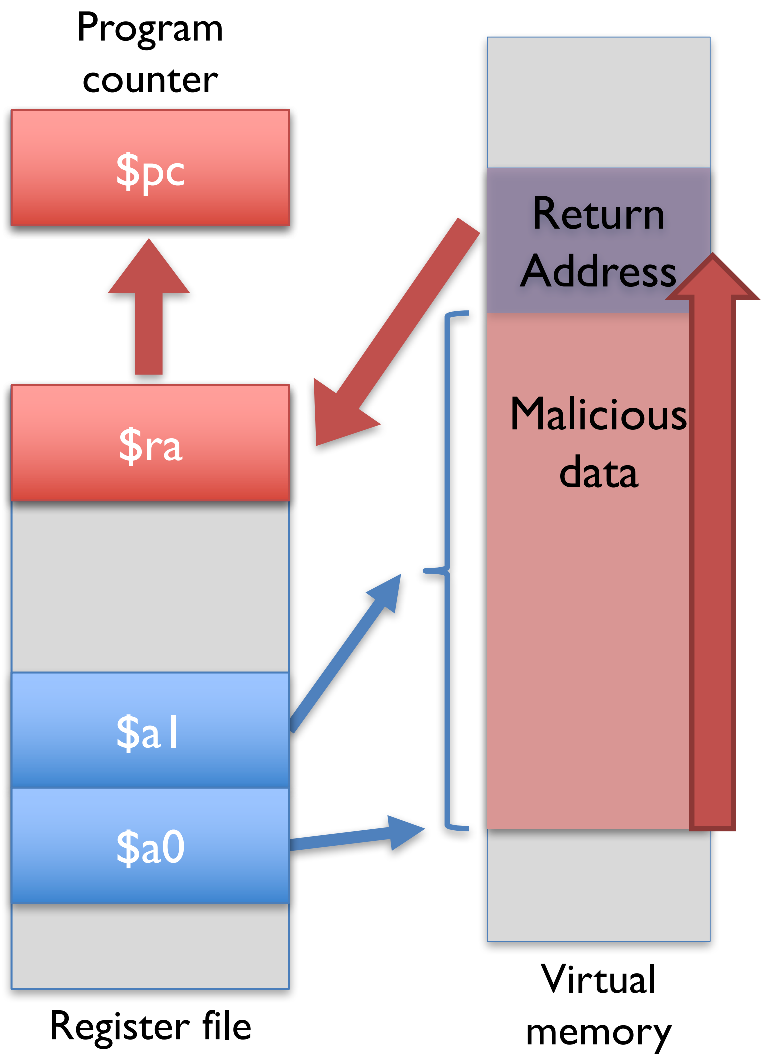
Every program and every privileged user of the system should operate using the **least amount of privilege necessary** to complete the job.

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

(Lack of) architectural least privilege

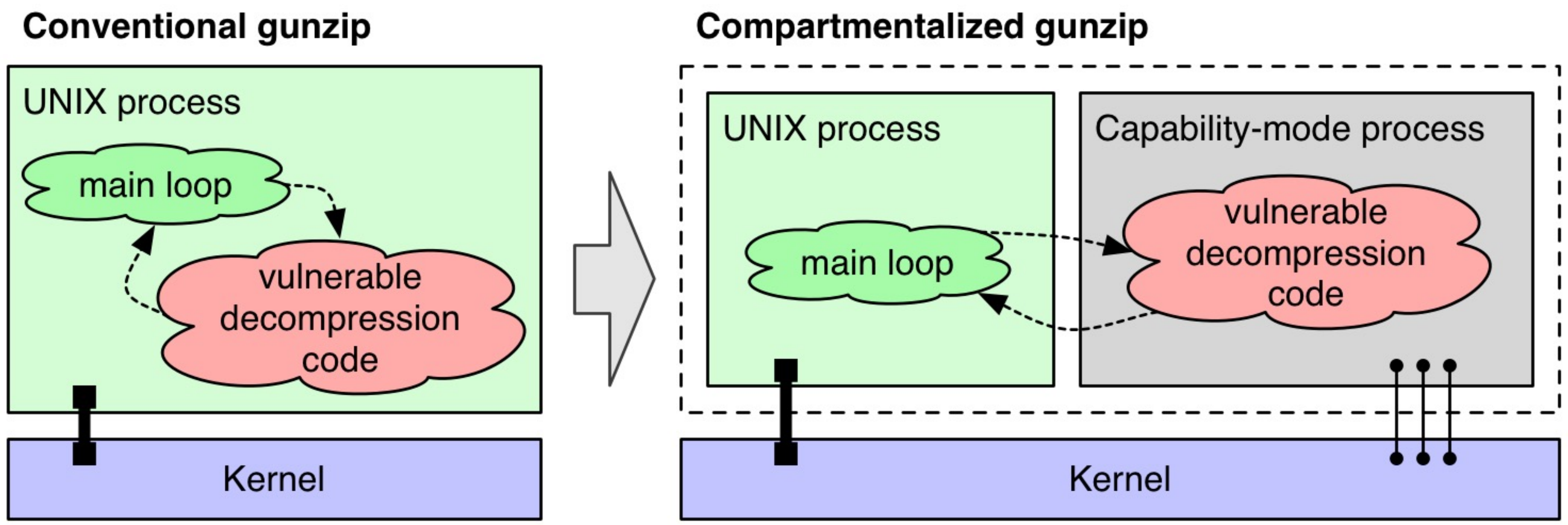
- Classical buffer-overflow + code reuse attack
  1. Buggy code overruns buffer, overwrites return address
  2. Overwritten return address is loaded and jumped to
- These privileges were not required by the C language; why allow code the ability to:
  - Write outside the target buffer?
  - Corrupt or inject a code pointer?
  - Execute data as code / re-use code?
- Limiting privilege doesn't fix bugs – but does provide **vulnerability mitigation**

Memory Management Units (MMUs) do not enable efficient granular privilege minimization



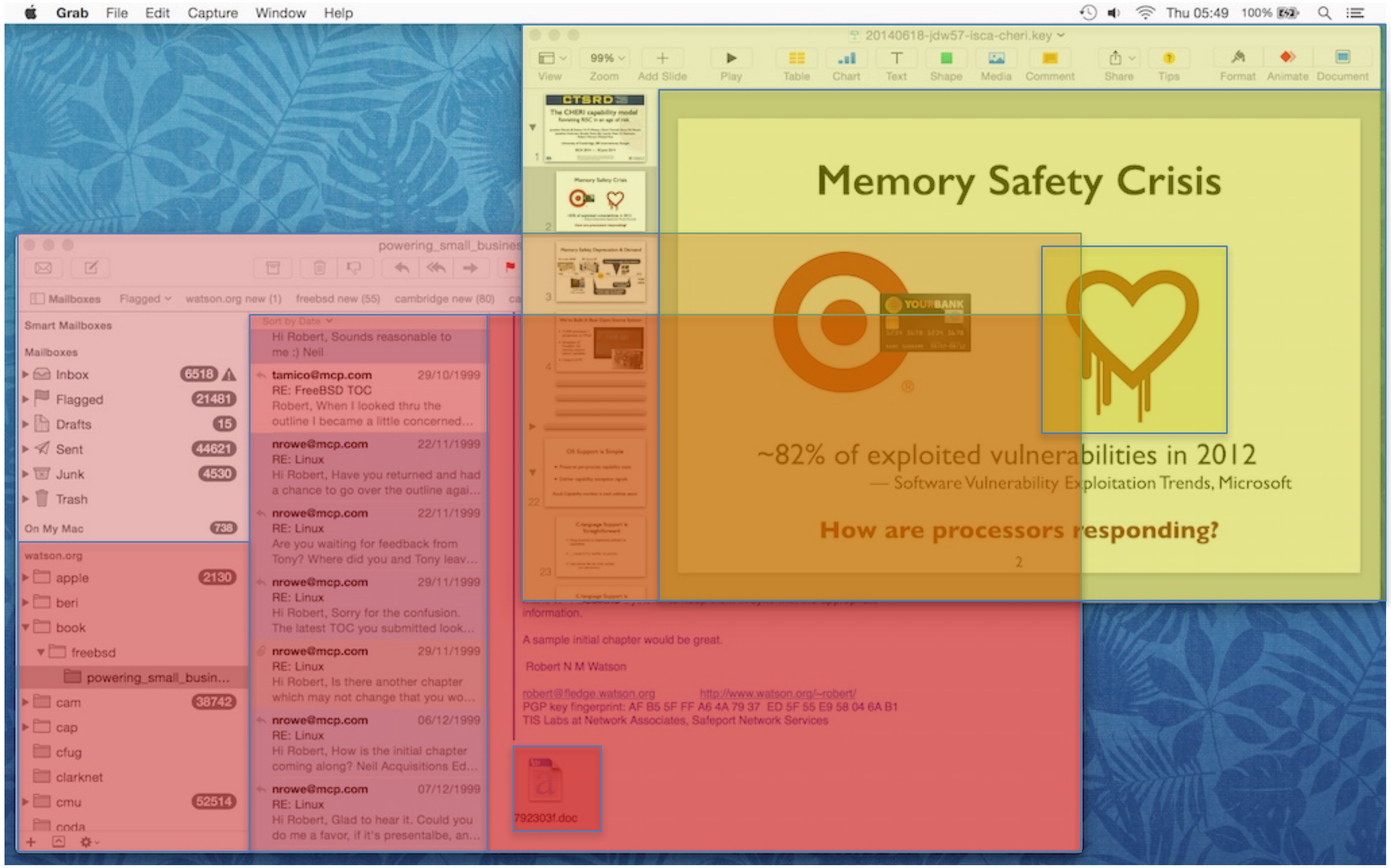
Application-level least privilege (1)

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 and exploits**

Application-level least privilege (2)


Grab File Edit Capture Window Help
Thu 05:49 100%
20140618-jdw57-isca-cheri.key
View Zoom Add Slide Play Table Chart Text Shape Media Comment Share Tips Format Animate Document
powering_small_busines
Mailboxes Flagged watson.org new (1) freebsd new (55) cambridge new (80)
Sort by Date
792303f.doc

Smart Mailboxes

Mailboxes

Inbox	6518
Flagged	21481
Drafts	15
Sent	44621
Junk	4530
Trash	
On My Mac	738
watson.org	
apple	2130
beri	
book	
freebsd	
powering_small_busin...	
cam	38742
cap	
cfug	
clarknet	
cmu	52514
coda	

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

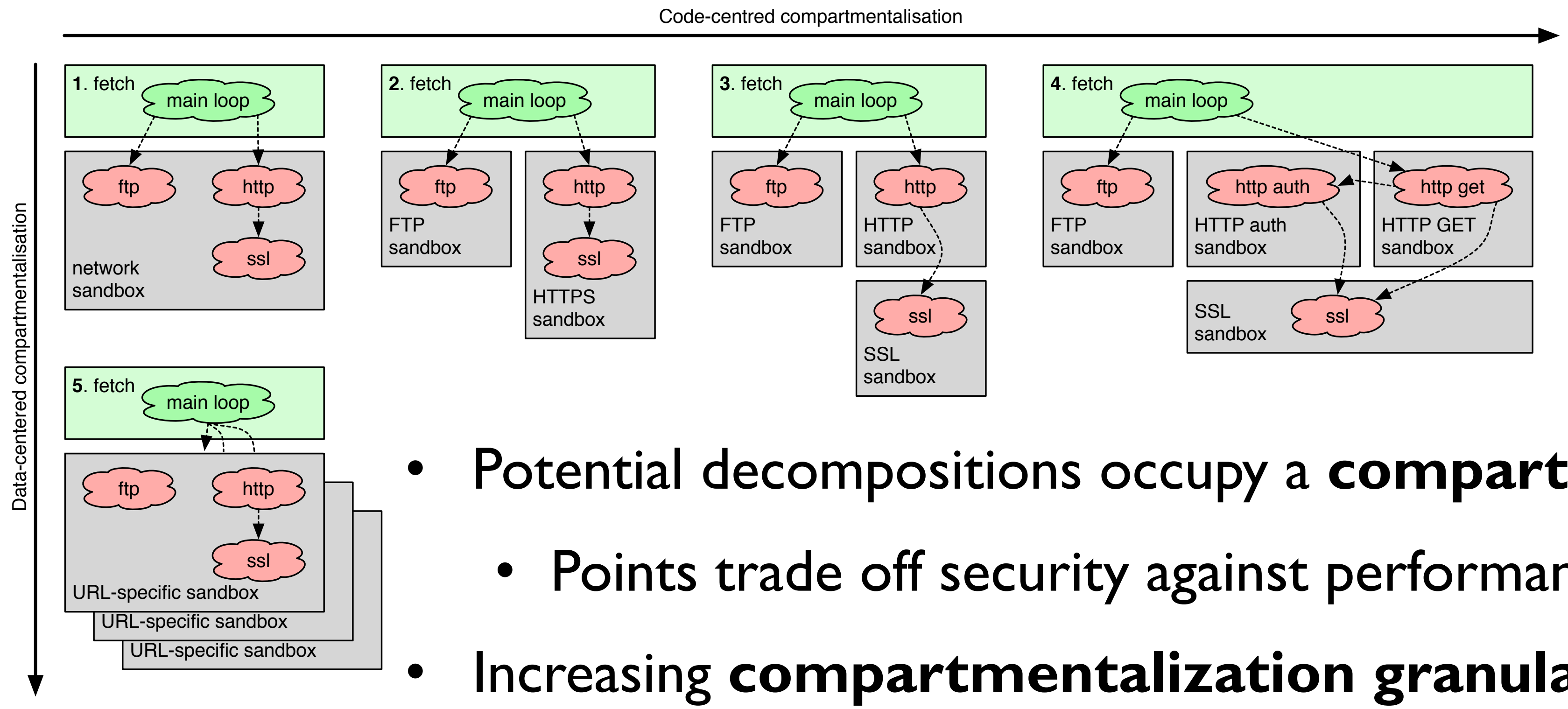
~82% of exploited vulnerabilities in 2012
— Software Vulnerability Exploitation Trends, Microsoft

How are processors responding?

2

SRI International

UNIVERSITY OF CAMBRIDGE



- Potential decompositions occupy a **compartmentalization space**:
  - Points trade off security against performance, program complexity
- Increasing **compartmentalization granularity** better approximates the principle of least privilege …
- … but **MMU-based architectures** do not scale to many 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, language runtimes, browsers, …
- **Granular spatial memory protection, pointer protection**
  - Buffer overflows, control-flow attacks (ROP, JOP), …
- **Foundations for temporal safety**
  - E.g., accurate C-language garbage collection
- **Higher-level language safety**
  - Safe interfaces to native code (e.g., impose Java memory safety on JNI)
  - Efficient memory safety (e.g., hardware assisted bounds checking)
- **Scalable in-process compartmentalization**
  - Facilitate greater use of exploit-independent mitigation techniques

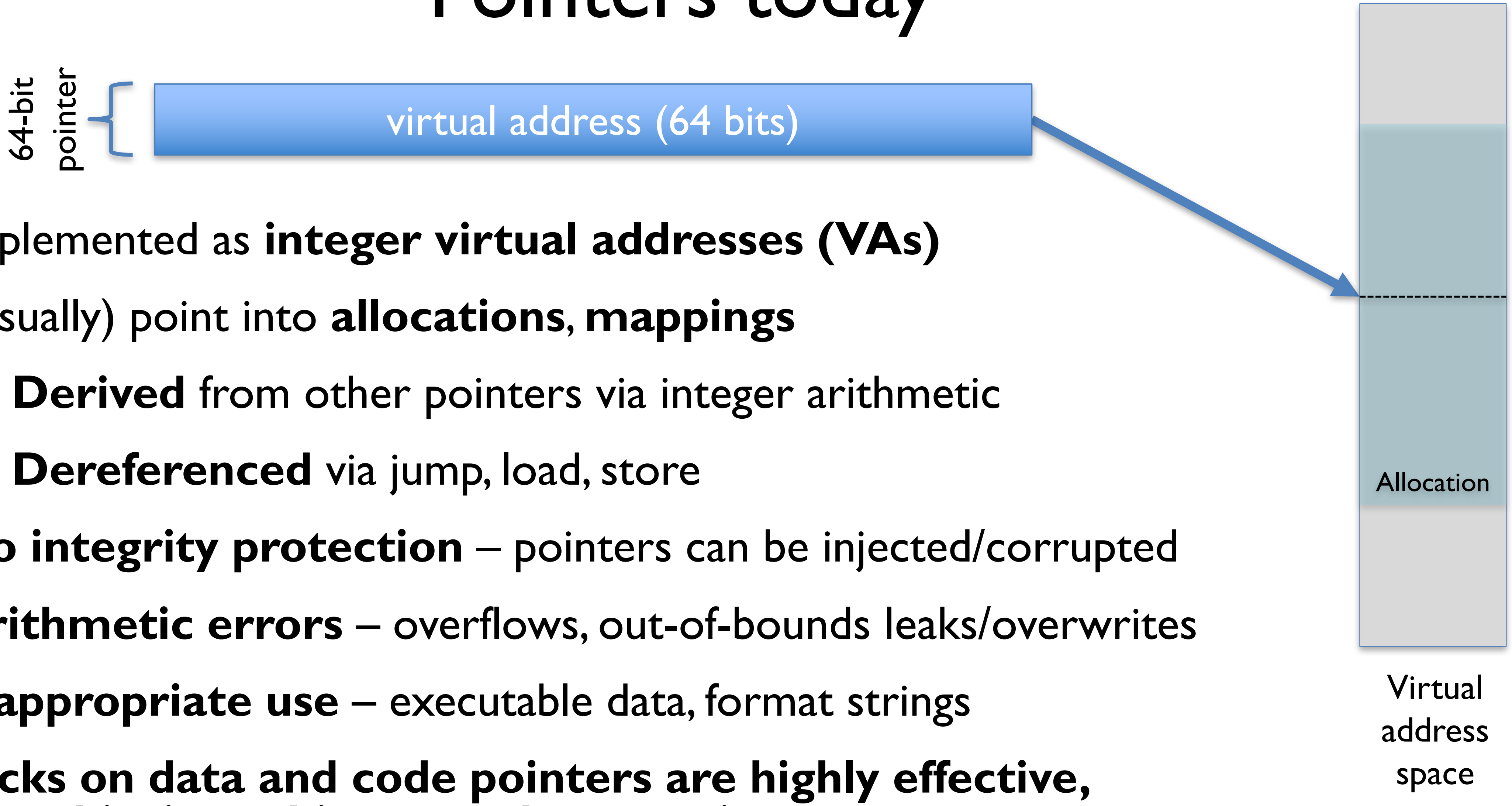
CHERI architectural goals (1)

- **De-conflate virtualization and protection**
  - Memory Management Units (MMUs) protect by **location** in memory
  - CHERI protects **references** to code, data, software objects
  - Add protections to **existing indirection (pointers)** – no new tables
- **Architectural mechanism** enforces **software policy**
  - **Language-based properties**
    (e.g., C/C++ compiler, linkers, OS model, runtime)
  - **New software abstractions**
    (e.g., confined objects for compartmentalization)

CHERI architectural goals (2)

- **Hybrid capability-system model**
  - **Capability systems** target the **principle of least privilege**
  - **Capabilities** are unforgeable, delegable tokens of authority
  - **Hybrid capability systems** compose cleanly w/current designs (RISC ISAs, MMUs, OSes, C-language software)
  - ISA design also utilizes **principle of intentional use**: Avoid implied privilege selection where possible (unlike an MMU)
- Performance goals:
  - **Low overhead** for **pointer protection and fine-grained memory protection** (goal: <2%)
  - **Significant performance gain** for **compartmentalization** (goal: >>1 order of magnitude)

Pointers today



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

Attacks on data and code pointers are highly effective, often achieving arbitrary code execution

CHERI protection model

- **RISC hybrid-capability architecture** supporting fine-grained, **pointer-based memory protection**:

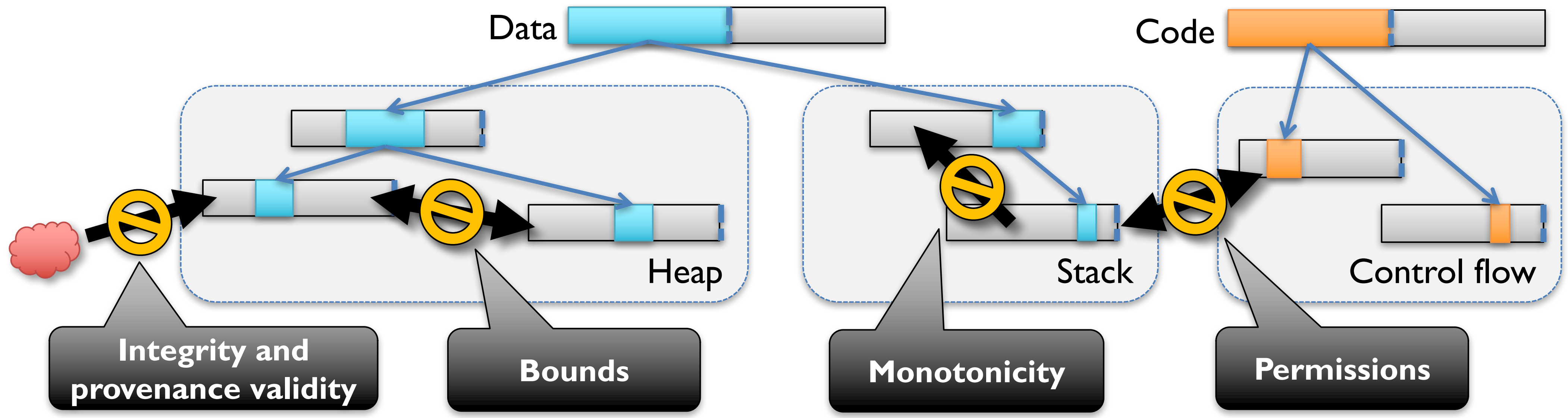
Protect **pointer**

  - **pointer integrity** (e.g., no pointer corruption)
  - **pointer provenance validity** (e.g., no pointer injection)

Protect **pointee**

  - **bounds checking** (e.g., no buffer overflows)
  - **permission checking** (e.g., W^X for pointers)
  - **monotonicity** (e.g., no privilege escalation / improper re-use)
  - **encapsulation** (e.g., protect software objects)

CHERI enforces protection semantics for pointers



➡ **Provence** and **monotonicity** control whether pointers can be **dereferenced**

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

➡ **Bounds** and **permissions** control how pointers are used, and can be **minimized**

- E.g., Pointers cannot be manipulated to access the wrong heap or stack object

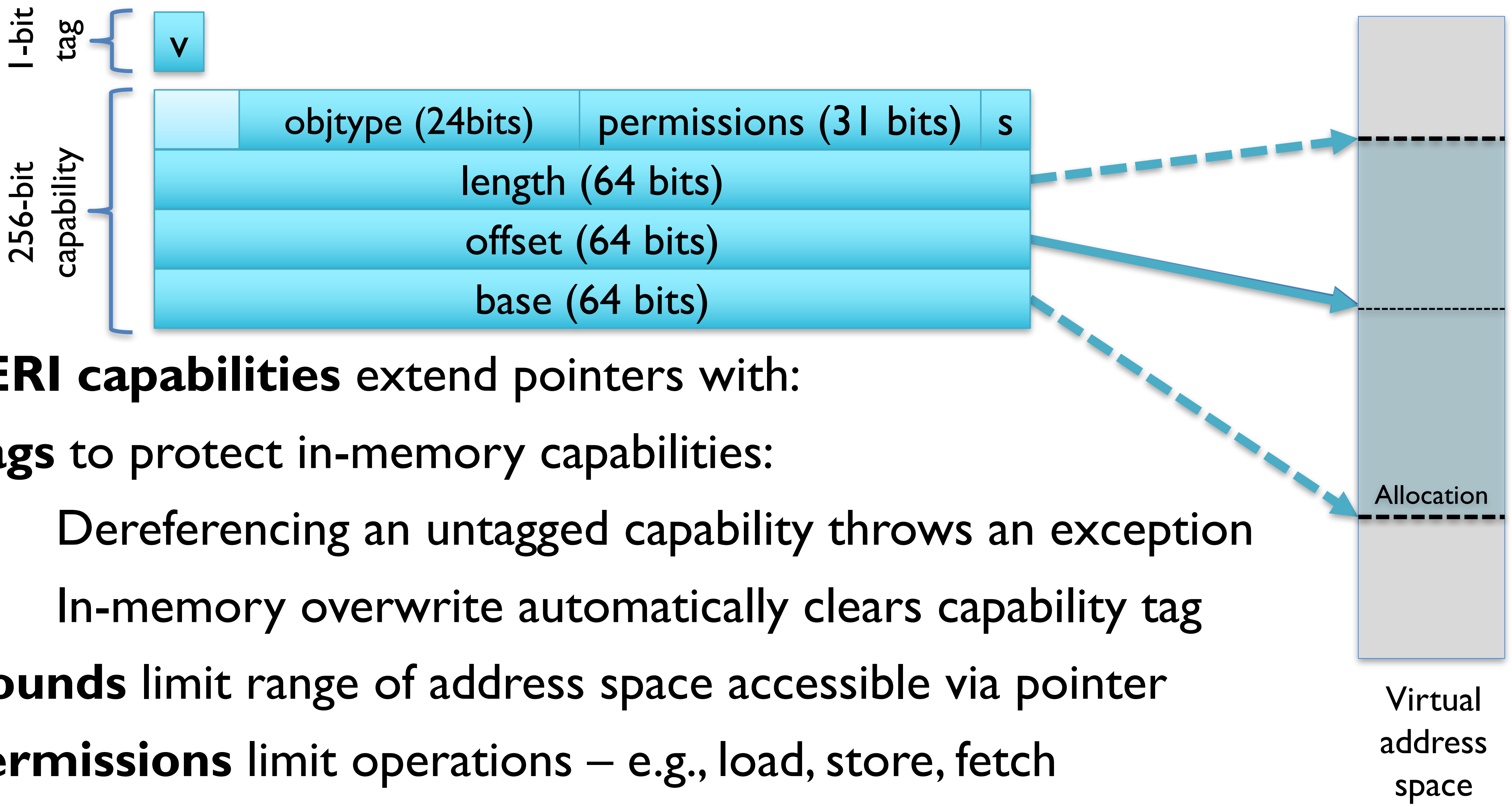
➡ Foundations for **software memory protection and compartmentalization**

CHERI-MIPS INSTRUCTION-SET ARCHITECTURE (ISA)

CHERI architectural approach

- **RISC ISA extensions** that avoid new microcode, table lookups, exceptions:
  - **MMUs** control the **implementation** of virtual addresses
  - **CHERI** protects **references** to virtual addresses
- **Pointers** can be implemented via **architectural capabilities**
  - **Capabilities: unforgeable, delegable tokens of authority**
  - **Tagged memory** protects **integrity**, **provenance** of capabilities in DRAM
  - **Metadata**, including **bounds** and **permissions**, limit capability use
  - **Capability monotonicity** is implemented via **guarded manipulation**
  - **Sealing** provides **immutable, software-defined capabilities**
  - **Exceptions, userspace CCall** implement **controlled non-monotonicity**
- **256-bit architectural model**, but **efficient 128-bit implementation**

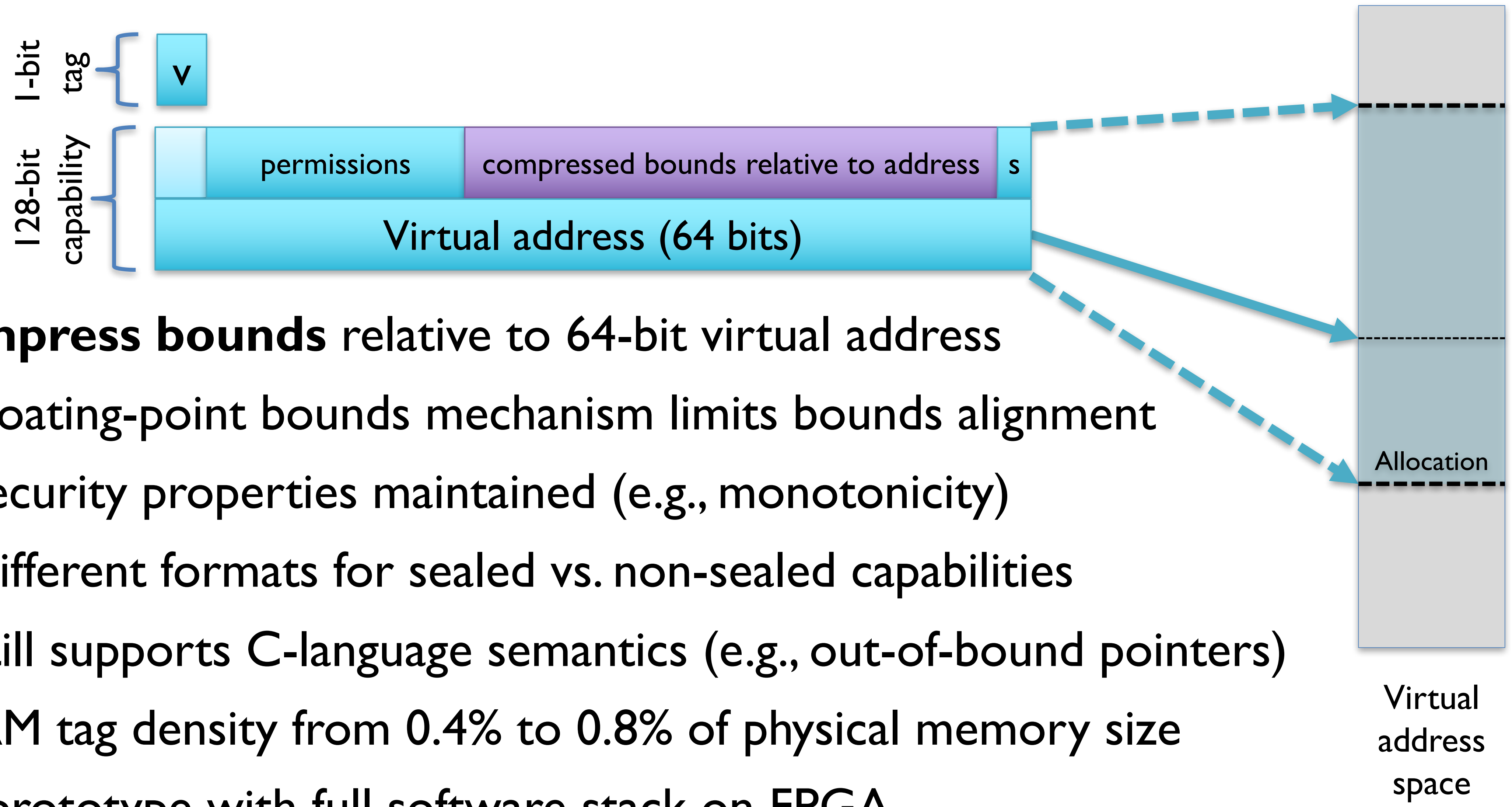
256-bit architectural capabilities



CHERI capabilities extend pointers with:

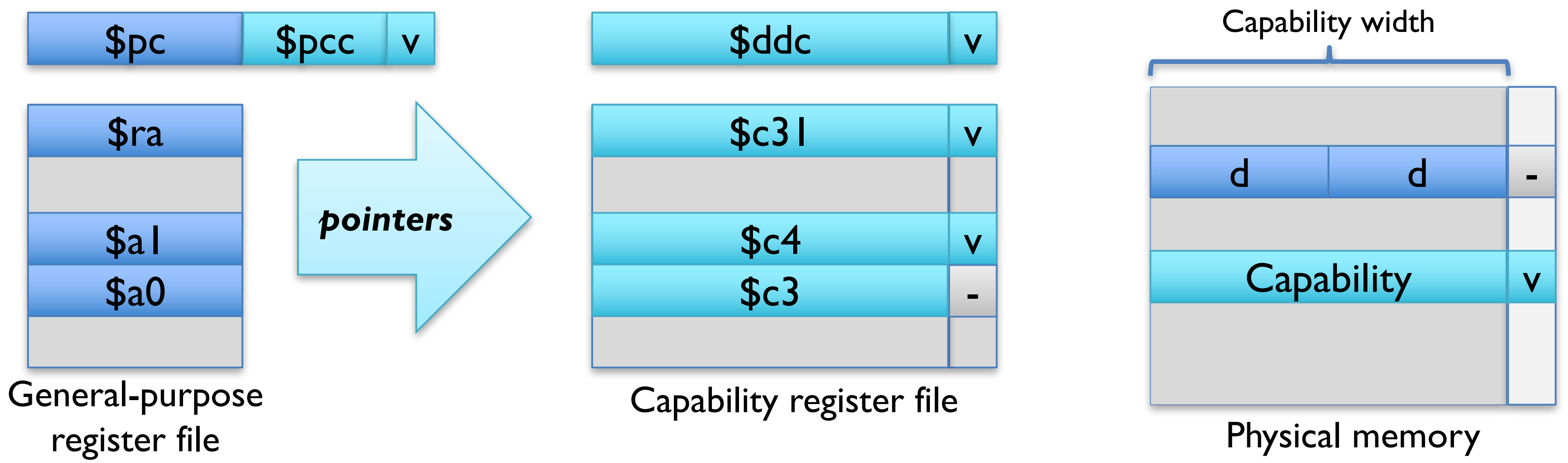
- **Tags** to protect in-memory capabilities:
  - Dereferencing an untagged capability throws an exception
  - In-memory overwrite automatically clears capability tag
- **Bounds** limit range of address space accessible via pointer
- **Permissions** limit operations – e.g., load, store, fetch
- **Sealing**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

128-bit micro-architectural capabilities



- **Compress bounds** relative to 64-bit virtual address
  - Floating-point bounds mechanism limits bounds alignment
  - Security properties maintained (e.g., monotonicity)
  - Different formats for sealed vs. non-sealed capabilities
  - Still supports C-language semantics (e.g., out-of-bound pointers)
- DRAM tag density from 0.4% to 0.8% of physical 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
- Other concrete ISA instantiations are possible: e.g., **merged register files**

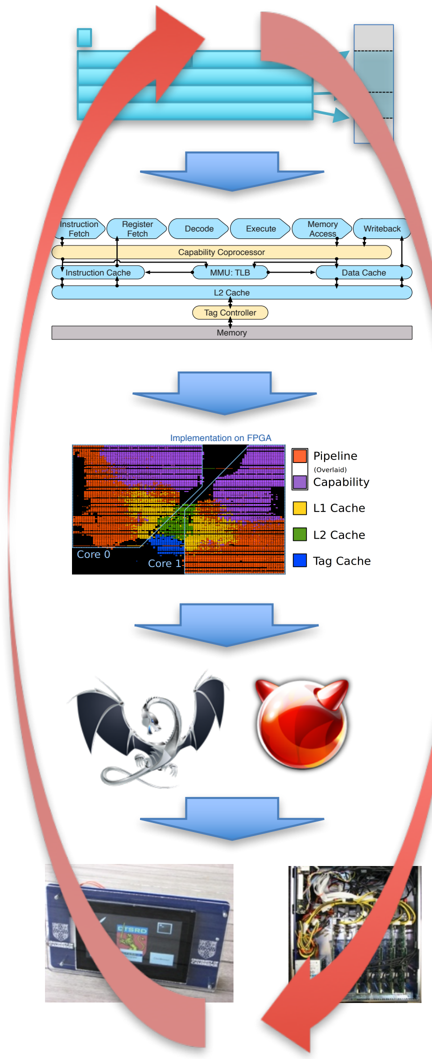
Virtual memory **and** capabilities

	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	In-address-space function calls or message passing
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 over 7 years



- Abstract **CHERI protection model** protects OS, C, linker, app.
- **CHERI-MIPS ISA** extends the 64-bit MIPS ISA
  - Human-readable CHERI ISA specification (tech report)
  - L3 + Sail MIPS + CHERI ISA formal models
  - Qemu-CHERI fast ISA emulator
- Bluespec SystemVerilog (BSV) pipelined, multicore **CHERI-MIPS CPU processor** – simple but realistic
  - C → Cycle-accurate software simulator
  - Verilog → FPGA @100MHz
- **CHERI software corpus**: FreeBSD, Clang/LLVM, applications: OpenSSH, PostgreSQL, nginx, …
- **Evaluation**: Performance, security, compatibility…

CHERI R&D Timeline

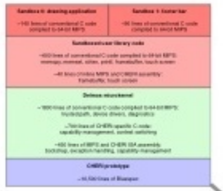
2010 · 2011 · 2012 · 2013 · 2014 · 2015 · 2016 · 2017

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

Jun. 2012: CheriBSD capability context switching

Jul. 2012: LLVM generates CHERI code

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

RESoLVE 2012: Hybrid MMU/ capability model

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

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

PLDI 2016: CHERI C-language formal semantics

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 Architectural portability: CHERI-x86 and CHERI-RISC-V sketches Exception-free domain transition

RISC + MMU + capabilities

Compartmentalization

In-kernel use; non-MIPS ISAs

C + capabilities

128-bit, code efficiency

CHERI SOFTWARE

What are CHERI's implications for software?

- Efficient fine-grained **architectural memory protection** enforces:

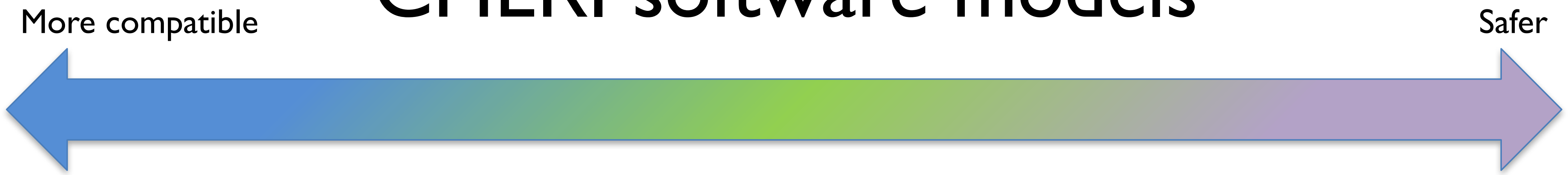
Provenance validity:	Where do pointers come from?
Integrity:	How do pointers get where they are going?
Bounds, permissions:	What rights should pointers carry?
Monotonicity:	Can real software play by these rules?

- Scalable fine-grained **software compartmentalization**

  Can we construct **isolation** and **controlled communication** using integrity, provenance, bounds, permissions, and monotonicity?

  Can **sealed capabilities**, **controlled non-monotonicity**, and **capability-based sharing** enable safe, efficient domain transition?

CHERI software models



Unmodified
All pointers are integers

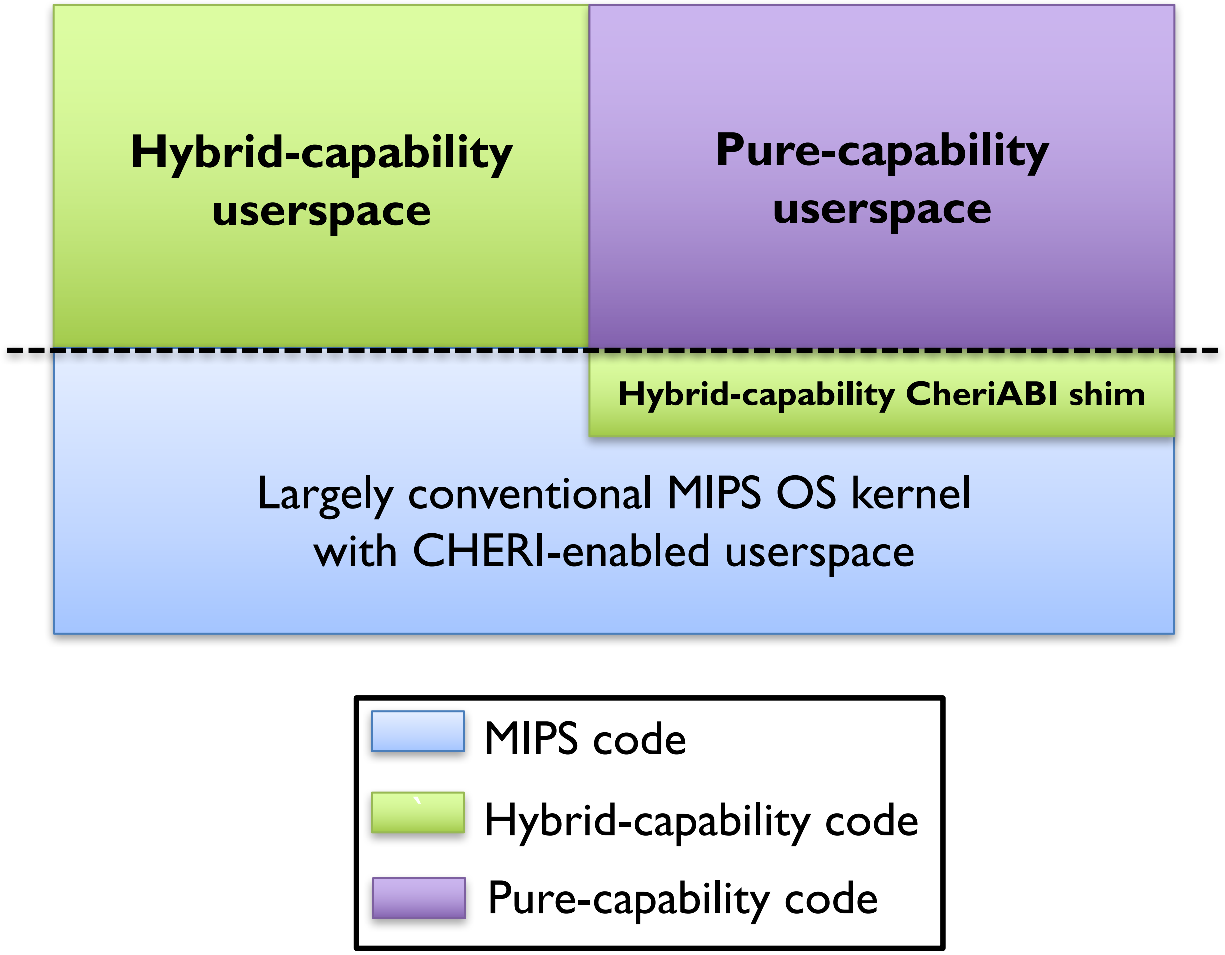
Hybrid
Annotated and automatically selected pointers are capabilities

Pure-capability
All pointers are capabilities

- **Source and binary compatibility** – multiple C-language, code-generation models:
  - **Unmodified code**: Existing n64 code runs without modification
  - **Hybrid code**: E.g., capabilities used in return addresses, annotated data/code pointers, specific types, etc. (MIPS n64-interoperable)

    … But "hybrid" is a spectrum between manual and automatic use
  - **Pure-capability code**: Ubiquitous data- and data-pointer protection. (Non-MIPS-n64-interoperable due to changed pointer size) – also a spectrum of choices
- **CHERI Clang/LLVM compiler prototype generates code for all**

From hybrid-capability code to pure-capability code


Hybrid-capability userspace
Pure-capability userspace
Hybrid-capability CheriABI shim
Largely conventional MIPS OS kernel with CHERI-enabled userspace
MIPS code
Hybrid-capability code
Pure-capability code

- **n64 MIPS ABI:** hybrid-capability code
  - Early investigation – manual annotation and C semantics
  - Many pointers are integers (including syscall arguments, most implied VAs)
- **CheriABI:** pure-capability code
  - The last two years – fully automatic use of capabilities wherever possible
  - All pointers, implied virtual addresses are capabilities (inc. syscall arguments)

CheriABI: A full pure-capability OS userspace

- Complete memory- and pointer-safe FreeBSD C/C++ userspace
  - **System libraries**: crt/csu, libc, zlib, libxml, libssl, …
  - **System tools and daemons**: echo, sh, ls, openssl, ssh, sshd, …
  - **Applications**: PostgreSQL, nginx; bringing up WebKit (C++)
- **Valid provenance**, **minimized privilege** for **all pointers, implied VAs**
  - Userspace capabilities originate in **kernel-provided roots**
  - Kernel, compiler, allocators, linker, … **refine** bounds and permissions
- Trading off **privilege minimization**, **monotonicity**, **API conformance**
  - Typically in memory management – realloc(), mmap() + mprotect()

Evaluating compatibility

Goal: **Little or no software modification** (BSD base system + applications)

	Pointer vs. integer	Pointer size & alignment	Pointer integrity	Function ABI	Unsupported features
BSD libraries	20	3	6	5	2
BSD programs	19	4	5	5	4
PostgreSQL	✓	✓	✓	-	-

BSD: 34 of 824 programs, 28 of 130 libraries modified. ~200 out of ~20,000 userspace C files/headers modified.

Goal: **Software that works** (BSD base + application test suites)

	Pass	Fail	Skip	Total
MIPS	2998	47	168	3213
Hybrid	2992	53	168	3213
CheriABI	2800	75	203	3078

Increase in "skip"s due to our not running with dynamic linking in our test environment currently.
Several memory-safety bugs in tests also found and fixed!

Evaluating protection

- Adversarial / historical analysis
  - ✓ Pointer integrity, provenance validity prevent ROP, JOP
  - ✓ Buffer overflows: Heartbleed (2014), Cloudbleed (2017)
  - ✓ Pointer provenance: Stack Clash (2017)
- Existing test suites – e.g., BOdiagsuite (buffer overflows)

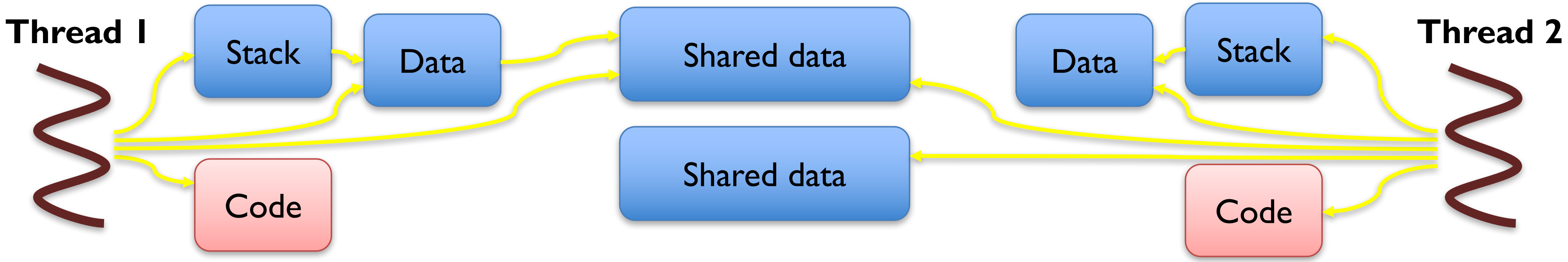
	OK	min	med	large
mips64	0	4	7	171
CheriABI	0	276	287	289
LLVM Address Sanitizer (asan) on x86	0	275	285	286

- Key evaluation concern: reasoning about a **CHERI-aware adversary**

CHERI COMPARTMENTALIZATION

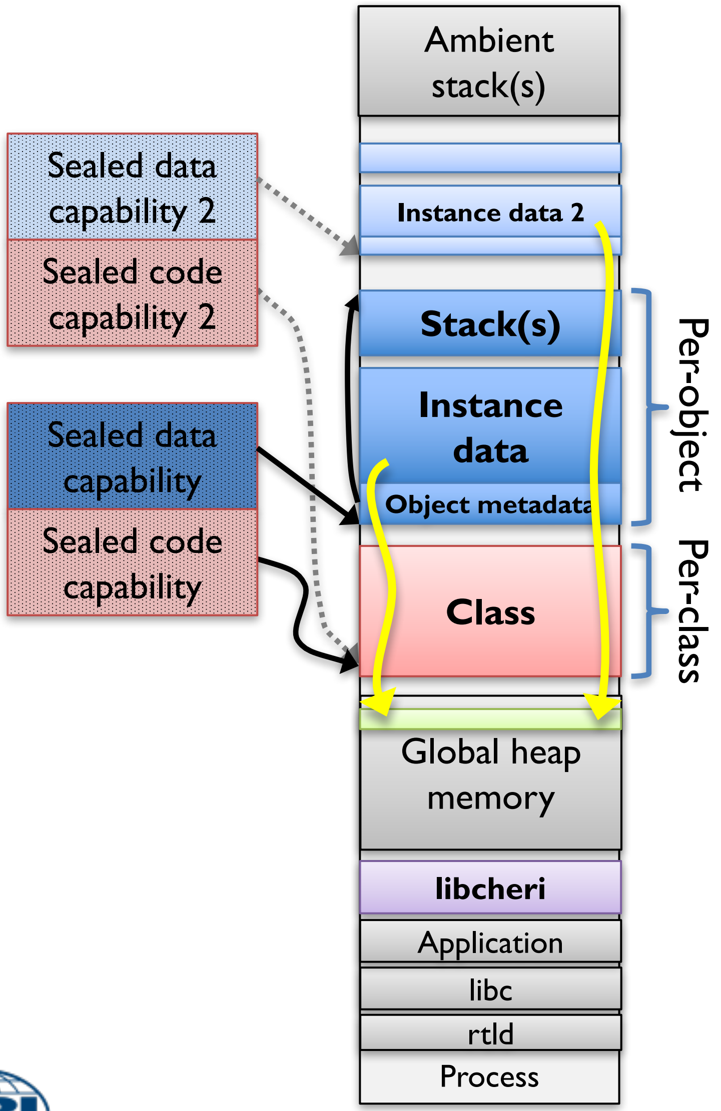
Principles of CHERI compartmentalisation

- A thread's **protection domain** is its **register-file capabilities** and **transitively reachable resources** (i.e., via held capabilities)
- Manipulation of the **capability graph** can implement **isolation**, **controlled communication**, and **domain transition**



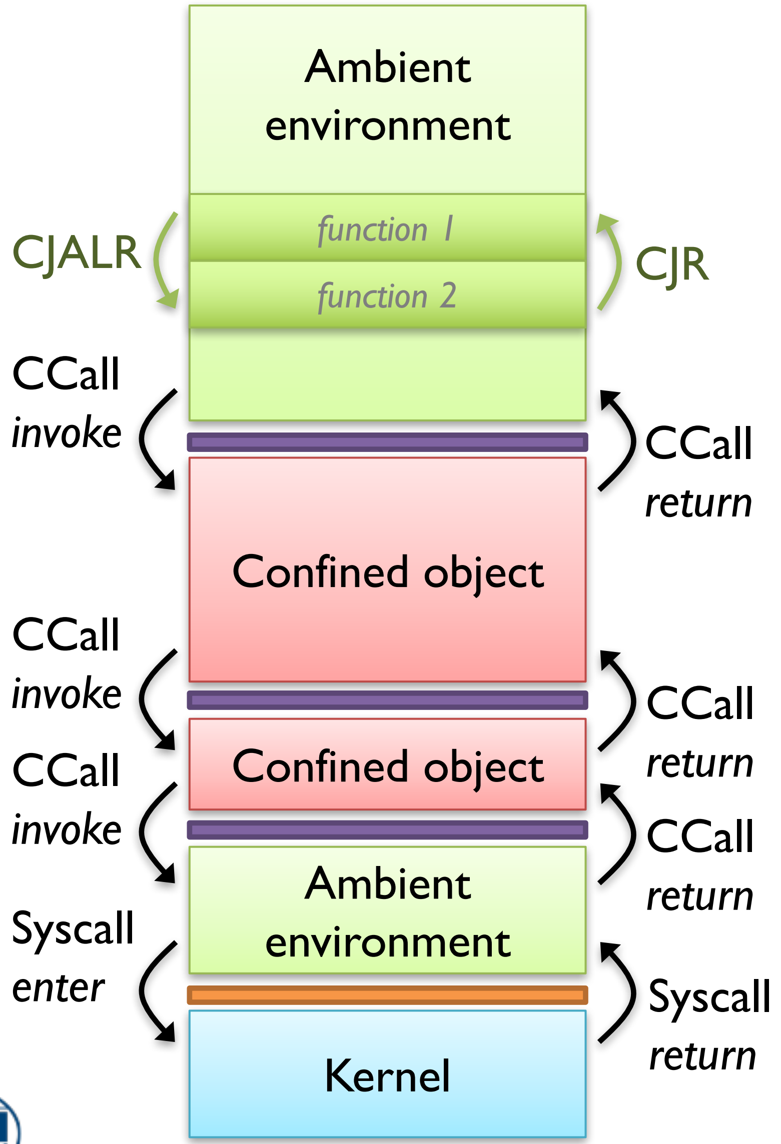
- We can then construct an **object-capability-based security model: classes, objects, shared memory,** and **object invocation**

CheriBSD in-process compartmentalization (sketch)



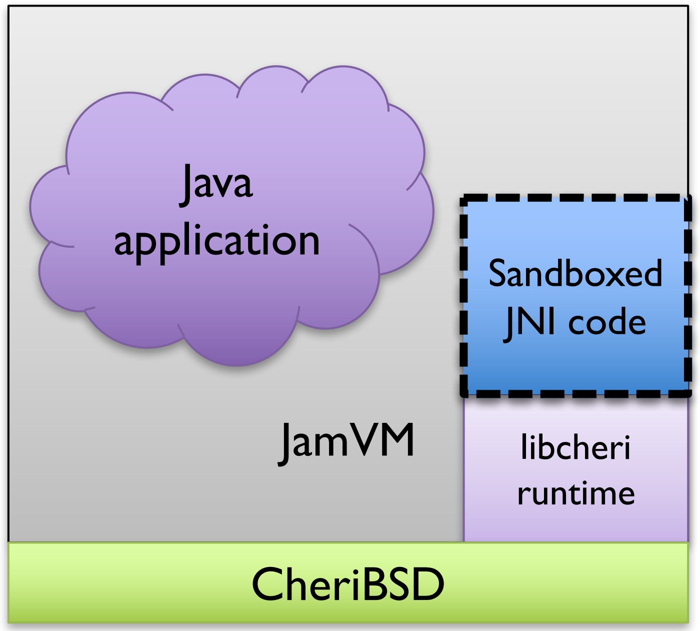
- CheriBSD userspace object-capability model
  - **libcheri** is a capability-based run-time linker
  - libcheri loads, links **classes**, instantiates **objects**
  - **Confined objects**: limited capabilities, no syscalls
- **Fast and robust protection-domain transition**
  - **Sealed capabilities** enforce encapsulation so that references can be safely delegated
  - **Invocation** of a sealed object triggers a **non-monotonic register-file transformation**
- **Efficient object and memory sharing**
  - **Delegate capabilities** across invocation, return

Object-capability invocation



- **Mutual trust** - robust **function calls**
  - CHERI-aware jump, jump-and-link instructions
  - Target, return capabilities protect control flow
  - Shared stack, globals, …
- **Mutual distrust** - object-capability **invocation**
  - Exception-free non-monotonic control transfer
  - Independent stacks, globals, … for encapsulation
- Per-thread **trusted stack** links object stacks
  - Reliable call-return semantics
  - Reliable recovery on uncaught exception
- Classes permissions limit system calls (vis. Java JNI)

CHERI-JNI: Protecting Java from JNI



- **Java Native Interface (JNI)** allows Java programs to use native code for performance, portability, functionality
  - Often fragile; sometimes overtly insecure
- Apply Java **memory-safety and security models** to JNI
  - Limit native-code access to JVM internal state
  - Pointer, spatial memory safety for native code
  - Temporal safety for JNI heap access w/C-language GC
  - Safe copy-free JNI access to Java buffers via capabilities
  - Enforces Java security model on JNI access to Java objects and system services (e.g., files, sockets)
- Prototyped using JamVM on CHERI-MIPS, CheriBSD

WHERE NEXT?

Ongoing research

Quantitative ISA optimization

Compiler optimization

Superscalar microarchitectures

Tag tables vs. native DRAM tags

Toolchain: linker, debugger, …

C++ compilation to CHERI

Grow software corpus

CHERI and ISO C/POSIX APIs

Sandbox frameworks into CHERI

MMU-free CHERI microkernel

Safe native-code interfaces (JNI)

Safe inter-language interoperability

C-language garbage collection

Accelerating managed languages

Formal proofs of ISA properties

Formal proofs of software properties

Verified hardware implementations

Non-volatile memory

Pointer-based security analysis from traces

Microarchitectural optimization opportunities from exposed software semantics

MMU-free HW designs for "IoT"

CHERI papers

ISCA 2014: Fine-grained, in-address-space memory protection hybridizes MMU, capabilities

ASPLOS 2015: Explore + refine C-language compatibility – capabilities + fat pointers

Oakland 2015: Efficient, capability-based compartmentalization in processes

ACM CCS 2015: Compartmentalization modeling using static analysis

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

IEEE Micro Journal Sep/Oct 2016: Hardware-assisted efficient domain switching

ASPLOS 2017: CHERI reinforcement for Java JNI

MIT Press book chapter 2017: Balancing disruption and deployability in CHERI

ICCD 2017: Efficient tagged memory through tag tables and caches

CHERI technical reports

Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (CHERI ISAv6)

- UCAM-CL-TR-907 – April/June 2017
- Kernel-mode compartmentalization, exception-free domain transition, architecture-abstracted efficient tag restoration, CHERI x86_64 and RISC-V sketches, explanation and rationale improvements

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

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
  - Strong, real-world C-language pointer and memory protection with low overhead
  - Scalable, fine-grained intra-process compartmentalization
- Substantial vulnerability-mitigation benefit validated against large, real-world software
- ISCA 2014, ASPLOS 2015, IEEE SSP 2015, ACM CCS 2015, PLDI 2016, IEEE Micro 2016; ASPLOS 2017, ICCD 2017, …
- Watson, et al. **Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6)**, UCAM-CL-TR-907, April 2017
- Open-source architecture, hardware, and software; specifications and prototypes

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

Q&A