Introducing CHERI-RISC-V

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

University of Cambridge and SRI International
RISC-V Week
Paris, 3-5 May 2022

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.
Approved for public release; distribution is unlimited.

This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 (“CTSRD”), with additional support from FA8750-11-C-0249 (“MRC2”), HR0011-18-C-0016 (“ECATS”), and FA8650-18-C-7809 (“CIFV”) as part of the DARPA CRASH, MRC, and SSITH research programs. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

This work was supported in part by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 105694.

We also acknowledge the EPSRC REMS Programme Grant (EP/K008528/1), the ERC ELVER Advanced Grant (789108), the Isaac Newton Trust, the UK Higher Education Innovation Fund (HEIF), Thales E-Security, Microsoft Research Cambridge, Arm Limited, Google, Google DeepMind, HP Enterprise, and the Gates Cambridge Trust.
Introduction: What is CHERI?

- CHERI = Capability Hardware Enhanced RISC Instructions
- CHERI is a new hardware technology that mitigates software security vulnerabilities
  - Developed by the University of Cambridge and SRI International starting in 2010, supported by DARPA and others
  - Arm collaboration from 2014
- CHERI for RISC-V is now mature, but being refined
- Today’s talk:
  - Why develop CHERI?
  - What is CHERI and how does it work?
  - What software will I be able to run on it?
  - What sort of evaluations have been run to date?
Why develop CHERI?

“Buffer overflows have not objectively gone down in the last 40 years. The impact of buffer overflows have if anything gone up.”

Ian Levy, NCSC

- Matt Miller (MS Response Center) @ BlueHat 2019:
  - From 2006 to 2018, year after year, 70% MSFT CVEs are memory safety bugs.
  - First place: spatial safety
    - Addressed directly by CHERI
  - Second place: use after free
    - Our recent work exploiting CHERI capability validity tags to precisely find pointers
More Motivation – Chromium Browser Safety

“70% of our serious security bugs are memory safety problems”

www.chromium.org/Home/chromium-security/memory-safety
Example 1

HeartBleed

Source: http://xkcd.com/1354/
HeartBleed

User Meg wants these 4 letters: BIRD. There are currently 34 connections open. User Brendan uploaded the file "Here's in car why". Note: Files for IP 375.381.83.17 are in /tmp/files-3843.

User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14853038534". Isabel wants pages about snakes but not too long. User Karen wants to change account password to "brendan".
User Meg wants the requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "ColloRasta". User Asher requests pages...
Went wrong? How do we do better?

• Classical answer:
  • The programmer forgot to check the bounds of the data structure being read
  • Fix the vulnerability in hindsight – one-line fix:
    \[
    \text{if } (1+2+\text{payload}+16 > s->s3->rrec.length) \text{ return } 0;
    \]

• Our answer:
  • Preserve bounds information during compilation
  • Use hardware (CHERI processor) to dynamically check bounds with little overhead and guarantee pointer integrity & provenance
Example 2: how to reduce the attack surface?

• The software attack surface keeps getting bigger
  • Applications just keep getting larger
  • Huge libraries of code aid rapid program development
  • Everything is network connected
• This aids the attacker: an expanding number of ways to break in
Software compartmentalization decomposes software into isolated compartments that are delegated limited rights. Able to mitigate not only unknown vulnerabilities, but also as-yet undiscovered classes of vulnerabilities and exploits.
Principles CHERI helps to uphold

- The **principle of intentional use**
  - Ensure that software runs the way the programmer intended, not the way the attacker tricked it
  - Approach: guaranteed pointer integrity & provenance, with efficient dynamic bounds checking

- The **principle of least privilege**
  - Reduce the attack surface using software compartmentalization
  - Mitigates known and unknown exploits
  - Approach: highly scalable and efficient compartmentalization
CHERI hardware adds a new type – the **Capability**

- CHERI Capability = bounds checked pointer with integrity
- Held in memory and in (new or extended) registers
A new type – the **Capability**

![Diagram showing virtual memory and capability components: permissions, compressed bounds (top, bottom), address.](image)
Software configures and uses capabilities to continuously enforce safety properties such as referential, spatial, and temporal memory safety, as well as higher-level security constructs such as compartment isolation.

CHERI capabilities are an architectural primitive that compilers, systems software, and applications use to constrain their own future execution.

The microarchitecture implements the capability data type and tagged memory, enforcing invariants on their manipulation and use such as capability bounds, monotonicity, and provenance validity.
Two key applications of the CHERI primitives

1. **Efficient, fine-grained memory protection for C/C++**
   - Strong source-level compatibility, but requires recompilation
   - Deterministic and secret-free referential, spatial, and temporal memory safety
   - Retrospective studies estimate $\frac{2}{3}$ of memory-safety vulnerabilities mitigated
   - Generally modest overhead (0%-5%, some pointer-dense workloads higher)

2. **Scalable software compartmentalization**
   - Multiple software operational models from objects to processes
   - Increases exploit chain length: Attackers must find and exploit more vulnerabilities
   - Orders-of-magnitude performance improvement over MMU-based techniques
     (<90% reduction in IPC overhead in early FPGA-based benchmarks)
**CHERI prototype software stack**

- **Complete open-source software stack** from bare metal up: compilers, toolchain, debuggers, hypervisor, OS, applications – all demonstrating CHERI
  - Rich CHERI feature use, but fundamentally incremental/hybridized deployment
  - Aim: Mature and highly useful research and development platform for Morello

<table>
<thead>
<tr>
<th>Open-source application suite</th>
<th>CheriBSD/Morello (funded by DARPA and UKRI)</th>
<th>Android (Arm)</th>
<th>Linux (Arm)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FreeBSD kernel + userspace, application stack</td>
<td>(Morello only)</td>
<td>(Morello only)</td>
</tr>
<tr>
<td></td>
<td>Kernel spatial and referential memory protection</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Userspace spatial, referential, and temporal memory protection</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Co-process compartmentalization</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Intra-process compartmentalization</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Morello-enabled bhyve Type-2 hypervisor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ARMv8-A 64-bit binary compatibility for legacy binaries</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CHERI-extended Google Hafnium hypervisor**

**CHERI Clang/LLVM compiler suite, LLD, LLDB, GDB**

**Baseline CHERI Clang/LLVM from SRI/Cambridge; Morello adaptation by Arm + Linaro**
Microsoft security analysis of CHERI C/C++

- Microsoft Security Research Center (MSRC) study analyzed all 2019 Microsoft critical memory-safety security vulnerabilities
  - Metric: “Poses a risk to customers → requires a software update”
  - Vulnerability mitigated if no security update required
  - Blog post and 42-page report
  - Concrete vulnerability analysis for spatial safety
  - Abstract analysis of the impact of temporal safety
  - Red teaming of specific artifacts to gain experience

- CHERI, “in its current state, and combined with other mitigations, it would have deterministically mitigated at least two thirds of all those issues”

https://msrc-blog.microsoft.com/2020/10/14/security-analysis-of-cheri-isa/
CHERI desktop ecosystem study: Key outcomes

Developed:

• 6 million lines of C/C++ code compiled for memory safety; modest dynamic testing

• Three compartmentalization case studies in Qt/KDE

Evaluation results:

• 0.026% LoC modification rate across full corpus for memory safety

• 73.8% mitigation rate across full corpus, using memory safety and compartmentalization

http://www.capabilitieslimited.co.uk/pdfs/20210917-cap ltd-cheri-desktop-report-version1-FINAL.pdf
Where to learn more?

- **Project web pages:**


- **Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)**, UCAM-CL-TR-951, October 2020

CHERI research and development timeline

Years 1-2: Research platform, prototype architecture

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

Years 4-7: Efficiency, CheriABI/C/C++/linker, ARMv8-A

Years 8-9: RISC-V, temporal safety, formal proof

Over 150 researcher years of effort by Cambridge & SRI
Many engineer years by Arm
Bridging the commercialisation chasm

Proven Technology

Fundamental Principles

University

Industry

Research in the Lab

Simulation/prototyping

Real World

TRL1: Basic principles
TRL2: Technology concept
TRL3: Proof of concept
TRL4: Simulations & initial results
TRL5: Validation on FPGA with modest toolchain
TRL6: Full OS, toolchain, several processors on FPGA
TRL7: Prototype demonstration in real world
TRL8: Product completed and qualified in real world
TRL9: Product proven in real world

CHERI-RISC-V & Arm Morello
First we made an FPGA-based hardware tablet
Open Source Stack: Research and Deployment

• CHERI-RISC-V developed open source:
  • Documentation (ISA ref, architecture overview, etc)
  • Specification in Sail
  • Simulators: Spike, Qemu
  • Clang/LLVM toolchain
  • OS support: CheriBSD, CheriFreeRTOS, CheriRTEMS
  • Hardware implementations
    • 3-stage, 5-stage and OoO cores on FPGA including AWS F1

Project URL:
http://cheri-cpu.org/
links to:
https://www.cl.cam.ac.uk/research/security/ctsrd/
Open Source CHERI-RISC-V Cores

• Piccolo 32b microcontroller:
  https://github.com/CTSRD-CHERI/Piccolo

• Flute 64b/32b scalar core:
  https://github.com/CTSRD-CHERI/Flute

• Toooba 64b out-of-order core based on MIT Riscy-OOO core:
  https://github.com/CTSRD-CHERI/Toooba
Arm Morello Demonstrator Board
Conclusions

• CHERI protections are completely deterministic and solve fundamental security issues

• CHERI provides the hardware with more semantic knowledge of what the programmer intended
  • Toward the principle of intentionality

• Efficient pointer integrity and bounds checking
  • Eliminates buffer overflow/over-read attacks (finally!)

• Provide scalable, efficient compartmentalisation
  • Allows the principle of least privilege to be exploited to mitigate known and unknown attacks

• Transitioning the technology via CHERI-RISC-V and Arm Morello
The CHERI-RISC-V Extension

Jessica Clarke, Peter Rugg, David Chisnall, Jonathan Woodruff, Alexandre Joannou
Robert N. M. Watson, Simon W. Moore, Peter G. Neumann, Hesham Almatary, Alasdair Armstrong, Peter Blandford-Baker,
John Baldwin, Hadrien Barrel, Thomas Bauereiss, Ruslan Bukin, Brooks Davis, Lawrence Esswood, Nathaniel W. Filardo,
Franz Fuchs, Dapeng Gao, Khilan Gudka, Brett Gutstein, Mark Johnston, Robert Kovacsics, Ben Laurie,
A. Theo Markettos, J. Edward Maste, Alfredo Mazzinghi, Prashanth Mundkur, Edward Napierala, George Neville-Neil,
Robert Norton-Wright, Lucian Paul-Trifu, Allison Randal, Ivan Ribeiro, Alex Richardson, Michael Roe, Peter Sewell,
Thomas Sewell, Stacey Son, Domagoj Stolfa, Andrew Turner, Munraj Vadera, Konrad Witaszczyk, and Hongyan Xia

University of Cambridge and SRI International
RISC-V Week
Paris, 3-5 May 2022
Approved for public release; distribution is unlimited.

This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 ("CTSRD"), with additional support from FA8750-11-C-0249 ("MRC2"), HR0011-18-C-0016 ("ECATS"), and FA8650-18-C-7809 ("CIFV") as part of the DARPA CRASH, MRC, and SSITH research programs. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

This work was supported in part by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 105694.

We also acknowledge the EPSRC REMS Programme Grant (EP/K008528/1), the ERC ELVER Advanced Grant (789108), the Isaac Newton Trust, the UK Higher Education Innovation Fund (HEIF), Thales E-Security, Microsoft Research Cambridge, Arm Limited, Google, Google DeepMind, HP Enterprise, and the Gates Cambridge Trust.
CHERI Overview

- CHERI capability words are hardware-defined pointer structures that include bounds and permissions.

- Capabilities are preserved atomically in both registers and memory.

- Capabilities can be dereferenced to load/store data (and other capabilities).

- All capabilities must be derived from more permissive capabilities.
Basic Requirements

- Load and store capabilities (2 * XLEN + tag)
- Load and store through capability pointers
- Jump to capabilities
- Manipulate capability metadata in registers (bounds, permissions)

- Must move capabilities atomically
- What are my bounds and permissions?
- Change PC with its bounds and permissions (ie PCC)
- Derive narrower capabilities
- Read capability fields
- Pointer arithmetic
CHERI-RISC-V Load and Store Capabilities

• Used reserved LQ/SQ for 128-bit capabilities

When using 64-bit capabilities in RV32, the RV64 instructions LD and SD are reused to behave as LC and SC respectively.

\[
\begin{array}{ccccccc}
31 & 25 & 24 & 20 & 19 & 15 & 14 & 12 & 11 & 7 & 6 & 0 \\
\hline
\text{imm} & \text{rs1} & 0x3 & \text{cd} & 0x3 & \text{LC cd, rs1, imm (RV32)} \\
\text{imm[11:5]} & \text{cs2} & \text{rs1} & 0x3 & \text{imm[4:0]} & 0x23 & \text{SC cs2, rs1, imm (RV32)} \\
\end{array}
\]

When using 128-bit capabilities in RV64, the RV128 instructions LQ and SQ (anticipated encoding) are reused to behave as LC and SC respectively.

\[
\begin{array}{ccccccc}
31 & 25 & 24 & 20 & 19 & 15 & 14 & 12 & 11 & 7 & 6 & 0 \\
\hline
\text{imm} & \text{rs1} & 0x2 & \text{cd} & 0xf & \text{LC cd, rs1, imm (RV64)} \\
\text{imm[11:5]} & \text{cs2} & \text{rs1} & 0x4 & \text{imm[4:0]} & 0x23 & \text{SC cs2, rs1, imm (RV64)} \\
\end{array}
\]

• Is this rv128? Not quite; will discuss later...
Load and Store through Capability Pointers

- New memory instruction encodings are expensive due to large immediates
- Use a mode bit for standard loads and stores to expect a capability address operand
  - Common cases: all integer pointers OR all capability pointers (depending on ABI)
  - Sacrifices intentionality in machine code
  - Mode bit is in PCC, so is naturally restored on function return
- Integer pointers use Default Data Capability (DDC) bounds
  Capability pointers use their own bounds
- Also add loads and stores explicit to each kind of pointer (with no immediate)

Uncompressed instructions affected by capability mode

<table>
<thead>
<tr>
<th>Integer load</th>
<th>LB</th>
<th>LH</th>
<th>LW</th>
<th>LD</th>
<th>LQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer load (unsigned)</td>
<td>LBU</td>
<td>LHU</td>
<td>LWU</td>
<td>LDU</td>
<td></td>
</tr>
<tr>
<td>Integer store</td>
<td>SB</td>
<td>SH</td>
<td>SW</td>
<td>SD</td>
<td>SQ</td>
</tr>
<tr>
<td>Floating-point load</td>
<td>FLW</td>
<td>FLD</td>
<td>FLQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-point store</td>
<td>FSW</td>
<td>FSD</td>
<td>FSQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomic</td>
<td>LR</td>
<td>SC</td>
<td>AMOSWAP</td>
<td>AMOADD</td>
<td>AMOAND</td>
</tr>
<tr>
<td>Atomic (cont)</td>
<td>AMOOR</td>
<td>AMOXOR</td>
<td>AMOMAX</td>
<td>AMOMIN</td>
<td></td>
</tr>
<tr>
<td>Address calculation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUIPC[^1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compressed instructions affected by capability mode

<table>
<thead>
<tr>
<th>Control flow</th>
<th>C.JALR</th>
<th>C.JR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compressed integer load</td>
<td>C.LW</td>
<td>C.LD</td>
</tr>
<tr>
<td>Compressed integer store</td>
<td>C.SW</td>
<td>C.SD</td>
</tr>
<tr>
<td>Compressed floating-point load</td>
<td>C.FLW</td>
<td>C.FLD</td>
</tr>
<tr>
<td>Compressed floating-point store</td>
<td>C.FSW</td>
<td>C.FSD</td>
</tr>
</tbody>
</table>

[^1]: Could we decide dynamically based on tag? Intentionality says no!
Split or Merged Register File

- New register file vs. Unified (& extended) register file
- C0 holds the NULL capability
- CHERI-RISC-V supports split or merged
  - Instructions are identical
  - Only semantics are changed

- So-far only implemented merged
  - Better scientific comparability with the baseline
  - Losing the chance to add more registers at almost no encoding cost!
  - Less context to save and restore
Jump to Capabilities

- Use explicit new jump to expect capability pointers
- Why not reuse the old jumps, repurposed with the mode bit? Because the mode bit is flipped by jumping to a capability!
- Helps intentionality, and encoding is cheap (just two operands)
- Compressed jumps do use the mode bit

Capabilities can point to code, not just data!

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mode</th>
<th>Mode</th>
<th>Mode</th>
<th>Mode</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7f</td>
<td>0xc</td>
<td>cs1</td>
<td>0x0</td>
<td>cd</td>
<td>0x5b</td>
</tr>
<tr>
<td>JALR. CAP cd, cs1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7f</td>
<td>0x14</td>
<td>cs1</td>
<td>0x0</td>
<td>cd</td>
<td>0x5b</td>
</tr>
<tr>
<td>JALR. PCC cd, cs1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7e</td>
<td>cs2</td>
<td>cs1</td>
<td>0x0</td>
<td>0x1</td>
<td>0x5b</td>
</tr>
<tr>
<td>CI invoke cs1, cs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Capability Manipulation Instructions

- Legacy integer instructions produce upper bits of the NULL capability (tag cleared)
- Pointer arithmetic uses dedicated instructions (IncOffset) rather than reusing legacy ones (ADD)
  - Makes pointer arithmetic intentional
  - Could help micro-architectural optimisation (IncOffset must check representability)
- Throw exception vs. clear tag on illegal transformations
  - Throwing an exception gives precise debugging
  - Clearing the tag is more convenient for software and micro-architecture, and is also safer
  - We're moving from exceptions to tag clearing

Morello is tag clearing

Product of merged register file!
Page Table Permissions

- Can track "capability free" pages to support sweeping for revocation
- Also supports experimental capability load detection

Five new bits for sv39 Page Table Entries (PTEs):

PTE bits for capability stores (mirrors W & D flags)

<table>
<thead>
<tr>
<th>CW</th>
<th>CD</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>Trap on capability stores (exception code 0x1B)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Capability stores atomically raise CD or fault (as above)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Capability stores permitted</td>
</tr>
</tbody>
</table>

PTE bits for capability loads

<table>
<thead>
<tr>
<th>CR</th>
<th>CRM</th>
<th>CRG</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Capability loads strip tags on loaded result</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Capability loads fault (exception code 0x1A)</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>1</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Capability loads are unaltered</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>Reserved for generational load barriers</td>
</tr>
</tbody>
</table>

Possibly more complex than commercially necessary for experimentation...
CSRs and Special Capability Registers

- PCC and DDC are universally accessible CSRs

- DDC and PCC hold the "almighty capability" on reset

- Scratch registers per privilege level

- New xtval for capability violations

- CSR whitelist when Access System Registers (ASR) is not set in PCC

### New CSRs

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Register</th>
<th>Modes</th>
<th>Access</th>
<th>Reset</th>
<th>Extends</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8C0</td>
<td>User capability control and status register (uccsr)</td>
<td>U, S, M</td>
<td>ASR</td>
<td>∞</td>
<td>utvec</td>
</tr>
<tr>
<td>0x9C0</td>
<td>Supervisor capability control and status register (scsrr)</td>
<td>U, S, M</td>
<td>ASR</td>
<td>∞</td>
<td>-</td>
</tr>
<tr>
<td>0xBC0</td>
<td>Machine capability control and status register (mcscr)</td>
<td>U, S, M</td>
<td>ASR</td>
<td>∞</td>
<td>uepc</td>
</tr>
</tbody>
</table>

### CSR whitelist

<table>
<thead>
<tr>
<th>CSR</th>
<th>Modes</th>
<th>Access</th>
<th>Reset</th>
<th>Extends</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycle(h)</td>
<td></td>
<td>Read-Only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time(h)</td>
<td></td>
<td>Read-Only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instret(h)</td>
<td></td>
<td>Read-Only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hmpcounter(h)</td>
<td></td>
<td>Read-Only</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Special Capability Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Modes</th>
<th>Access</th>
<th>Reset</th>
<th>Extends</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>PC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>utvec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>uepc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>stvec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>sepc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td>mtvec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td>mepc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Both RV32 and RV64 are supported

- RV32 (CHERI64) and RV64 (CHERI128) are specified

- Would be interesting to experiment with RV64 with CHERI64 (sv32) to learn about implications of RV128 in CHERI128
CHERI128 Relationship with RV128

• Not quite RV128
  • Added Load and Store Quad instructions
  • But lacking 128-bit arithmetic instructions

• If an implementation had both, would we want new loads and stores to preserve intentionality? (New tag-clearing LQ/SQ)

sv64 (64-bit virtual addresses) with RV128 is not unreasonable; i.e. XLEN = 2×"VA-LEN"
Conclusions

• CHERI academic research has moved fully to the CHERI-RISC-V architecture

• Lots of implementations:

  - The Sail-CHERI-RISC-V model (CHERI-64/CHERI-128)
  - QEMU simulator (CHERI-64/CHERI-128)
  - CHERI Ibex (CHERI-64) (not up-to-date)
  - CHERI Piccolo (CHERI-64)
  - CHERI Flute (CHERI-64/CHERI-128)
  - CHERI RiscyOO CHERI-RISC-V (CHERI-128)

• Fully featured LLVM compiler support

• CheriBSD with full software stack building in pure-capability mode
Four CHERI-RISC-V Micro-Architectures

Peter Rugg, Jonathan Woodruff, Alexandre Joannou, Ivan Ribeiro, Robert N. M. Watson, Simon W. Moore, Peter Sewell, Peter G. Neumann


University of Cambridge and SRI International

RISC-V Week
Paris, 3-5 May 2022
Approved for public release; distribution is unlimited.

This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 (“CTSRD”), with additional support from FA8750-11-C-0249 (“MRC2”), HR0011-18-C-0016 (“ECATS”), and FA8650-18-C-7809 (“CIFV”) as part of the DARPA CRASH, MRC, and SSITH research programs. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

This work was supported in part by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 105694.

We also acknowledge the EPSRC REMS Programme Grant (EP/K008528/1), the ERC ELVER Advanced Grant (789108), the Isaac Newton Trust, the UK Higher Education Innovation Fund (HEIF), Thales E-Security, Microsoft Research Cambridge, Arm Limited, Google, Google DeepMind, HP Enterprise, and the Gates Cambridge Trust.
Contents

• How to add CHERI to a core
• Shared components
  • Tag controller
  • Compression library
• Four implementations
• Per-core optimizations
CHERI-RISC-V Microarchitectural Changes

- Extend decode, registers, arithmetic, and memory access for capabilities
- Capability CSRs and new capability exception behaviors
- Widen caches and add tag support
- Tagged AXI interconnect
- Performance counters
- Analyze performance, identify bottlenecks, and optimize microarchitecture to meet timing and performance goals
CHERI-RISC-V Pipeline Changes

- **Instruction Fetch**
  - Fetch address distinguished from sandbox offset (PC)

- **Decode**
  - Capability manipulation added in parallel with ALU

- **Execute**
  - Bounds check added in parallel with cache access
  - PC extended to be a capability

- **Memory Access**
  - Capability offsets and bounds legacy memory operations

- **Writeback**
  - Registers extended with capability metadata and tags

- **Branch Predict**
  - PC extended to be a capability
CHERI-RISC-V Reused Components

• AXITagController
  Tag controller module manages combining tags with data

• Capability encoding library
  Memory, register, and pipeline capability formats with functions for each

• TestRIG testing framework
  CHERI instruction definitions and templates for testing deep capability states
Tagging Capabilities

• Capabilities have a hidden validity tag
  • In registers and memory

• Tag bit is critical to security
  • Conventional operations (arith, memory) clear the tag
  • Only capability instructions preserve the tag and guarantee monotonic decrease in rights

• One hidden bit per 128-bits avoids using other integrity measures (no crypto needed….)
Propagating tags from registers to DRAM

- Tags stored in registers and caches with data to ensure consistency
- Off-chip storage:
  - Tags stored in upper 1% of commodity DRAM
  - Tag cache per DRAM controller reduces DRAM traffic
  - No consistency issues
Hierarchical Tag Compression

- Size tag cache line length to 64-byte DDR4 burst transfer size ⇒ one line covers tags for 8KiB of memory (128-bit capabilities)

- Many lines don’t contain tags (code, large blocks of data, disk cache, etc.)
  - So handling tag sparseness is important
  - Only want to pay for tagging when needed
Tag Compression

- 2-level tag table
- Each bit in the root level indicates all zeros in a leaf group
- Reduces tag cache footprint
- Amplifies cache capacity

![Diagram of 2-level tag table with root and leaf tables. Each 64-byte block represents 8KiB of data, with 1 bit per 8KiB indicating tag presence. The root table signifies all zeros in a leaf group.]

1 bit per 8KiB of data: 0 for no tags set
Capability Compression

Capabilities encode at least 3 64-bit fields:

But we can encode the Top and Bottom relative to the Address:

• Larger objects require greater alignment
• Address must be “near” the Top and Bottom
Capability Compression Shared Library

- Capabilities partially decompressed in stages throughout pipeline
- Shared library between all four implementations

Memory Capability

- XOR + Shift + Multiplex

Register Capability

- “Exponent” and “Mantissa” extracted
  - 152 bits

Pipeline Capability

- Compression edge-cases identified
  - 162 bits

Architectural “CHERI-concentrate” format
- 128 bits
Baseline Processors

- Piccolo - Bluespec
  32-bit 3-stage in-order microcontroller (Bluespec Inc.)
- Ibex - Verilog
  32-bit highly area-optimised microcontroller (lowRISC)
- Flute - Bluespec
  64-bit 5-stage in-order microcontroller (Bluespec Inc.)
- Toooba - Bluespec
  64-bit out-of-order superscalar (Bluespec Inc.) based on RiscyOO (MIT)
CHERI-RISC-V Scaling Across Cores

- CHERI Piccolo and CHERI Ibex – Power and Area
  Small core (no MMU or FPU), so logic overhead is most significant
  Where present, very small cache, so DRAM traffic (i.e. power) overhead is pronounced

- CHERI Flute – Frequency and Prediction
  Deeper pipeline, higher frequency (100Mhz), so sensitive to timing
  Performance depends on predicting branches and forwarding results

- CHERI Toooba – Concurrency, Prediction, and Multicore
  Superscalar execution and performance is sensitive to parallelism
  Higher cost of misprediction and performance is very sensitive to accuracy
CHERI Flute – Challenge 1 – Frequency

- Highest target frequency (100MHz). Initial design failed timing.

- Solution: careful optimization
  - Make forwarding independent of bounds check
  - Add latch between instruction cache and AXI interconnect
  - Expose cache invariants to synthesis tool
  - Latch computed SCR value
  - Refactor check for misprediction
  - Delay capability trap infrequent path (performance penalty only on trap)
  - Rebalance floating point latching
CHERI Flute – Challenge 2 – Prediction

- First core with prediction: fetch requests are sent to the instruction memory before the capability used is known.
- Various possibilities, e.g. predict entire capability, predict address or offset.
- Solution: Divide core into front-end working with predicted raw addresses, and back-end working with full capabilities.
- Also need to speculate "capability encoding mode": add to predicted state.

```
Fetch
\downarrow
Decode
\downarrow
Execute
\downarrow
Memory Access
\downarrow
Writeback
```

```
Branch predictor
\uparrow

Addresses
\uparrow

Capabilities
\uparrow

Check: predicted address == PCC.address
```
CHERI Flute – Challenge 3 – DRAM traffic

- DRAM overhead was high initial CHERI-Flute implementation (16.9% overhead)

  Write-through caches: capability writes double DRAM traffic of integer writes. Pushing to the stack twice as expensive with purecap!

- Page table miss traffic was a large contribution to DRAM traffic

  The page table walks (uncached) directly accessed DRAM

  2-way associative TLB meets the power/DRAM 5% target (-0.6% overhead)
CHERI Toooba – Challenge 1 – Reorder Buffer

**Initial Reorder Buffer** (state for 64 in-flight instructions) **184%** overhead

- **Solution 1:** Change enum structure to avoid explosion
  
  Bluespec compiler was inefficient for sparse enum assignments; CHERI Toooba used these more than Toooba

- **Solution 2:** Eliminate three 64-bit registers from records
  
  Derivable from other state (TVAL) or unnecessary in our configuration (dest_data, store_data)

- **Solution 3:** MUX out of Reorder Buffer rows into ALU
  
  64 MUXs (1 per record) -> 2 MUXs (1 per ALU pipe)

Currently **2%** overhead for Reorder buffer

Reorder Buffer Contribution to Logic Overhead in CHERI-Toooba

Percentage Contribution

- Initial
- After Optimisation

Old Design

```
rob 0
rob 1
...
rob 63
```

Solution 3

```
rob 0
rob 1
...
rob 63
```
CHERI Toooba – Challenge 2 – PCC Metadata

- Prediction/fetch/decode granularity
  32 bits -> 16 bits for compressed instructions

Each granule has PCC and predicted next PCC:
256 bits of address metadata per 16-bit granule

- The distinct PC segments guaranteed to be 1/8 the maximum granules

Fetch blocks of up to 4 granules share a segment
Any predicted PC will be shared with the next block

- Introduce compression table in 4 stages of Fetch

PC now represented by 15 bits
30 bits per 16-bit granule

```haskell
typedef struct {
    PcLSB lsb;
    PcIdx idx;
} PcCompressed deriving(Bits,Eq,FShow);

IndexedMultiset#(PcIdx, PcMSB, SupSizeX2) pcBlocks <- mkIndexedMultisetQueue;
function CapMem decompressPc(PcCompressed p) = {pcBlocks.lookup(p.idx), p.lsb};
```
CHERI Toooba – Challenge 3 – BTB Contention

- CHERI pointers cause higher DRAM traffic
- Misprediction causes wasted traffic in Toooba
  
  Deep, out-of-order speculation means unnecessary memory traffic caused by misspeculated loads

- Use a 2-way associative Branch Target Buffer (BTB) to compensate for DRAM overhead

  13.07% -> 4.6% DRAM Overhead

- Segmented BTB to more-than-compensate for the area overhead
Conclusions

• CHERI security extensions have been applied to a range of cores
  • Microcontrollers to a superscalar core for CHERI-RISC-V
  • Arm Morello SoC also demonstrates that CHERI can be added to a commercial core (Neoverse N1)

• CHERI comes with some costs, but careful microarchitectural optimization reduces these substantially
The CHERI-RISC-V Software Ecosystem and Toolchain

Alex Richardson, Jessica Clarke, David Chisnall, Brooks Davis, John Baldwin

University of Cambridge and SRI International
RISC-V Week
Paris, 3-5 May 2022
Approved for public release; distribution is unlimited.

This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 (“CTSRD”), with additional support from FA8750-11-C-0249 (“MRC2”), HR0011-18-C-0016 (“ECATS”), and FA8650-18-C-7809 (“CIFV”) as part of the DARPA CRASH, MRC, and SSITH research programs. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

This work was supported in part by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 105694.

We also acknowledge the EPSRC REMS Programme Grant (EP/K008528/1), the ERC ELVER Advanced Grant (789108), the Isaac Newton Trust, the UK Higher Education Innovation Fund (HEIF), Thales E-Security, Microsoft Research Cambridge, Arm Limited, Google, Google DeepMind, HP Enterprise, and the Gates Cambridge Trust.
Compiling for CHERI RISC-V

• How can we make use of CHERI capabilities for C/C++ code?

• Use CHERI LLVM: https://github.com/CTSRD-CHERI/llvm-project

```c
#include <stdio.h>

int main(void) {
    printf("Hello world\n");
}
```

<table>
<thead>
<tr>
<th>C RISC-V Assembly</th>
<th>CHERI RISC-V Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>-march=rv64gc -mabi=lp64d</td>
<td>-march=rv64gcxcheri -mabi=l64pc128d</td>
</tr>
</tbody>
</table>

1. Adjust stack pointer
2. Save return address

```assembly
main:
    addi sp, sp, -16
    sd ra, 8(sp)
.LBB0_1:
    auipc a0, %pcrel_hi(.Lstr)
    addi a0, a0, %pcrel_lo(.LBB0_1)
    call puts
    mv a0, zero
    ld ra, 8(sp)
    addi sp, sp, 16
    ret

.Lstr:
    .asciz "Hello world"
```

```assembly
main:
    cincoffset csp, csp, -16
    csc cra, 0(csp)
.LBB0_1:
    auipcc ca0, %captab_pcrel_hi(.Lstr)
    clc ca0, %pcrel_lo(.LBB0_1)(ca0)
    ccall puts
    mv a0, zero
    clc cra, 0(csp)
addi sp, sp, 16
    cincoffset csp, csp, 16
    cret
```

3. Generate pointer to .Lstr
4. Call printf()
5. Set return value to zero
6. Restore return address
7. Adjust stack pointer
8. Return
Automatic capability bounds (1)

- OS kernel, run-time linker, memory allocator(s) and compiler take care of **automatic capability bounds refinement**
- On `execve()` initial (bounded) capabilities set up by OS kernel
- Run-time linker loads data & code for libraries using `mmap()`
- Kernel returns a new bounded capability for `mmap()`
- Run-time linker then processes relocations and creates bounded capabilities for global variables and functions
- `malloc()` ensures that allocation is correctly bounded
The compiler automatically adds bounds e.g. for stack allocations:
Automatic capability bounds (3)

Opt-in support for preventing sub-object overflows (at a moderate compatibility cost).

Sub-object overflow could corrupt UID or the function pointer (although CHERI ensures the latter is not callable!)

Callee can only access the 16 bytes of the name array

Callee gets access to the entire allocation
Memory protection for the language and the language runtime

**Language-level memory safety**
- Pointers to heap allocations
- Function pointers
- Pointers to global variables
- Pointers to memory mappings
- Pointers to TLS variables
- Pointers to sub-objects

**Sub-language memory safety**
- GOT pointers
- Vararg array pointers
- PLT entry pointers
- ELF aux arg pointers
- C++ vtable pointers
- Stack pointers
- Return addresses

- Capabilities are refined by the kernel, run-time linker, compiler-generated code, heap allocator, …
- Protection mechanisms:
  - Referential memory safety
  - Spatial memory safety + privilege minimization
  - Temporal memory safety
- Applied automatically at two levels:
  - **Language-level pointers** point explicitly at stack and heap allocations, global variables, …
  - **Sub-language pointers** used to implement control flow, linkage, etc.
- Sub-language protection mitigates bugs in the language runtime and generated code, as well as attacks that cannot be mitigated by higher-level memory safety
Portability when compiling for CHERI-RISC-V

- In general, most code will just work as-is or be flagged by compiler warnings.
- However, a few common issues exist:
  - Insufficient alignment (e.g. in memory allocators)
    - Loading/storing pointers needs stricter alignment (16 bytes for CHERI-128)
    - Allocators should use `alignof(max_align_t)` instead of hardcoding 8
  - Incorrectly casting pointers to or from `long`
    - `long` can only hold the address part, bounds and validity are lost when casting
    - Please use C99 `uintptr_t` instead (using `long` is not allowed by C standard)
  - Updating pointers after `realloc()` may require auditing
Operating system support

- Most mature OS is CheriBSD, FreeBSD with full support for pure-capability CHERI code
  - [https://github.com/CTSRD-CHERI/cheribsd](https://github.com/CTSRD-CHERI/cheribsd)
- Even runs a memory safe KDE graphical desktop!
  - 0.026% LoC modification rate across full corpus for memory safety
  - 73.8% mitigation rate across full corpus, using memory safety and compartmentalization
Debugging

• We have a version GDB with support for CHERI-RISC-V available at https://github.com/CTSRD-CHERI/gdb

• Generally works just as you are used to it

Starting program: /opt/cheri-exercises/buffer-overflow-stack-cheri

Program received signal SIGPROT, CHERI protection violation
Capability bounds fault caused by register ca1.
0x0000000000101dae in write_buf (buf=0x3fffdfff5c [rwRW,0x3fffdfff5c-0x3fffdfff6c] "", ix=16) at src/exercises/buffer-overflow-stack/buffer-overflow-stack.c:13
13 src/exercises/buffer-overflow-stack/buffer-overflow-stack.c: No such file or directory.
(gdb) bt
#0 0x00000000000101dae in write_buf (buf=0x3fffdfff5c [rwRW,0x3fffdfff5c-0x3fffdfff6c] "", ix=16) at src/exercises/buffer-overflow-stack/buffer-overflow-stack.c:13
#1 0x00000000000101e98 in main () at src/exercises/buffer-overflow-stack/buffer-overflow-stack.c:31
(gdb)
Debugging

- We have a version GDB with support for CHERI-RISC-V available at https://github.com/CTSRD-CHERI/gdb

- Generally works just as you are used to it

Starting program: /opt/cheri-exercises/buffer-overflow-stack

Program received signal SIGPROT, CHERI protection violation
Capability bounds fault caused by register ca1.
0x0000000000101dae in write_buf (buf=0x3fffdfff5c [rwRW,0x3fffdfff5c-0x3fffdfff6c] "", ix=16) at src/exercises/buffer-overflow-stack/buffer-overflow-stack.c:13
13 src/exercises/buffer-overflow-stack/buffer-overflow-stack.c: No such file or directory.
(gdb) bt
#0 0x0000000000101dae in write_buf (buf=0x3fffdfff5c [rwRW,0x3fffdfff5c-0x3fffdfff6c] "", ix=16) at src/exercises/buffer-overflow-stack/buffer-overflow-stack.c:13
#1 0x0000000000101e98 in main () at src/exercises/buffer-overflow-stack/buffer-overflow-stack.c:31
(gdb)
Emulators

- QEMU: [https://github.com/CTSRD-CHERI/qemu](https://github.com/CTSRD-CHERI/qemu)
  - Fast and mature emulator including instruction tracing and GDB support for bare-metal debugging
  - Easiest way to get started with CHERI – no need for an FPGA
- Sail model: [https://github.com/CTSRD-CHERI/sail-cheri-riscv](https://github.com/CTSRD-CHERI/sail-cheri-riscv)
  - Reference model for 32 and 64-bit CHERI-RISC-V
Tying it all together - cheribuild

• Yet another meta build system for CHERI software
• Builds all the projects needed to run CheriBSD (and much more)
• To get started: cheribuild.py run-riscv64-purecap -d
• Can also cross-compile hundreds of additional projects that aren’t packaged yet for CheriBSD, e.g. kde-x11-desktop
• Automates various steps such as build and installation, booting CheriBSD and running (cross-compiled) test suites
• Available at https://github.com/CTSRD-CHERI/cheribuild
Conclusions

• CHERI RISC-V has a mature software ecosystem including OS support (CheriBSD), emulators (QEMU) and debuggers (GDB)

• All of these projects are open-source and available on GitHub: https://github.com/CTSRD-CHERI/

• Key takeaways from this talk:
  • Using CHERI-RISC-V should feel essentially the same as RISC-V
  • Even if you don’t plan on using CHERI for your code: please use (u)intptr_t when casting pointers to integers
CHERI-RISC-V demo
Demonstrating Memory-safety Features under CheriBSD on a Multi-core, Superscalar Softcore

Franz Fuchs, Robert N. M. Watson, Simon W. Moore, Peter Sewell, Peter G. Neumann
Hesham Almatary, Jonathan Anderson, Alasdair Armstrong, Peter Blandford-Baker,
John Baldwin, Hadrien Barrel, Thomas Bauereiss, Ruslan Bukin, David Chisnall, Jessica Clarke, Nirav Dave, Brooks Davis,
Lawrence Esswood, Nathaniel W. Filardo, Dapeng Gao, Khilan Gudka, Brett Gutstein, Alexandre Joannou,
Mark Johnston, Robert Kovacsics, Ben Laurie, A. Theo Markettos, J. Edward Maste, Alfredo Mazzinghi,
Alan Mujumdar, Prashanth Mundkur, Steven J. Murdoch, Edward Napierala, George Neville-Neil, Robert Norton-Wright,
Philip Paeps, Lucian Paul-Trifu, Allison Randal, Ivan Ribeiro, Alex Richardson, Michael Roe, Colin Rothwell, Peter Rugg,
Hassen Saidi, Peter Sewell, Thomas Sewell, Stacey Son, Domagoj Stolfa, Andrew Turner, Munraj Vadera,
Konrad Witaszczyk, Jonathan Woodruff, Hongyan Xia, and Bjoern A. Zeeb

University of Cambridge and SRI International
RISC-V Week
Paris, 3-5 May 2022
Approved for public release; distribution is unlimited.

This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 ("CTSRD"), with additional support from FA8750-11-C-0249 ("MRC2"), HR0011-18-C-0016 ("ECATS"), and FA8650-18-C-7809 ("CIFV") as part of the DARPA CRASH, MRC, and SSITH research programs. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

This work was supported in part by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 105694.

We also acknowledge the EPSRC REMS Programme Grant (EP/K008528/1), the ERC ELVER Advanced Grant (789108), the Isaac Newton Trust, the UK Higher Education Innovation Fund (HEIF), Thales E-Security, Microsoft Research Cambridge, Arm Limited, Google, Google DeepMind, HP Enterprise, and the Gates Cambridge Trust.
This presentation is based on material prepared by the CHERI project:

Adversarial CHERI Exercises and Missions:
https://ctsrd-cheri.github.io/cheri-exercises/

CHERI Workshops:
https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/workshops/
Cross-compiling CHERI Binaries

```
clang --target riscv64-unknown-freebsd --sysroot=/path/to/sysroot-
riscv64-purecap/ -mno-relax -march=rv64gc -mabi=lp64d source.c -o binary

clang --target riscv64-unknown-freebsd --sysroot=/path/to/sysroot-
riscv64-purecap/ -mno-relax -march=rv64gcxcheri -mabi=lp64pc128d
-Wcheri source.c -o binary
```

- **Compile for baseline**
- **Compile for CHERI-RISC-V**
- Enable all CHERI warnings
- Compile for 128-bit capability CHERI target
int main(void){
    printf("size of pointer: %zu\n", sizeof(void *));
    printf("size of address: %zu\n", sizeof(ptraddr_t));
    return (0);
}
CHERI tag protection (1/2)

```c
char buf[0x1FF];
volatile union {
    char *ptr;
    char bytes[sizeof(char*)];
} p;

for (size_t i = 0; i < sizeof(buf); i++) {
    buf[i] = i;
}

p.ptr = &buf[0x10F];
char *q = (char*)((uintptr_t)p.ptr & ~0xFF);
printf("q=%" PRINTF_PTR " (0x%zx into buf)\n", q, q - buf);
printf("*q=%02x\n", *q);

p.bytes[0] = 0;
char *r = p.ptr;
printf("r=%" PRINTF_PTR " (0x%zx)\n", r, r - buf);
printf("*r=%02x\n", *r);
```

- Declare a character buffer and a union struct
- Fill the buffer with sequence of values
- Assign `p.ptr` to part of `buf`
- Assign `q` to an aligned address in `buf`, and print status
- Attempt to assign `r` to an aligned address in `buf` via the `p.bytes` field and print status
Writing to pointer and writing to a data array are both valid on the baseline.

Write to capability with a capability operation works fine.

Data write strips validity tag.

Dereferencing an invalid capability leads to an exception.
Buffer Overflow (1/3)

```c
char upper[0x10];
char lower[0x10];

printf("upper = %p, lower = %p, diff = %zx\n",
    upper, lower, (size_t)(upper - lower));

upper[0] = 'a';
printf("upper[0] = %c\n", upper[0]);

lower[sizeof(lower)] = 'b';
printf("upper[0] = %c\n", upper[0]);
```

- Declare two buffers at consecutive addresses
- Print memory layout of buffers
- Write to buffer `upper` and print status
- Out-of-bounds write to `lower` and print status of `upper` to see whether buffer overflow worked
Buffer Overflow (2/3)

Buffer overflow successful due to missing bounds information on baseline

upper and lower are guarded by capabilities limiting each buffer to 16 byte bounds

Out-of-bounds write leads to CHERI exception
Buffer Overflow (3/3)

<setup_cap>:
cincoffset ca0, csp, 48
csetbounds cs0, ca0, 16

cincoffset ca0, csp, 64
csetbounds cs1, ca0, 16

...

<write_to_lower>:
addi a1, zero, 16
cincoffset cs0, cs0, a1

addi a1, zero, 98
csb a1, 0(cs0)

Create capability for buffer lower and set 16 byte bounds
Create capability for buffer upper at consecutive addresses and set 16 byte boundaries
Move capability address out of bounds because a1 = 16
Store character 'b' in a1
Store byte to memory; CHERI exception due to address out-of-bounds
CHERI C/C++: pointer provenance validity (1/2)

• An integer data type cast to a pointer data type results in a NULL-derived capability without a tag;

• However, there are data types that can hold pointer or integer values (e.g., uintptr_t).

• In the CHERI memory protection model, capabilities are derived from a single other capability;

• In CHERI C/C++, a capability can be a result of a complex expression with multiple data types and casts.

CHERI C/C++ Programming Guide, Section 4.2, 4.2.1, and 4.2.3 (https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf)
• Ideally, we would like to recompile source code for CheriABI and automatically gain security;

• Unfortunately, there is a lot of software that use incorrect data types to hold values that fit in them but have different semantics.
CHERI LLVM can identify capability-related issues and print warnings:

- Loss of provenance (-Wcheri-capability-misuse);
- Ambiguous provenance (-Wcheri-provenance);
- Underaligned capabilities of packed structures (-Wcheri-capability-misuse);
- Underaligned load of capability type (-Wcheri-inefficient).

CHERI C/C++ Programming Guide, Chapter 6
Example broken cat program

We modified the cat(1) program from CheriBSD/FreeBSD to introduce two bugs:

1. Loss of provenance.

2. Provenance-free integer type to pointer type cast.
Let’s try to compile cat!

Potential loss of provenance: we have given the compiler multiple choices for a source of provenance and the compiler might have picked the wrong one.

We are trying to case a non-pointer type to a pointer type; we need a valid capability as a source of provenance.
Conclusions

• CHERI HW/SW stack is fully working
• CHERI enforces intentionality
• Provenance validity and bounds checking lead to strong spatial memory safety that eliminates security bugs
• CHERI LLVM helps a developer to adapt a C/C++ program to CHERI C/C++
• Source code changes are needed where source of provenance is not clear, where non-pointer to pointer casting is done, and where alignments enforced by the developer are incorrect
WRAP UP
Wrap Up

• CHERI is a stable platform:
  • Demonstrated on RISC-V and Arm (previously on MIPS)
  • Full software stack (compiler, linker, OSs, etc.)
  • Formal verification of key security properties

• Our aim:
  • CHERI on every platform with no IP restrictions
  • Looking to ratify CHERI as an official RISC-V extension

Project website: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/