CHERI
A Hybrid Capability-System Architecture


University of Cambridge, SRI International

ETH Zurich / NewOS Workshop – 16-17 February 2016
Motivation

The Eternal War in Memory*

Example bug: **Heartbleed**

…allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users.

Yet another memory safety bug!

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)
Principle of least privilege (2)

• **Access control**
  • Minimize privileges held by users (and hence their processes) in accordance to policy

• **Fault tolerance**
  • Limit the impact of software/hardware faults

• **Vulnerability and Trojan mitigation**
  • Constrain rights gained as a result of software supply-chain compromise (Karger IEEE S&P 1987)
  • Motivation for sandboxing, privilege separation, and software compartmentalization used to mitigate vulnerabilities in contemporary applications
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:
  • 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?
• Wouldn’t eliminate the bug – but would provide effective vulnerability mitigation
Software compartmentalization decomposes software into **isolated compartments** that are delegated **limited rights**

Able to mitigate not only unknown vulnerabilities, but also **as-yet undiscovered classes of vulnerabilities/exploits**!
Application-level least privilege (2)
### Compartmentalization options for software

- Each trade off security against performance and programming complexity

### But MMU-based processes are problematic:
- Poor spatial protection granularity
- Limited simultaneous-process scalability
- Multi-address-space programming model
REVISITING RISC IN AN AGE OF RISK
CTSRD: Revisiting the hardware-software interface for security

Oct. 2010: Project starts

Oct. 2011: Capability microkernel runs sandbox on FPGA

Jun. 2012: CheriBSD capability context switching

Jul. 2012: LLVM generates CHERI code

Nov. 2012: Sandboxed code on CheriBSD; trojan mitigation demo

Dec. 2013: Lightweight CheriBSD domain switching

Nov. 2014: tcpdump uses multiple domain switches per packet

Jan. 2014: CheriBSD + CHERI LLVM

Jun. 2015: 128-bit LLVM and CheriBSD

Sep. 2015: CheriABI helloworld

LAW 2010: capabilities revisited

RESolVE 2012: hybrid capability-system model

ISCA 2014: hybrid MMU/capability model, architecture

ASPLOS 2015: C-language compatibility

IEEE S&P 2015: operating systems, compartmentalization

ACM CCS 2015: program analysis, compartmentalization
A hybrid capability-system model

• De-conflate **virtualization** and **protection**

• Retain **Memory Management Unit (MMU)** to implement (and protect with) **virtual addresses**
  • OS processes, machine 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-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 model offers strong binary and source-code compatibility
CHERI ISA-level features

- **RISC**: simple, compiler-focused ISA extensions avoid microcode and table walking
- **C pointers** map cleanly into ISA-level capabilities
- **Tagged capabilities** protect code and data pointer integrity in registers and memory
- **Pointer metadata**, including **bounds** and **permissions**, limits undesired (re-)use
- **Guarded manipulation** implements **capability monotonicity** and **sealing** for **least privilege**
- **256-bit architectural model**; unpublished efficient 128-bit micro-architectural implementation
### CHERI architectural elements

- **Tagged memory** protects capability-sized words in DRAM as pointers
- **Capability register file** holds in-use capabilities (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
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
Tags for integrity and provenance

- Capability register tags indicate valid capabilities
  - Untagged dereferences throw CPU exceptions
- Tagged memory retains tags when loaded/stored
  - Implement pointers embedded within 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 / 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
Sealed capabilities

• **Sealed bit** provides **strong, software-defined encapsulation**
  - Enforce a **software TCB-defined calling convention**
  - 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**

Virtual address space
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 **micro-architectural** implementation
128-bit micro-architectural capabilities

- Exchange **bounds precision** for **reduced capability size**
  - **Floating-point**(-like) bounds relative to pointer
  - Supports **out-of-bound C pointers** – unlike prior schemes
  - Retains **monotonicity** for safe delegation!
  - Care required with **security-imprecision trade offs**
- DRAM tag density from 0.4% to 0.8% of memory size
- Fully functioning prototype with software stack on FPGA

Virtual address space
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
SOFTWARE DEPLOYMENT
# Virtual memory and capabilities

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

**CHERI hybridizes the two models:** pick the **best** for each problem to solve!
Binary and source-code compatibility

More compatible

Safer

N64
All pointers are registers

Hybrid
Some pointers are capabilities; e.g., annotated data pointers, stack and/or code pointers

Pure-capability
All code and data pointers are capabilities

• **Hybrid code**: annotated use for data/code pointers, automatic use in return addresses, some stack pointers, etc.; N64-interoperable.

• **Pure-capability code**: ubiquitous data-pointer protection, strong Control Flow Integrity (CFI). Non-N64-interoperable.

• **Strong C-language compatibility**: capabilities are designed to represent pointers, support almost all common C-language idioms

• **CHERI Clang/LLVM prototype** supports both code models
Software deployment models

Hybrid capability/MMU OSes

- Legacy application + capability libraries
  - Virtual address spaces
  - Address-space executive
  - OS kernel
  - CHERI CPU

- Pure-capability application
  - Virtual address spaces
  - Address-space executive
  - OS kernel
  - CHERI CPU

- Capability-based OS with legacy libraries
  - Single address space
  - Address-space executive
  - CHERI CPU

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

Single-address-space systems are possible but not our focus
Capability-aware system-call ABI

- CheriBSD kernel implemented the 64-bit MIPS ABI
  - Hybrid-ABI shims within processes
- **CheriABI** adds pure-capability syscall ABI, C runtime, libraries
  - Pure-capability userspace binaries
  - Majority of C-language FreeBSD userspace “just works” – e.g., SSH!
  - Support for many more pure-capability applications/benchmarks
- **Ubiquitous memory protection for critical TCBs**
COMPARTMENTALIZATION
In-process object-capability model

- Intra-process protection domain
  - Capability register file contents
  - Transitive closure of capabilities
- Domain transition
  - Per-thread capability register-file transformation ("Call", "Return")
- libcheri implements classes, objects
  - Encapsulation, mutual distrust
  - Objects are sealed code + data capabilities with identical types
- Capability arguments / return values allow efficient delegation
Object-capability call and return

- **Default object** has ambient authority: full address space and system calls

- **Compartmentalization runtime** constructs constrained objects with explicitly delegated rights

- **Synchronous function-call-like** \( \text{CCall/CReturn} \) supports current application/library interfaces

- **Trusted stack** stitches together call chains of mutually distrusting objects

- **CCall/CReturn ABI** clears unused registers to prevent data/capability leakage between objects
# Application implications

## Pros

- Single address-space programming model
- Referential integrity matches programmer model
- Only modest work to insert protection-domain boundaries
- Objects permit mutual distrust
- Constant (low) overhead relative to function calls even with large memory flows

## Cons

- Still have to reason about the security properties
- Shared memory is more subtle than copy semantics
- Capability overhead in data cache is real and measurable
- ABI subtleties between MIPS and CHERI compiled code
- Lower overhead raises further cache side-channel concerns
VALIDATION AND REFINEMENT
CTSRD: Revisiting the hardware-software interface for security

Oct. 2010: Project starts

Oct. 2011: Capability microkernel runs sandbox on FPGA

Jun. 2012: CheriBSD capability context switching

Jul. 2012: LLVM generates CHERI code

Nov. 2012: Sandboxed code on CheriBSD; trojan mitigation demo

Dec. 2013: Lightweight CheriBSD domain switching

Nov. 2014: tcpcdump uses multiple domain switches per packet

Jan. 2014: CheriBSD + CHERI LLVM

Jun. 2015: 128-bit LLVM and CheriBSD

Sep. 2015: CheriABI helloworld

Nov. 2015: 128-bit CHERI ISAv4 specification


LAW 2010: capabilities revisited

RESolVE 2012: hybrid capability-system model

ISCA 2014: hybrid MMU/capability model, architecture

ASPLOS 2015: C-language compatibility

IEEE S&P 2015: operating systems, compartmentalization

ACM CCS 2015: program analysis, compartmentalization
**CHERI I experimental prototype**

- **Hardware:**
  - 64-bit MIPS + CHERI ISA extensions
  - Formal ISA model (in Cambridge L3)
  - BSV HDL prototypes (FPGA target)
  - Pipelined, L1/L2 caches, MMU, multicore
  - Capability extensions, tagged memory
  - 256-bit and 128-bit prototypes

- **Software:**
  - CheriBSD operating system
  - CHERI clang/LLVM compiler
  - Adapted applications
  - Open-source HW and SW
• ‘Capability coprocessor’ provides capability registers, instructions
• $\ddc, \pcc$ interpose on MIPS load/store ISA, instruction fetch
• Processing ‘before’ MMU makes capabilities address-space relative
• Tag controller associates tags with in-memory capabilities
• Our implementation: memory partitioned, with a region holding all tags
Demo Tablet Platform

Terasic DE-4 tablet hosting 100MHz CHERI processor, CheriBSD OS
Pointer-intensive benchmarks for pure-capability code (worst case)

- Primary cost: D-cache footprint from pointer-size increase
- Cycles overhead vs. data-size parameter (range of working-set sizes)
  - 8.1% - 80.1% 256-bit capabilities
  - 2.5% - 24.3% 128-bit capabilities
- “In the noise” for Dhrystone & tcpdump (256-bit capabilities)
- Other security/performance options – e.g., only return-address capabilities
Sandboxing: Domain-switching overhead

cycles overhead

- 1: socket
- 2: pipe
- 3: shm+pipe
- 4: shm+sem
- 5: pthread+sem
- 6: invoke
- 7: func

payload/bytes

process-based separation approaches

Inter-thread baseline

CHERI domain X

Function-call baseline
Library compartmentalization

- Compartmentalize within libraries without disturbing public API/ABI
- Allows unmodified applications to benefit from compartmentalization of key system classes/libraries
- Memory-based APIs are extremely inefficient to pass between processes
- Very efficient between CHERI compartments as pointers delegate memory access
**CHERI papers (1)**

- **ISCA 2014**: Fine-grained, in-address-space memory protection
  - Deconflate virtualization and protection
  - Hybrid model adds capabilities while retaining an MMU
  - Capabilities: pointers with tags, permissions, bounds
  - Manual annotations protect selected stack/heap pointers
  - C-language TCBs: OSes, language runtimes, etc.

- **ASPDAC 2015**: Explore and refine C-language compatibility
  - Converge fat-pointer and capability models
  - Binary-compatibility models and C compilation
  - Large-scale software study of C-language compatibility
CHERI papers (2)

- **Oakland 2015**: Hybrid hardware-software compartmentalization
  - **Sealed capabilities and object types**
  - Hardware-enforced **object-capability model**
  - Efficient, in-address-space **HW-SW domain transition**
- **ACM CCS 2015**: Compartmentalization modeling and analysis
  - **Conceptual model** for software compartmentalization
  - **LLVM-based static analysis tools** to analyze compartmentalized designs to validate security goals
  - **Annotations** for security goals, compartments, sensitive data, vendor information, past vulnerabilities, …
- **Analyses** of Chromium, OpenSSH; KDE compartmentalization
Current R&D directions

• Improve architecture, micro-architectural performance
  • Converge register files, 128-bit “compressed” capabilities
  • Opcode footprint reduction through ISA load/store reuse
• Explore and mature software security and development models
  • Compiler, linker, and ABI refinement
  • Control-Flow Integrity (CFI)
  • Compartmentalization programming models
  • Selected system calls within compartments (a la Capsicum)
  • Complete pure-capability CheriBSD implementation
• Temporal safety (e.g., accurate C garbage collection)
Broader implications

• Model is applicable to other RISC ISAs – ARMv8, RISC-V, etc.
  • Some design decisions are **deep** – e.g., tags, monotonicity
  • Others are **shallow** – e.g., separate vs. merged register files
• Many incremental SW paths, security/performance tradeoffs
  • Deploy selectively for data/code pointers? (e.g., stack, CFI)
  • Deploy in key class libraries? (no need to recompile applications)
• Language runtimes / JIT: Java, Javascript, memory safety
• Kernel compartmentalization (i.e., microkernels)
• Single-address-space systems (de-emphasise conventional MMU)
• Reduce protection pressure on the TLB/page-table system
  • Restore memory protection at PB-scale (HP’s “The Machine”)
Conclusions

• RISC ISA and CPU design implement capability model
• In-address-space pointers become capabilities
  • Complements MMU-based virtual memory
  • Fine-grained memory protection for code, data
  • Scalable compartmentalization
  • Strong compatibility with C-Language TCBs
• Open-source implementation, ISA specification: http://www.cheri-cpu.org/
Q&A

- **Oct. 2011:** Capability microkernel runs sandbox on FPGA
- **Jun. 2012:** CheriBSD capability context switching
- **Jul. 2012:** LLVM generates CHERI code
- **Nov. 2012:** Sandboxed code on CheriBSD; trojan mitigation demo
- **Dec. 2013:** Lightweight CheriBSD domain switching
- **Nov. 2014:** tcpdump uses multiple domain switches per packet
- **Sep. 2015:** CheriABI helloworld
- **Oct. 2010:** Project starts
- **Nov. 2011:** FPGA tablet + microkernel
- **May 2012:** FPGA prototype + FreeBSD
- **April 2013:** multi-FPGA CheriCloud
- **Jun. 2014:** 'fat capabilities' first ISA and FPGA prototype
- **Jun. 2015:** 128-bit FPGA prototype
- **Nov. 2015:** 128-bit CHERI ISA v4 specification
- **ACM CCS 2015:** program analysis, compartmentalization
- **IEEE S&P 2015:** operating systems, compartmentalization

**LAW 2010:** capabilities revisited

**RESolVE 2012:** hybrid capability-system model

**ISCA 2014:** hybrid MMU/capability model, architecture

**ASPLOS 2015:** C-language compatibility