

CHERI: Architectural Support for Memory Protection and Compartmentalization

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

Hesham Almatary, Jonathan Anderson, John Baldwin, Hadrien Barrel, Ruslan Bukin, David Chisnall, Nirav Dave, Brooks Davis, Lawrence Esswood, Nathaniel W. Filardo, 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 Stolf, Andrew Turner, Munraj Vadera, Jonathan Woodruff, Hongyan Xia, and Bjoern A. Zeeb

University of Cambridge and SRI International

2 April 2019

Introduction

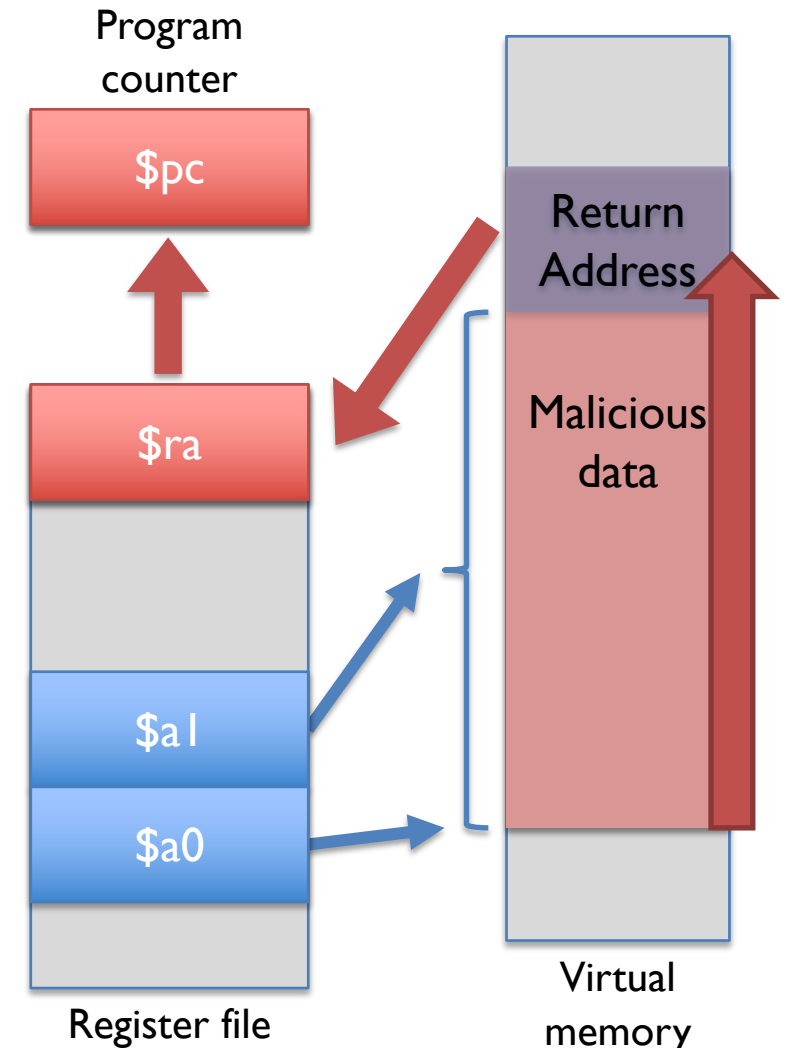
- A little about the CHERI architecture
- Software implications of architectural memory protection at scale
 - (Fine-grained software compartmentalization is another talk)
- To learn more about the CHERI architecture and prototypes:

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

- Watson, et al. **Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6)**, Technical Report UCAM-CL-TR-907, Computer Laboratory, April 2017.
- Davis, et al. **CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment**, ASPLOS 2019.
- Also of interest: Watson, et al. **Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks**, Technical Report UCAM-CL-TR-916, Computer Laboratory, February 2018.

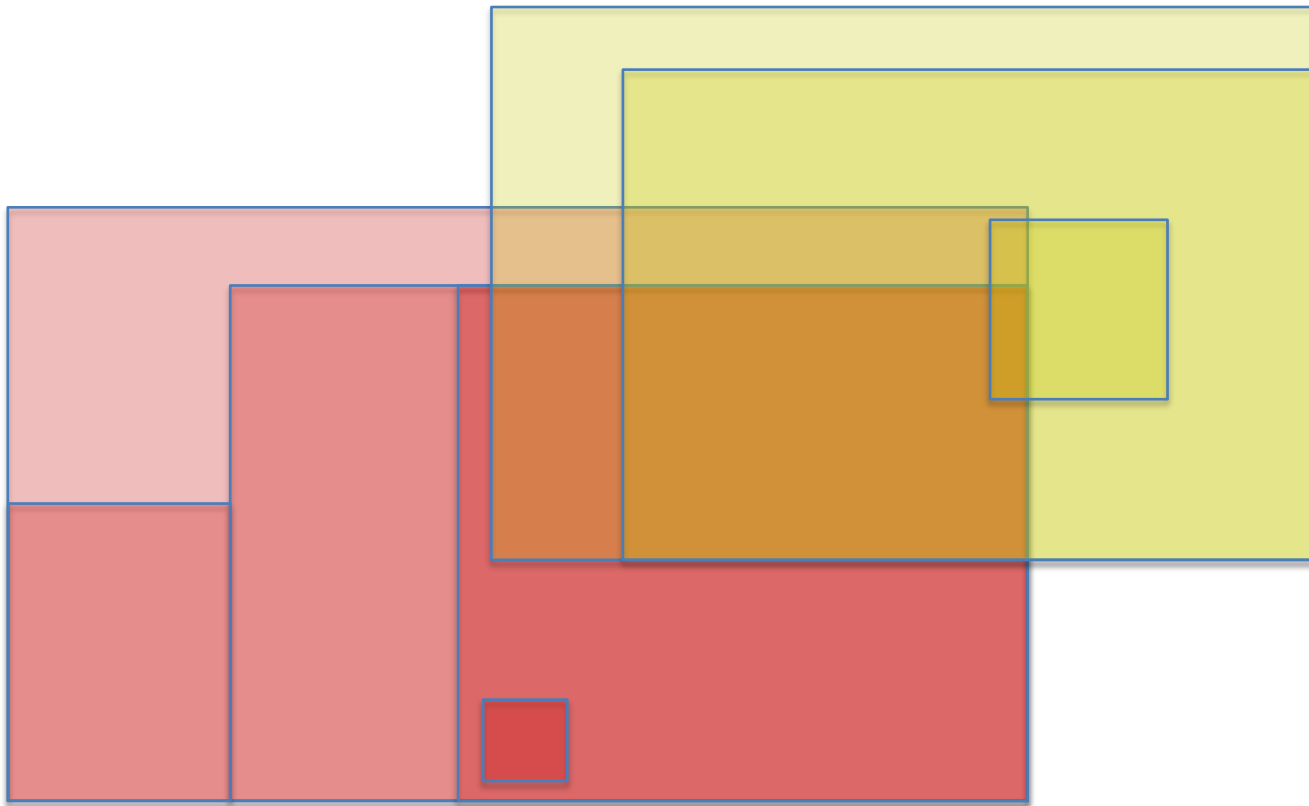
(Lack of) architectural least privilege

- Classical buffer-overflow attack
 1. Buggy code overruns a buffer, overwrites return address with attacker-provided value
 2. Overwritten return address is loaded and jumped to, allowing the attacker to manipulate control flow
- 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, fine-grained privilege reduction**



Application-level least privilege

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

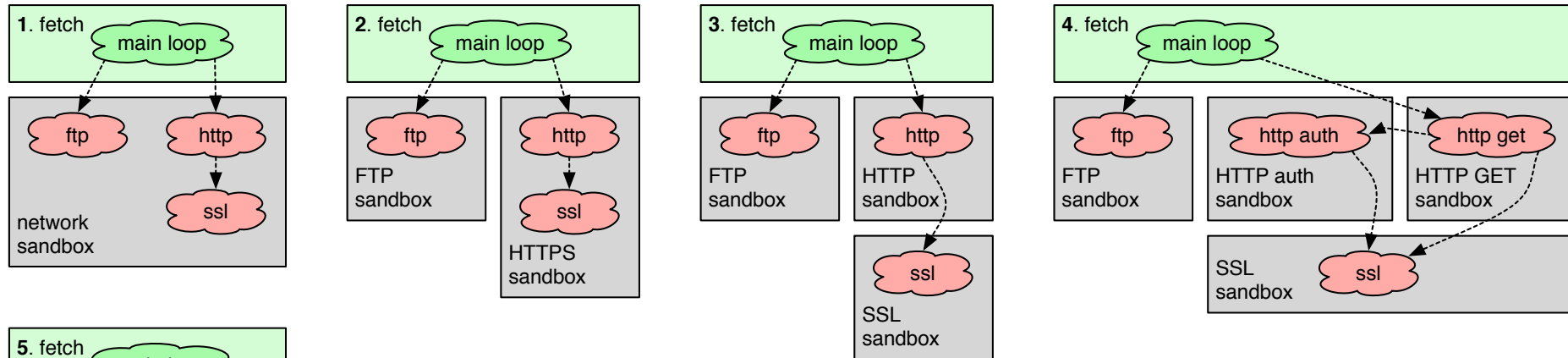


Potential compartmentalization boundaries matching reasonable user expectations for **least privilege** can be found in many user-facing apps.

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**

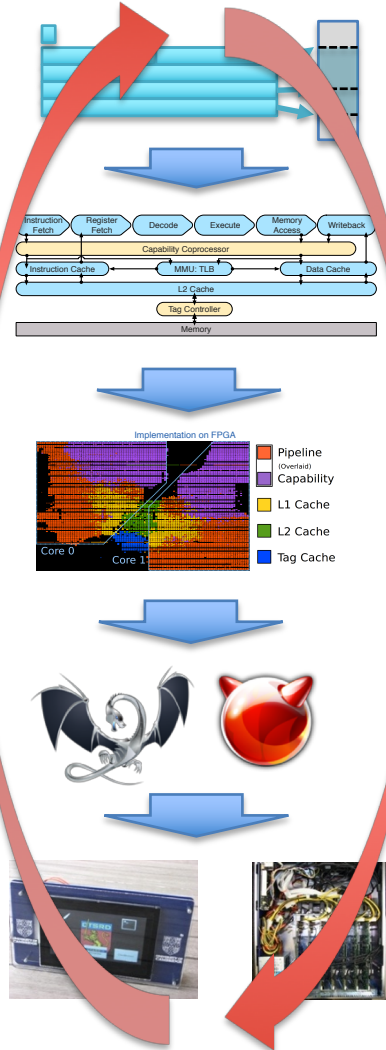
Code-centred compartmentalisation



- 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

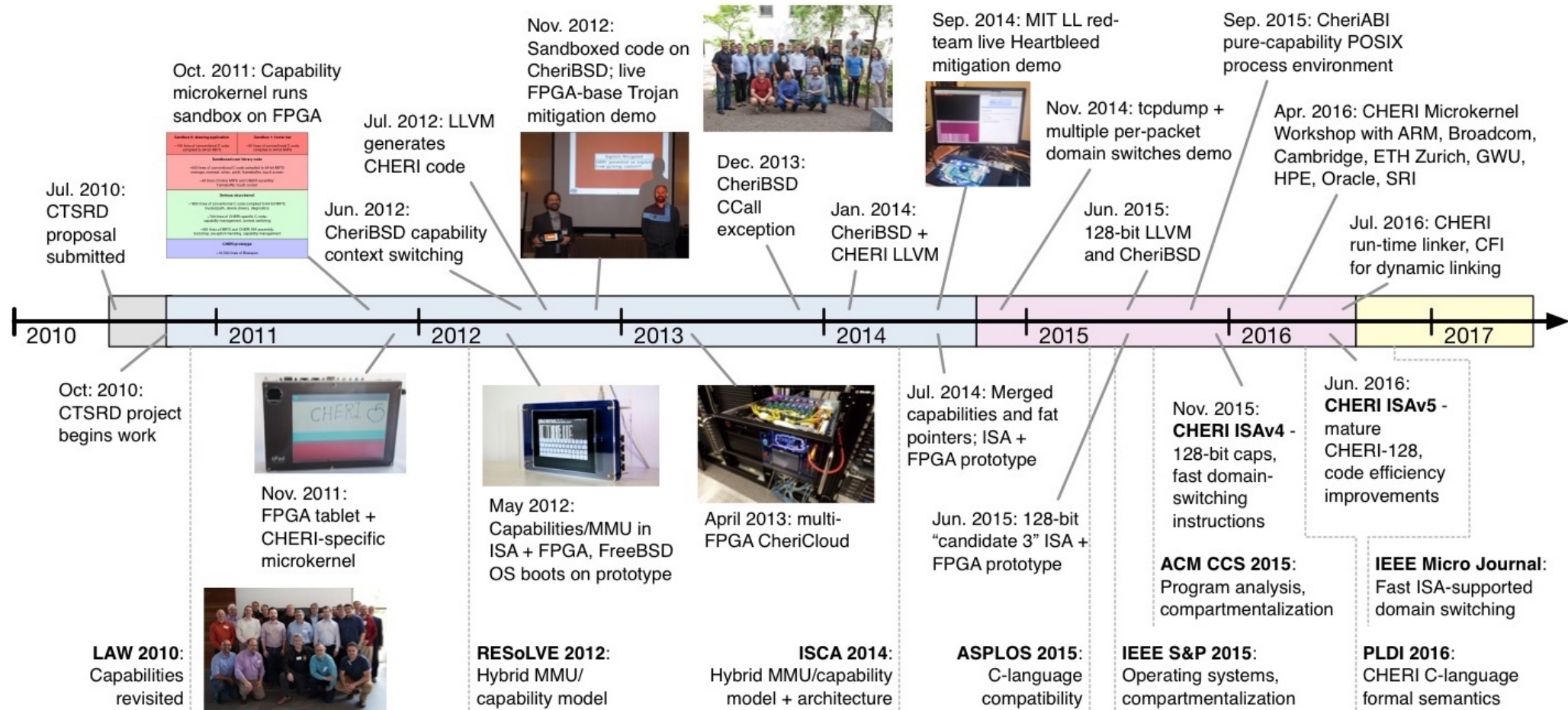
HARDWARE-SOFTWARE CO-DESIGN FOR CHERI

Hardware-software co-design over 8 years



- SRI + Cambridge over three DARPA programs (~\$26M), EPSRC REMS, (£5.6M)
Industrial: Google / DeepMind / Arm / HPE / ... (~£750K)
- Architectural mitigation for C/C++ TCB vulnerabilities
 - Tagged memory, capability pointer representation
 - Fine-grained pointer and memory protection
 - Highly scalable software compartmentalization
 - Hybrid capability system for incremental adoption
- Least-privilege, capability-oriented design mitigates many known (and unknown future) classes of vulnerabilities + exploit techniques
- Hardware-software-model co-design + concrete prototyping:
 - CHERI abstract protection model, CHERI-MIPS concrete ISA
 - 2x CHERI-MIPS ISA formal models, Qemu-CHERI, FPGA prototypes
 - CHERI Clang/LLVM, CheriBSD OS, C/C++-language applications
 - Repeated iteration to improve {overhead, security, compatibility, ..}

CHERI research and development timeline



Years 1-2: Research platform, prototype architecture

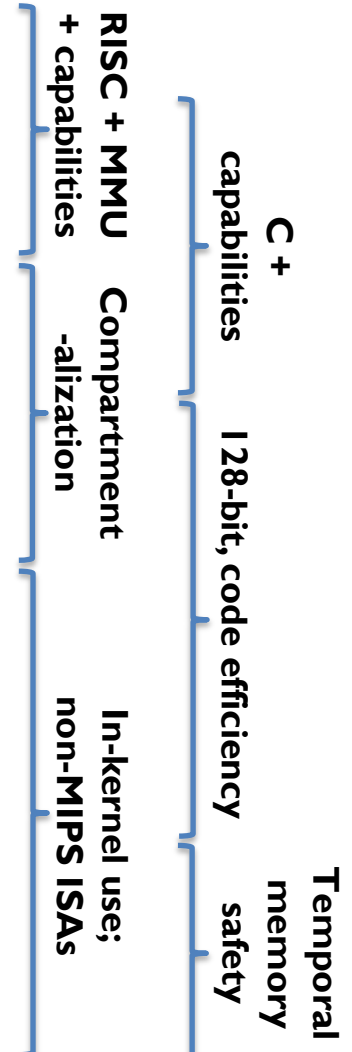
Years 4-7: Efficiency, software stack at scale

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

CHERI ISAv6 in 2017; **CHERI ISAv7** due 2019

CHERI ISA Refinement over 9 years

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 and CHERI-RISC-V sketches Exception-free domain transition
2019	ISAv7	64-bit capabilities for 32-bit architectures Elaborated draft CHERI-RISC-V ISA Architectural performance optimization for C++ applications Temporal memory safety Microarchitectural side-channel resistance features



CHERI PROTECTION MODEL AND ARCHITECTURE

CHERI design goals and approach (I)

- **Architectural security** to mitigate **C/C++ TCB vulnerabilities**
 - Efficient primitives allow software to ubiquitously employ the **principle of least privilege** and **principle of intentional use**
- **De-conflate virtualization and protection**
 - Memory Management Units (MMUs) protect by **location** in memory
 - CHERI protects **references (pointers)** to code, data, objects
 - Capabilities can also be used to describe **scalable isolated compartments with efficient sharing** within address spaces
 - Capabilities add protection properties to **existing indirection** (pointers), avoiding adding new architectural table lookups

CHERI design goals and approach (2)

- **Hybrid capability architecture**
 - Model **composes naturally** with RISC ISAs, MMUs, MMU-based systems software, C/C++ languages
 - Capabilities protect resources **within virtual address spaces**
 - Supports **incremental software deployment paths**
- **Architectural mechanism** can enforce various **software policies**
 - **Language-based properties** – e.g., referential, spatial, and temporal integrity (e.g., C/C++ compiler, linkers, OS model, runtime)
 - **New software abstractions** – e.g., software compartmentalization (e.g., confined objects for in-address-space isolation)

CHERI design goals and approach (3)

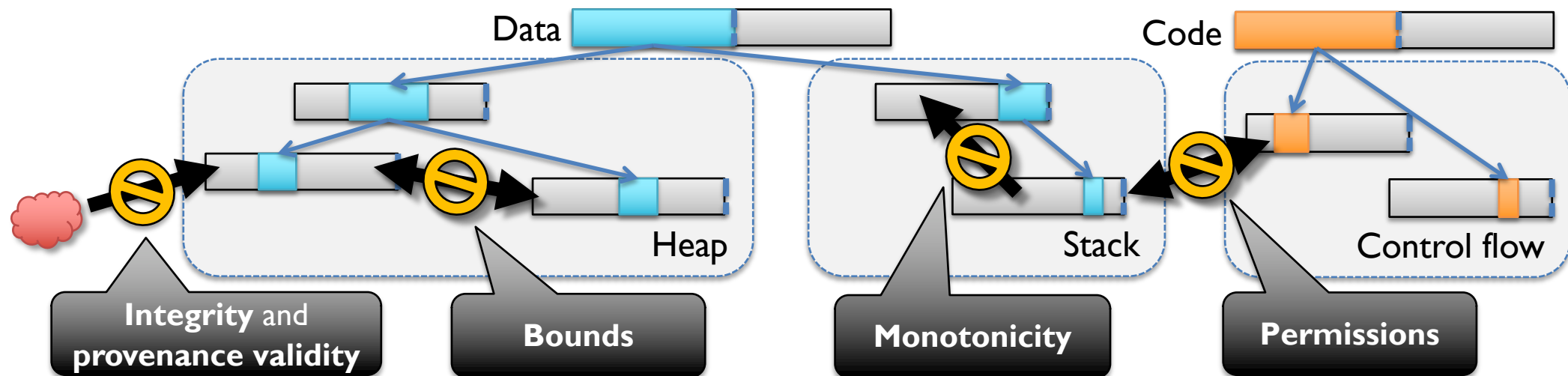
- **Limited + selective disruption** to current architecture, microarchitecture
 - Retain almost vast majority of current RISC / load-store ISAs including register structure and supervisor features such as the MMU
 - **Introduce capability registers and instructions**
 - **Introduce compressed capability model**
 - **Interpose on I-fetch and legacy load/store instructions**
 - **Constrain privileged instructions to allow kernel sandboxing**
 - **Introduce capability-width physical memory tagging**
 - Implementation is **consistent with current design tenets** for in-order and superscalar processors, cache-based memory subsystems
 - Key contributions include **capability compression, tag support**

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., HW assist on bounds checking)
- **Scalable in-process compartmentalization**
 - Facilitate exploit-independent mitigation techniques

C/C++-language memory and pointer safety is the focus of this talk

CHERI enforces protection semantics for pointers

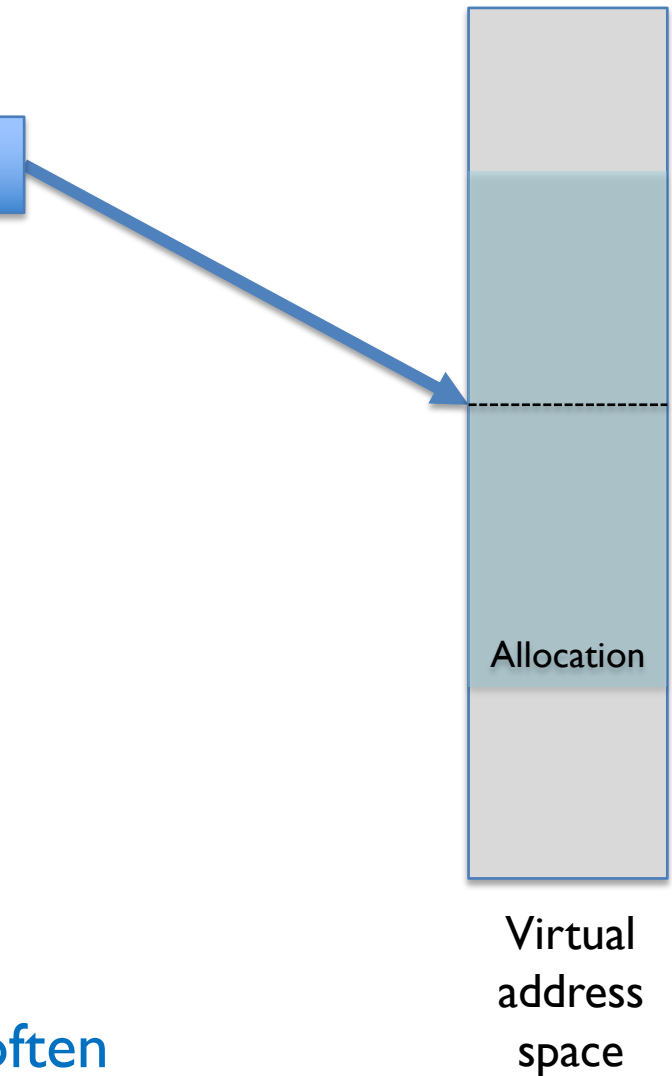


- **Integrity and provenance validity** ensure that valid pointers are derived from other valid pointers via valid transformations; **invalid pointers cannot be used**
 - E.g., Received network data cannot be interpreted as a code or data pointer
- **Bounds** prevent pointers from being manipulated to 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

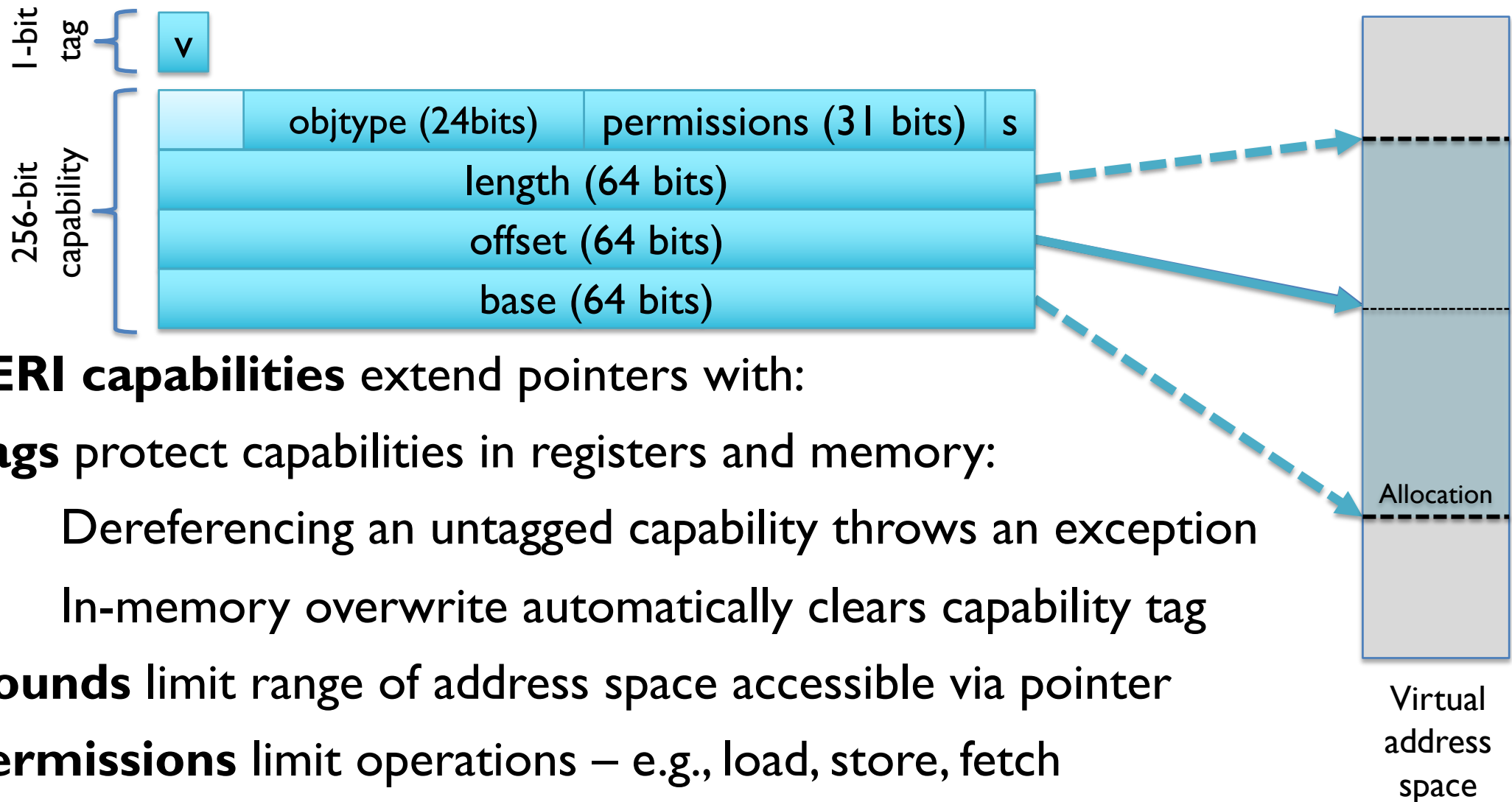
Pointers today



- 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** – 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, often achieving **arbitrary code execution**



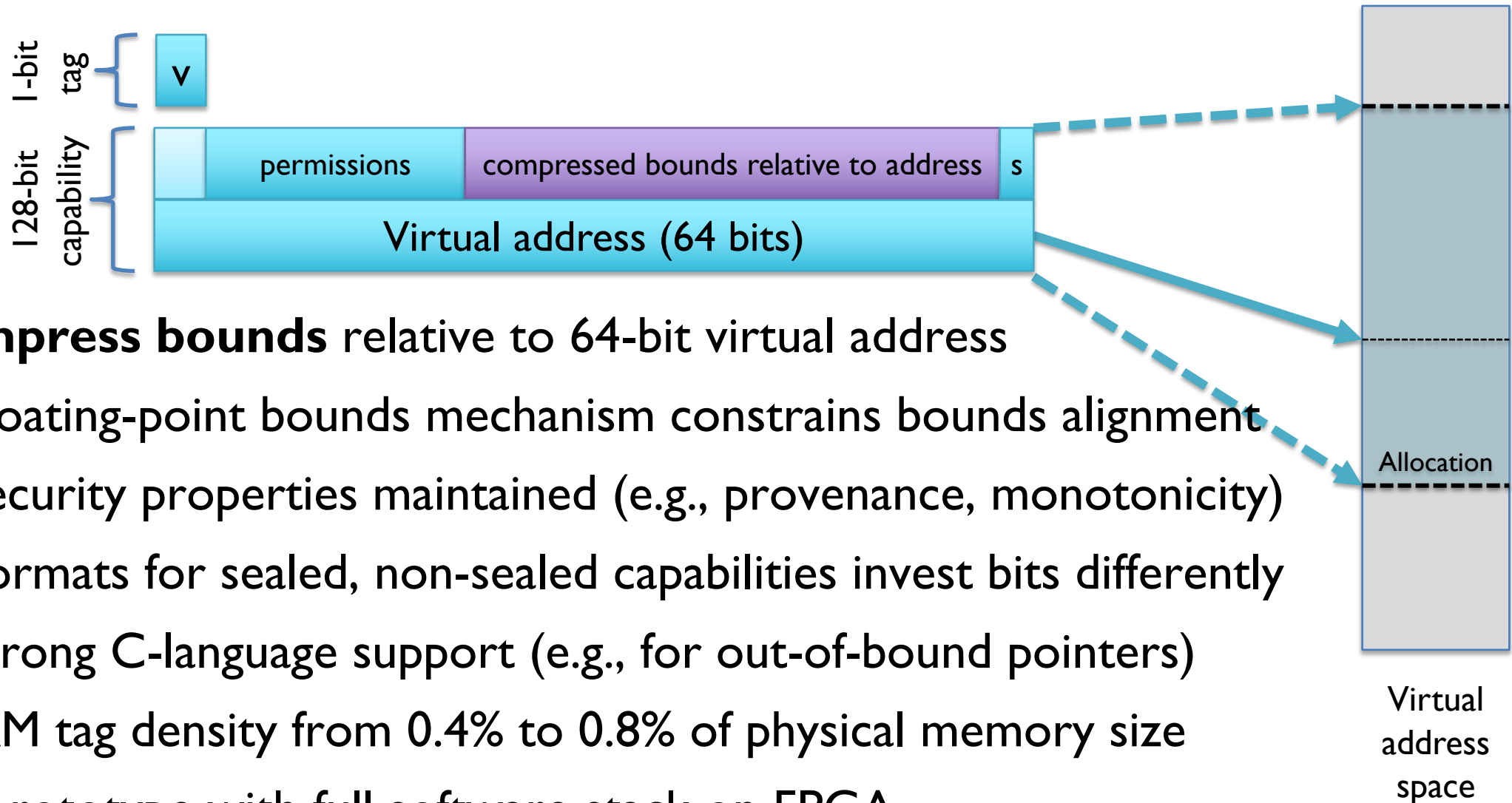
Protection model: 256-bit capabilities



CHERI capabilities extend pointers with:

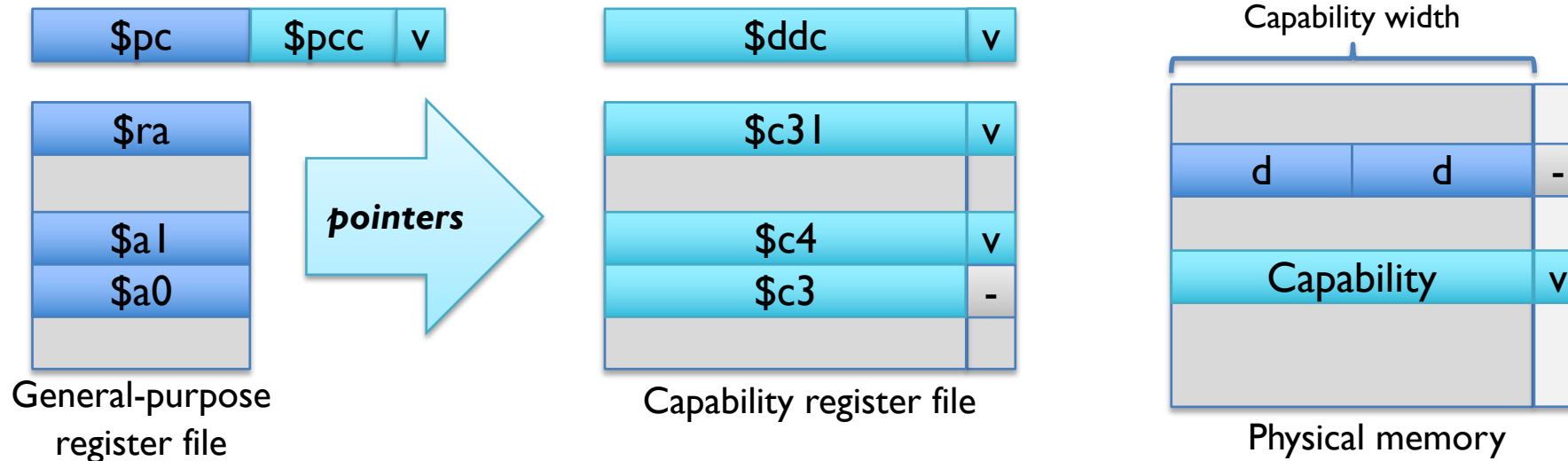
- **Tags** protect capabilities in registers and memory:
 - Dereferencing an untagged capability throws an exception
 - In-memory overwrite automatically clears capability tag
- **Bounds** limit range of address space accessible via pointer
- **Permissions** limit operations – e.g., load, store, fetch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

Architecture: 128-bit compressed capabilities



- **Compress bounds** relative to 64-bit virtual address
 - Floating-point bounds mechanism constrains bounds alignment
 - Security properties maintained (e.g., provenance, monotonicity)
 - Formats for sealed, non-sealed capabilities invest bits differently
 - Strong C-language support (e.g., for 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**

FINE-GRAINED MEMORY PROTECTION

What are CHERI's implications for software?

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

Provenance validity: Q: Where do pointers come from?

Integrity: Q: How do pointers move in practice?

Bounds, permissions: Q: What rights should pointers carry?

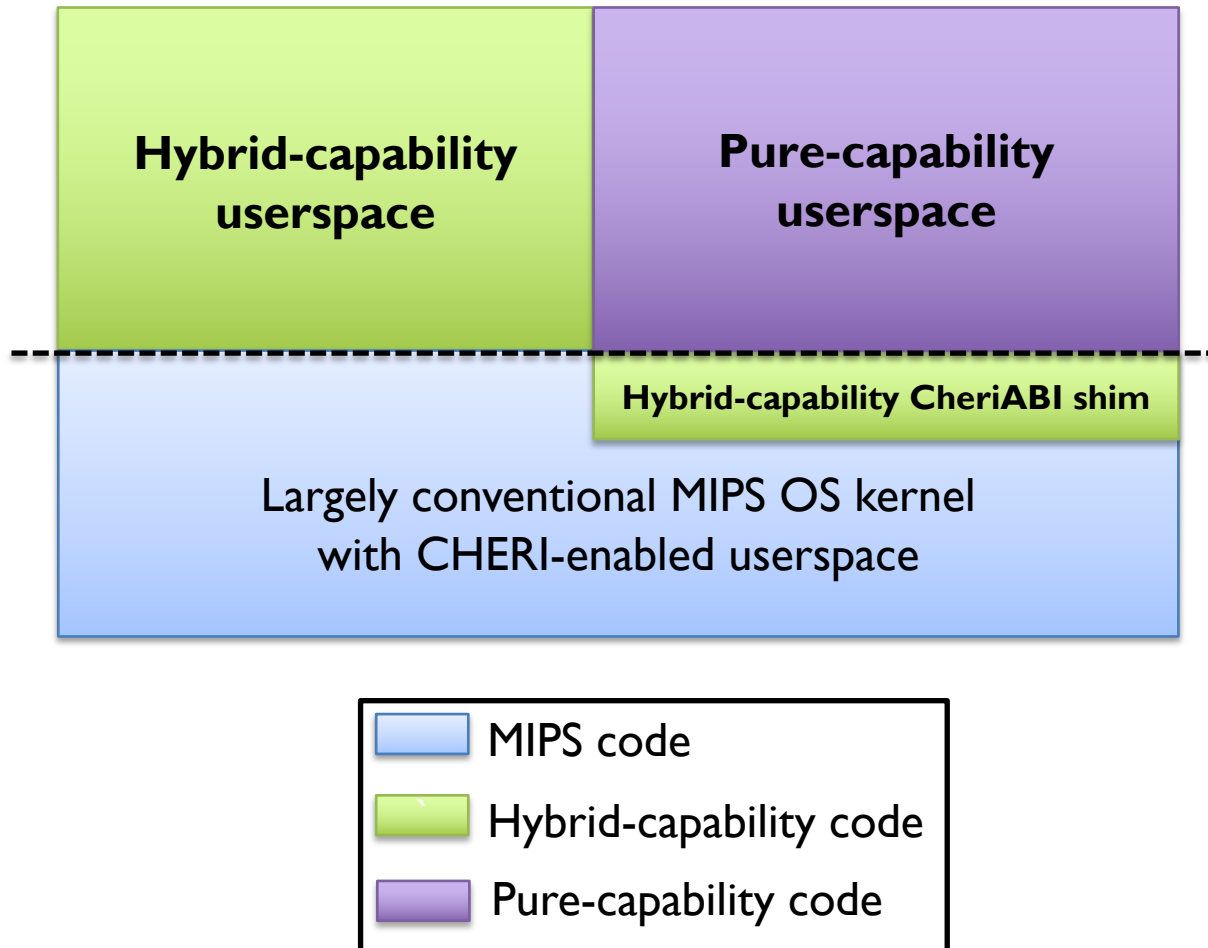
Monotonicity: Q: Can real software play by these rules?

- Scalable fine-grained **software compartmentalization**

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

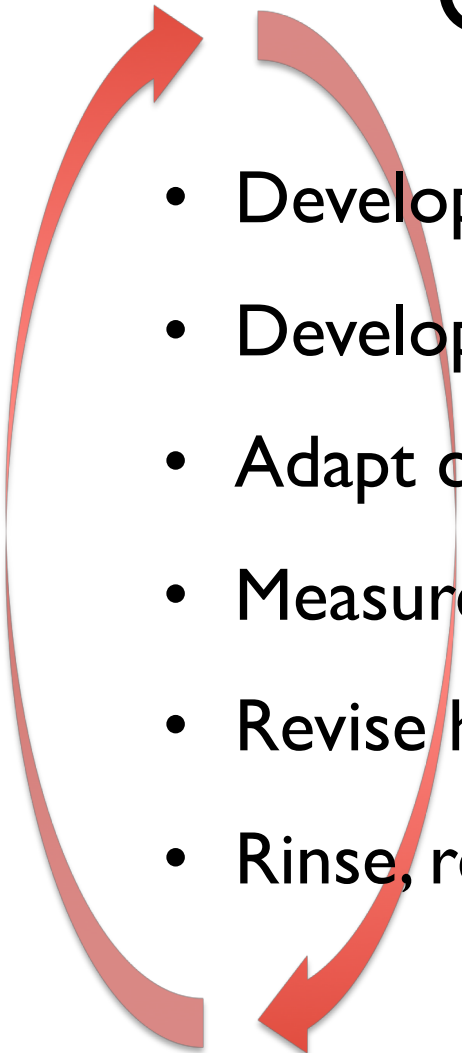
Q: Can **sealed capabilities**, **controlled non-monotonicity**, and **capability-based sharing** enable safe, efficient compartmentalization?

From hybrid-capability code to 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)
- Now investigating pure-capability kernel

CheriABI co-design methodology

- 
- Develop pure-capability CHERI Clang/LLVM compiler suite
 - Develop pure-capability CheriABI POSIX process environment
 - Adapt complete UNIX system and its applications
 - Measure compatibility, performance, protection, ...
 - Revise hardware, architecture, compiler/linker, OS, applications
 - Rinse, repeat

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, WebKit (C++)
- **Valid provenance, minimized privilege for pointers, implied VAs**
 - Userspace capabilities originate in **kernel-provided roots**
 - Compiler, allocators, run-time linker, etc., **refine** bounds and perms
- Trading off **privilege minimization, monotonicity, API conformance**
 - Typically in memory management – realloc(), mmap() + mprotect()

OS changes required for CheriABI

(A grand tour of low-level OS behavior)

Hybrid ABI = MIPS ABI + ...

- Kernel support for tagged memory, capability context switching, etc.
- Tag-preserving libc: memory copy, memory move, sort, ...
- Bounds-aware malloc(), realloc(), free(), ...
- setjmp(), longjmp(), sigcontext / signal delivery, pthreads updates for capabilities
- Run-time linkage for capability-based references to globals, code, vtables, etc. (bounds, permissions, ...)
- Debugging APIs such as ptrace()

CheriABI = Hybrid ABI + ...

- Kernel support for pure-capability userspace
- C start-up/runtime (CSU/CRT) changes
- Initial process state: reduced initial capability registers, ELF aux args, sigcode, etc.
- Pointer arguments/return values for syscalls are now capabilities, ...
- Review and fix tag preservation, integer/pointer provenance and casts
- Run-time linkage for globals, code, vtables, etc. (bounds, permissions, ...)

Evaluating memory-protection compatibility

Approach: Prototype (1) “pure-capability” **C compiler** (Clang/LLVM) and (2) **full OS** (FreeBSD) that use capabilities for all explicit or implied userspace pointers

Goal: Little or no software modification (BSD base system + utilities)
 Small changes to source files for 34 of 824 programs, 28 of 130 libraries.
 Overall: modified ~200 of ~20,000 user-space C files/header

	Pointer + integer integrity, prov.	Pointer size & alignment	Monotonicity	Calling conventions	Unsupported features
BSD headers	11	6	0	2	0
BSD libraries	83	36	4	41	22
BSD programs	24	9	1	11	2

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

	Pass	Fail*	Skip	Total
MIPS	3501 (91%)	90	244	3835
Pure capability	3301 (90%)	122	246	3669

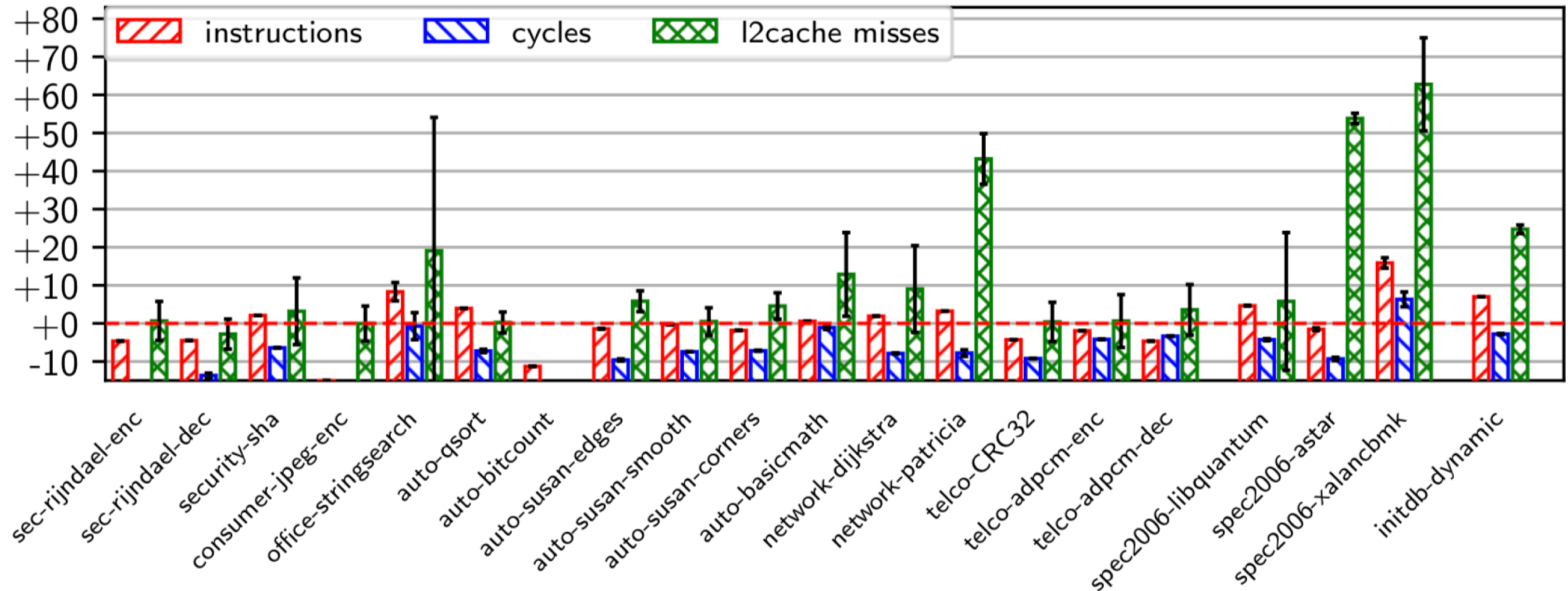
Evaluating memory-protection impact

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

	OK	min	med	large
mips64	0	4	8	175
CheriABI	0	279	289	291
LLVM Address Sanitizer (asan) on x86	0	276	286	286

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

Performance



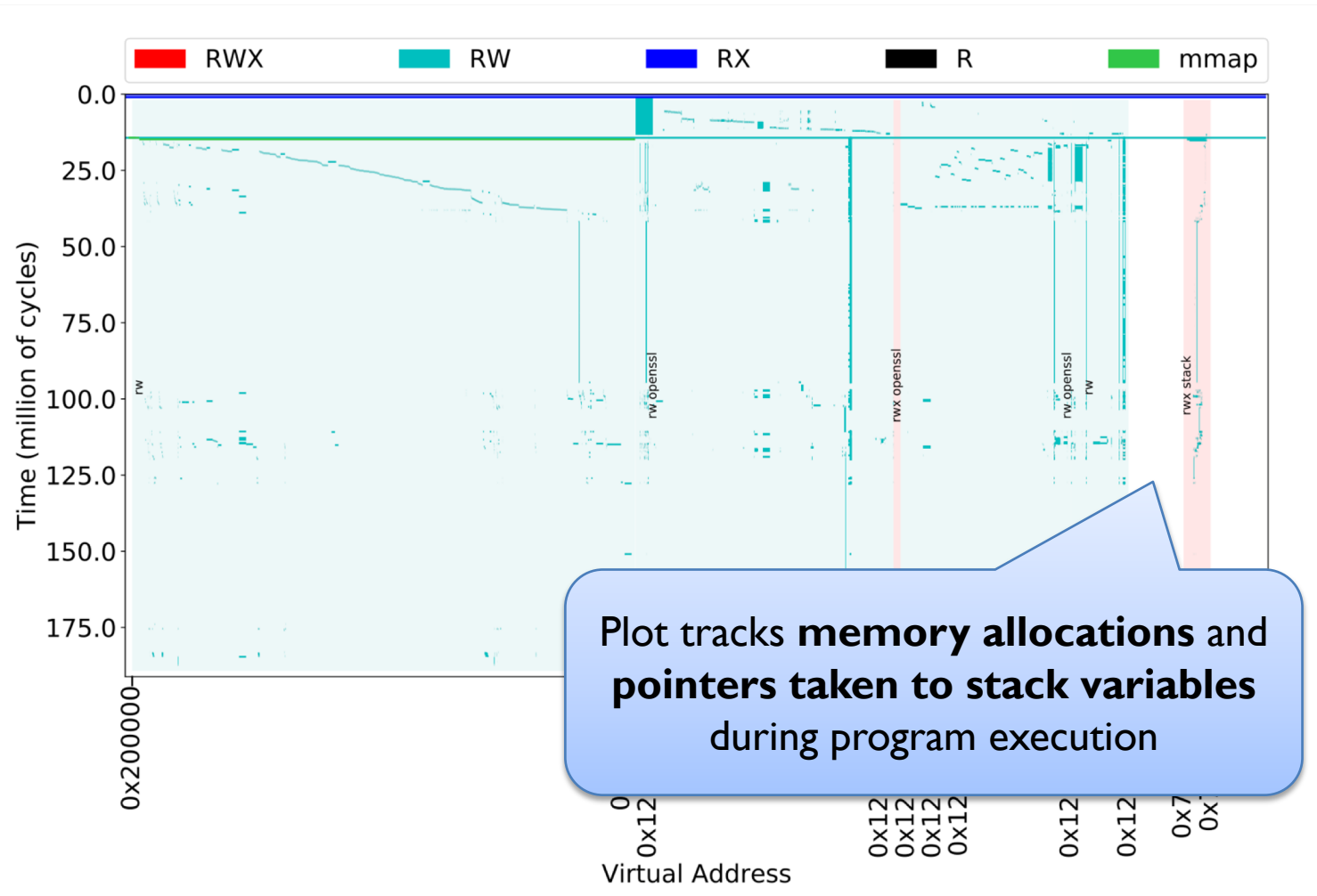
Impact in CHERI-MIPS typically less than 5%

Key overhead: higher pointer-density applications see increased cache pressure

Small print: 100MHz pipelined 64-bit MIPS core with CHERI extensions, 32KiB L1, 256 KiB L2 on FPGA

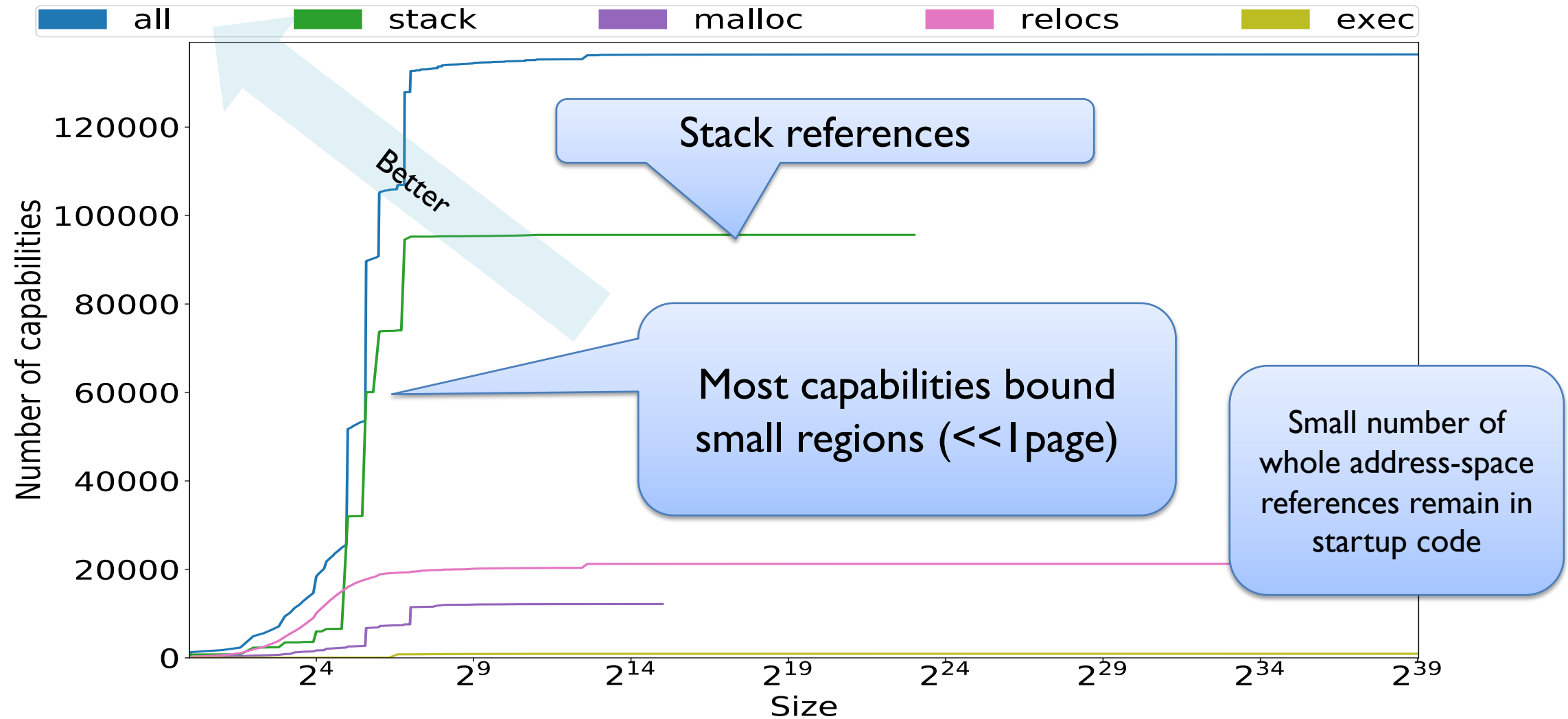
Similar core to ARM Cortex-A53 but without prefetching.

Trace-based analysis using tagged pointers



- CHERI tags pointers in hardware, so we can find them in registers and memory
 - Extract detailed execution traces in Qemu and FPGA
 - Construct pointer provenance graphs
- Pattern match and measure:
 - allocations (stack, heap, ...),
 - propagation of capabilities,
 - rights refinements,
 - data and capability leaks, etc.
- Evaluate temporal aspects

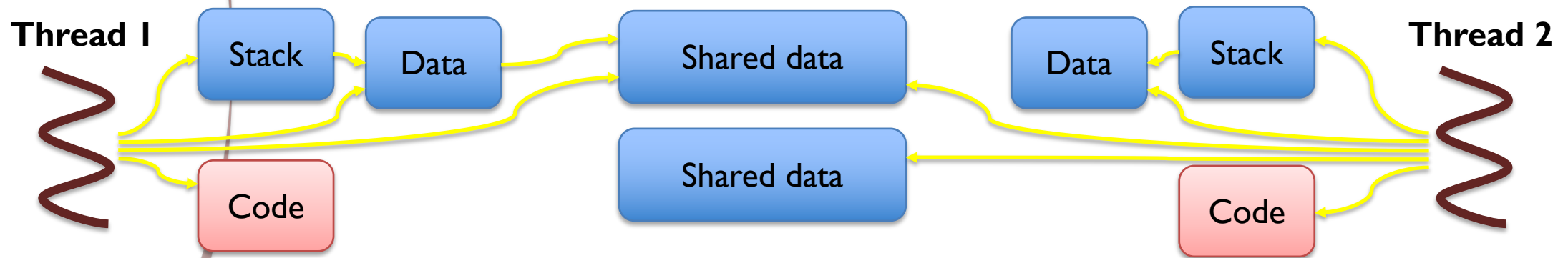
Capability bounds (OpenSSL)



SOFTWARE COMPARTMENTALIZATION

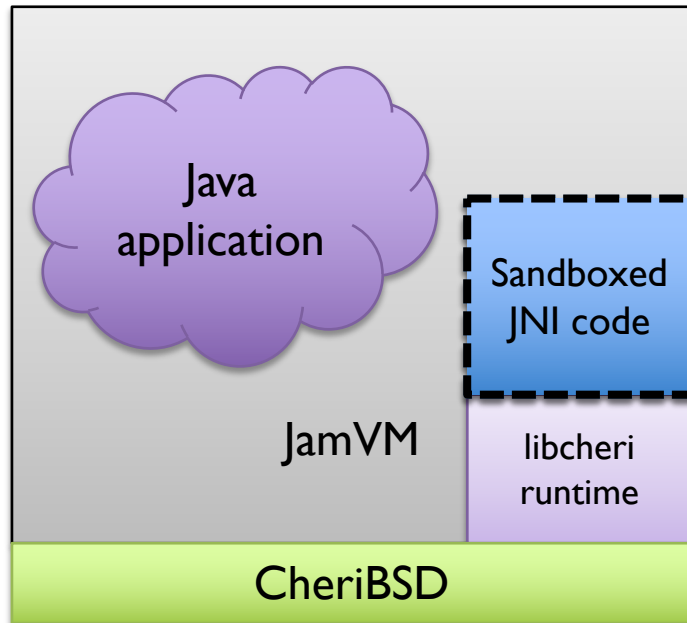
Principles of CHERI compartmentalization (Oakland 2015)

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



- We can then construct an **compartmentalized security models**; e.g., **classes, objects, shared memory**, and **object invocation**

CHERI-JNI: Protecting Java from JNI (ASPLOS 2017)



- **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

Growing the 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”

Ongoing HW-SW security research projects

- EPSRC IOSEC – Research into I/O-originated adversaries
 - NDSS 2019:Thunderclap – OS IOMMU vulnerabilities
- DARPA ECATS – CHERI + SoCs – SRI, Cambridge, ARM Research
 - CHERI for 32-bit microcontrollers
 - CHERI-RISC-V
 - CHERI interactions with DMA and heterogenous compute
 - Containing untrustworthy IP cores in CHERI-aware SoCs
- DARPA CIFV – Formal modeling/reasoning – SRI, Cambridge, Arm Research
 - Formal models of CHERI-enabled architectures
 - Formal verification of CHERI architectural security properties

Extensive open-source ecosystem and academic publication record

- Unique hardware – software – formal-model co-design process
- Memory protection + compartmentalization for MIPS, RISC-V, ARMv8, ARM-M
- Papers at ISCA'14, ASPLOS'15, IEEE S&P'15, ACM CCS'15, PLDI'2016, ASPLOS'17, ICCD'17, ICCD'18, POPL'19, NDSS'19, and ASPLOS'19
- Research featured in **the Register** (2019), the **New Scientist** (2018), the **Economist** (2014), and **New York Times** (2012)
- Sail formal ISA models of CHERI-MIPS (and soon CHERI-RISC-V) convert to Isabelle, HOL, and Coq to allow formal verification of security properties
- Open-source CHERI-MIPS and CHERI-RISC-V CPU cores in Bluespec SystemVerilog (BSV) targeted at FPGA and cycle-accurate C simulation
- Open-source compiler, linker, debugger, and OS including Clang/LLVM and full memory-safety FreeBSD UNIX implementation
- Typical cycle overheads <5% for workloads on multiple microarchitectures
- Multi-year collaboration with Arm

Conclusion

- New architectural primitives require software adaptation and rich evaluation
 - Primitives support many potential usage patterns, use cases
 - Applicable uses depend on compatibility, performance, effectiveness
 - Best validation approach: full hardware-software prototype
 - Co-design methodology: hardware $\leftrightarrow$ architecture $\leftrightarrow$ software
- CheriABI explores ubiquitous pointer and spatial memory protection in the MMU-based POSIX process model
 - Tradeoffs around language semantics, security effects
 - Good compatibility, strong protection, reasonable overheads
- Exposing greater program semantics to architecture assists with efficient protection – but could it have other benefits (e.g., in microarchitecture?)

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

Learning more about CHERI

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

Watson, Moore, Neumann, et al. **Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6)**, Technical Report UCAM-CL-TR-907, Computer Laboratory, April 2017.

CHERI ISAv7-alpha4 (draft) available on request; technical report due for release in early 2019