CTSRD

CRASH-worthy Trustworthy Systems Research and Development

CHERI

A Hybrid Capability Architecture

Robert N. M. Watson

Simon W. Moore, Peter G. Neumann, Jonathan Woodruff, Jonathan Anderson, Hadrien Barral, Ruslan Bukin, David Chisnall, Nirav Dave, Brooks Davis, Lawrence Esswood, Khilan Gudka, Alexandre Joannou, Chris Kitching, Ben Laurie, A. Theo Markettos, Alan Mujumdar, Steven J. Murdoch, Robert Norton, Philip Paeps, Alex Richardson, Michael Roe, Colin Rothwell, Hassen Saidi, Stacey Son, Munraj Vadera, Hongyan Xia, and Bjoern Zeeb

University of Cambridge, SRI International

CHERI Microkernel Workshop – 23 April 2016

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

Architectural protection for pointers

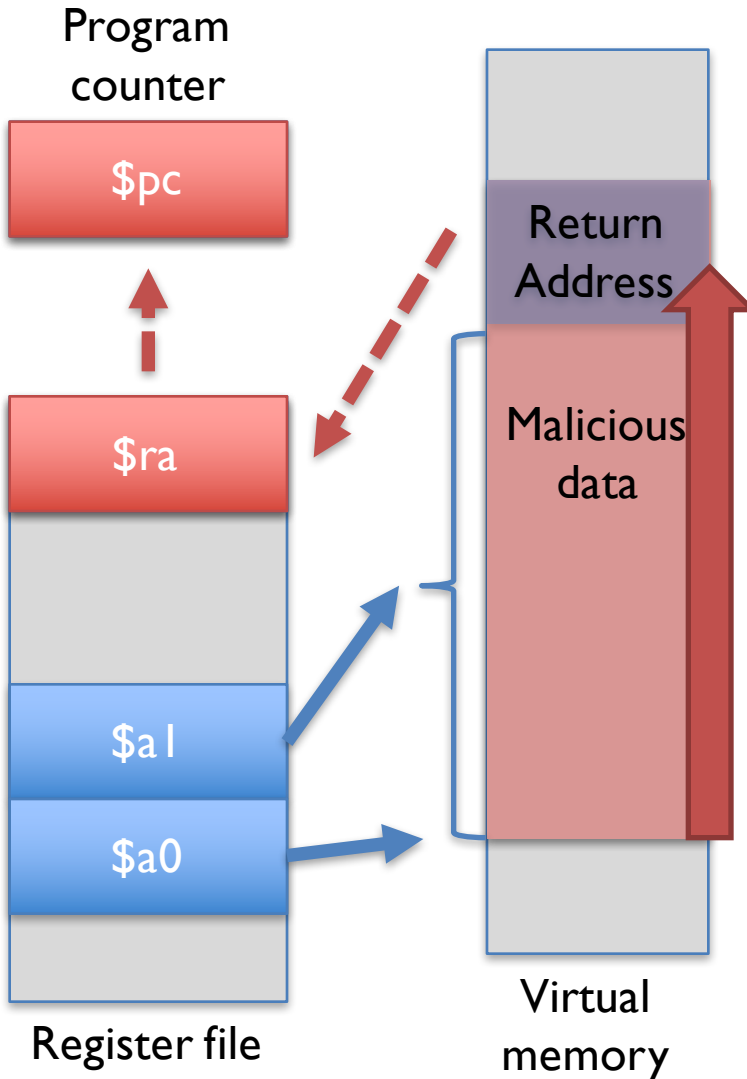
- De-conflate **virtualization** and **protection**
- **Hybrid model**: retain **Memory Management Unit** (MMU) implementing **multiple address spaces**
  - OS processes, full-system virtualization, …
- Add **ISA-level capabilities** to implement and protect **pointers within address spaces**
  - Fine-grained, compiler-driven **memory protection** for code and data
  - Fine-grained, scalable **compartmentalization**

CHERI software protection goals

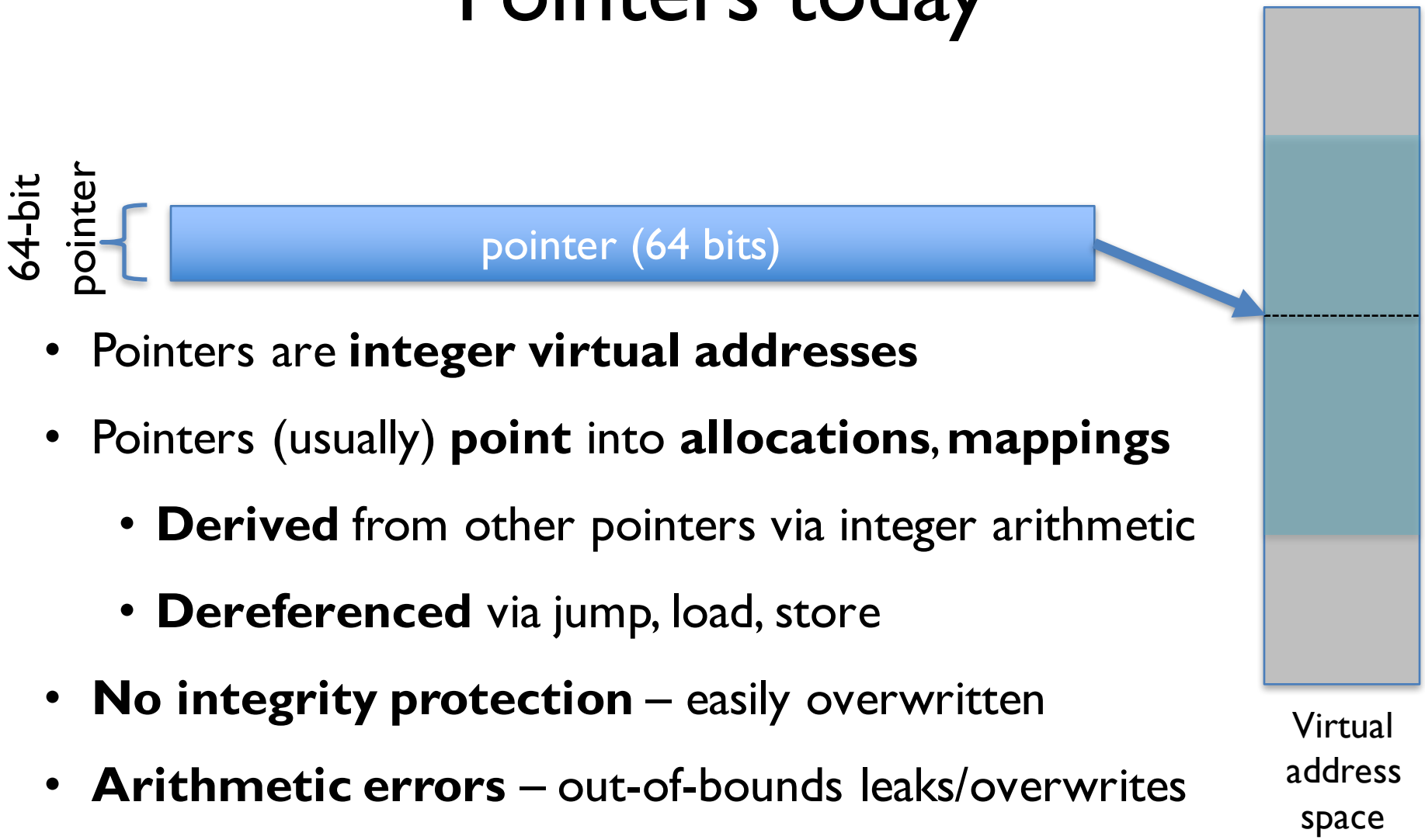
- Target **C/C++-language TCBs** – OS kernels, monolithic applications, language runtimes, …:
  - **Spatial safety** protects against many pointer-misuse vulnerabilities
  - **Temporal safety** supports software models that protect against memory re-use attacks
  - **Scalable compartmentalization** provides exploit-independent mitigation
- **Hybrid capability-system model** provides strong compatibility with current software models

Architectural least privilege

- Classical buffer-overflow attack
  - Buggy code overruns a buffer, overwriting an on-stack return address
  - Overwritten return address is loaded and jumped to, corrupting control flow
- Why did we allow these privileges not required by the language model:
  - Ability to overrun the buffer?
  - Ability to inject a code pointer that can be used as a jump target?
  - Ability to execute data as code?
- Limiting these privileges wouldn't prevent the bug – but would provide effective **architectural vulnerability mitigation**

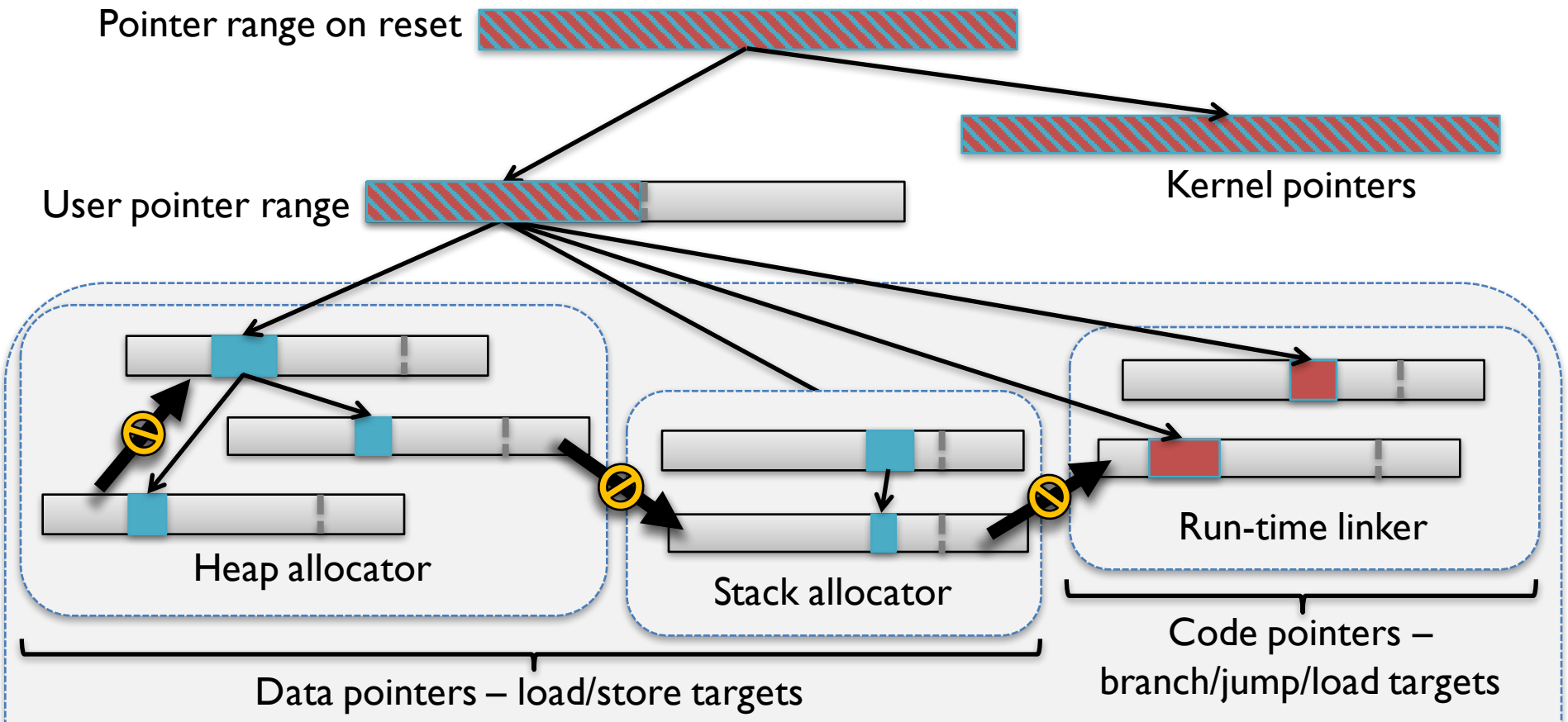


Pointers today



- Pointers are **integer virtual addresses**
- Pointers (usually) **point** into **allocations, mappings**
  - **Derived** from other pointers via integer arithmetic
  - **Dereferenced** via jump, load, store
- **No integrity protection** – easily overwritten
- **Arithmetic errors** – out-of-bounds leaks/overwrites
- **Inappropriate use** – executable data, format strings

Enforcing pointer provenance, bounds, permissions, and monotonicity

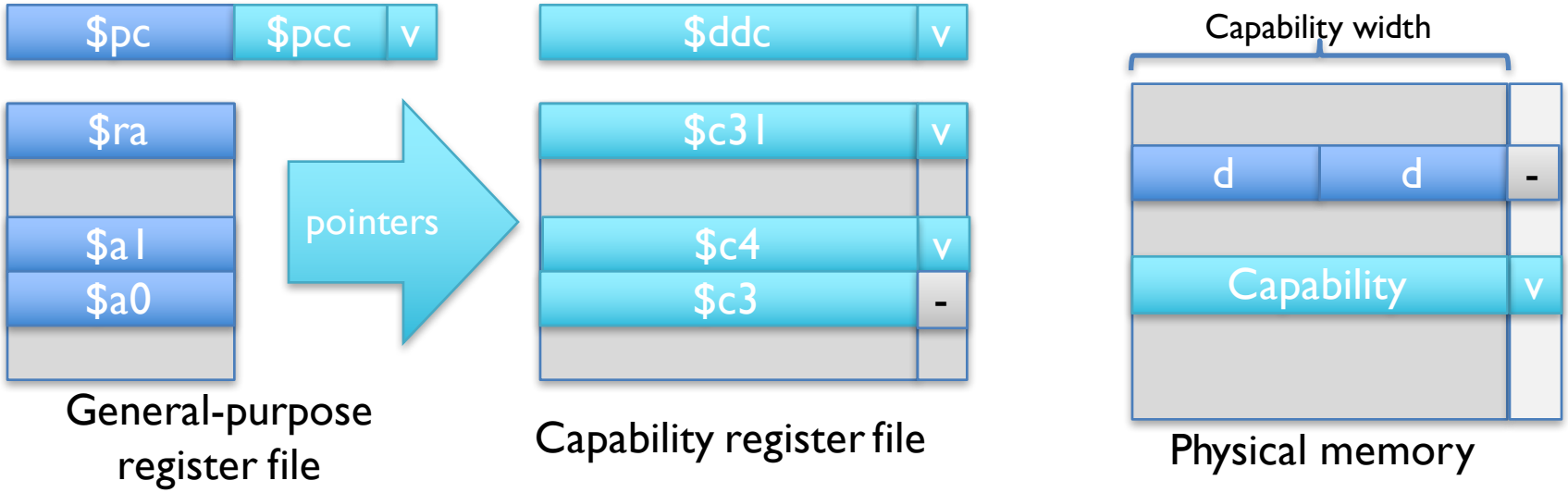


Valid userspace pointer set – pointers not generated using derivation rules are not part of the valid provenance tree and should not be dereferenceable

CHERI architectural approach

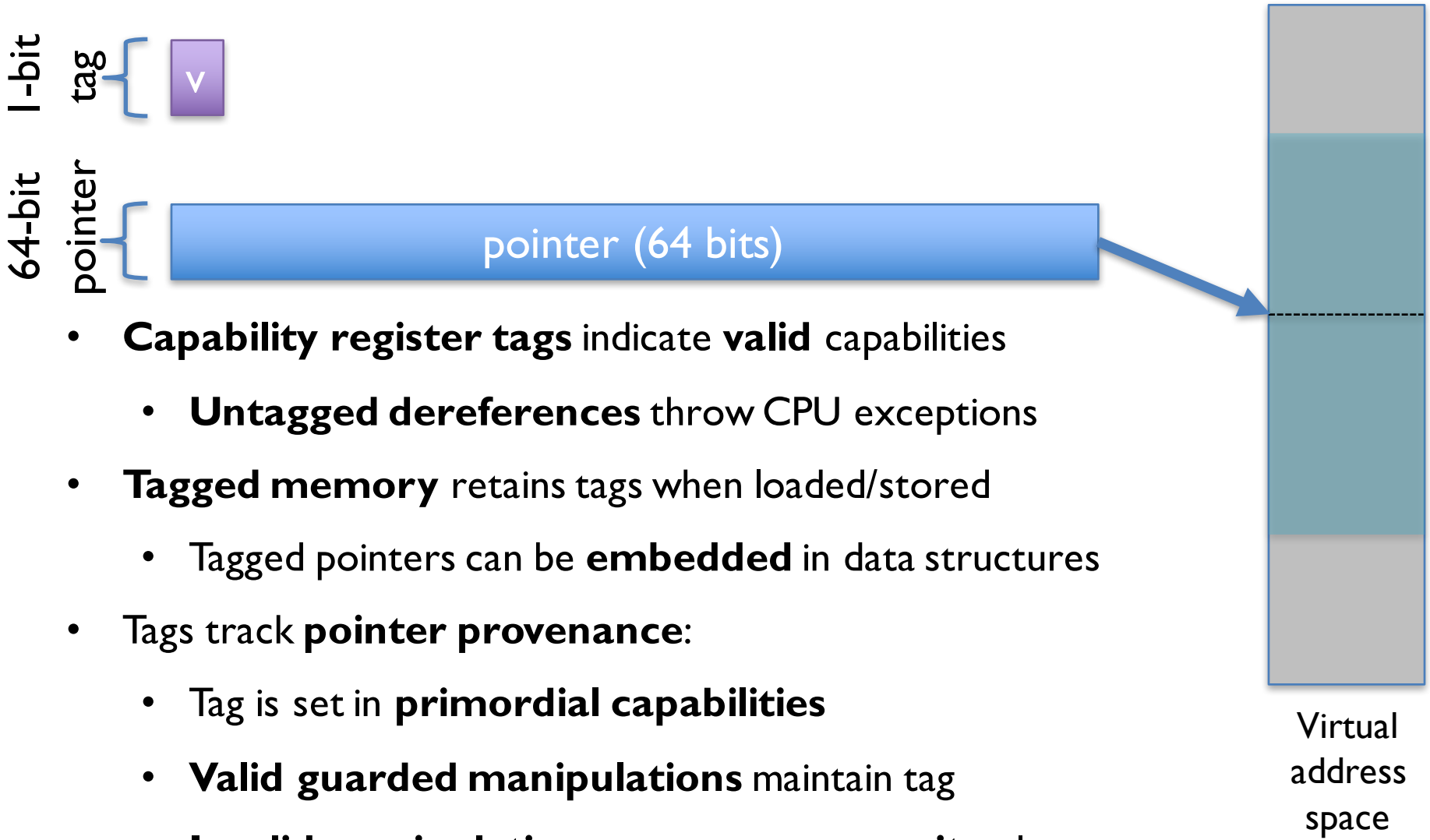
- **RISC**: simple, compiler-focused ISA extensions avoid microcode and table-based data structures
- **Pointers** implemented via **architectural capabilities**
  - **Tagged capabilities** protect code and data pointer integrity in both registers and memory
  - **Pointer metadata**, including **bounds** and **permissions**, limits undesired use
  - **Guarded manipulation** implements **capability monotonicity** and **sealing** for **least privilege**
- **256-bit architectural model** – 64-bit addresses, etc.
- Efficient **128-bit microarchitectural implementation**

CHERI-MIPS: capabilities protect pointers in registers and memory



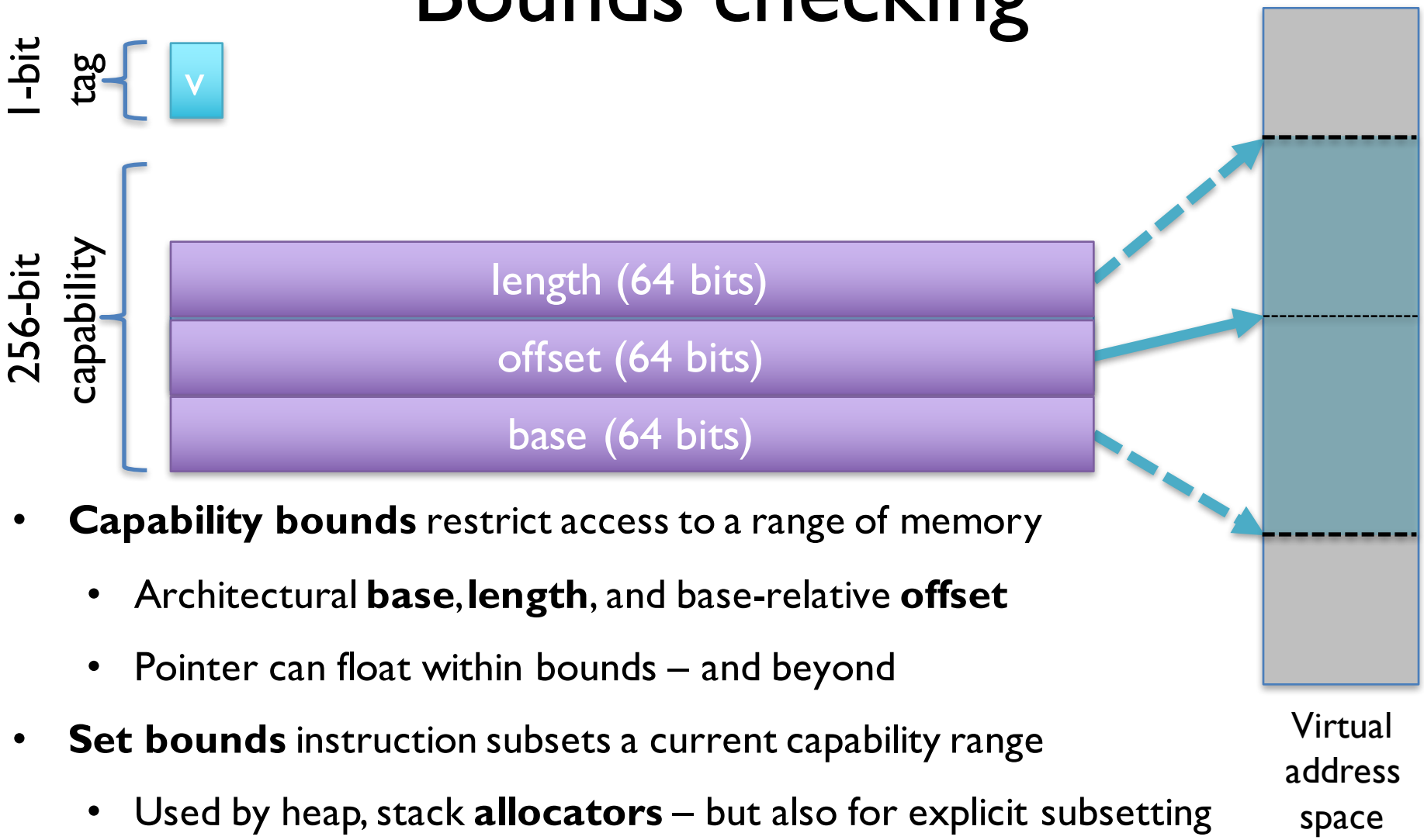
- **Capability register file** holds in-use capabilities (pointers)
- **Tagged memory** protects capability-sized words in DRAM as pointers
- **Program counter capability** ($pcc) extends program counter
- **Default data capability** ($ddc) controls legacy RISC loads/stores
- **System control registers**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

Tags for integrity and provenance



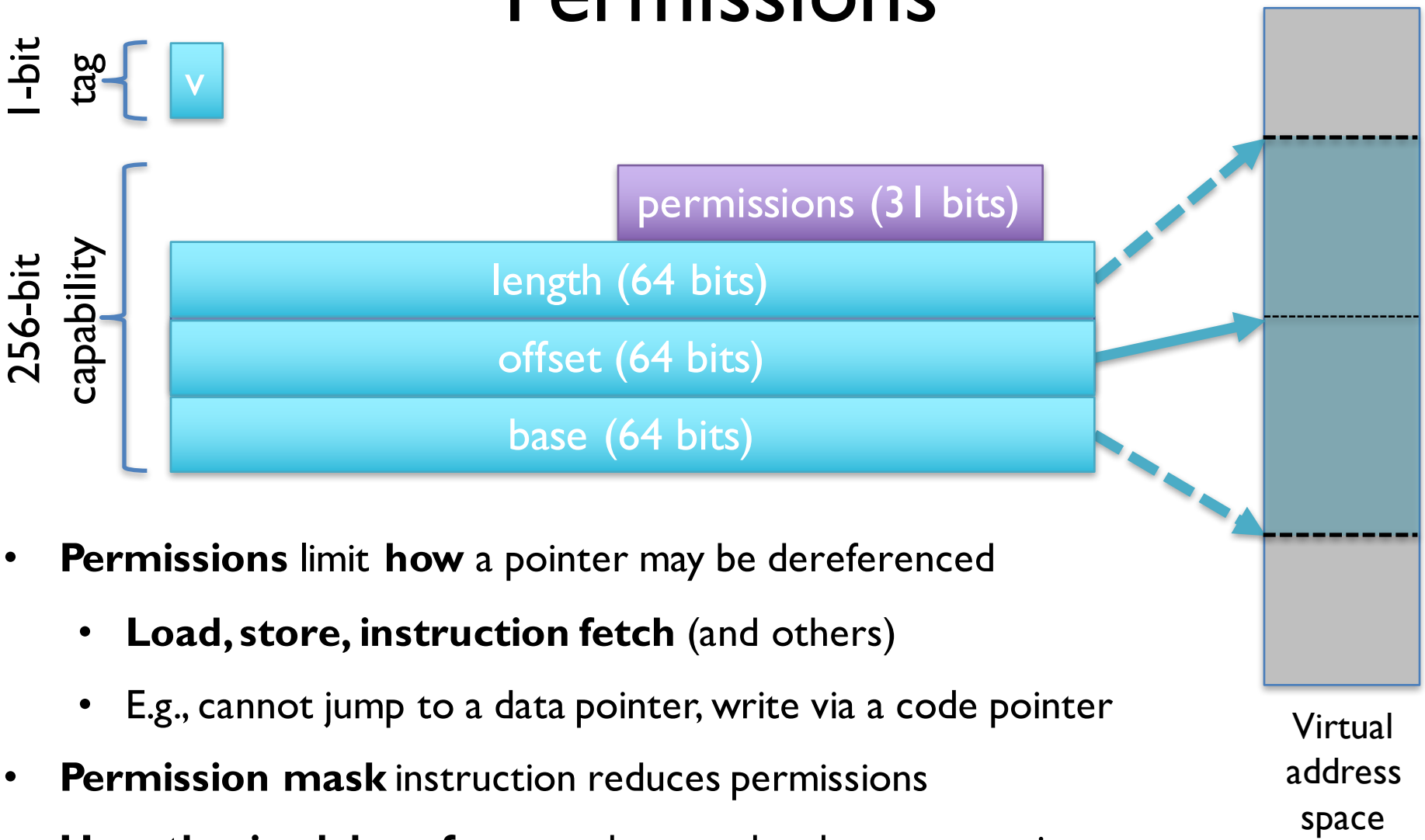
- **Capability register tags** indicate **valid** capabilities
  - **Untagged dereferences** throw CPU exceptions
- **Tagged memory** retains tags when loaded/stored
  - Tagged pointers can be **embedded** in data structures
- Tags track **pointer provenance**:
  - Tag is set in **primordial capabilities**
  - **Valid guarded manipulations** maintain tag
  - **Invalid manipulations, memory overwrite** clear tag

Bounds checking



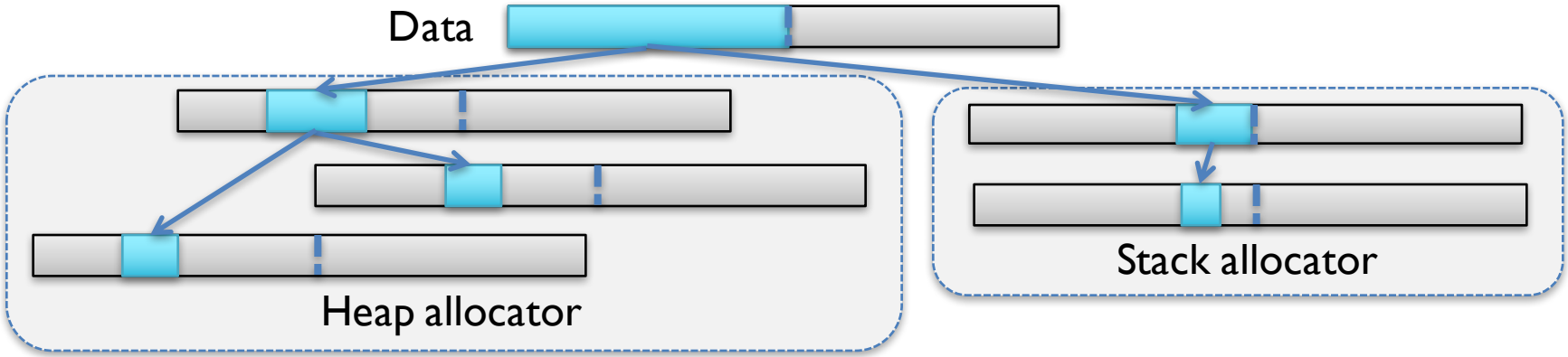
- **Capability bounds** restrict access to a range of memory
  - Architectural **base**, **length**, and base-relative **offset**
  - Pointer can float within bounds – and beyond
- **Set bounds** instruction subsets a current capability range
  - Used by heap, stack **allocators** – but also for explicit subsetting
- **Out-of-bounds dereference** throws a hardware exception

Permissions



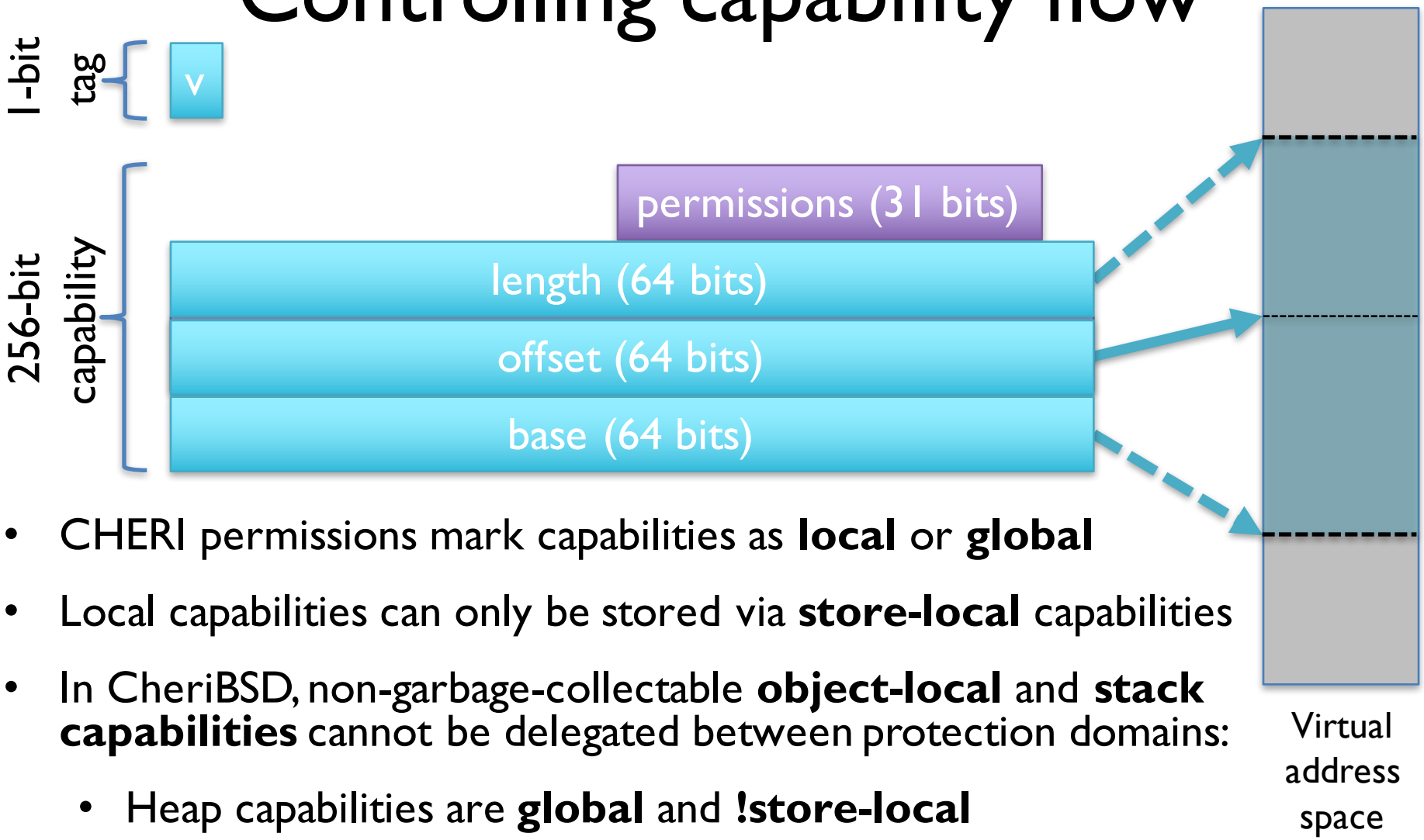
- **Permissions** limit **how** a pointer may be dereferenced
  - **Load, store, instruction fetch** (and others)
  - E.g., cannot jump to a data pointer, write via a code pointer
- **Permission mask** instruction reduces permissions
- **Unauthorized de-reference** throws a hardware exception

Pointer provenance and monotonicity



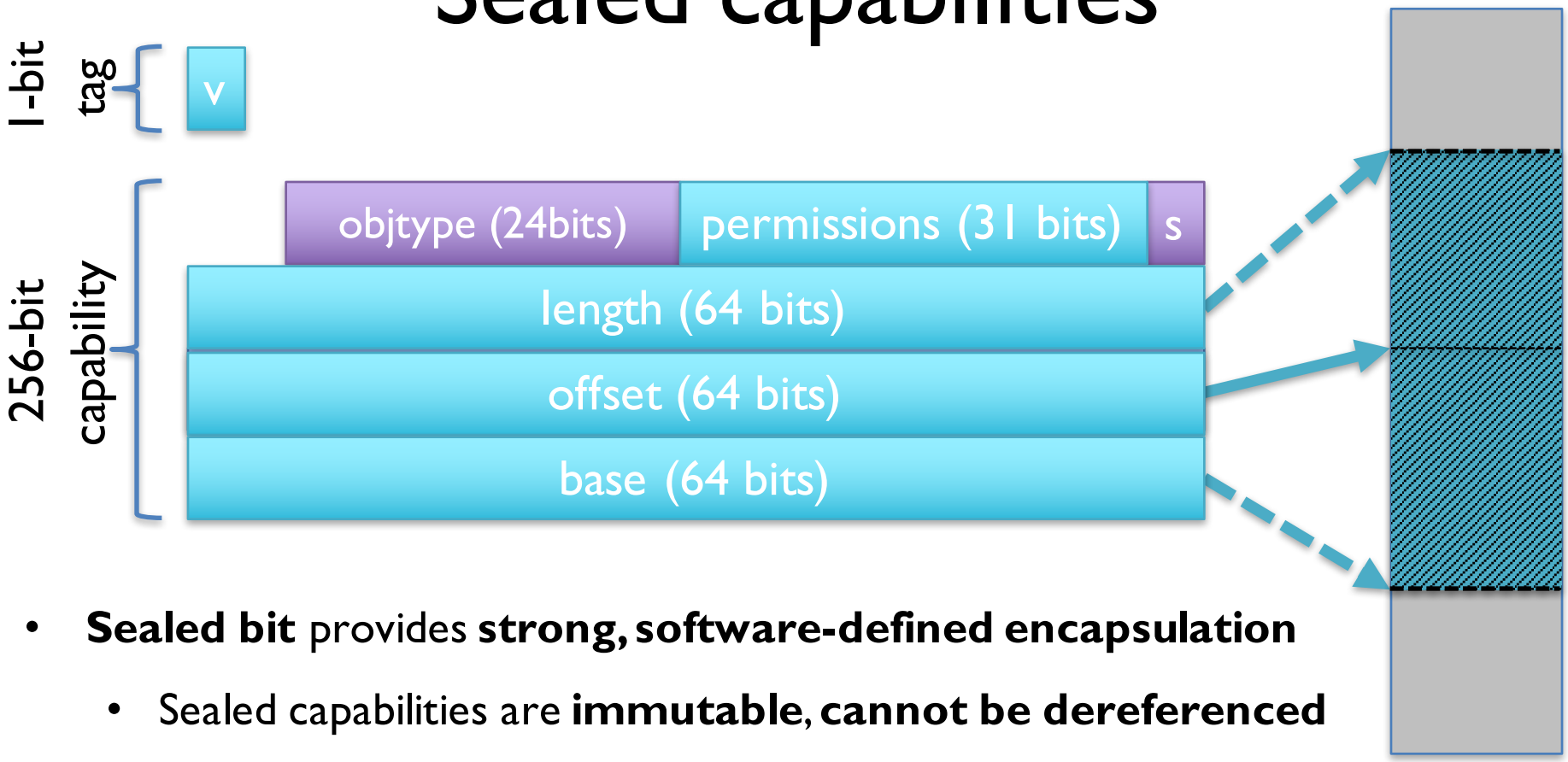
- **Pointer provenance:** pointers must be derived from other pointers
- **Guarded manipulation** implements **capability monotonicity:**
  - **Tags** can be cleared but not set
  - **Bounds** can be narrowed but not widened
  - **Permissions** can be cleared but not set
- E.g., received network data cannot be interpreted as a **code pointer**
- E.g., **data pointers** cannot be manipulated to access other heap objects

Controlling capability flow



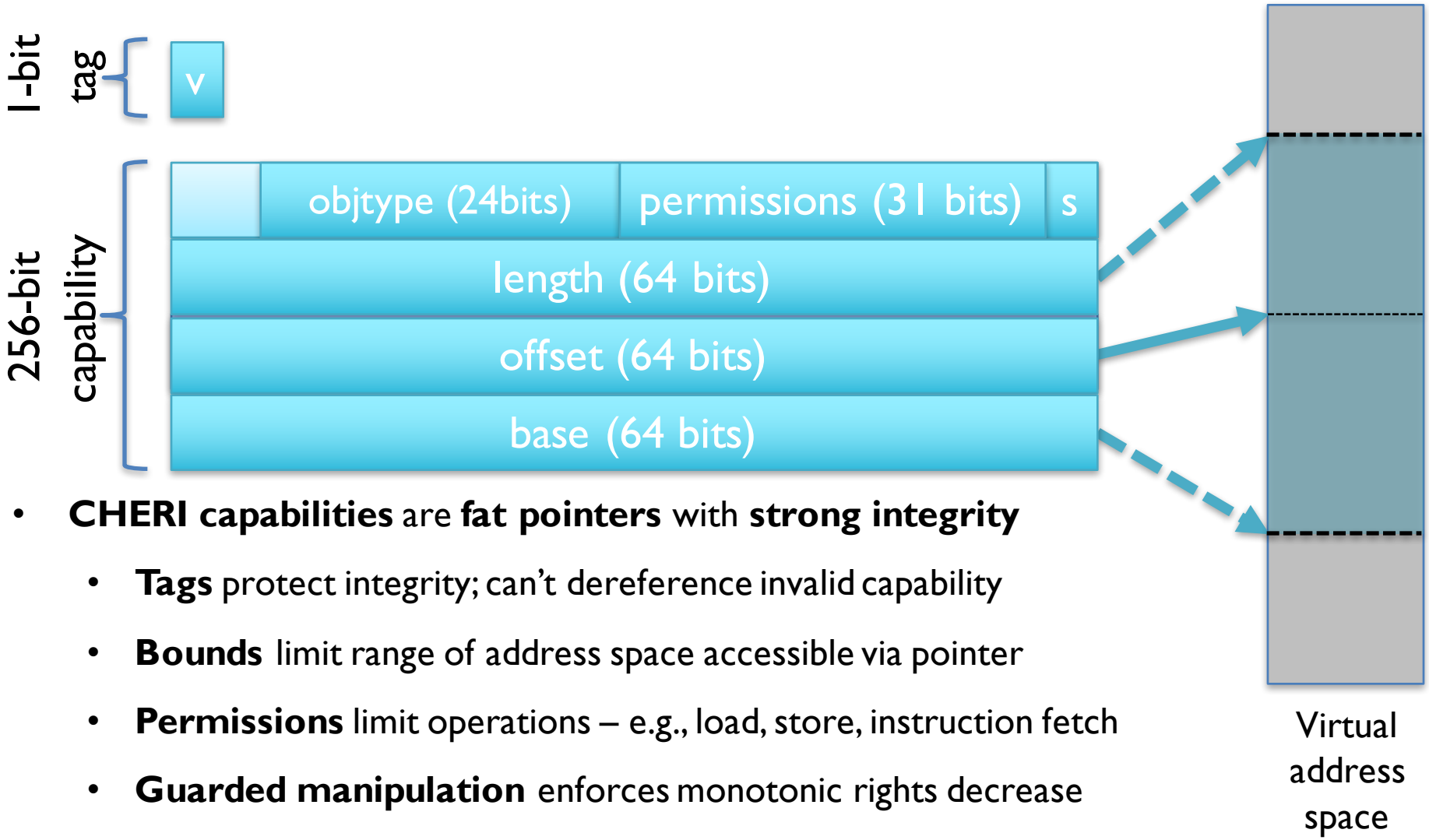
- CHERI permissions mark capabilities as **local** or **global**
- Local capabilities can only be stored via **store-local** capabilities
- In CheriBSD, non-garbage-collectable **object-local** and **stack capabilities** cannot be delegated between protection domains:
  - Heap capabilities are **global** and **!store-local**
  - Object/stack capabilities are **local** and **store-local**

Sealed capabilities



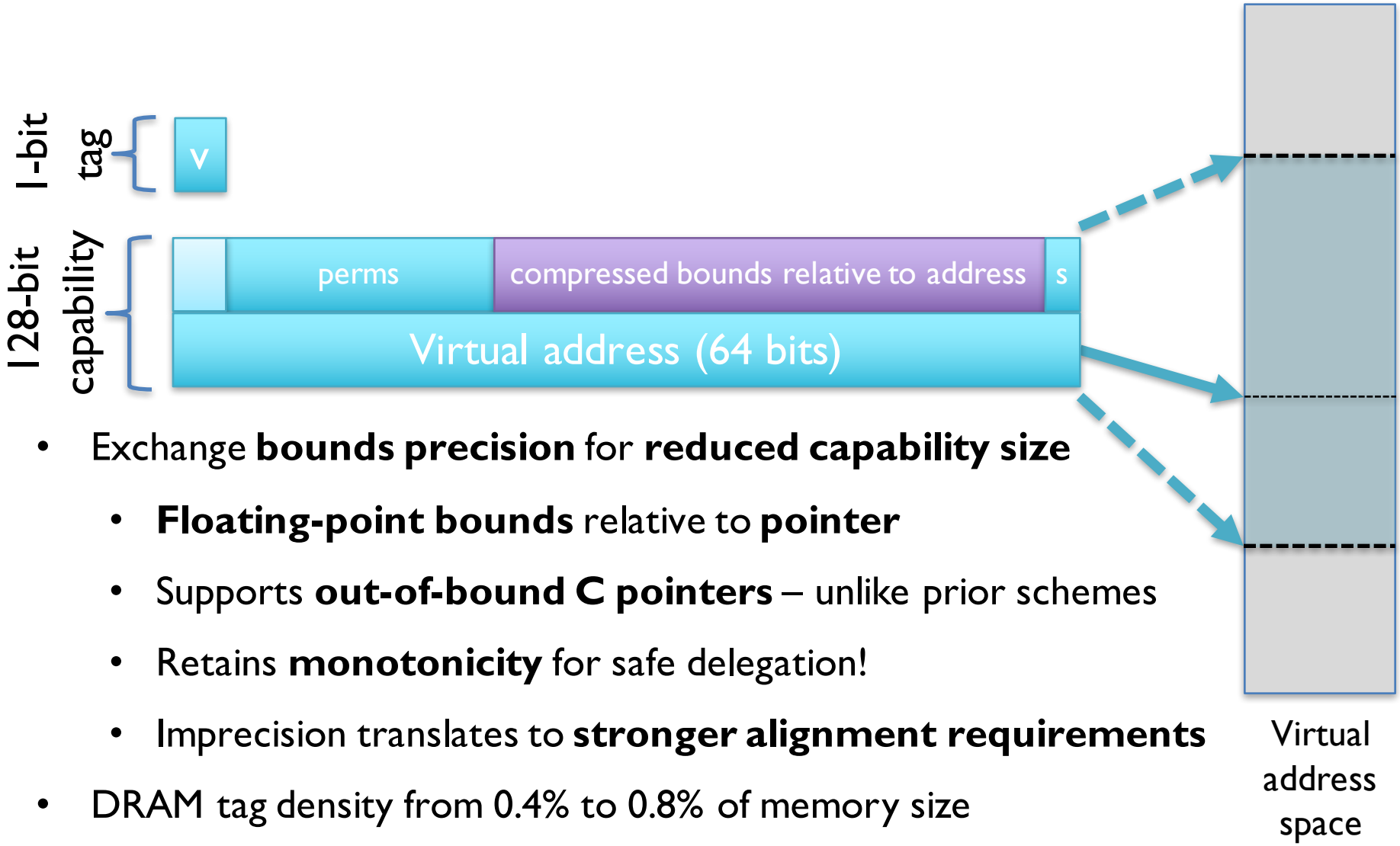
- **Sealed bit** provides **strong, software-defined encapsulation**
  - Sealed capabilities are **immutable**, **cannot be dereferenced**
- **Object types** atomically link multiple capabilities
  - **Object capabilities** pair **code** and **data capabilities**
  - Foundation for **secure hardware-software object invocation**

256-bit architectural capabilities



- **CHERI capabilities** are **fat pointers** with **strong integrity**
  - **Tags** protect integrity; can't dereference invalid capability
  - **Bounds** limit range of address space accessible via pointer
  - **Permissions** limit operations – e.g., load, store, instruction fetch
  - **Guarded manipulation** enforces monotonic rights decrease
- **Architectural description** not the **microarchitectural implementation**

128-bit micro-architectural capabilities



- Exchange **bounds precision** for **reduced capability size**
  - **Floating-point bounds** relative to **pointer**
  - Supports **out-of-bound C pointers** – unlike prior schemes
  - Retains **monotonicity** for safe delegation!
  - Imprecision translates to **stronger alignment requirements**
- DRAM tag density from 0.4% to 0.8% of memory size
- Fully functioning prototype with software stack on FPGA

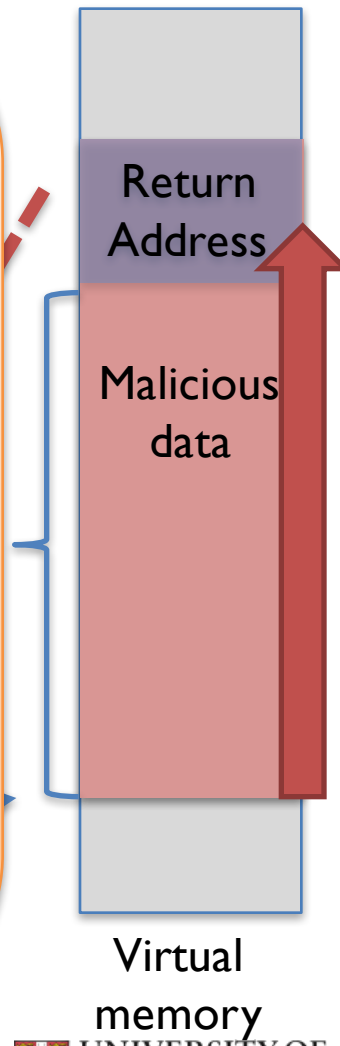
Architectural least privilege

CHERI memory protection:

- Eliminates out-of-bounds accesses
- Prevents injected data use as a code or data pointer
- Data pointers cannot be used as branch or jump targets
- Control-Flow Integrity (CFI) limits code-pointer reuse
- Scalable compartmentalization mitigates as-yet undiscovered attack techniques and supply-chain attacks

While:

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



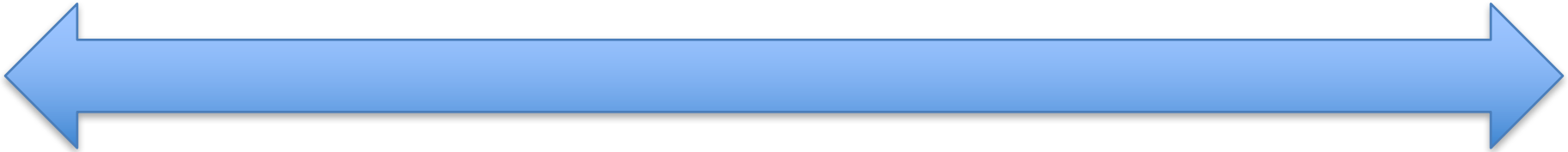
Virtual memory **and** capabilities

	Virtual Memory	Capabilities
Protects	Virtual addresses and pages	References (pointers) to C code, data structures
Hardware	MMU,TLB	Capability registers, tagged memory
Costs	TLB, page tables, lookups, shootdowns	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:
pick the **best** for each problem to solve!

CHERI software models

More compatible — Safer



Unmodified
All pointers are registers

Hybrid
Annotated and automatically selected pointers are capabilities

Pure-capability
All code and data pointers are capabilities

- **Source and binary compatibility**: common **C-language idioms,** various **ABIs**
  - **Unmodified code**: Existing n64 code runs without modification
  - **Hybrid code**: e.g., used solely in return addresses, for annotated data/code pointers, for specific types, stack pointers, etc.; n64-interoperable.
  - **Pure-capability code**: ubiquitous data-pointer protection, strong Control Flow Integrity (CFI). Non-n64-interoperable.
- **CHERI Clang/LLVM prototype** generates code for all three

CHERI technical reports

- **Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture**. (UCAM-CL-TR-876).
  - ISAv4 released in November 2015
  - ISAv4: experimental 128-bit capabilities, domain-switching optimizations, further C-language support; chapters describing software protection model
- **Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide**. (UCAM-CL-TR-877).
  - New document released in November 2015
  - Much more detail on compiler, OS internals
- New ISA specification due in May 2016: mature 128-bit capabilities, instructions for more efficient code generation

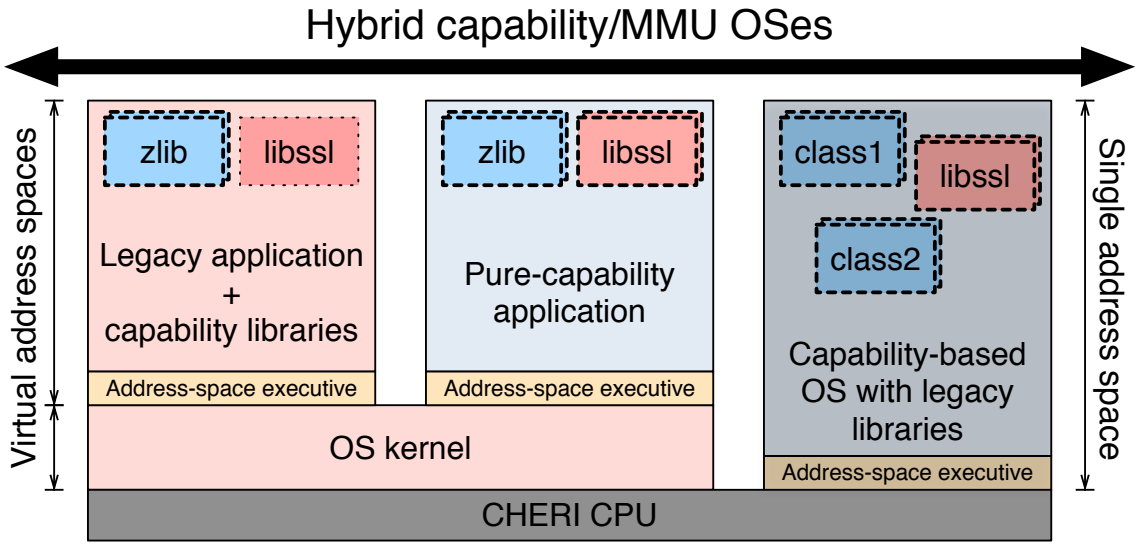
CHERI papers

- **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 and analysis
- **PLDI 2016**: C-language semantics and extension
- **IEEE Micro Journal 2016 submission**: Hardware assistance for efficient domain switching
- **IEEE Micro 2016 submission**: Compressed 128-bit capabilities for reduced cache footprint

Q&A

BACKUP SLIDES

CHERI OS considerations



- Prototyped on FreeBSD operating system (+/- 4 KLoC)
- Process model extended for tagged capabilities
  - Register-file setup and maintenance (exec, switch, thread create)
  - VM support for physical tags; signal handling; debugging
- Fine-grained, in-address-space object-capability security model
  - CCall/CReturn exception handlers; sandboxed syscalls restricted
  - Userspace compartmentalization runtime

CHERI instructions

Instruction class	Instructions
Inspect capabilities	CGetBase, CGetOffset, CGetLen, CGetTag, CGetSealed, CGetPerm, CGetType, CToPtr, CPtrCmp
Manipulate capabilities	CClearRegs, CIncOffset, CSetBounds, CSetBoundsExact, Cmove, CClearTag, CAndPerm, CSetOffset, CGetPCC, CFromPtr, CSub
Memory access to, and via, capabilities	CL[BHWD][U], CLC, CLLC, CLL[BHWD][U], CSCC, CS[BHWD], CSC, CSC[BHWD], CSSC
Control flow	CBTU, CBTS, CJR, CJALR
Sealed capabilities	CCheckPerm, CCheckType, CSeal, CUnseal, CCall, CReturn
Exception handling	CGetCause, CSetCause