CHERI
Capability Hardware Enhanced RISC Instructions

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


University of Cambridge and SRI International
Web Slide Deck – 16 January 2024

Approved for public release; distribution is unlimited. This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 (“CTSRD”), with additional support from FA8750-11-C-0249 (“MRC2”), HR0011-18-C-0016 (“ECATS”), FA8650-18-C-7809 (“CIFV”), HR001122C0110 (“ETC”), and HR001122S0003 (“MTSS”). 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 in part by the Innovate UK project 105694 (“Digital Security by Design (DSbD) Technology Platform Prototype”, and Innovate UK project 10027440 (“Developing and Evaluating an Open-Source Desktop for Arm Morello”).

This work was also supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-11-C-0237 (“CTSRD”), with additional support from FA8750-11-C-0249 (“MRC2”), HR0011-18-C-0016 (“ECATS”), FA8650-18-C-7809 (“CIFV”), HR001122C0110 (“ETC”), and HR001123C0031 (“MTSS”) as part of the DARPA I2O CRASH, I2O MRC, and MTO 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.

We further acknowledge EPSRC REMS (EP/K008528/1), EPSRC CHaOS (EP/V000292/1), ERC ELVER (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 introduction

• **CHERI** is a new processor technology that mitigates software security vulnerabilities
  
  • Developed by the University of Cambridge and SRI International starting in 2010, supported by DARPA
  
  • Arm collaboration from 2014, supported by DARPA; Arm Morello CPU, SoC; board announced 2019, with support from UKRI Shipping as of Jan 2022
  
  • Microsoft CHERIoT (RISC-V) Ibex core announced Sep 2022: Open sourced in February 2023; lowRISC FPGA board announced Sep 2023
  
• Today’s talk:
  
  • What is CHERI?
  
  • Transition efforts including Arm, Google, Microsoft, and beyond …
  
  
Capability systems

- The capability system is a **design pattern** for how CPUs, languages, OSes, ... can control access to resources
  - **Capabilities** are communicable, unforgeable tokens of authority
  - In **capability-based systems**, resources are reachable only via capabilities
- Capability systems limit the **scope and spread of damage** from accidental or intentional software misbehavior
- They do this by making it **natural and efficient** to implement, in software, two security design principles:
  - The **principle of least privilege** dictates that software should run with the minimum privileges to perform its tasks
  - The **principle of intentional use** dictates that when software holds multiple privileges, it must explicitly select which to exercise

What is CHERI?

• CHERI is a processor architectural protection model
  • Composes a capability-system model with hardware and software
  • Adds new security primitives to Instruction-Set Architectures (ISAs)
  • Implemented by microarchitectural extensions to the CPU and SoC
  • Enables new security behavior in software

• CHERI mitigates vulnerabilities in C/C++ Trusted Computing Bases
  • Hypervisors, operating systems, language runtimes, browsers, ….
  • Fine-grained memory protection deterministically closes many arbitrary code execution attacks, and directly impedes common exploit-chain tools
  • Scalable compartmentalization mitigates many vulnerability classes .. even unknown future classes .. by extending the idea of software sandboxing

• CHERI-RISC-V research architecture and prototype FPGA implementations
• Arm Morello industrial demonstrator CPU, board; Microsoft CHERIoT CPU
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**.
An Introduction to CHERI

  - Architectural capabilities and the CHERI ISA
  - CHERI microarchitecture
  - ISA formal modeling and proof
  - Software construction with CHERI
  - Language and compiler extensions
  - OS extensions
  - Application-level adaptations

NB: Predates public announcement of Morello
(Lack of) architectural least privilege

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

• These privileges were not required by the C language; why allow code the ability to:
  • Write outside the target buffer?
  • Corrupt or inject a code pointer?
  • Execute data as code / re-use code?

• Limiting privilege doesn’t fix bugs – but does provide **vulnerability mitigation**

- Memory Management Units (MMUs) do not enable efficient, fine-grained privilege reduction
Application-level least privilege

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

Potential compartmentalization boundaries matching reasonable user expectations for least privilege can be found in many user-facing apps. E.g., a malicious email attachment should not be able to gain access to other attachments, messages, folders, accounts, or the system as a whole.

Able to mitigate not only unknown vulnerabilities, but also as-yet undiscovered classes of vulnerabilities and exploits.
• Potential decompositions occupy a compartmentalization space:
  • Points trade off security against performance, program complexity
  • Increasing compartmentalization granularity better approximates the principle of least privilege …
  • … but MMU-based architectures do not scale to many processes:
    • Poor spatial protection granularity
    • Limited simultaneous-process scalability
    • Multi-address-space programming model
HARDWARE-SOFTWARE CO-DESIGN FOR CHERI
Hardware-software-semantics co-design

• University of Cambridge and SRI International from 2010 supported by DARPA
• Architectural mitigation for C/C++ TCB vulnerabilities
  • Tagged memory, new hardware capability data type
  • Model hybridizes cleanly with contemporary hardware and software designs
  • New hardware enables incremental software deployment
• Hardware-software-semantics co-design + concrete prototyping:
  • CHERI abstract protection model; concrete ISA instantiations in 64-bit MIPS, 32/64-bit RISC-V (+ Microsoft CHERIoT), 64-bit Armv8-a (Arm Morello)
  • Formal ISA models, Qemu-CHERI, and multiple FPGA prototypes
  • Formal proofs that ISA security properties are met, automatic testing
  • CHERI Clang/LLVM/LLD, CheriBSD, C/C++-language applications
  • Repeated iteration to improve {performance, security, compatibility, ..}
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 8-12: RISC-V, temporal safety, proof, Arm Morello, Microsoft CHERI Ibex
## CHERI ISA refinement over 13 years

<table>
<thead>
<tr>
<th>Year</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010-2012</td>
<td>ISAv1</td>
<td>RISC capability-system model w/64-bit MIPS Capability registers, tagged memory Guarded manipulation of registers</td>
</tr>
<tr>
<td>2012</td>
<td>ISAv2</td>
<td>Extended tagging to capability registers Capability-aware exception handling Boots an MMU-based OS with CHERI support</td>
</tr>
<tr>
<td>2014</td>
<td>ISAv3</td>
<td>Fat pointers + capabilities, compiler support Instructions to optimize hybrid code Sealed capabilities, CCall/CRReturn</td>
</tr>
<tr>
<td>2015</td>
<td>ISAv4</td>
<td>MMU-CHERI integration (TLB permissions) ISA support for compressed 128-bit capabilities HW-accelerated domain switching Multicore instructions: full suite of LL/SC variants</td>
</tr>
<tr>
<td>2016</td>
<td>ISAv5</td>
<td>CHERI-128 compressed capability model Improved generated code efficiency Initial in-kernel privilege limitations</td>
</tr>
<tr>
<td>2017</td>
<td>ISAv6</td>
<td>Mature kernel privilege limitations Further generated code efficiency Architectural portability: CHERI-x86, CHERI-RISC-V sketches Exception-free domain transition</td>
</tr>
<tr>
<td>2019</td>
<td>ISAv7</td>
<td>Architectural performance optimization for C++ applications Microarchitectural side-channel resistance features Architecture-neutral CHERI protection model All instruction pseudocode from a formal model CHERI Concentrate capability compression Improved C-language support, dynamic linking, sentry capabilities Elaborated CHERI-RISC-V ISA 64-bit capabilities for 32-bit architectures Accelerated tag operations for temporal memory safety</td>
</tr>
<tr>
<td>2020</td>
<td>ISAv8</td>
<td>MMU temporal memory-safety assist; e.g., capability dirty bit Optimizations for sentry capabilities CHERI-RISC-V privileged support, general maturity Further C-language semantics improvements</td>
</tr>
<tr>
<td>2023</td>
<td>ISAv9</td>
<td>CHERI-RISC-V now the reference architecture CHERI-RISC-V maturity for standardization, including tag stripping CHERI-x86 userspace sketch maturity</td>
</tr>
</tbody>
</table>
CHERI ISA v7 – June 2019

- Key features:
  - Architecture-neutral CHERI model
  - Elaborated CHERI-RISC-V ISA
  - CHERI Concentrate capability compression (IEEE TC 2019)
  - Side-channel resistance features
  - Improved C-language compatibility, dynamic linkage, performance optimizations (ASPLOS 2019)
  - Experimental features including 64-bit capabilities for 32-bit architectures (ICCD 2018), temporal safety (IEEE Micro 2019, IEEE SSP 2020)
  - All instruction pseudocode derived from Sail formal models, formally proven properties (IEEE SSP 2020)
CHERI ISAv8 (October 2020)

- Key changes
  - Capability compression is now part of the abstract protection model
  - Both 32-bit and 64-bit architectural address sizes are supported
  - Various experimental features are now mature: Sentry capabilities, CHERI-RISC-V
  - New MMU temporal memory-safety mechanisms based on load-side barrier model (ASPLOS 2024)
  - CHERI microarchitecture chapter
- Synchronized with Arm Morello (IEEE MICRO Journal 2023)
CHERI ISA v9 (September 2023)

Most recent specification version (released about every two years)

Key changes

- CHERI-RISC-V is now the reference architecture
- Numerous CHERI-RISC-V improvements for standardization
- CHERI-MIPS removed
- CHERI-x86 better elaborated
- Complete shift to tag stripping from exception throwing for non-monotonic capability operations
CHERI: From research to product

- Starting in 2010, hardware-software co-design using FPGAs, open-source software, created a CHERI-MIPS CPU + software stack at Cambridge and SRI
- Arm collaboration from 2014, supported by DARPA; Arm Morello CPU, SoC; board announced 2019, with support from InnovateUK; Shipped in Jan 2022
  - High-performance 2.5GHz, multicore, out-of-order prototype CPU design
- Microsoft CHERIoT RISC-V CPU open sourced Feb 2023
  - 3-stage pipeline for small embedded / IoT / root-of-trust; based on Ibex
  - lowRISC FPGA board for CHERIoT announced Sep 2023; ship date in 2024
  - SCI SoC using CHERIoT announced Nov 2023; ship date in 2024
- Codasip CHERI RISC-V CPU announced in Nov 2023; ship date in 2024
  - 7-stage in-order processor line targeted at high-end embedded
CHERI PROTECTION MODEL
AND ARCHITECTURE
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.

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 design goals and approach

• De-conflate memory virtualization and protection
  • Memory Management Units (MMUs) protect by location (address)
  • CHERI protects existing references (pointers) to code, data, objects
  • Reusing existing pointer indirection avoids adding new architectural table lookups

• Architectural mechanism that enforces software policies
  • Language-based properties – e.g., referential, spatial, and temporal integrity (C/C++ compiler, linkers, OS model, runtime, …)
  • New software abstractions – e.g., software compartmentalization (confined objects for in-address-space isolation, …)
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**
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**
  - Valid pointers, once removed, cannot be reintroduced solely unless rederived from other valid pointers
  - E.g., Received network data cannot be interpreted as a code/data pointer – even previously leaked pointers
- **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
- These primitives not only allow us to implement **strong spatial and temporal memory protection**, but also higher-level policies such as **scalable software compartmentalization**
CHERI 128-bit capabilities

- **Capabilities** extend integer memory addresses
- **Metadata** (bounds, permissions, ...) control how they may be used
- **Guarded manipulation** controls how capabilities may be manipulated; e.g., **provenance validity** and **monotonicity**
- **Tags** protect capability integrity/derivation in registers + memory
CHERI 128-bit capabilities

- **CHERI capabilities** are a new architectural data type extending integer addresses
- **Capability metadata** (bounds, permissions, …) control how a capability may be used
- **Capability tags** protect the integrity + safe derivation of capabilities in registers and memory

![Diagram showing Capability metadata and tags](image)

- **GPRs extended to 129 bits**
  - $pc$
  - $ra$
  - $a1$
  - $a0$
  - $pcc$
  - $c4$
  - $c3$

- **Capability width**
  - Upper bound
  - Pointer address
  - Lower bound

- **Virtual address space**
- **Tagged physical memory**
  - Memory allocation
  - 1-bit tags added to DRAM

### Bounds compressed relative to address

<table>
<thead>
<tr>
<th>Type</th>
<th>Offset</th>
<th>Permissions</th>
<th>Otype</th>
<th>Bound Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lower</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Upper</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Memory allocation

- Privileged
- User
- Data
- Stack
- Code
- Text
- Bss
- Heap

### Virtual address space

- **Upper bound**: $d$
- **Pointer address**: $d$
- **Lower bound**: -
- **Capability**: $v$
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
  - Floating-point compressed 64-bit lower and upper bounds
  - Strengthens larger allocation alignment requirements
  - Out-of-bounds pointer support essential to C-language compatibility

- **Permissions** limit operations – e.g., load, store, fetch

- **Sealing**: immutable, non-dereferenceable capabilities – used for non-monotonic transitions
Merged capability register file + tagged memory
(as found in Morello and CHERI-RISC-V; MIPS used a split register file)

- **64-bit general-purpose registers (GPRs)** are extended with 64 bits of metadata and a 1-bit validity tag
- **Program counter (PC)** is extended to be the program-counter capability ($PCC$)
- **Default data capability ($DDC$)** constrains legacy integer-relative ISA load and store instructions
- **Tagged memory** protects capability-sized and -aligned words in DRAM by adding a 1-bit validity tag
- **Various system mechanisms** are extended (e.g., capability-instruction enable control register, new TLB/PTE permission bits, exception code extensions, saved exception stack pointers and vectors become capabilities, etc.)
CHERI-RISC-V formal ISA model

- CHERI RISC-V ISA model extends RISC-V formal ISA specification, in Sail
- Sail RISC-V ISA specification developed by UCam + SRI
  - Selected as official RISC-V spec by the Foundation
  - Sail is a custom first-order imperative language for expressing ISA specifications, usable by engineers but with static type checking of bitvector lengths etc.
  - The Sail spec is inlined in versions of the unprivileged and privileged RISC-V manuals
- Sail auto-generates a C emulator, theorem-prover definitions, and SMT definitions
- Machinery for configuring model WRT YAML from compliance group
  - Readable, precise definition of ISA behavior, usable as test oracle for testing hardware against and for software bring-up, and providing prover definitions if you want more rigorous reasoning
- Paper on earlier CHERI-MIPS L3 modelling and proof work at IEEE SSP 2020
- Most recently completed monotonicity proofs for the Arm Morello architecture
**Abstract**

There are two fundamental problems here. First, mainstream systems are typically developed using conventional software stack to make use of these. Despite decades of research, memory safety bugs are still limited to the permission to access the pointed-to object. On the other hand, a conventional C/C++ program implicitly uses permission to its entire memory region for accesses via a pointer, making run only with the permissions it needs to function. For example, a conventional C/C++ program implicitly uses permission to its entire memory region for accesses via a pointer, making run only with the permissions it needs to function. For example, a conventional C/C++ program implicitly uses permission to its entire memory region for accesses via a pointer, making run only with the permissions it needs to function. For example, a conventional C/C++ program implicitly uses permission to its entire memory region for accesses via a pointer, making run only with the permissions it needs to function.

We do this for CHERI, an architecture with hardware support for fine-grained memory protection, aiming to provide practically deployable performance and compatibility. CHERI achieves this by extending commodity architectures with new security mechanisms, and adapting a design that support and improve normal engineering practice – as a path for existing software. CHERI is a maturing research architecture design provides specific security properties.

First, the protection mechanisms provided by the mainstream processor architecture and C/C++ language abstractions, dating back to the 1970s and become ever more acute. These methods have historically sufficed commercially for much on test-and-debug methods, with (at best) prose specifications. Foreground, they fail to prevent large numbers on a larger scale, the JavaScript execution engine of a browser, and implementation and software stack. We use formal models of processor architecture, with its accompanying hardware implementation and software stack. We use formal models of processor architecture, with its accompanying hardware implementation and software stack. We use formal models of processor architecture, with its accompanying hardware implementation and software stack. We use formal models of processor architecture, with its accompanying hardware implementation and software stack.

We formalise key intended security properties of the design and engineering process, both in lightweight ways and for running software, and for test generation – and for formal documentation, in emulators used as a test oracle for hardware verification. We formalise key intended security properties of the design and engineering process, both in lightweight ways and for running software, and for test generation – and for formal documentation, in emulators used as a test oracle for hardware verification. We formalise key intended security properties of the design and engineering process, both in lightweight ways and for running software, and for test generation – and for formal documentation, in emulators used as a test oracle for hardware verification. We formalise key intended security properties of the design and engineering process, both in lightweight ways and for running software, and for test generation – and for formal documentation, in emulators used as a test oracle for hardware verification.

In this paper we show how more rigorous engineering methods can be used to improve as-""
CHERI MICROARCHITECTURE AND PROTOTYPES
Architectural primitives for software security

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**.
CHERI hardware research prototypes

• Original research based on our home-grown pipelined BERI MIPS core (CHERI-MIPS)

• We have transitioned our CHERI research to extended versions of open-source off-the-shelf BSV RISC-V cores (CHERI-RISC-V)
  • CHERI-Piccolo 3-stage pipeline, 32-bit, no MMU
  • CHERI-Flute 5-stage pipeline, 32- or 64-bit, MMU
  • CHERI-Toooba Superscalar, 64-bit, MMU

• Novel microarchitectural contributions include capability compression model, tagged memory implementation techniques

• All of our CPU designs are open source

• We also provide a QEMU full-system and userlevel simulators for CHERI-RISC-V
Changes to the Piccolo core (RISC-V 3-stage pipeline):
- capability arithmetic
- capability load/store operations with bounds checking
- extended exception model
- PC becomes a capability (PCC)
- default data capability (DDC)
- new control/status registers
- merged integer & capability register file

Memory subsystem:
- AXI user-field added to transport tag bits & data width doubled
- caches extended to include tags

DRAM changes:
- New tag controller uses a hierarchical tag table to efficiently store tag bits backed by top of DRAM
Microarchitectural tag storage for off-the-shelf DRAM

Efficient Tagged Memory
Alexandre Joannou*, Jonathan Woodruff*, Robert Kovacsics*, Simon W. Moore*, Alex Bradbury*, Hongyan Xia*, Robert N. M. Watson*, David Chisnall†, Michael Roe†, Brooks Davis†, Edward Napierala*, John Baldwin†, Khilan Gudka†, Peter G. Neumann†, Alfredo Mazzinghi*, Alex Richardson*, Stacey Son†, A. Theodore Markettos*

*Computer Laboratory, University of Cambridge, Cambridge, UK †SRI International, Menlo Park, CA, USA
Website: www.cl.cam.ac.uk/research/comparch Website: www.sri.com

Abstract—We characterize the cache behavior of an in-memory tag table and demonstrate that an optimized implementation can typically achieve a near-zero memory traffic overhead. Both industry and academia have repeatedly demonstrated tagged memory as a key mechanism to enable enforcement of power-ful security invariants, including capabilities, pointer integrity, and information-flow tracking. A single-bit tag use cases into information flow ([3], [9], [11], [14]–[16], [20]–[25]). We might divide ambitious and useful functions that solve difficult problems into three categories: no tag, a single-bit tag (SBT), or a multi-bit tag (MBT).

Ⅱ. Shift from flat to hierarchal tag table to hold tags in DRAM

• Exploit inconsistent density of tags in physical memory
• Reduces DRAM access overhead for a variety of workloads

• Published in the IEEE International Conference on Computer Design (ICCD) 2017

For simplicity, we identify three points in the tagging design space: no tag, a single-bit tag (SBT), or a multi-bit tag (MBT). We sweep parameter spaces and evaluate against a range of benchmarks with diverse characteristics. We identify a practical approach: an in-DRAM tag table caches whose hit rates can be surprisingly high, with a tag cache next to the DRAM controller, including identifying a practical approach: an in-DRAM tag table cache-ability. We then demonstrate tag-table compression patterns sufficiently to inform implementations or further optimizations.

The contributions of this paper include:
• A characterization of an elegantly simple and highly effective compression scheme for three tag use cases,
• A characterization of the dynamic workload of tag-memory is used by many research systems to enforce security from early days of computer architecture ([1], [2], and tagged tagging pointer integrity ([3], [4], enabling unforgeable capabilities in nearly unmodified programs, including tracking watchpoints, and information-flow tracking. A single-bit tag shadowspace is the most commonly proposed requirement, as enforcing types. We survey various tag shadowspace approaches tagging untyped data word and any number of new hardware-agnostic types (and therefore interpret the tag bit directly ([11], [12]), or instrumenting computer systems with high performance. We might divide ambitious and useful functions that solve difficult problems into three categories: no tag, a single-bit tag (SBT), or a multi-bit tag (MBT).

Various tagged architectures share the requirement of a shadowspace: no tag, a single-bit tag (SBT), or a multi-bit tag (MBT). We sweep parameter spaces and evaluate against a range of benchmarks with diverse characteristics. We identify a practical approach: an in-DRAM tag table cache-ability. We then demonstrate tag-table compression patterns sufficiently to inform implementations or further optimizations.

For simplicity, we identify three points in the tagging design space: no tag, a single-bit tag (SBT), or a multi-bit tag (MBT). We sweep parameter spaces and evaluate against a range of benchmarks with diverse characteristics. We identify a practical approach: an in-DRAM tag table cache-ability. We then demonstrate tag-table compression patterns sufficiently to inform implementations or further optimizations.

The contributions of this paper include:
• A characterization of an elegantly simple and highly effective compression scheme for three tag use cases,
• A characterization of the dynamic workload of tag-memory is used by many research systems to enforce security from early days of computer architecture ([1], [2], and tagged tagging pointer integrity ([3], [4], enabling unforgeable capabilities in nearly unmodified programs, including tracking watchpoints, and information-flow tracking. A single-bit tag shadowspace is the most commonly proposed requirement, as enforcing types. We survey various tag shadowspace approaches tagging untyped data word and any number of new hardware-agnostic types (and therefore interpret the tag bit directly ([11], [12]), or instrumenting computer systems with high performance. We might divide ambitious and useful functions that solve difficult problems into three categories: no tag, a single-bit tag (SBT), or a multi-bit tag (MBT).

Various tagged architectures share the requirement of a shadowspace: no tag, a single-bit tag (SBT), or a multi-bit tag (MBT). We sweep parameter spaces and evaluate against a range of benchmarks with diverse characteristics. We identify a practical approach: an in-DRAM tag table cache-ability. We then demonstrate tag-table compression patterns sufficiently to inform implementations or further optimizations.

The contributions of this paper include:
• A characterization of an elegantly simple and highly effective compression scheme for three tag use cases,
• A characterization of the dynamic workload of tag-memory is used by many research systems to enforce security from early days of computer architecture ([1], [2], and tagged tagging pointer integrity ([3], [4], enabling unforgeable capabilities in nearly unmodified programs, including tracking watchpoints, and information-flow tracking. A single-bit tag shadowspace is the most commonly proposed requirement, as enforcing types. We survey various tag shadowspace approaches tagging untyped data word and any number of new hardware-agnostic types (and therefore interpret the tag bit directly ([11], [12]), or instrumenting computer systems with high performance. We might divide ambitious and useful functions that solve difficult problems into three categories: no tag, a single-bit tag (SBT), or a multi-bit tag (MBT).

Various tagged architectures share the requirement of a shadowspace: no tag, a single-bit tag (SBT), or a multi-bit tag (MBT). We sweep parameter spaces and evaluate against a range of benchmarks with diverse characteristics. We identify a practical approach: an in-DRAM tag table cache-ability. We then demonstrate tag-table compression patterns sufficiently to inform implementations or further optimizations.

The contributions of this paper include:
• A characterization of an elegantly simple and highly effective compression scheme for three tag use cases,
• A characterization of the dynamic workload of tag-memory is used by many research systems to enforce security from early days of computer architecture ([1], [2], and tagged tagging pointer integrity ([3], [4], enabling unforgeable capabilities in nearly unmodified programs, including tracking watchpoints, and information-flow tracking. A single-bit tag shadowspace is the most commonly proposed requirement, as enforcing types. We survey various tag shadowspace approaches tagging untyped data word and any number of new hardware-agnostic types (and therefore interpret the tag bit directly ([11], [12]), or instrumenting computer systems with high performance. We might divide ambitious and useful functions that solve difficult problems into three categories: no tag, a single-bit tag (SBT), or a multi-bit tag (MBT).

Various tagged architectures share the requirement of a shadowspace: no tag, a single-bit tag (SBT), or a multi-bit tag (MBT). We sweep parameter spaces and evaluate against a range of benchmarks with diverse characteristics. We identify a practical approach: an in-DRAM tag table cache-ability. We then demonstrate tag-table compression patterns sufficiently to inform implementations or further optimizations.

The contributions of this paper include:
• A characterization of an elegantly simple and highly effective compression scheme for three tag use cases,
• A characterization of the dynamic workload of tag-memory is used by many research systems to enforce security from early days of computer architecture ([1], [2], and tagged tagging pointer integrity ([3], [4], enabling unforgeable capabilities in nearly unmodified programs, including tracking watchpoints, and information-flow tracking. A single-bit tag shadowspace is the most commonly proposed requirement, as enforcing types. We survey various tag shadowspace approaches tagging untyped data word and any number of new hardware-agnostic types (and therefore interpret the tag bit directly ([11], [12]), or instrumenting computer systems with high performance. We might divide ambitious and useful functions that solve difficult problems into three categories: no tag, a single-bit tag (SBT), or a multi-bit tag (MBT).

Various tagged architectures share the requirement of a shadowspace: no tag, a single-bit tag (SBT), or a multi-bit tag (MBT). We sweep parameter spaces and evaluate against a range of benchmarks with diverse characteristics. We identify a practical approach: an in-DRAM tag table cache-ability. We then demonstrate tag-table compression patterns sufficiently to inform implementations or further optimizations.

The contributions of this paper include:
• A characterization of an elegantly simple and highly effective compression scheme for three tag use cases,
Compressing capability bounds

CHERI Concentrate: Practical Compressed Capabilities


Abstract—We present CHERI Concentrate, a new fat-pointer compression scheme applied to CHERI, the most developed capability-pointer system at present. Capability fat pointers are a primary candidate to enforce fine-grained and non-bypassable security properties in future computer systems, although increased pointer size can severely affect performance. Thus, several proposals for capability compression have been suggested elsewhere that do not support legacy instruction sets, ignore features critical to the existing software base, and also introduce design inefficiencies to RISC-style processor pipelines. CHERI Concentrate improves on the state-of-the-art region-encoding efficiency, solves important pipeline problems, and eases semantic restrictions of compressed encodings, allowing it to protect a fully legacy software stack. We present the first quantitative analysis of compressed capability encodings, showing that CHERI Concentrate improves on the state-of-the-art for both 32-bit and 64-bit processors.

- Published in IEEE Transactions on Computers, April 2019
- Efficient compressed capabilities for 32-bit and 64-bit processors
  - Reduces size of capabilities from 4x machine word size to 2x
  - Large reduction in cache overheads
  - Efficiently fits into a RISC pipeline with negligible impact on clock frequency
  - Maintains all security and software compatibility properties
Arm Morello (2022)

- $225M government, academia, and industrial research program led by UK Research and Innovation (UKRI)
  - Announced partners: Arm, Google, Microsoft
  - 15+ UK universities with research grants
  - 70+ funded business incubation projects
- Baseline for design: Neoverse N1 core
  - 2.5GHz quad-core, superscalar
  - Implements CHERI extensions
  - Runs full CHERI-enabled software stacks
- Definitely a prototype, but a very powerful one!
  - Roughly a thousand chips manufactured for use by research + development labs

Abstract

Memory safety issues are a persistent source of security vulnerabilities, with conventional architectures and the C/C++ codebase chronically prone to exploitable errors. The CHERI project has explored a novel architectural approach to ameliorate such issues using unforgeable hardware capabilities to implement pointers.

Morello is an Arm experimental platform for evaluation of CHERI in the Arm architecture context, to explore its potential for mass-market adoption. This paper describes the Morello Evaluation Platform; covering the motivation; the functionality of the Morello architectural hardware extensions, their potential for fine-grained memory safety and software compartmentalization; their formally proven security properties; their impact on the microarchitecture of the high-performance out-of-order multi-processor Arm Morello processor; and the software enablement program by Arm, University of Cambridge, and Linaro. Together, this allows a wide range of researchers in both industry and academia to explore and assess the Morello platform.

Introduction

Arm believes that security is the greatest challenge that computing needs to address to meet its full potential. Arm technology is used in products that are transforming every industry by enabling access to data and communications, and by extracting information and meaning from that data. This transformation continues in our society wherever the application of computing resources can make people's lives easier and more connected. Unfortunately, this increasing reliance on computing has created unprecedented opportunities for criminals, as can be seen in the ever-growing cost of cybercrime. In addition, the growing reliance of national infrastructure on technology means that computer security is part of National Security. Given this context, seems likely that the boundaries of the computing revolution will be determined by the security of our computing systems.

There is ample evidence that memory safety issues such as buffer overflows and use-after-free have been a persistent source of vulnerabilities for many years, and this continues in many ecosystems. While languages such as Rust offer the prospect of more inherent memory safety, the reality is that there is a huge body of C and C++ code being used, written, and adapted every day, and there are many undetected vulnerabilities waiting to be exploited. Arm has introduced the Memory Tagging Extensions in recent years to provide a mechanism to help identify memory safety issues, and these have demonstrated that ordinary code has a great number of latent memory safety errors.

For the purpose of open access, the authors have applied a Creative Commons Attribution (CC BY) license.
Microsoft CHERIoT core (2023)

- Production CHERI-extended Ibex microcontroller
- Small-scale microcontroller used in OpenTitan, etc.
- CHERI-RISC-V tuned for small microcontrollers
- Clean-slate memory-safe, compartmentalized embedded OS for high-risk applications
- Open sourced in February 2023
- RISC-V embedded standardization candidate
- Collaboration across Microsoft Research, MSRC, Azure Silicon, and Azure Edge + Platform
- lowRISC Sunburst FPGA board reference platform
- Published in IEEE MICRO 2023
lowRISC Sunburst (announced 2023)

- lowRISC-designed/manufactured low-cost FPGA prototyping platform for CHERIoT
- Open consultation on board design and requirements
- Anticipated ship date in 2024
- Supported by UKRI / DSbD
Codasip (announced 2023)

• Commercial CHERI-RISC-V core based on existing RISC-V IP + tooling product

• Codasip is contributing heavily to the CHERI-RISC-V standardization effort

• RISC-V core baseline is pipelined, multicore, MMU-enabled design
RISC-V CHERI SIG and TG

• Ambition: Standardize CHERI use with the RISC-V ISA, given multiple companies building prototypes and products
  • SIG created in October 2022, SIG chair is Alex Richardson (Google), co-chair Simon Moore (Cambridge)
  • TG created in January 2024, same acting chairs
• SIG has been meeting every two weeks for over a year working through use cases, implications for different microarchitecture, various RISC-V standardization considerations
  • First draft specification on verge of being released for community discussion, review, extension
• First CHERI-RISC-V products won’t conform as standard not complete, but working hard with industrial partners (e.g., Codasip, Google) to ensure useful convergence
HOW SOFTWARE WORKS ON CHERI
Architectural primitives for software security

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 C/C++ MEMORY PROTECTION
Early questions:

- 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?
More recent questions: CHERI implications for software?

• But also higher-level protection properties:

  **Heap temporal memory safety**  Q: Do applications use – or compare pointers after free (e.g., for lockless algorithms)?

  **Safety for custom allocators**  Q: Can application-specific allocators also benefit from spatial and temporal safety?

  **Robustness for code generation**  Q: Can software that intentionally introduces new code – kernels, run-time linkers, language runtimes – benefit?

  **Safe isolation and communication**  Q: Can mutually distrusting software modules communicate safely across strong boundaries?
What do we mean by C/C++ memory safety?

• Complex question, as while memory unsafety is clearly present, neither language defines what memory safety could mean.

• Our thoughts from over a decade working on CHERI:
  • **Memory safety** for C++ is (pragmatically) anything that would have defended you from memory-safety vulnerabilities.
  • **Vulnerability mitigation** deterministically coerces bugs that are currently vulnerabilities back into bugs – i.e., you would no longer urgently patch them.
  • **Exploit mitigation** interferes with attack techniques exploiting a lack of memory safety.
  • **Deterministic mitigation** means that defenses always work regardless of information leakage, attempts to brute force, and so on.

• Our ambition for CHERI C/C++ memory safety is to **mitigate the vast majority** (>70%) of memory-safety vulnerabilities with full determinism.
A space of C memory-protection models

- C does not define a memory-protection model
  - We have therefore had to (organically) grow one

- Optimization goals have been:
  - Works well with CHERI (changing CHERI allowed, subject to PPA)
  - %LoC source-code modification rates
  - ABI / code-generation / optimization model alignment with status quo
  - Dynamic performance overhead (e.g., cycles)
  - Vulnerability mitigation (ideally deterministic)

- There is a rich space of potential memory-protection models
  - Points combine (or not) different protection options
  - E.g., Sub-object bounds, heap/stack temporal safety, …
  - Today’s trade-off point hits around 70% of memory-safety vulnerabilities
  - Compartmentalization shifts adversary model to arbitrary code execution
Memory-safe CHERI C/C++

- Capabilities used to implement all pointers
  - **Implied** – Control-flow pointers, stack pointers, GOTs, PLTs, …
  - **Explicit** – All C/C++-level pointers and references
- Strong referential, spatial, and heap temporal safety
- Minor changes to C/C++ semantics; e.g.,
  - All pointers must have well defined single provenance
  - Increased pointer size and alignment
  - Care required with integer-pointer casts and types
  - Memory-copy implementations may need to preserve tags
Memory protection for the language and the language runtime

**Language-level memory safety**

- 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
  - (e.g., union type confusion)

**Sub-language memory safety**

- Got pointers
- Return addresses
- Stack pointers
- Got pointers
- Vararg array pointers
- C++ v-table pointers
- ELF aux arg pointers
- PLT entry pointers
- Function pointers
- Pointers to global variables
- Pointers to memory mappings
- Pointers to tls variables
- Pointers to sub-objects
- Pointers to stack allocations
- Pointers to heap allocations
• Capabilities are substituted for integer addresses throughout the address space
• Bounds and permissions are minimized by software including the kernel, run-time linker, memory allocator, and compiler-generated code
• Hardware permits fetch, load, and store only through granted capabilities
• Tags ensure integrity and provenance validity of all pointers
struct timezone tz;

time_t get_unix_time(void)
{
    struct timeval tv;
    gettimeofday(&tv, &tz);
    return tv.tv_sec;
}

---

get_unix_time_riscv:
  addi  sp, sp, -32
  sd   ra, 24(sp)
  addi  a0, sp, 8
  .LBB0_1:
    auipc a1, %pcrel_hi(tz)
    addi a1, a1, %pcrel_lo(.LBB0_1)
    call  gettimeofday
    (expands to auipc, possibly cld, cjalr)
  .LBB0_2:
    auipc a1, %pcrel_lo(.LBB0_1)(cal)
  cjalr  cra, ca2
  cld   a0, 0(csp)
  clc   cra, 16(csp)
  cincoffset  csp, csp, 32
  cret

get_unix_time_cheririscv:
  cincoffset  csp, csp, -32
  csc   cra, 16(csp)
  cincoffset  ca0, csp, 0
  csetbounds  ca0, ca0, 16
  .LBB0_1:
    auipc  ca1, %pcrel_hi(tz)
    clcall, %pcrel_lo(.LBB0_1)(cal)
    .LBB0_2:
      auipcc ca2, %pcrel_hi(gettimeofday)
      clcca2, %pcrel_lo(.LBB0_2)(ca2)
      cjalr  cra, ca2
      cld   a0, 0(csp)
      clc   cra, 16(csp)
      cincoffset  csp, csp, 32
      cret

---

- The general code structure is unchanged except that:
  - The integer stack pointer becomes a capability stack pointer
  - The pointer to a local stack allocation becomes capability
  - Compiler-specified bounds are set on the local variable pointer before use
  - The loaded jump target is a capability rather than an integer address

---

RISC-V vs. CHERI-RISC-V generated code

1. Adjust stack address/capability
2. Save return address/capability
3. Create address/capability to local ‘tv’
4. Generate address/capability to global ‘tz’
5. Call gettimeofday()
6. Load return value from ‘tv’
7. Load return address/capability
8. Restore stack address/capability
9. Return
CheriBSD: A pure-capability operating system

• Complete memory- and pointer-safe FreeBSD C/C++ kernel + userspace
  • **OS kernel**: Core OS kernel, file systems, networking, device drivers, …
  • **System libraries**: crt/cs, ld-elf.so, libc, zlib, libxml, libssl, …
  • **System tools and daemons**: echo, sh, ls, openssl, ssh, ssd, …
  • **Applications**: PostgreSQL, nginx, WebKit (C++)

• **Valid provenance, minimized privilege for pointers, implied VAs**
  • Userspace capabilities originate in *kernel-provided roots*
  • Compiler, allocators, run-time linker, etc., refine bounds and perms

• Trading off **privilege minimization, monotonicity, API conformance**
  • Typically in memory management – realloc(), mmap() + mprotect()
Pure-capability UNIX process environment

CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment

Brooks Davis∗
brooks.davis@sri.com

Robert N. M. Watson†
robert.watson@cl.cam.ac.uk

Alexander Richardson†
alexander.richardson@cl.cam.ac.uk

Peter G. Neumann∗
peter.neumann@sri.com

Simon W. Moore†
simon.moore@cl.cam.ac.uk

John Baldwin‡
john@araratriver.co

David Chisnall§

Jessica Clarke†

Nathaniel Vesperino†

• Received best paper award at ASPLOS, April 2019
• Complete pure-capability UNIX OS userspace with spatial memory safety
  • Usable for daily development tasks
  • Almost vast majority of FreeBSD tests pass
  • Management interfaces (e.g. ioctl), debugging, etc., work
• Large, real-world applications have been ported: PostgreSQL and WebKit
Heap temporal memory safety

Cornucopia: Temporal Safety for CHERI Heaps


Abstract—Use-after-free violations of temporal memory safety continue to plague software systems, underpinning many high-impact exploits. The CHERI capability system shows great promise in preventing such vulnerabilities. However, conventional capability systems are not designed to support heap temporal memory safety. In this paper, we present the implementation of heap temporal memory safety for CHERI heaps using capability revocation. While use-after-free heap vulnerabilities are ultimately due to application misuse of the malloc() and free() interface, complete sanitization of this vast key C library interface remains an unsolved problem. Our implementation achieves temporal memory safety for C/C++-language heaps using capability revocation.

- IEEE Symposium on Security and Privacy ("Oakland"), May 2020
- Hardware and software support for deterministic temporal memory safety for C/C++-language heaps using capability revocation
- Hardware enables fast tag searching using MMU-assisted tracking of tagged values, tag controller and cache
Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety (ASPLOS 2024)

- **Cornucopia heap temporal safety** (IEEE SSP 2020), is a GC-inspired, quarantining technique
  - The kernel virtual-memory subsystem tracks “capability dirty” pages
  - A long “stop-the-world” phase - as much as 30 milliseconds measured in practice
- **Cornucopia Reloaded** (ASPLOS 2024) moves to a GC-inspired “load-barrier”
  - VM invariant is that accessible pages have already undergone revocation
  - Depend on 1-bit capability generation added to VM PTEs, implemented by Morello
  - Stop-the-world pauses 10s of microseconds
- Enabled by default in CheriBSD 23.11
Ongoing temporal memory-safety deployment

- Shipped in CheriBSD 23.11 release
  - Experimenting with larger-scale software, such as desktop stack
  - Enabled by default in 23.11 to gain exposure; easy to disable
- Looking for increased experience:
  - Semantic impact on any applications vs. bugs/vulnerabilities discovered
  - Acceptability of performance behavior, optimization opportunities
  - Use in higher-level allocators – e.g., APR, Chromium, etc.
  - Support for strong isolation needed for compartmentalization
  - Enabling safe inter-compartment communication via shared memory
Formal Mechanised Semantics of CHERI C: Capabilities, Undefined Behaviour, and Provenance (ASPLOS 2024)

- Research paper on a formal semantics and behaviour of CHERI C:
  - CHERI C Semantics Design Questions
  - CHERI C executable semantics
  - Validation and Experimental Comparison
- Considers topics such as, “What are compiler optimizations allowed to do when they recognize undefined behavior such as out-of-bounds accesses”
## CHERI C compatibility: CheriBSD Code Changes

<table>
<thead>
<tr>
<th>Area</th>
<th>Files total</th>
<th>Files modified</th>
<th>% files</th>
<th>LoC total</th>
<th>LoC changed</th>
<th>% LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>11,861</td>
<td>896</td>
<td>7.6</td>
<td>6,095k</td>
<td>6,961</td>
<td>0.18</td>
</tr>
<tr>
<td>• Core</td>
<td>7,867</td>
<td>705</td>
<td>9.0</td>
<td>3,195k</td>
<td>5,787</td>
<td>0.18</td>
</tr>
<tr>
<td>• Drivers</td>
<td>3,994</td>
<td>191</td>
<td>4.8</td>
<td>2,900k</td>
<td>1,174</td>
<td>0.04</td>
</tr>
<tr>
<td>Userspace</td>
<td>16,968</td>
<td>649</td>
<td>3.8</td>
<td>5,393k</td>
<td>2,149</td>
<td>0.04</td>
</tr>
<tr>
<td>• Runtimes (excl. libc++)</td>
<td>1,493</td>
<td>233</td>
<td>15.6</td>
<td>207k</td>
<td>989</td>
<td>0.48</td>
</tr>
<tr>
<td>• libc++</td>
<td>227</td>
<td>17</td>
<td>7.5</td>
<td>114k</td>
<td>133</td>
<td>0.12</td>
</tr>
<tr>
<td>• Programs and libraries</td>
<td>15,475</td>
<td>416</td>
<td>2.7</td>
<td>5,186k</td>
<td>1,160</td>
<td>0.02</td>
</tr>
</tbody>
</table>

**Notes:**
- Numbers from cloc counting modified files and lines for identifiable C, C++, and assembly files
- Kernel includes changes to be a hybrid program and most changes to be a pure-capability program
  - Also includes most of support for CHERI-MIPS, CHERI-RISC-V, Morello
  - Count includes partial support for 32 and 64-bit FreeBSD and Linux binaries.
  - 67 files and 25k LoC added to core in addition to modifications
  - Most generated code excluded, some existing code could likely be generated
Pure-capability CheriBSD kernel

- Full UNIX operating-system kernel compiled with CHERI C
  - Roughly 2.4MLoC core kernel excluding device drivers
  - Referential safety for all explicit and implied pointers
  - Spatial safety for mappings, stack and heap allocations, globals; with sub-object bounds
  - Temporal memory safety is not yet supported, work is being planned.
- 1.4% LoC change, 7.7% files changed
  - Includes support for hybrid kernel with CheriABI userspace, which requires capability annotations for system-call arguments
  - We will have better data on a pure purecap kernel soon, stripping hybrid support, which should substantially reduce %LoC change
Pure-capability CheriBSD kernel: Vulnerabilities

• Security analysis based on retrospective vulnerability study over 22 years
• 56% of total vulnerabilities (113 of 200) are memory-safety; of these:
  • 54% mitigated through referential and spatial safety (implemented); of these, 8% of memory safety w/sub-object
  • 72% mitigated if including heap temporal memory safety (white-board design)
  • 26% unmitigated are uninitialized values; at least 5% of memory safety would likely be mitigated by LLVM stack initialization
  • Handful of unmitigated vulnerabilities: stack temporal safety, VM vulnerabilities, …
• 1 FTE for ~2.5 years for MIPS, RISC-V, and Morello; most time on common code
• Be aware of selection bias in vulnerability discovery – e.g., KASAN finding use-after-free vulnerabilities with fuzzing, but not subobject bounds overflows
Pure-capability CheriBSD kernel: Sub-object bounds

struct example {
    int ex_int;
    char ex_arr[16];
    int ex_secret;
};

// Example allocation
struct example *p;
p = malloc(sizeof(*p));

// Narrow bounds on ex_arr
char *arrp = p->ex_arr;

// Over flowing copy triggers
// bounds violation
memcpy(arrp, src, 20);

- Sub-object bounds are an optional compilation mode for CHERI C
- Additional protection at slightly greater friction due to containerof()
Pure-capability CheriBSD kernel: Sub-object bounds

- Automatic bounds narrowing does not cause porting problems in most of the cases:
  - Core kernel requires changes to ~80 files (13% of the core kernel files)
  - Changes consist of simple annotations, magnitude of changes is small.
  - Detecting sub-object bounds incompatibilities can only be done at run-time. Limited by test coverage.
  - Kernel drivers are known to have poor test coverage. Likely that additional changes will be required here.

- Kernel uses "subobject-safe" policy for bounds narrowing:
  - Enforces sub-object bounds everywhere except for array indexing
  - In practice this does not affect the ability to mitigate past vulnerabilities
Capability graph visualization and analysis

• Pointers are now directly visible in hardware – in memory, ISA-level traces, and so on
  • We can directly analyze capability delegation with CHERI

• New extraction tools scan virtual addresses spaces and binaries to enable:
  • Visualization
  • Validation
  • Debugging and optimization

• Allows direct analysis of attacker-visible resources and attack surfaces
<table>
<thead>
<tr>
<th>START</th>
<th>END</th>
<th>PRT</th>
<th>ro</th>
<th>rw</th>
<th>rx</th>
<th>rwx</th>
<th>TOTAL</th>
<th>DENSITY</th>
<th>FLAGS</th>
<th>TP</th>
<th>PATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x101000</td>
<td>0x101000</td>
<td>----</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>------</td>
<td>----</td>
<td>print-pointer</td>
</tr>
<tr>
<td>0x110000</td>
<td>0x110000</td>
<td>r-xR-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>CN</td>
<td>vn</td>
<td>print-pointer(.plt)</td>
</tr>
<tr>
<td>0x111000</td>
<td>0x120000</td>
<td>----</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>------</td>
<td>----</td>
<td>print-pointer</td>
</tr>
<tr>
<td>0x120000</td>
<td>0x122000</td>
<td>r-R-</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>21</td>
<td>0.24</td>
<td>C</td>
<td>vn</td>
<td>print-pointer(.got)</td>
</tr>
<tr>
<td>0x122000</td>
<td>0x131000</td>
<td>----</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>------</td>
<td>----</td>
<td>print-pointer</td>
</tr>
<tr>
<td>0x131000</td>
<td>0x132000</td>
<td>rw-RW</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>0</td>
<td>14</td>
<td>0.16</td>
<td>------</td>
<td>sw</td>
<td>print-pointer(.bss)</td>
</tr>
<tr>
<td>0x40131000</td>
<td>0x40139000</td>
<td>r-R-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>CN</td>
<td>vn</td>
<td>ld-elf.so.1</td>
</tr>
<tr>
<td>0x40139000</td>
<td>0x40148000</td>
<td>----</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>------</td>
<td>----</td>
<td>ld-elf.so.1</td>
</tr>
<tr>
<td>0x40148000</td>
<td>0x40162000</td>
<td>r-xR-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>------</td>
<td>----</td>
<td>ld-elf.so.1</td>
</tr>
<tr>
<td>0x40162000</td>
<td>0x40171000</td>
<td>----</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>------</td>
<td>----</td>
<td>ld-elf.so.1</td>
</tr>
<tr>
<td>0x40171000</td>
<td>0x40174000</td>
<td>rw-RW</td>
<td>462</td>
<td>89</td>
<td>5</td>
<td>0</td>
<td>556</td>
<td>6.37</td>
<td>C</td>
<td>vn</td>
<td>ld-elf.so.1(.got)</td>
</tr>
<tr>
<td>0x40174000</td>
<td>0x40183000</td>
<td>----</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>------</td>
<td>----</td>
<td>ld-elf.so.1</td>
</tr>
<tr>
<td>0x40183000</td>
<td>0x40184000</td>
<td>rw-RW</td>
<td>32</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>39</td>
<td>0.45</td>
<td>C</td>
<td>vn</td>
<td>ld-elf.so.1</td>
</tr>
<tr>
<td>0x40184000</td>
<td>0x40186000</td>
<td>rw-RW</td>
<td>3</td>
<td>36</td>
<td>21</td>
<td>11</td>
<td>71</td>
<td>0.81</td>
<td>------</td>
<td>sw</td>
<td>ld-elf.so.1(.bss)</td>
</tr>
<tr>
<td>0x40186000</td>
<td>0x4018e000</td>
<td>rw-RW</td>
<td>0</td>
<td>316</td>
<td>4</td>
<td>10</td>
<td>330</td>
<td>3.78</td>
<td>------</td>
<td>sw</td>
<td>unknown</td>
</tr>
<tr>
<td>0x4018e000</td>
<td>0x401f0000</td>
<td>----</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>CN</td>
<td>gd</td>
<td>Guard</td>
</tr>
<tr>
<td>0x401f0000</td>
<td>0x40211000</td>
<td>r-R-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>CN</td>
<td>vn</td>
<td>libc.so.7</td>
</tr>
<tr>
<td>0x40211000</td>
<td>0x40230000</td>
<td>----</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>CN</td>
<td>gd</td>
<td>libc.so.7</td>
</tr>
<tr>
<td>0x40230000</td>
<td>0x4035e000</td>
<td>r-xR-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>CN</td>
<td>vn</td>
<td>libc.so.7(.plt)</td>
</tr>
<tr>
<td>0x4035e000</td>
<td>0x4036e000</td>
<td>----</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>------</td>
<td>----</td>
<td>libc.so.7</td>
</tr>
<tr>
<td>0x4036e000</td>
<td>0x4038a000</td>
<td>r-R-</td>
<td>3643</td>
<td>699</td>
<td>537</td>
<td>0</td>
<td>4879</td>
<td>55.92</td>
<td>C</td>
<td>vn</td>
<td>libc.so.7(.got)</td>
</tr>
<tr>
<td>0x4038a000</td>
<td>0x40399000</td>
<td>----</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>------</td>
<td>----</td>
<td>gdb</td>
</tr>
<tr>
<td>0x40399000</td>
<td>0x403a4000</td>
<td>rw-RW</td>
<td>127</td>
<td>49</td>
<td>1151</td>
<td>0</td>
<td>1327</td>
<td>15.21</td>
<td>C</td>
<td>vn</td>
<td>libc.so.7</td>
</tr>
<tr>
<td>0x403a4000</td>
<td>0x407dd000</td>
<td>rw-RW</td>
<td>1</td>
<td>269</td>
<td>4</td>
<td>0</td>
<td>274</td>
<td>3.14</td>
<td>sw</td>
<td>libc.so.7</td>
<td></td>
</tr>
<tr>
<td>0x407dd000</td>
<td>0x407fe000</td>
<td>rw-RW</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0.07</td>
<td>sw</td>
<td>unknown</td>
<td></td>
</tr>
<tr>
<td>0x407fe000</td>
<td>0x40850000</td>
<td>----</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>CN</td>
<td>gd</td>
<td>Guard</td>
</tr>
<tr>
<td>0x40850000</td>
<td>0x40815000</td>
<td>rw-RW</td>
<td>0</td>
<td>57</td>
<td>0</td>
<td>0</td>
<td>57</td>
<td>0.65</td>
<td>sw</td>
<td>unknown</td>
<td></td>
</tr>
<tr>
<td>0x40815000</td>
<td>0x40a15000</td>
<td>rw-RW</td>
<td>0</td>
<td>210</td>
<td>0</td>
<td>0</td>
<td>210</td>
<td>2.41</td>
<td>sw</td>
<td>unknown</td>
<td></td>
</tr>
<tr>
<td>0x40a15000</td>
<td>0x60000000</td>
<td>rw-RW</td>
<td>1</td>
<td>140</td>
<td>0</td>
<td>0</td>
<td>141</td>
<td>1.62</td>
<td>sw</td>
<td>unknown</td>
<td></td>
</tr>
<tr>
<td>0x60000000</td>
<td>0x80000000</td>
<td>rw-RW</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>22</td>
<td>0.25</td>
<td>sw</td>
<td>unknown</td>
<td></td>
</tr>
<tr>
<td>0x80000000</td>
<td>0xfffffbff0000</td>
<td>rw-RW</td>
<td>1</td>
<td>28</td>
<td>1</td>
<td>1</td>
<td>31</td>
<td>0.36</td>
<td>sw</td>
<td>unknown</td>
<td></td>
</tr>
<tr>
<td>0xfffffbff0000</td>
<td>0xfffffbff8000</td>
<td>----</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>------</td>
<td>----</td>
<td>gdb</td>
</tr>
<tr>
<td>0xfffffbff8000</td>
<td>0xfffffff6000</td>
<td>----</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>------</td>
<td>----</td>
<td>gdb</td>
</tr>
<tr>
<td>0xfffffff6000</td>
<td>0xfffffff8000</td>
<td>rw-RW</td>
<td>7</td>
<td>640</td>
<td>101</td>
<td>0</td>
<td>748</td>
<td>8.57</td>
<td>---D</td>
<td>sw</td>
<td>Stack</td>
</tr>
<tr>
<td>0xfffffff8000</td>
<td>0x100000000000</td>
<td>r-x-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>ph</td>
<td>unknown</td>
<td></td>
</tr>
</tbody>
</table>
MSRC: Security analysis of CHERI C/C++

- Study analyzed all 2019 critical security vulnerabilities
- Metric: “Poses a risk to customers → requires a software update”
- 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 build CHERI experience
- Potential adversarial techniques post-CHERI
- Recently shifted from CHERI-MIPS to CHERI-RISC-V and Arm Morello
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/
Security Analysis of CHERI ISA

Is it possible to get to a state where memory safety issues would be deterministically mitigated? Our quest to mitigate memory corruption vulnerabilities led us to examine CHERI (Capability Hardware Enhanced RISC Instructions), which provides memory protection features against many exploited vulnerabilities, or in other words, an architectural solution that breaks exploits. We’ve looked at how CHERI would break class-specific categories of vulnerabilities and considered additional mitigations to put in place to get to a comprehensive solution. We’ve assessed the theoretical impact of CHERI on all the memory safety vulnerabilities we received in 2019, and concluded that in its current state, and combined with other mitigations, it would have deterministically mitigated at least two thirds of all those issues.

We’ve reviewed revision 7 and used CheriBSD running under QEMU as a test environment. In this research, we’ve also looked for weaknesses in the model and ended up developing exploits for various security issues using CheriBSD and qtwebkit. We’ve highlighted several areas that warrant improvements, such as vulnerability classes that CHERI doesn’t mitigate at the architectural level, the importance of using reliable and CHERI compliant memory management mechanisms, and multiple exploitation primitives that would still allow memory corruption issues to be exploited. While CHERI does a fantastic job at breaking spatial safety issues, more is needed to tackle temporal and type safety issues.

Your feedback is extremely important to us as there’s certainly much more to discover and mitigate. We’re looking forward to your comments on our paper.

Nicolas Joly, Saif ElSherei, Saar Amar – Microsoft Security Response Center (MSRC)

https://msrc-blog.microsoft.com/2020/10/14/security-analysis-of-cheri-isa/
Ease of adoption compared to high-level languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Approximate open-source LoC*</th>
<th>Memory safe</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>10,317,799,775</td>
<td>✗ → ✓ with CHERI</td>
</tr>
<tr>
<td>C++</td>
<td>2,937,552,905</td>
<td>✗ → ✓ with CHERI</td>
</tr>
<tr>
<td>Java</td>
<td>2,614,800,470</td>
<td>✓</td>
</tr>
<tr>
<td>Rust</td>
<td>39,538,172</td>
<td>✓</td>
</tr>
</tbody>
</table>

Worth pondering: In the past 6 months, the CHERI project has adapted more lines of open-source code to memory safety than the Rust project has created in its entire history.

* Synopsys Black Duck Open Hub: [https://www.openhub.net/languages](https://www.openhub.net/languages) - Stats taken 13 December 2023
Could we achieve practical memory safety* for multi-BLoC C/C++ software stacks within 4 years without a ground-up rewrite?

* There's a very long discussion to have about what “memory-safe C/C++” means, but Microsoft’s practical definition of ”deterministically mitigates security vulnerabilities” seems a good place to start.
How should people ask for memory safety?

• Transition appears to be even harder than developing the technology in the first place

• One key challenge is how people can ask for memory safety
  • Poorly satisfied by today’s mitigation techniques – stack canaries, PAC, ASLR, ...
  • Well satisfied by “up and coming” technologies such as CHERI, Rust, etc.

• How can you request (and be satisfied that you will receive) memory safety on your government procurement form?

• Will require engagement with technical and procedural aspects of the problem

• Host a series of workshops focused on both ..? Your thoughts very welcome!
**How does CHERI relate to other non-C/C++ memory-safety technologies?**

<table>
<thead>
<tr>
<th>CHERI C/C++</th>
<th>Rust</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires new, multi-vendor hardware rollout</td>
<td>Requires rewrite of all source code in a new programming language</td>
</tr>
<tr>
<td>Modest changes even to large software packages (Often around ~0.02 %LoC)</td>
<td>Extensive use of “unsafe Rust” can undermines safety for TCBs and in some use cases</td>
</tr>
<tr>
<td>Requires more significant changes to specific packages – e.g., kernels, language runtimes</td>
<td></td>
</tr>
<tr>
<td>The rollout can be done incrementally .. Once there is new hardware</td>
<td>The rollout can be done incrementally .. On current hardware</td>
</tr>
<tr>
<td>Dynamic enforcement prevents run-time exploitation – but means that crashes may occur</td>
<td>Most memory-safety bugs eliminated at compile time, supporting design changes to prevent bugs</td>
</tr>
<tr>
<td>Compartmentalization avoids trust in the compiler, handles code generated by adversaries</td>
<td>Strong trust in compiler, and no model for handling code generated by an adversary</td>
</tr>
</tbody>
</table>
But what if we put Rust and CHERI together?

- Research question: Is the whole greater than the sum of the parts?
- Lots of reasons to imagine that this might be true, including:
  - Enable a fully memory-safe software ecosystem without 100% software rewrite
  - Reduce total trust in the Rust compiler, enabling downloadable precompiled Apps, device driver sandboxing, …
  - Enforce basic spatial and temporal memory protection for unsafe Rust
  - Use CHERI sub-language protection with Rust to reduce exposure to compiler bugs, new exploit techniques
  - Contain vulnerabilities in C/C++ libraries and other system TCBs
- But .. All of these ideas unimplemented and unevaluated
CHERI SOFTWARE
COMPARTMENTALISATION
What is software compartmentalization?

- Fine-grained decomposition of a larger software system into isolated modules to constrain the impact of faults or attacks.
- Goals is to minimize privileges yielded by a successful attack, and to limit further attack surfaces.
- Usefully thought about as a graph of interconnected components, where the attacker’s goal is to compromise nodes of the graph providing a route from a point of entry to a specific target.

CheriFreeRTOS components and the application execute in compartments. CHERI contains an attack within TCP/IP compartment, which access neither flash nor the internals of the software update (OTA) compartment.
Software compartmentalization at scale

- Current CPUs limit:
  - The number of compartments and rate of their creation/destruction
  - The frequency of switching between them, especially as compartment count grows
  - The nature and performance of memory sharing between compartments
  - CHERI is intended to improve each of these – by at least an order of magnitude

CHERI contains attack within compartment, preventing access to other data
• Isolated compartments can be created using closed graphs of capabilities, combined with a constrained non-monotonic domain-transition mechanism.
Compartmentalization scalability

• CHERI dramatically improves compartmentalization scalability
  • More compartments
  • More frequent and faster domain transitions
  • Faster shared memory between compartments
  • Many potential use cases – e.g., sandbox processing of each image in a web browser, processing each message in a mail application
  • Unlike memory protection, software compartmentalization requires careful software refactoring to support strong encapsulation, and affects the software operational model

Early benchmarks show a 1-to-2 order of magnitude performance inter-compartment communication improvement compared to conventional designs
Operational models for CHERI compartmentalization

• An architectural protection model enabling new software behavior

• As with virtual memory, multiple operational models can be supported
  
  • E.g., with an MMU: Microkernels, processes, virtual machines, etc.
  
  • How are compartments created/destroyed? Function calls vs. message passing? Signaling, debugging, …?

• We have explored multiple viable CHERI-based models to date, including:

  Isolated dynamic libraries   Efficient but simple sandboxing in processes

  UNIX co-processes          Multiple processes share an address space

• Improved performance and new paradigms using CHERI primitives

• Both will be available in CheriBSD/Morello
Proposed operational models:
Isolated libraries and UNIX co-processes

Isolated dynamically linked libraries

• New API loads libraries into in-process sandboxes.
• Calling functions in isolated libraries performs a domain transition, with overheads comparable to function calls.
• Simple model eschews asynchrony, independent debugging, etc.

UNIX co-processes

• Multiple processes share a single virtual address space, separated using independent CHERI capability graphs.
• CHERI capabilities enable efficient sharing, domain transition.
• Rich model associates UNIX process with each compartment.

Prototype to appear in CheriBSD 22.10; updates in 23.10

Prototype to appear in future CheriBSD release
Example: Robust shared libraries

- User compartments exist within individual UNIX processes (“robust shared libraries”):
  - CHERI isolates compartments within each address spaces
  - Compartment switcher is itself a trusted userspace library
  - Compartments have strict subset of OS rights of the process
- Intra-process domain switches take no architectural exceptions and do not enter the kernel
- Multiple processes + IPC required if differing OS right sets needed
Shared library compartmentalization (1/3)

- Run-time linker limits shared libraries to accesses enabled by ELF
  - Adversary model assumes arbitrary code execution within library
  - Run-time linker delegates capabilities for linked functions, globals via GOT/PLT
  - Domain transitions implemented by trampolines interposed on inter-object calls / returns

- Running prototype on Arm Morello
  - Low measured overheads in early experiments (e.g., ~1% for image decompression sandboxing)
  - Released in CheriBSD 22.12 in December
  - Debugging, tracing, and performance enhancements in CheriBSD 23.10
Shared library compartmentalization (2/3)

- Library compartmentalization inserts domain-transition trampolines into inter-library calls and returns
  - PLT entries are initialized with sealed trampoline capabilities that provide strong encapsulation
  - Per-target trampolines are used for branch-prediction reasons (still more tradeoffs to explore here)
  - A single “return trampoline” provides a branch-predictable reverse transition path
- Trampolines perform a number of operations relating to capability register setup/clearing, setting up return path, stack changes, etc.
Shared library compartmentalization (3/3)

- Domain transitions on inter-library calls + returns
  - Inter-domain frames protect control flow between domains
- Stack temporal safety is hard, so we approximate
  - Per-thread trusted stack tracks domain transitions
  - Reentrant per-thread, library stack pools
  - CHERI sealing mechanism protects code transitions, data pointers from corruption
From shared libraries to kernel modules

• Can this userspace model work in the kernel as well? The kernel is actually:
  • Integrated main binary with kernel run-time linker
  • Collection of kernel modules implementing drivers, services, …
  • The same model likely applies, with suitable adaptation to the kernel run-time environment
  • We are developing an early prototype implementing this model
Example: CHERI co-process model

- CHERI isolates **multiple processes** within a single virtual address space
  - Kernel-provided trusted compartment switcher runs in userspace (actually a microkernel)
  - CHERI-based inter-process memory sharing + domain switching
  - A compartment’s OS rights correspond to the owning process
- Inter-process context switches take **no architectural exceptions** and **do not enter the kernel**
- CHERI can be pitched as **improving IPC performance** while retaining a (largely) conventional process model
CHERI TRANSITION
Morello and CHERI-RISC-V

• We are pursing two CHERI adaptations to post-MIPS ISAs:
  
  • 2014 Joint with Arm, an experimental adaptation of 64-bit ARMv8-A Arm Morello multicore SoC, development board, etc. (announced Oct. 2019; experimental SoC shipped 2022)
  
  • 2017 An experimental adaptation of 32/64-bit RISC-V (open-source research processors on FPGA)
  
• Complete elaborations of the full hardware-software stack for each ISA:
  
  • All aspects of the architectures (e.g., ARMv8-A VM features, etc.)
  
  • Formal models + proofs, hardware implementations, compilers, OSes
  
• Potential for transition through both paths
# CHERI target architectures

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Features</th>
<th>CHERI challenges</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit MIPS</td>
<td>1990s RISC architecture (CHERI baseline)</td>
<td>Our legacy research architecture. Poor code density and addressing modes: harder to differentiate ‘essential’ CHERI costs; few transition opportunities with MIPS</td>
</tr>
<tr>
<td>64-bit ARMv8-A</td>
<td>Mature and widely deployed load-store architecture</td>
<td>Feature-rich; exception-adverse; rich address modes; constrained opcode space; hardware page tables; virtualization features; ecosystem</td>
</tr>
<tr>
<td>32-bit and 64-bit RISC-V</td>
<td>Open RISC ISA in active development (MIPS + 10 years?)</td>
<td>Limited addressing modes (expects micro-op fusion); hardware page tables; only partially standardized; features missing (e.g., hypervisor); immature software stack</td>
</tr>
</tbody>
</table>
What’s the smallest variety of CHERI?

The Portmeiron project is a collaboration between Microsoft Research Cambridge, Microsoft Security Response Center, and Azure Silicon Engineering & Solutions. Over the past year, we have been exploring how to scale the key ideas from CHERI down to tiny cores on the scale of the cheapest microcontrollers. These cores are very different from the desktop and server-class processors that have been the focus of the Morello project.

Microcontrollers are still typically in-order systems with short pipelines and tens to hundreds of kilobytes of local SRAM. In contrast, systems such as Morello have wide and deep pipelines, perform out-of-order execution, and have gigabytes to terabytes of DRAM hidden behind layers of caches and a memory management unit with multiple levels of page tables. There are billions of microcontrollers in the world and they are increasingly likely to be connected to the internet. The lack of virtual memory means that they typically don’t have any kind of process-like abstraction and so run unsafe languages in a single privilege domain.

This project has now reached the stage where we have a working RTOS running existing C/C++ components in compartments. We will be open sourcing the software stack over the coming months and are working to verify a production-quality implementation of our proposed ISA extension based on the lowRISC project’s Ibex core, which we intend to contribute back upstream.

RISC-V CHERI Special Interest Group (SIG)

- Created in early October 2022, SIG acting chair is Alex Richardson (Google)
- Preparing to create first standardization task group pursuing:
  - 64-bit CHERI-RISC-V building on SRI/Cambridge’s ISA
- Once IP issues are resolved, can proceed with second task group:
  - Microcontroller CHERI building on Microsoft’s recent work
- Significant ISA refinement and need for high-quality reference implementation of higher-end 64-bit design
CHERI-x86 seedling

- Explore application of CHERI to the widely used x86 architecture
  - Initial prototype ISA developed and formally modeled
  - Focused on compiler targeted (“userlevel”) instruction set
  - Automatically generated test suite from formal model to enable potential future simulator and hardware implementation
  - Early low-level toolchain support; compiler support now beginning
- Proof-of-concept prototype allows design-space exploration prior to industrial engagement
CHERI-ARM research since 2014

• Since 2014, in collaboration with Arm, we have been pursuing joint research to experimentally incorporate CHERI into ARMv8-A:
  • Develop CHERI as an architecture-neutral and portable protection model implemented in multiple concrete architectures
  • Refine and extend the CHERI architecture – e.g., capability compression, tagging µarch, domain transition, and temporal safety
  • Apply concept of architecture neutrality to the CHERI-enabled software stack, including compiler, OS, and applications
  • Expand software: large-scale application experiments, OS use, debuggers, …
  • Extend work in formal modeling and proofs to an industrial-scale architecture
  • Solve arising practical {hardware, software, …} problems as part of the research
  • Build evidence, demonstrations, SW templates to support potential CHERI adoption
ISCF: Digital Security by Design (UKRI)

- 5-year Digital Security by Design UKRI program: £70M UK gov. funding, £117M UK industrial match, to create CHERI-ARM demonstrator SoC + board with proven ISA

- Leap supply-chain gap that makes adopting new architecture difficult – in particular, validation of concepts in microarchitecture, architecture, and software “at scale”

- Support industrial and academic R&D (EPSRC, ESRC, InnovateUK)

- Baseline CPU is Neoverse N1; reuses existing SoC/board designs

- Collaborative review distillation of CHERI ISAv8; experimental additions relating to temporal safety, compartmentalization

- Science designed allowed: Multiple architectural + microarchitectural design choices for software-based evaluation

- 2020 emulation models; 2022 Morello board shipped!
Challenges with creating substantially new architecture

Required to justify

New Hardware

New Software Models

Required to develop

UK Research and Innovation
Why is Arm interested in the CHERI architecture

• Arm had been working with UoCambridge on CHERI for some 4-5 years
• Big step to addressing security based on strong fundamental principles
• Addresses spatial memory safety robustly and some ideas for temporal safety
  • Memory safety issues reported to be involved with ~70% of vulnerabilities (Matt Miller, BlueHat IL, 2019)
• Has scope to be the foundation of a new mechanism for compartmentalisation
  • Potentially far cheaper than using translation tables
• Interesting scope to address temporal safety issues as well as spatial ones....
• Many of the Arm software vendors are similarly interested in the possibilities of CHERI
  • Microsoft, Google and others have expressed strong interest in exploring the concept...
  • ... but lots of questions about the real-world performance costs and usage models
  • ...understanding the intended usage models is important to refine the architectural features
• But is a novel thing to do with additional costs to the system and software
  • Adding a 129th tag bit has a lot of impacts to the memory system
  • it is an ABI change, so non-trivial costs for compatibility for some uses
IP Position

- Today’s CPU architectures have largely the same basic functionality
  - “Similar but different” approaches to most aspects of system architecture
  - Small scale optimisations exist
- This position very beneficial for the porting of system software
  - Anything that fundamentally changes the system software architecture is likely to be ignored
- Arm believes that this reality needs to continue with capabilities
  - Implication is that we’d like the world’s leading architectures to adopt capabilities
  - The Digital Security by Design program
Arm Morello specification

- Experimental application of CHERI ISAv8 to ARMv8-A
- Much richer base ISA .. Much longer spec - 2,155 pages excluding additional material!
- Describes ISA as implemented in Arm Morello FVP and processor/SoC
- Includes recent features such as sentry and load-side barrier support
The Morello Board

• An Industrial Demonstrator of a Capability architecture
  • Uses a prototype capability extension to the Arm Architecture
    • Prototype is a “superset” of what could be adopted into the Arm architecture
  • Use of a superset of the architecture is very unusual
    • Also unrealistic as a commercial product – there will be some frequency effects
    • However, there are tight timescales so architecture is nearly complete now
  • The superset of the architecture will allow a lot of software experimentation
    • Various different mechanisms for compartmentalisation
    • Collection of features for which the justification is unclear
    • Techniques for holding the capability tag bit

• Architecture will have formally proved security properties (with UoC and UoE)

• Morello Board will be the ONLY physical implementation of this prototype architecture
  • Learnings from these experiments will be adopted into a mainstream extension to the Arm architecture
  • NO COMMITMENT TO FULL BINARY COMPATIBILITY TO THE PROTOTYPE ARCHITECTURE
    • But successful concepts are expected to be carried forward into the architecture and can be reused there
Morello Board overview (subject to change)

- Quad core bespoke high-end CPU with prototype capability extensions
  - Backwards compatibility with v8.2 AArch64-only
  - Based on Neoverse N1 core
    - Multi-issue out-of-order superscalar core with 3 levels of cache
  - Build in 7nm process
  - Targeting clock frequency around 2GHz

- Reasonable performance GPU and Display controller
  - Standard Mali architecture core – not extended with capability
  - Supports Android

- PCIe and CCIx interfaces including to FPGA based accelerators
- FPGA for peripheral expansion
- SBSA compliant system
- 16GB of System Memory (expandable to 32GB – tbc)
Morello Board: Capability Hardware Prototype Platform

- Silicon implementation of a Capability Hardware CPU Instruction Set Architecture
- Implements Morello Profile for A-class Prototype Architecture
- Two clusters each of two Rainier CPUs
- Interconnect and Memory Controller support for tagged memory
- Two channel DDR4 DRAM interface
- PCIe Gen3 and Gen4 x16 interface
- CCIX (Cache Coherent Interconnect for Accelerators) interface
- Mid-range GPU, display processor and HDMI output
- On standard uATX form factor board
Arm Morello Programme: Architectural security goals and known limitations (July 2023)

- Framing security direction and disclaimers:
  - Architectural security aims and experimental validation
  - Constraints of the Armv8.2-A baseline ISA
  - Limitations of the experimental software stack
  - Limitations on the hardware threat model
  - Important to understand what Morello can do – and cannot; e.g.,
    - Has enabled 50+MLoC CHERI C/C++ code corpus
    - No expectation to resist Spectre or Rowhammer

Early performance results from the prototype Morello microarchitecture (September 2023)

- Performance analysis of SPECint 2006 on Morello
  - Reminder: Morello is prototype architecture and microarchitecture; no production optimization cycle possible on DSbD timeline
  - Baseline Morello microarchitecture “as shipped”
  - Modified Morello designs on FPGA addressing discovered limitations / re-tuning parameters
- “Benchmark ABI” and “P128” code models to improve predictions for future mature microarchitecture
- Best available spatial safety overhead on Morello prototype microarchitecture, with refinements, for SPECint 2006: 5.7%
- Worst projected spatial safety overhead on anticipated mature microarchitecture for SPECint 2006: 1.8% - 3.0%

Early performance results from the prototype Morello microarchitecture (live website)

- Live version of the website will be updated as understanding improves
- Currently in sync with TR, but will see further updates in coming months (see Version History)
  - Looking at topics such as the impact of dynamic linking
- Complements Benchmarking Guidance section in Getting Started with CheriBSD

UK EPSRC DSbD research program 2020-2023

- 9 EPSRC projects funded across 10 UK universities
- Several InnovateUK industrial projects supporting exploration, evaluation, demonstration

EPSRC Competition

- £10M Research funding
- £7M from ISCF/DSbD
- £3m from DCMS

- The EPSRC call covered 3 areas:
  - Capability enabled hardware proof and software verification
  - Impact on system software and libraries
  - Future implications of capability enabled Hardware
- Projects starting July-Oct

Selected Projects

- **AppControl**: Enforcing Application Behaviour through Type-Based Constraints
  Dr Wim Vanderbauwhede (University of Glasgow)

- **CapableVMs** – Capable Virtual Machines
  Dr Laurence Tratt (King’s College London) & Dr Jeremy Singer (University of Glasgow)

- **CAPcolerate**: Capabilities for Heterogeneous Accelerators
  Dr Timothy Jones (University of Cambridge)

- **CapC**: Capability C semantics, tools and reasoning
  Dr Mark Batty (University of Kent)

- **CAP-TEE**: Capability Architectures for Trusted Execution
  Dr David Oswald (University of Birmingham)

- **CHaOS**: CHERI for Hypervisors and Operating Systems
  Dr Robert Watson (University of Cambridge)

- **CloudCAP**: Capability-based Isolation for Cloud-Native Applications
  Prof Peter Pietzuch (Imperial College London)

- **HD-Sec**: Holistic Design of Secure Systems on Capability Hardware
  Professor Michael Butler (University of Southampton)

- **SCoCH**: Secure Code for Capability Hardware
  Dr Giles Reger (The University of Manchester)
  Prof Daniel Kroening (University of Oxford)
Digital Security by Design (DSbD) runs the Technology Access Programme (TAP) for UK-based companies to experiment with CHERI and Morello.

- We have collaborated with ~35 companies that have been porting their products or prototyping new projects on Morello boards.
- Several of these companies reported that, using Morello, they found vulnerabilities in their code and analysed past vulnerabilities against CHERI.
DSbD TAP Cohort 1-4

Programme scale so far:
• +15 million lines of code ported to Morello by Cohorts 1, 2, 3 and 4
• 32 networking and learning events
• Multi-sector and cross-discipline involvement

Source: Digital Catapult, DSbD TAP Showcase booklet
DSbD TAP Cohort 1-4: example projects

- Cohort 1, RealVNC: Memory-safe remote access VNC client and server
- Cohort 2, CAN-PHANTOM: Memory-safe CAN-based vehicle immobiliser based on libusb from CheriBSD
- Cohort 3, JET Connectivity: Memory-safe 5g-enabled base station
- Cohort 4, Integrity: Memory-safe and compartmentalised user-space storage stack based on SPDK and DPDK
CHERI REFERENCE SOFTWARE STACK
Why port the CHERI stack to Morello?

• **Validate** the Morello architecture (functional, sufficient)

• **Evaluate** the Morello implementation (performance, energy use, …)

• **Provide reference software semantics** (spatial and temporal safety, compartmentalization, POSIX integration, OS kernel use, …) that will be applicable to other adaptations

• **Act as a template and prototyping platform** for at-scale industrial and academic demonstration, including providing adaptations of common software dependencies (e.g., widely used libraries)

• **Provide a platform for future software research**, asking questions about what we can use CHERI for in {operating systems, compilers, language runtimes, applications, …}

• **Enable a growing academic and industrial community** around CHERI and Morello, including dozens of UK universities and companies associated with DSbD
Caution: Research software!

- The baseline compiler toolchain and OS stack are themselves research
  - This means unknown risks, hard-to-predict schedules, and inevitable direction changes
- Application Binary Interface (ABI) stability
  - ABIs are a key research area; there are 2x Morello ABIs, and there will be [many?] more
  - This limits long-term binary compatibility guarantees for compiled software (for example)
- Software performance optimization with a limited corpus
  - Right now, we’re just happy things are working, but we will get beyond that soon!
- Supporting a large and diverse audience of consumers with different objectives
  - Engineering constraints limit objectives and support (e.g., software updates)
- Software adaptation workload
  - Some code ports trivially (e.g., Qt/KDE stack) and other code doesn’t (e.g., JITs)
CHERI prototype software stack on Morello

- **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

### Open-source application suite
(KDE Plasma, Wayland, WebKit, Python, OpenSSH, nginx, …)

**CheriBSD/Morello** (funded by DARPA and UKRI)  
(Morello and CHERI-RISC-V)

- FreeBSD kernel + userspace, application stack
- Kernel spatial and referential memory protection
- Userspace spatial, referential, and temporal memory protection
- Co-process compartmentalization (development branch)
- Linker-based compartmentalization
- Morello-enabled bhyve Type-2 hypervisor
- ARMv8-A 64-bit binary compatibility for legacy binaries

**Android** (Arm)  
(Morello only)

**Linux** (Arm)  
(Morello only)

### CHERI Clang/LLVM compiler suite, Morello GCC, LLD, LLDB, GDB
(At least) two code generation / ABI targets

- **Hybrid code** is primarily aarch64 but with selected capability use:
  - Kernel: Mostly aarch64 with capability use for system-call arguments, context switching, virtual memory, signals
  - Userspace: Runs off-the-shelf arm64 programs without modification

- **Pure-capability code** implements all data and control-flow pointers with capabilities:
  - Kernel and userspace both spatially and referentially space
  - In the future userspace temporally safe
## FreeBSD base, ports/packages

<table>
<thead>
<tr>
<th>Base</th>
<th>Base FreeBSD OS including kernel and key libraries, shells, daemons, and command-line tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ports</td>
<td>Build infrastructure + FreeBSD adaptation patches – roughly 30,000 mainstream open-source libraries, runtimes, and application</td>
</tr>
<tr>
<td>Packages</td>
<td>Prebuilt binary packages built from ports, installed and managed using the pkg(8) package manager</td>
</tr>
</tbody>
</table>

We provide a full set of ~20K-30K aarch64 (non-CHERI) packages to run on CheriBSD/Morello to use while the CheriABI collection matures.
## Maturing CHERI software artifacts

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>3rd-party packages (Hybrid)</td>
<td>23K <strong>memory-unsafe</strong> software packages with strong functionality expectations</td>
<td>Since May 2022 (22.05 release)</td>
</tr>
<tr>
<td>3rd-party packages (CheriABI)</td>
<td>11K <strong>memory-safe</strong> software packages with mixed functionality expectations</td>
<td>Since May 2022 (22.05 release) Up from 9k packages in 23.11</td>
</tr>
<tr>
<td>Morello GPU device drivers</td>
<td>Memory-safe kernel and user drivers,</td>
<td>Since December 2022 (22.12 release)</td>
</tr>
<tr>
<td>Benchmark ABI support (+3rd-party packages)</td>
<td>Support for modified code generation addressing Morello bounds prediction</td>
<td><strong>Shipping in 23.11</strong> (roughly the same packages as CheriABI)</td>
</tr>
<tr>
<td>Userlevel heap temporal safety</td>
<td>Prototype implements strong temporal safety, developed with Microsoft; testing required</td>
<td><strong>Shipping in 23.11</strong> (pretty experimental)</td>
</tr>
<tr>
<td>Linker-based compartmentalization</td>
<td>Introduces strong encapsulation boundaries around UNIX libraries with no modification</td>
<td>Since 22.12 (very experimental); <strong>Significant improvements in 23.11</strong></td>
</tr>
<tr>
<td>bhyve (Type-2) hypervisor</td>
<td>Prototype boots pure-capability guest OS, validation required</td>
<td><strong>Shipping in 23.11</strong> (very experimental)</td>
</tr>
<tr>
<td>Co-process compartmentalization</td>
<td>Prototype runs some compartmentalized software (e.g., OpenSSL); API co-design</td>
<td>Planning to ship in 2024</td>
</tr>
</tbody>
</table>
Ease of adoption compared to high-level languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Approximate open-source LoC*</th>
<th>Memory safe</th>
<th>Memory safe with CHERI</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>10,317,800,000</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>C++</td>
<td>2,937,550,000</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Java</td>
<td>2,600,000,000</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Rust</td>
<td>39,500,000</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

In the past 6 months, the CHERI project has converted more lines of open-source code to memory safety than the Rust project has created in its entire history.

* Synopsys Black Duck Open Hub: [https://www.openhub.net/languages](https://www.openhub.net/languages)
Could we achieve practical memory safety* for C/C++ desktop/server/embedded stacks within 4 years without a total software rewrite?

* There’s a very long discussion to have about what “memory-safe C/C++” means, but Microsoft’s practical definition of ”deterministically mitigates security vulnerabilities” seems a good place to start.
Getting Started with CheriBSD

- Introduces CheriBSD
- Steps you through installation on a Morello board using a USB stick image that you can download
- Describes third-party package system and pkg64/pkg64c
- Illustrates “hello world” compilation and debugging
- Describes some known issues
- Explains how to get support

https://ctsrd-cheri.github.io/cheribsd-getting-started/
Adversarial CHERI Exercises and Missions

- CHERI training exercises for developers, red teams, and bug bounties
- Adversarial missions where we want to understand exploitation better
- CHERI software adaptation
- Assume a strong level of knowledge about C, code generation, exploitation
  - (E.g., GOTs, PLTs, ROP, and JOP)
- Targets Morello and CHERI-RISC-V

https://ctsrd-cheri.github.io/cheri-exercises/
CHERI software stack support channels

• cheri-cpu.slack.com Slack
  • Visit the CHERI website to request an invitation email/link
• Forthcoming mailing lists (not yet live)
  • cl-cheribsd-announce Low-traffic announcement
  • cl-cheribsd-discuss General discussion and support
  • cl-cheribsd-security Report security issues
• Sundry issue trackers in the github.com/CTSRD-CHERI organization
• Not just “How do I get the software to work”, but also to assist with experimental design, interpreting results, and seeking improvements
How to obtain and install the CHERI software stack

- One build tool to rule them all: cheribuild
  https://github.com/CTSRD-CHERI/cheribuild

- Builds, installs, and/or runs:
  - QEMU CHERI-RISC-V and Morello, Morello FVP
  - CheriBSD/CHERI-RISC-V and Morello disk images
  - Small suite of adapted third-party applications
  - Up and running with one command (CHERI-RISC-V):
    ./cheribuild.py --include-dependencies run-riscv64-purecap
CHERI/MORELLO DESKTOP STUDY
2021 desktop pilot study results

Developed:

- 6 million lines of C/C++ code compiled for memory safety; modest dynamic testing
- Three compartmentalization whiteboard 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

Useful observation to be made about memory safety: Not enough to address the de facto threat model of quite a few libraries …
2022.12 Morello memory-safe desktop software stack

Roughly 30MLoC on a shipping Arm Morello board today, with memory-safe:

- CheriBSD kernel with DRM + Panfrost drivers
- CheriBSD userspace with libraries, OpenSSH, ...
- OpenGL, Wayland display server
- Plasma, KDE base applications including Dolphin, Okular, Konsole.

Also shipping in December 2022 with:

- Aarch64 CHERI/Morello-aware GDB debugger
- 9K CheriABI packages, 20K aarch64 ("legacy") packages; notable exclusions for language runtimes
Now on to the grand challenges

• We are now within reach of an exciting – and historically highly vulnerable – application corpus to which we can apply CHERI protections

• Memory-safe desktop applications at scale – especially those that contain one or more language runtimes:
  • Web browsers
  • Mail readers
  • Office suites

• Extending this to fine-grained compartmentalization as software prototypes mature – library compartmentalization, coprocesses, further models, …

• For example: UKRI- and Google-funded efforts around the Chromium web browser at CapLtd, Kings College London, Arm, and Cambridge
Memory Safety Grand challenge: Google Chromium

• “The real thing”:
  • Foundation for Google Chrome, Microsoft Edge, Microsoft Teams, Electron, …
  • Over 35MLoC, >190 library dependencies
  • V8, an intimidatingly real language runtime
  • Code from numerous diverse origins and in countless forms of idiomatic C and C++
  • Vast wealth of past vulnerabilities to use in evaluation
  • Performance critical components
  • Memory-safety and compartmentalization objectives
  • ~9 staff months so far, most effort went into V8 adaptation
  • V8 now running test suite with complete JIT support

Pilot project supported by UKRI, and Google
CONCLUSION
Some potential software research areas

- **Clean-slate OSes and languages**
  
  Current research has focused on incremental CHERI adoption within current software and languages. How would we design new OSes, languages, etc., assuming CHERI as an ISA baseline?

- **Compilers, language runtimes, and JITs**
  
  How can we mitigate the performance overheads of more pointer-dense executions, such as with language runtimes? Are vulnerabilities in code generated by compilers and JIT susceptible to mitigation using CHERI? How does CHERI break or potentially improve current compiler analyses and optimization?

- **Further C/C++ protections with CHERI**
  
  We have focused on spatial, referential, and temporal memory safety for C/C++. But the CHERI primitives could assist with data-oriented protections, garbage collection, type checking, etc. Could these improve security, and at what performance cost?

- **Safe and managed languages**
  
  Languages such as Java, Rust, C#, OCaml, etc., offer strong safety properties, but frequently depend on C/C++ runtimes and FFI-linked native code. Can CHERI provide stronger foundations for higher-level language stacks?

- **Virtualization**
  
  Can memory protection usefully harden hypervisors? Can we compartmentalize hypervisors? Can CHERI offer a better mechanism for virtualizing code than an MMU?

- **Debuggers and tracing**
  
  Debugging/tracing tools rely on high levels of privilege to operate. How can we reduce their privilege to mitigate vulnerabilities in these tools? With stronger architectural semantics, is new dynamic analysis possible?

- **Software compartmentalization tools**
  
  Granular software compartmentalization offers vulnerability mitigation through privilege reduction and strong encapsulation. How should current applications be refactored, and new applications be designed, to accomplish maintainable and more secure software?

- **Security evaluation and adversarial research**
  
  What is the impact of CHERI on known vulnerabilities and attack techniques? How does a CHERI-aware attacker change their behavior? Could formal models and proofs support stronger security arguments for CHERI?
Conclusion

• New architectural primitives require rich HW and SW 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

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


Lessons learned: Split vs. merged register files

• CHERI-MIPS has **split register files** following coprocessor conventions
• … but new register files add control logic, increasing area overhead
• Instead merge register files along the lines of 32-bit → 64-bit extension
• Key design choice in CHERI-RISC-V: Implement both approaches, evaluate
From hybrid-capability code to pure-capability code

- **n64 MIPS ABI**: hybrid-capability code
  - **Early investigation** – manual annotation and C semantics
  - Many pointers are integers (including syscall arguments, most implied VAs)
- **CheriABI**: pure-capability code
  - **More recently** – fully automatic use of capabilities wherever possible
  - All pointers, implied virtual addresses are capabilities (inc. syscall arguments)
- Now investigating pure-capability kernel
OS changes required for CheriABI
(A grand tour of low-level OS behavior)

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

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

**Approach:** Prototype (1) “pure-capability” **CHERI C/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

<table>
<thead>
<tr>
<th></th>
<th>Pointer + integer integrity, prov.</th>
<th>Pointer size &amp; alignment</th>
<th>Monotonicity</th>
<th>Calling conventions</th>
<th>Unsupported features</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSD headers</td>
<td>11</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>BSD libraries</td>
<td>83</td>
<td>36</td>
<td>4</td>
<td>41</td>
<td>22</td>
</tr>
<tr>
<td>BSD programs</td>
<td>24</td>
<td>9</td>
<td>1</td>
<td>11</td>
<td>2</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Pass</th>
<th>Fail*</th>
<th>Skip</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS</td>
<td>3501 (91%)</td>
<td>90</td>
<td>244</td>
<td>3835</td>
</tr>
<tr>
<td>Pure capability</td>
<td>3301 (90%)</td>
<td>122</td>
<td>246</td>
<td>3669</td>
</tr>
</tbody>
</table>

*Test failure investigation remains a work-in progress; we believe these can be resolved*
Evaluating memory-protection impact

• Adversarial / historical vulnerability analysis
  ✓ Pointer integrity, provenance validity prevent ROP, JOP
  ✓ Pointer provenance: Stack Clash (2017)

• Existing test suites – e.g., BOdiagsuite (buffer overflows)

<table>
<thead>
<tr>
<th></th>
<th>OK</th>
<th>min</th>
<th>med</th>
<th>large</th>
</tr>
</thead>
<tbody>
<tr>
<td>mips64</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>175</td>
</tr>
<tr>
<td>CheriABI</td>
<td>0</td>
<td>279</td>
<td>289</td>
<td>291</td>
</tr>
<tr>
<td>LLVM Address Sanitizer (asan) on x86</td>
<td>0</td>
<td>276</td>
<td>286</td>
<td>286</td>
</tr>
</tbody>
</table>


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