CTSRD

CRASH-worthy Trustworthy Systems Research and Development

A Hardware-Software Total-System View of Trustworthiness

Peter G. Neumann

Chief Scientist, SRI International Computer Science Lab

Menlo Park CA 94025-3493

Neumann@CSL.SRI.COM

First Annual seL4 Summit

Dulles Airport Hilton, Virginia

14-16 November 2018

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

Outline of the talk

- An overview of the problems to be confronted: the current state of the art, and what is missing
- Past, recent, and current research and development advances
- Principles for designing and developing trustworthy systems
- Some lessons from experience
- Overview of the Capability Hardware Enhanced RISC Instructions (*CHERI*) hardware-software total system, ongoing joint work of the University of Cambridge and SRI
- Conceivable composition of CHERI hardware and seL4?
- Conclusions and relevant references

The Current State of the Art

- Most system and network implementations are not sufficiently trustworthy – with more than 109,000 common vulnerabilities (CVEs – cve.mitre.org), about 15,000 added thus far in 2018!
- Efforts to make operating systems/OS kernels/VMMs more secure are limited by micro-architectural compromises -- e.g., Spectre, Foreshadow[-NG], side channels, and inadequate DMA protection from embedded microcontrollers and malicious USB access.
- Relevant research in trustworthiness, quality software engineering, and formal methods is not finding its way into development.
- Gaps prevail in research, development practice, education, training.

What Is Missing?

- We need meaningfully trustworthy hardware on which to build much more trustworthy operating systems, more trustworthy networks, and sensible applications that take suitable advantage of such hardware and its supporting technology.
- We need much greater attention to software development in theory and practice, including realistic methodologies for applying formal methods, and proactive designs for trustworthiness.
- We need ubiquitous attention to advanced computer literacy, including K-12, college, graduate school, and ongoing training.

Some Significant Advances

- Multics (1965) was seminal, with protected segmentation, paging, rings, hierarchical directories, access-control lists, separate I/O coprocessor with controlled DMA, hardware-ensured stack buffer overflow avoidance, dynamic linking, Y2K avoidance(!), …
- Considerable past constructive R&D (Multics, capability-based systems incl. PSOS) has been largely ignored in practice (e.g., new hardware, total-system architectures, applied formal methods).
- Capsicum, seLinux, seL4 microkernel, GreenHills separation kernel, CertiKOS virtual-machine hierarchy are considerable software advances, with significant use of formal methods.
- CHERI's highly principled clean-slate hardware-software co-design is a potential fundamental advance, with extensive formal analysis of the hardware instruction-set architecture.

Trent Jaeger's talk extensively amplifies this slide, next after my talk.

Principles for Trustworthy Systems (1)

Overall Architectural Principles

1. Sound conceptual total-system architectures, with realistic implementability and compositional/layered assurance
2. Minimization of what must be trusted -- including avoidance or explicit mitigation of otherwise adverse dependence on components that may not be trustworthy
3. Open design (e.g., Kerckhoffs `Law' focusing on cryptographic keys & key management, assuming attackers know everything else but the secret keys) and open analyses

Based on Saltzer/Schroeder 1975 + Kaashoek 2009, Neumann CHATS 2004. CHERI Principles 2018

Principles for Trustworthy Systems (2)

Design and Implementation Principles, part one:

4. Least privilege (allocating only what is needed, and not more)
5. Intentional use (minimize confusions among multiple possible privileges)
6. Resilient dependency despite questionable components (cf. 2)
7. Composable modular abstractions with complete mediation, strict encapsulation, and precisely defined/layered interfaces)
8. Layered and predictably sound compositional assurance
9. Object and type integrity (for strongly typed objects)

Principles for Trustworthy Systems (3)

Summary of Design and Implementation Principles, part two:

10. Separation of domains, privileges (to facilitate least privilege), policy vs. mechanism (to facilitate evolvability, scalability, duplicated functionality), roles, duties (to partition superuser privileges), and compartments -- isolation with controlled sharing as required

11. Sound sufficiently fine-grained access controls, authentication, authorization, accountability (e.g., forensics-worthy monitoring and nonrepudiatable auditing), and administrative control

12. Usability, manageable complexity, hiding of security details except where essential

Principles Enhance Predictable Composition

- *Composition* is relevant to instruction-set architectures (ISAs), specs, hardware and software, requirements, source/object code,, layered/compositional formal analyses, …
- *Composability* ➔ preservation of properties. *Compositionality* ➔ acceptability of constructive emergent properties of a composition, perhaps remediation of undesirable ones.
- *Modular abstraction with strict encapsulation* greatly aids sound composition. Hierarchically layered assurance is essential to compositionality and total-system trustworthiness. Other principles are also beneficial to predictable composition.

Neumann CHATS 04 and Principles 2018, Mark Miller PhD Thesis 2006

Examples of Layered Assurance (1)

- SRI's Provably Secure Operating System (*PSOS*, 1973-1980 -- designed using SRI's Hierarchical Development Methodology, *HDM*) was an early example of a clean-slate hardware-software co-design, with formal specifications for 7 layers of hardware abstraction and 9 layers of the operating system. HDM could enable proofs of satisfaction of desired properties one layer at a time, up from the bottom layer, with formal interlayer abstract implementations and state mappings: *http://www.csl.sri.com/neumann/psos/psos80.pdf with subsequent revisiting: http://www.csl.sri.com/neumann/psos03.pdf*
- seL4's elegant hierarchical five-layer proof structure. Klein et al., ACM Trans. Computer Systems 2014, particularly Figure 3.

Examples of Layered Assurance (2)

- Zhong Shao's *CertiKOS* (2012 to date): clean-slate formally analyzed multi-layer secure virtual-machine operating system kernel, developed HDM-ish. *flint.cs.yale.edu/certikos/*
- *CHERI* (2010 to date): clean-slate formally specified capability-based hardware instruction-set architecture (ISA), with formal analysis of the ISA, can enable software analysis based on proofs of properties of the hardware specifications. Analysis of consistency of the ISA with hardware implementations and supply-chain trustworthiness must also be considered. More on the CHERI ISA and system architecture follows. *cl.cam.ac.uk/research/security/ctsrd/cheri/*

Some Lessons from Experience (1)

- *Absence or oversimplification* of well-vetted system requirements, system architectures, design specifications, appropriate design and development tools, and proactive consideration of assurance often leads to serious structural impediments to trustworthiness.
- *Widespread disregard for complexities* associated with achieving total-system trustworthiness and predictably sound compositions of modules, subsystems, networking, proofs, etc., defeat mistaken beliefs that security can be added later!
- *Intrinsically flawed or otherwise inadequate hardware* can undermine improvements in operating systems, compilers, and tools for software development and analysis.

Some Lessons from Experience (2)

- *Highly principled developments* of hardware and software have enormous long-term advantages over conventional practice, with respect to evolvability, adaptability, and especially trustworthiness. They are also likely to be more amenable to formal analysis.
- *Judicious use of formal methods* in specifying and analyzing functional hardware specifications is now realistic, with proofs of consistency with formally specified requirements, along with their applicability to low-layer software. Advances in theorem provers, model checkers, and satisfiability-modulo-theory analyzers (SMT solvers) have progressed significantly to support such efforts.

CTSRD

CRASH-worthy Trustworthy Systems Research and Development

CHERI: Architectural Support for Memory Protection and Compartmentalization

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

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 Stolfa, Andrew Turner, Munraj Vadera, Jonathan Woodruff, Hongyan Xia, and Bjoern A. Zeeb

SRI International and the University of Cambridge

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.

DARPA CRASH Program

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

…what would you change?

Howie Shrobe, DARPA I2O, 2010 (now once again at MIT)

Principle of least privilege

Every program and every privileged user of the system should operate using the **least amount of privilege** with **finest granularity necessary for the given purpose**.

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

Principle of Intentional Use

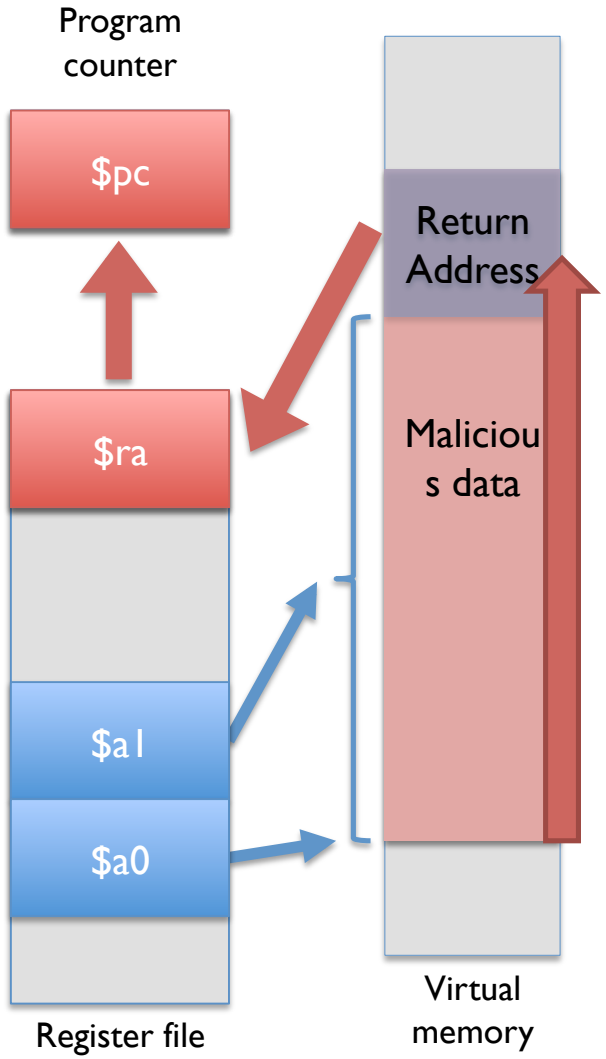
Authorize only what is really intended, not what is merely expedient or overly broad.
Privileges should be allocated explicitly according to situational needs, e.g., avoiding implicit opportunities for potential confusions when selecting among commingled simultaneously held multiple privileges.

Fundamental to CHERI; Norman Hardy, The Confused Deputy – ACM SIGOPS Review, November 1988.

(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.**



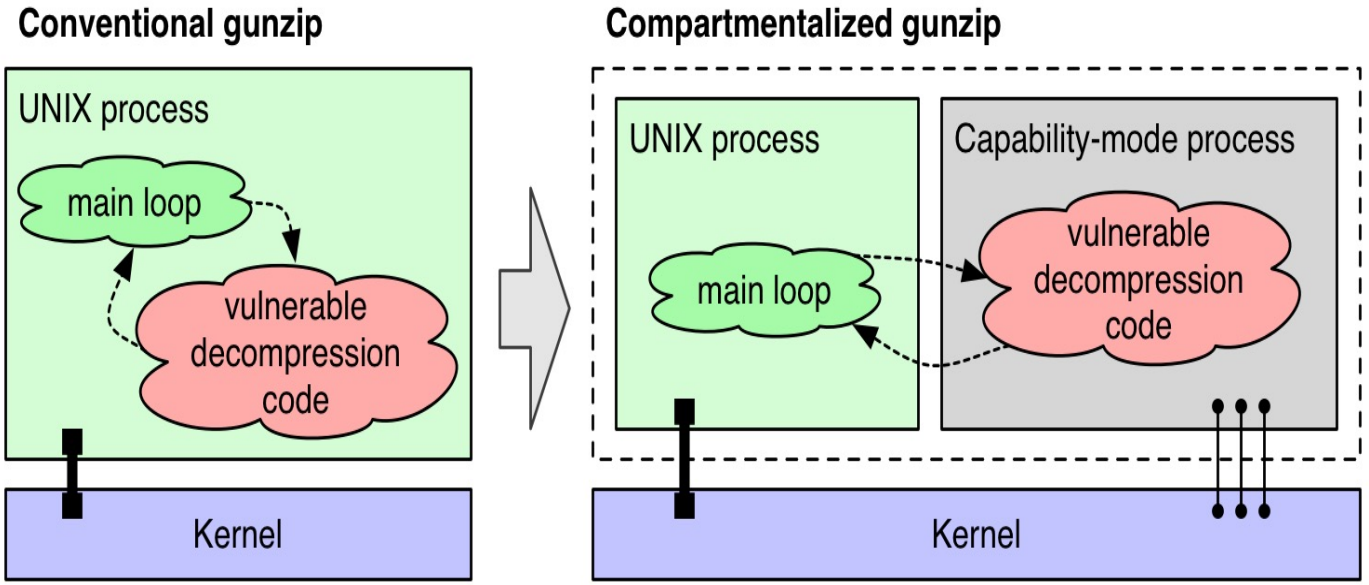
Motivation – The Eternal War in Memory*

- Many security vulnerabilities exploit memory safety violations.

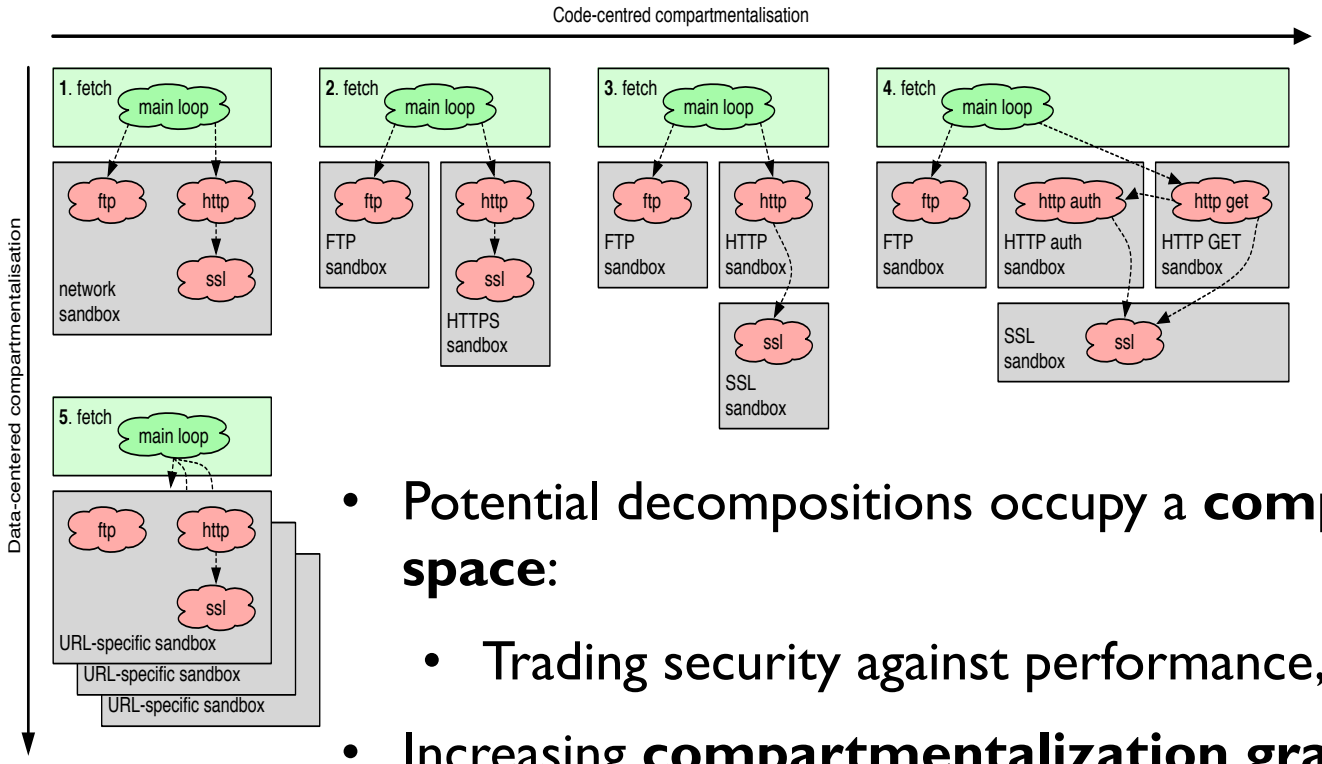
* Oakland 2013 paper title: SoK: Eternal War in Memory, László Szekeres, Mathias Payer, Tao Wei, Dawn Song

Application-level least privilege (1)

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



Can mitigate unknown vulnerabilities and **as-yet undiscovered classes of vulnerabilities and exploits.**



- Potential decompositions occupy a **compartmentalization space**:
  - Trading 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 AND ARCHITECTURE

CHERI design goals and approach (1)

- **Architectural security** to mitigate **C/C++ TCB vulnerabilities**
  - Efficient primitives allow software to ubiquitously employ the **principle of least privilege** & **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.

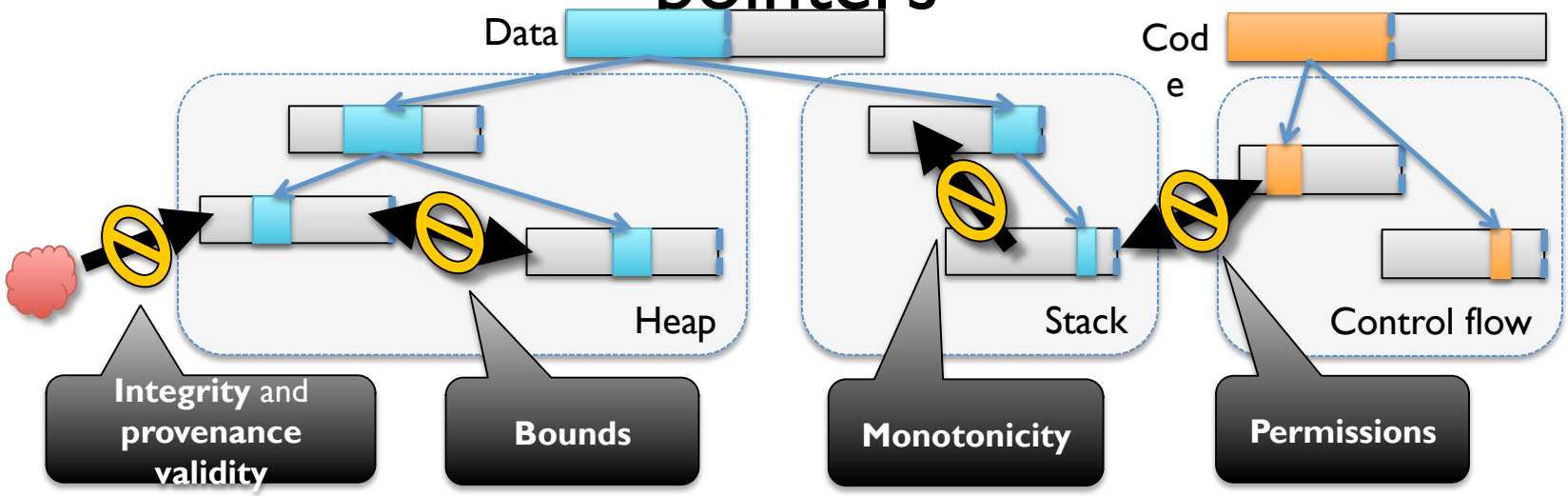
CHERI design goals and approach (2)

- **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), object types

CHERI software protection goals

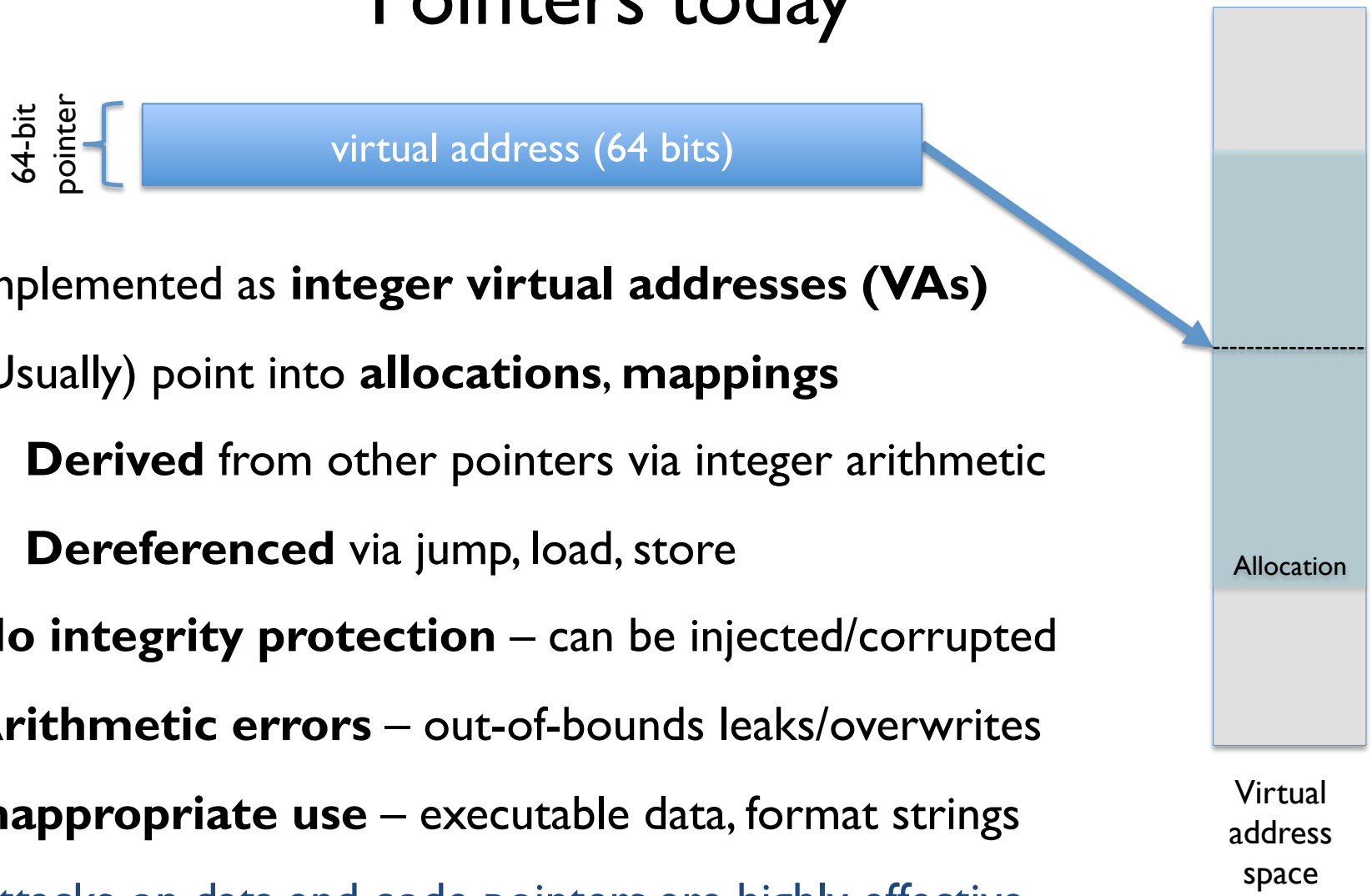
- **C/C++-language TCBs:** kernels, language runtimes, browsers, …
- **Granular spatial memory protection, pointer protection**
  - Buffer overflows, control-flow attacks (ROP, JOP), …
- **Foundations for temporal safety**
  - E.g., accurate C-language garbage collection
- **Higher-level language safety**
  - Safe interfaces to native code (e.g., impose Java memory safety on JNI – see slide 43)
  - Efficient memory safety (e.g., HW assist on bounds checking)
- **Scalable in-process compartmentalization**
  - Facilitate exploit-independent mitigation techniques

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.

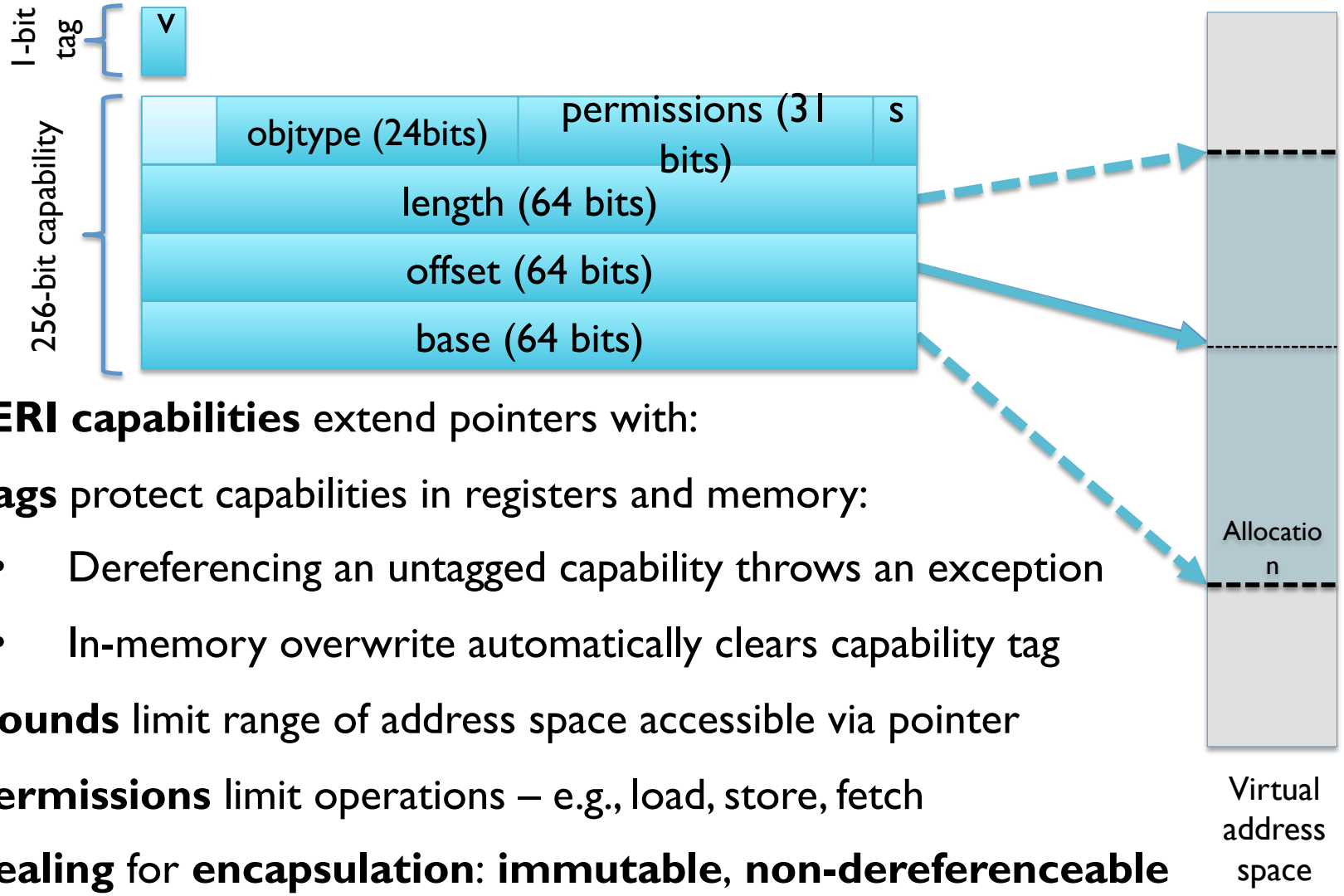
Pointers today



- 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**

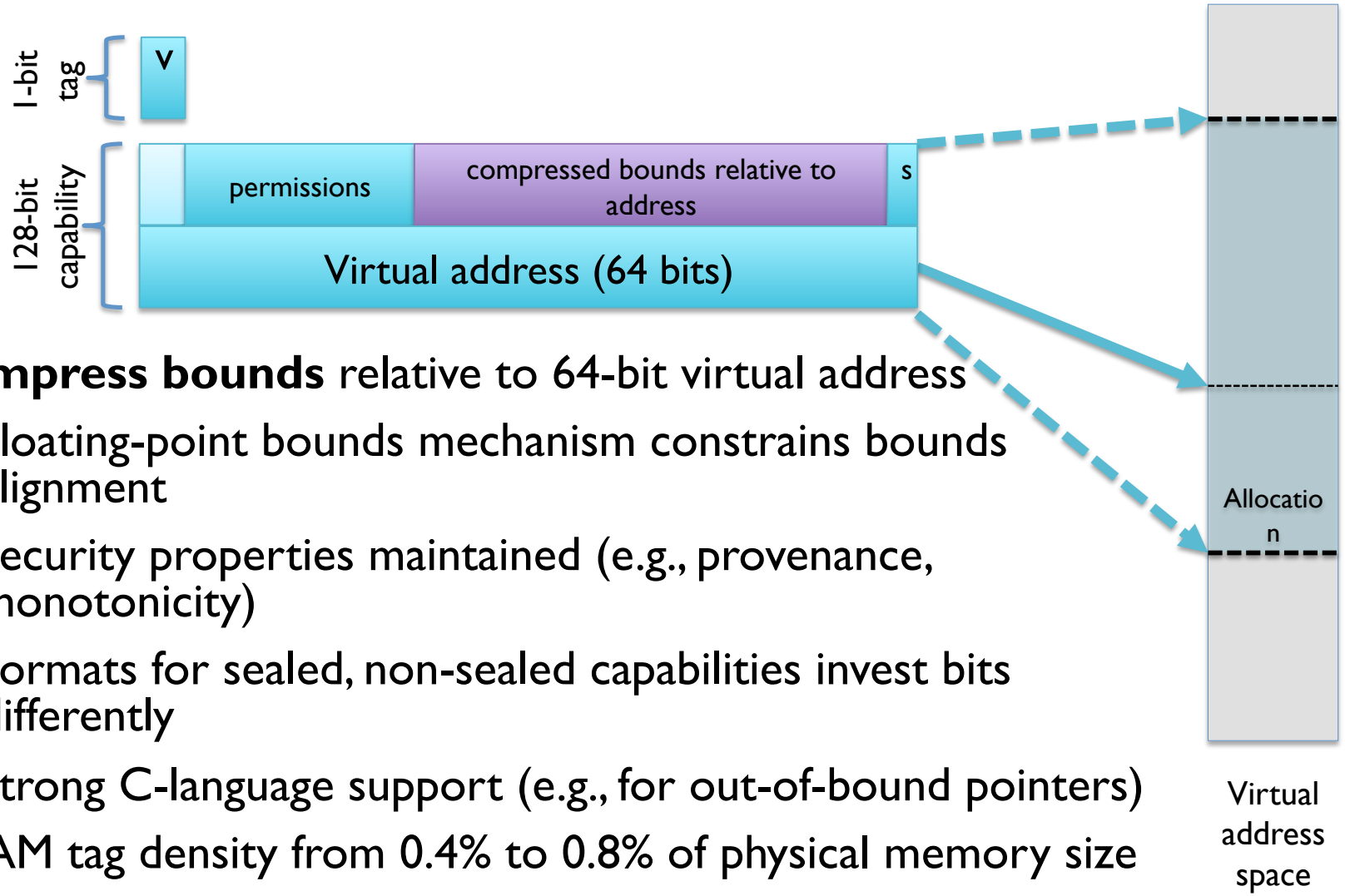
Protection model: 256-bit capabilities



CHERI capabilities extend pointers with:

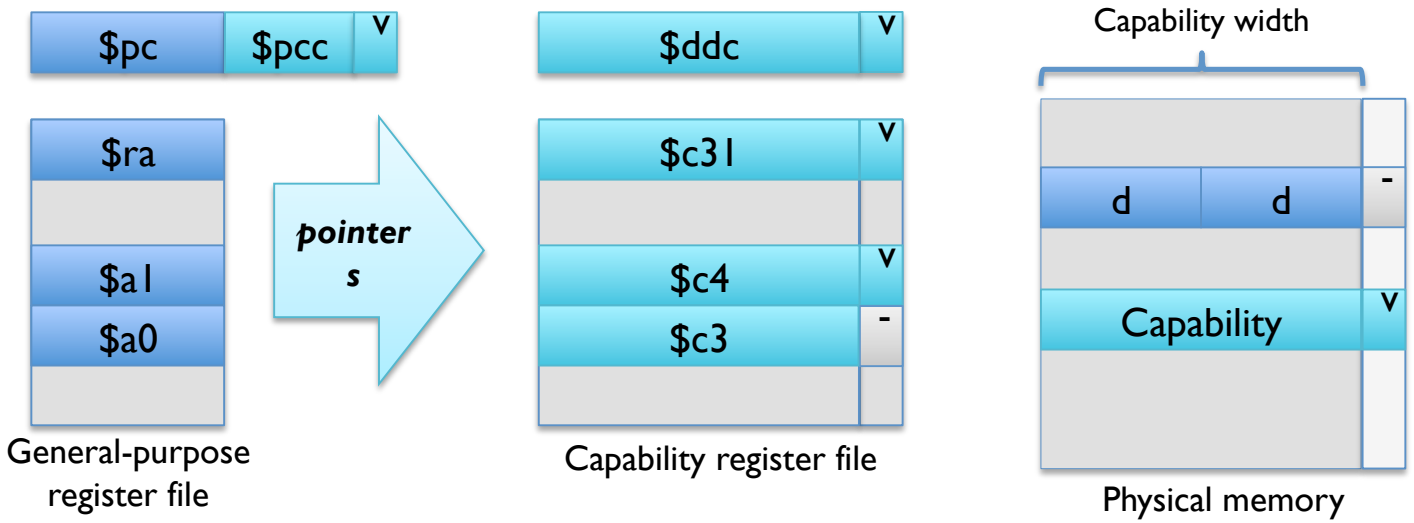
- **Tags** protect capabilities in registers and memory:
  - Dereferencing an untagged capability throws an exception
  - In-memory overwrite automatically clears capability tag
- **Bounds** limit range of address space accessible via pointer
- **Permissions** limit operations – e.g., load, store, fetch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

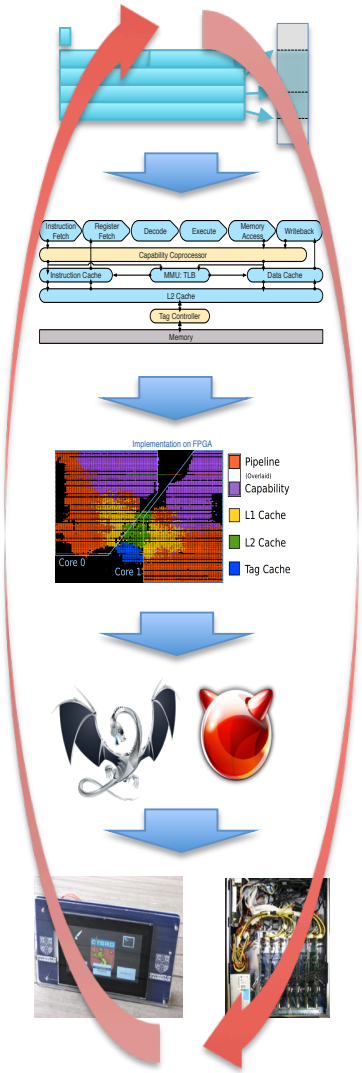
Mapping CHERI into 64-bit MIPS



- **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**

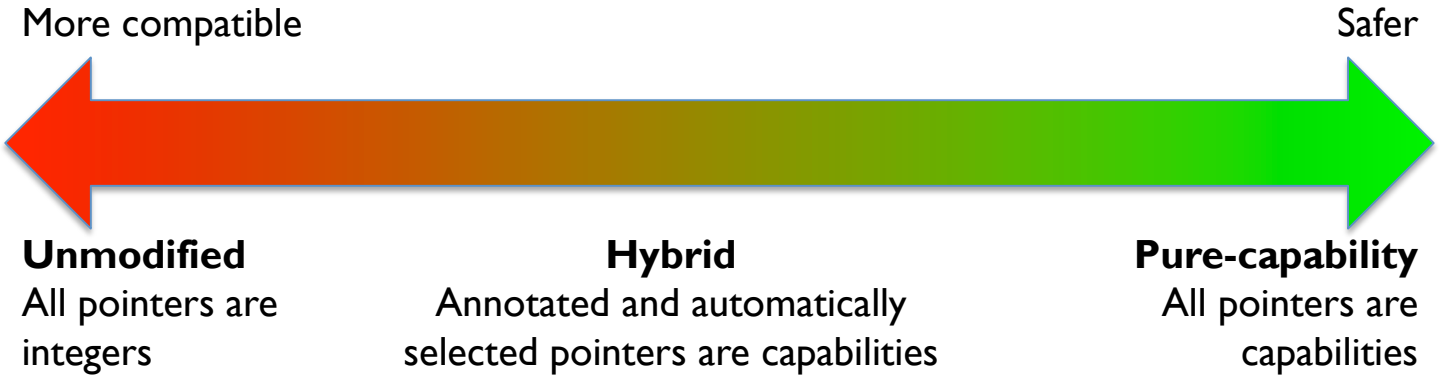
HARDWARE-SOFTWARE CO-DESIGN FOR CHERI

CHERI hardware-software co-design since 2010



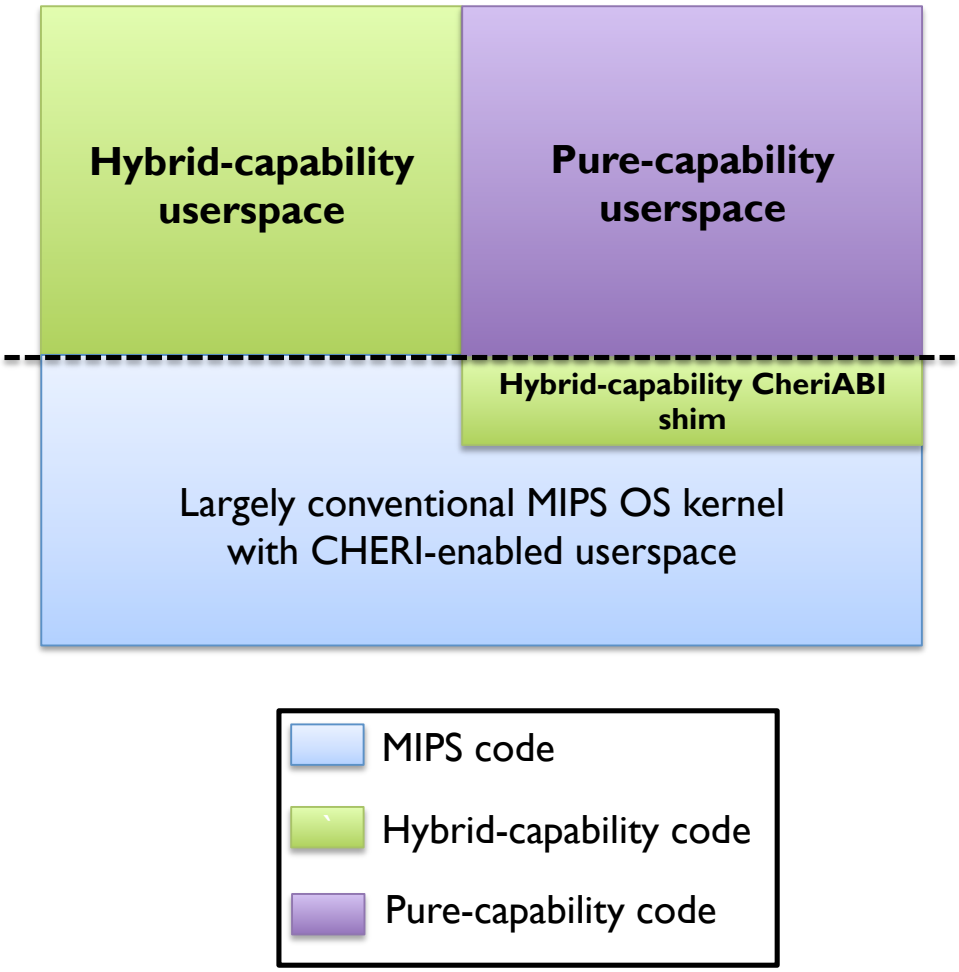
- SRI and Cambridge joint work over three DARPA projects (two new)
- Architectural mitigation for C/C++ TCB vulnerabilities
  - Tagged memory, capability pointer representation
  - Fine-grained pointer and memory protection
  - Highly scalable software compartmentalization
  - Hybrid capability system 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
  - CHERI-MIPS ISA formal model, Qemu-CHERI, FPGA prototypes
  - CHERI Clang/LLVM, CheriBSD OS, C/C++-language applications
  - Repeated iteration to improve {overhead, security, compatibility, ..}

Low-level CHERI software models



- **Source, binary compatibility**: **C-language idioms,** multiple **ABIs**
  - **Unmodified code**: Existing object code runs without modification.
  - **Hybrid code**: E.g., used in return addresses, for annotated data/code pointers, for specific types, stack pointers, etc. (However, "hybrid" is a spectrum: many different choices for manual and automatic selection of integers vs. capabilities, API and ABI impacts.)
  - **Pure-capability code**: Ubiquitous data- and data-pointer protection, but not interoperable with legacy code due to changed pointer size.
- **CHERI Clang/LLVM compiler prototype** generates code for each.

Hybrid-capability code to pure-capability code

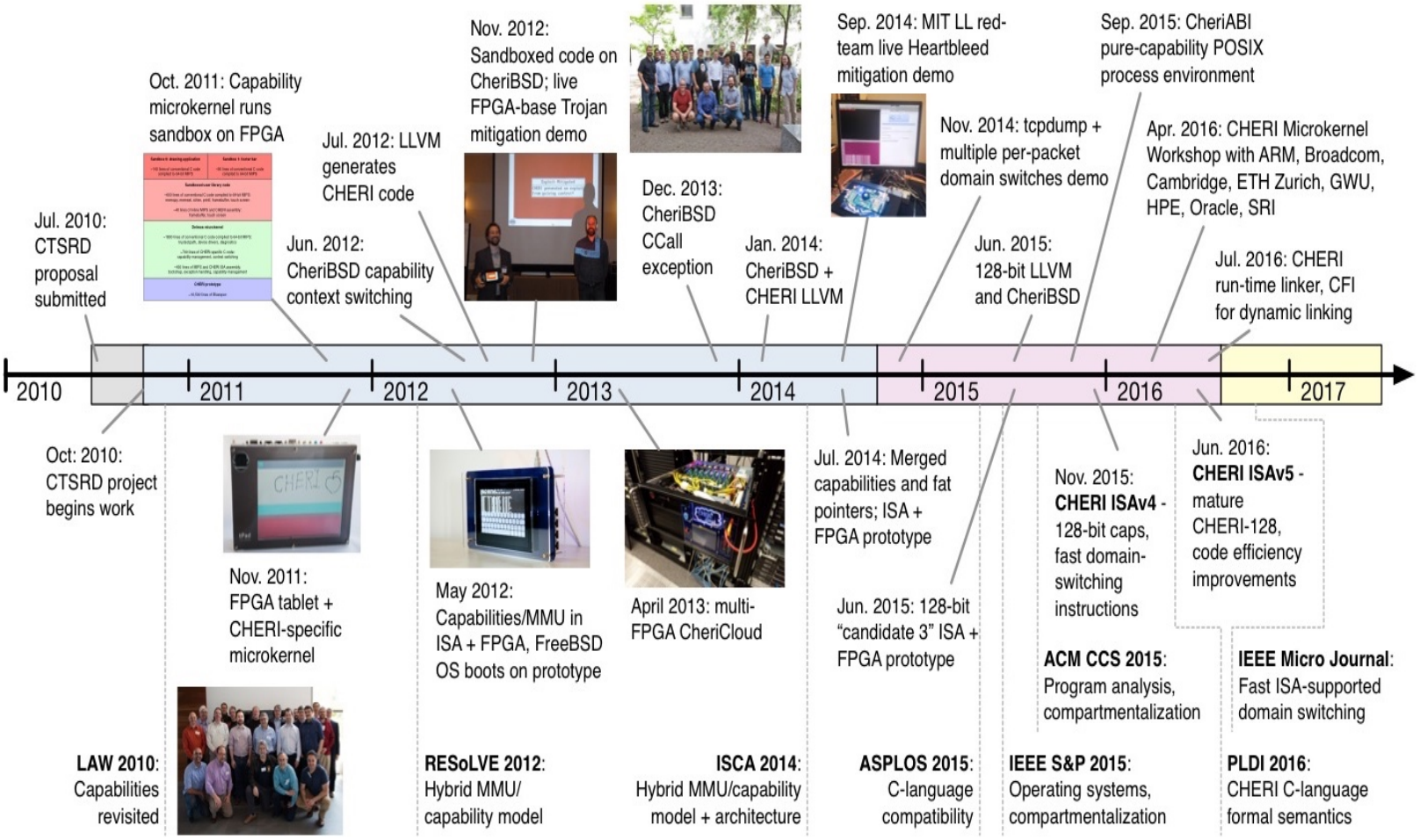


- **n64 MIPS ABI:** hybrid capabilities
  - **Early investigation** – manual annotation and C semantics: *Many* pointers/implied VAs are integers (including syscall arguments,)
- **CheriABI:** pure-capability code
  - **The past two years** – fully automatic use of capabilities wherever possible
  -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

Pure Capability Code → Needs CheriABI

- CheriABI goals
  - Compatibility layer to the OS
  - Allow capabilities to be used in place of pointers
  - Somewhat like a 32-bit compatibility layer for a 64-bit OS
- Result: We can now recompile large corpuses of C code into a pure capability form, with almost no source code changes.
- Paper: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment (in review)

CHERI research and development timeline



Years 1-2: Research platform, prototype

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

Years 4-7: Efficiency, software stack at scale

CHERI ISAv6 in 2017; **CHERI ISAv7** 2018

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?

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

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

*Test failure investigation remains a work in progress; believed resolvable

Evaluating memory-protection impact

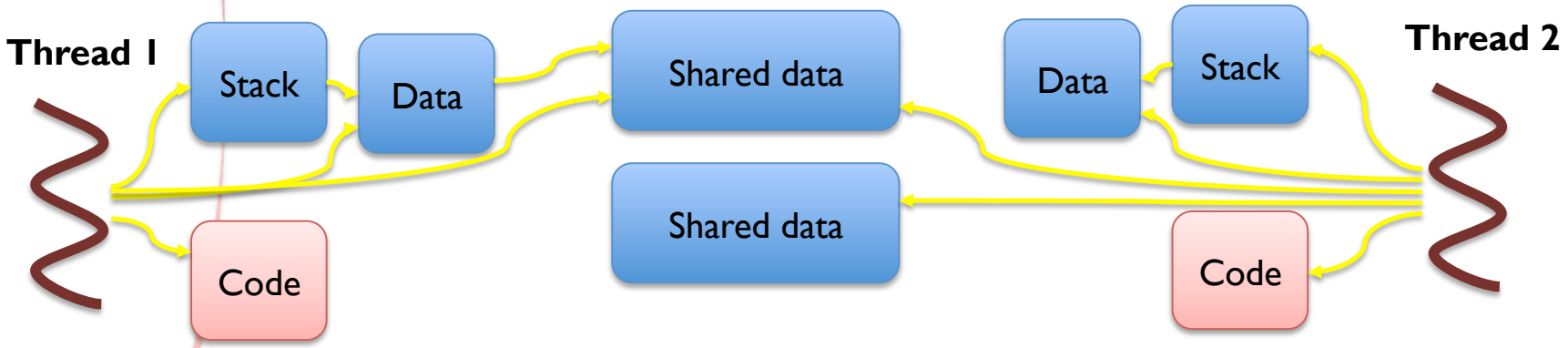
- Adversarial / historical vulnerability analysis
  - ✓Pointer integrity, provenance validity prevent ROP, JOP
  - ✓Buffer overflows: Heartbleed (2014), Cloudbleed (2017)
  - ✓Pointer provenance: Stack Clash (2017)
- Existing test suites – e.g., BOdiagsuite (buffer overflows)
- **Recently: microarchitectural side channels such as Meltdown/Spectre**

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

SOFTWARE COMPARTMENTALIZATION

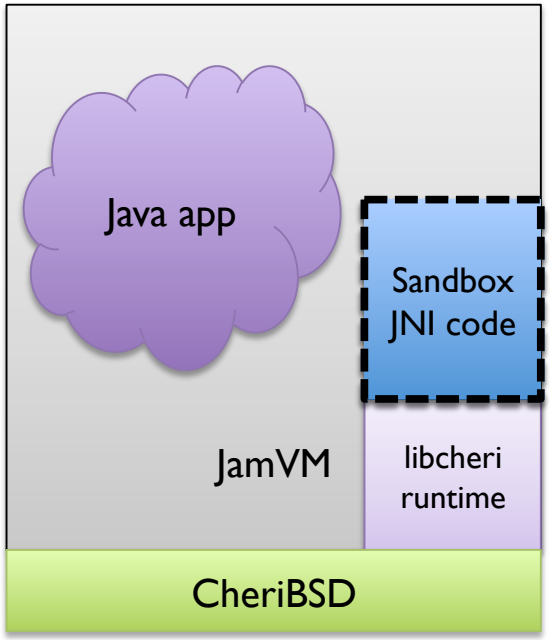
Principles of CHERI compartmentalization (SSP 2015)

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



- **Compartmentalized models (classes, objects, sharing, object invocations)**

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
- Java **memory-safety and security models** for 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

Ongoing CHERI-related 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"

Related HW-SW security research projects

- EPSRC IOSEC – Research into I/O-originated adversaries (Cambridge)
- DARPA MTO ECATS: CHERI and full SoCs (SRI, Cambridge, ARM Research)
  - CHERI-RISC-V
  - CHERI for 32-bit microcontrollers
  - CHERI interactions with DMA and heterogeneous computing
  - Containing untrustworthy IP cores in CHERI-aware SoCs
- DARPA I2O/MTO CIFV – Formal modeling and reasoning (SRI, Cambridge)
  - Formal models of CHERI instruction-set architectures
  - Formal verification of CHERI architectural security properties

Conclusions Relating to CHERI

- 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 ↔ architecture ↔ 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 have other benefits (e.g., in microarchitecture?).

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

More Conclusions Relating to CHERI

- CHERI enables hardware with more semantic knowledge of what programmers actually intend, with the **principle of intentional use.**
- Enables efficient **pointer integrity** and **bounds checking,** eliminating bounds overflow/over-read attacks (finally!).
- Provides scalable and efficient compartmentalization: the **principle of least privilege** helps **mitigate known and unknown attacks;** big potential performance improvement over process-based compartmentalization.
- We are working with industry and the RISC-V community to bring the technology to realization and to market.
- Thanks to sponsors: **DARPA I2O/AEO/MTO**, **Arm**, **Google**, EPSRC, HEIF, Isaac Newton Trust, Thales E-Security, Google DeepMind, HP Labs -- and to the entire set of project team members.

COMPOSING CHERI, seL4, RISC-V???

(FOUR CONTEMPLATIVE SLIDES)

Possible Paths to Total-System Security

- Total-system assurance could benefit significantly from something like SRI's Hierarchical Development Methodology, with formally specified hardware, and analyses from the hardware specs up through operating system kernels and applications. (Fabrication consistency and supply-chain issues must still be considered.)
- Such formally based efforts are now beginning to approach feasibility today, considering advances such as the CHERI ISA proofs in progress, the seL4 proofs, the CertiKOS layered architecture, and the Green Hills kernel evaluation for Common Criteria EAL6+.

Composing seL4 with CHERI hardware?

- *seL4* has relegated much of user-critical trustworthiness to user space, making it particularly relevant to embedded systems in which security of application code could be proven statically. However, seL4's overall security also depends on the trustworthiness of the hardware ISA spec and its hardware implementations.
- *CHERI*'s formal analyses of ISA security properties can facilitate increased trustworthiness in OSs and user code – assuming that its hardware implementation is sufficiently consistent with the ISA.
- Intuitively, it would seem that a careful integration of both might provide total-system trustworthiness, especially for embedded systems. However, knowing that trustworthiness is inherently challenging, we need to explore the details of such an approach.

CHERI-Enhanced seL4: CE-seL4/RISC-V ?

Certain limitations restricting more general applicability of seL4 could be eased by the presence of a more trustworthy ISA – such as CHERI.

- *CHERI-enhanced seL4/RISC-V* could yield greater trustworthiness for untrusted applications, exploiting CHERI's capability-based least privilege and compartmentalization, CHERI-compliant LLVM compiler, and ongoing development of interposed DMA controls for potentially untrusted I/O and embedded active devices, with CHERI ISA proofs leading to formal analysis of user-space security.
- Emerging CHERI DMA capability extensions could provide improved protections for the seL4 kernel itself, increasing total-system assurance – even with untrusted USB-like hardware devices.

CHERI-Enhanced seL4: CE-seL4/RISC-V (2)

Additional considerations and opportunities

- *Performance improvements* in seL4 application use cases, even without CHERI compartmentalization for untrustworthy applications. Examples: fine-grained within address space app compartments; seL4 IPC adopting CHERI-based isolation & communication primitives rather than MMU-based ones.
- *Conclusion:* CE-seL4/RISC-V is worth exploring further. With formal analysis and carefully documentation, it could inspire development of more trustworthy hardware-software systems in the future. It could also help improve seL4, CHERI, and RISC-V as a byproduct!

Learning more about CHERI

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

Watson, Moore, Neumann, et al. **Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7)**, Technical Report UCAM-CL-TR-927, Cambridge Computer Laboratory, November 2018.

Watson, Moore, Neumann, 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.

Two new papers, accepted in November

A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson, **Thunderclap: Exploring Vulnerabilities of Operating-System IOMMU Protection to DMA from Untrustworthy Peripherals,** NDSS 2019, San Diego CA, 24-27 February 2019.

Brooks Davis et al., **CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment,** ASPLOS 2019, Providence, Rhode Island, 13-17 April 2019.

Additional References

CHERI-relevant papers and architecture report are at cheri-cpu.org:
http://www.cl.cam.ac.uk/research/security/ctsrd/cheri/

Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, L.Robinson, PSOS 1980; http://www.csl.sri.com/neumann/psos/psos80.pdf and PSOS Revisited **http://www.csl.sri.com/neumann/psos03.pdf**

Neumann, CHATS 2004: http://www.csl.sri.com/neumann/chats04.pdf

Neumann, Principles 2018, **Fundamental Trustworthiness Principles in CHERI,** Chapter 6, *New Solutions for Cybersecurity* (Shrobe, Shrier and Pentland, eds.), MIT Press, 2018.

More relevant background: http://www.csl.sri.com/neumann/