CHERI-RISC-V: standardising an open-source CHERI-enhanced ISA

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

University of Cambridge and SRI International
CHERITech in Glasgow
31 March 2023
Approved for public release; distribution is unlimited.

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

This work was also supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 ("CTSRD"), with additional support from FA8750-11-C-0249 ("MRC2"), HR0011-18-C-0016 ("ECATS"), FA8650-18-C-7809 ("CIFV"), and HR001122C0110 ("ETC") as part of the DARPA CRASH, MRC, and SSITH research programs. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

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

- CHERI = Capability Hardware Enhanced RISC Instructions
  - CHERI is a new hardware technology that mitigates software security vulnerabilities
- Developed by the University of Cambridge and SRI International starting in 2010, supported by DARPA and others
- Arm collaboration from 2014
- From 2019: UKRI Digital Security by Design initiative has brought in many more collaborators and led to the Arm Morello prototype chip+system
Reminder: Why develop CHERI?

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

Ian Levy, NCSC

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

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

www.chromium.org/Home/chromium-security/memory-safety
Software Memory Safety

Executive summary
Modern society relies heavily on software-based automation, implicitly trusting developers to write software that operates in the expected way and cannot be compromised for malicious purposes. While developers often perform rigorous testing to prepare the logic in software for surprising conditions, exploitable software vulnerabilities are still frequently based on memory issues. Examples include overflowing a memory buffer and leveraging issues with how software allocates and de-allocates memory. Microsoft® revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. [1] Google® also found a similar percentage of memory safety vulnerabilities over several years in

https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF
Example 1

HeartBleed

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

SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).

User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about snakes but not too long. User Karen wants to change account password to "BloodHorse")

User Meg wants these 4 letters: BIRD. There are currently 346 connections open. User Brendan uploaded the file "visualize.txt".

Note: Files for IP 375.381.83.17 are in /tmp/files-3843.
User Meg wants the connection. Jake requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about snakes but not too long. User Karen wants to change account password to "ColHeBaSt". User Amber requests pages.
Went wrong? How do we do better?

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

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

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

- **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 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**
Principles CHERI aims to uphold

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

• The **principle of least privilege**
  • Reduce the attack surface using software compartmentalization
  • Mitigates known and unknown exploits
  • Approach: highly scalable and efficient compartmentalization
CHERI-RISC-V ISA
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
There are two fundamental problems here. First, mainstream architectures with new security mechanisms, and adapting a capability model to a type-safe language, or to radically change hardware mechanisms provided by the mainstream processor architecture and software stack to make use of these.

The CHERI Context

Architecture design provides specific security properties.

A. The CHERI Context

B. CHERI: A Checking, Encapsulated Register-Integer architecture

C. Formal proof of compartmentalization for CHERI

D. Formal ISA models CHERI-MIPS, CHERI-RISC-V, and Morello

E. Formal proof of compartmentalization for CHERI-MIPS, Morello

Verified Security for the Morello Capability-enhanced Prototype Arm Architecture

Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell

University of Cambridge, Cambridge, UK

first.last@cl.cam.ac.uk
Merged capability register file + tagged memory
(as found in CHERI-RISC-V and ARM Morello)

- **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 MICROARCHITECTURE
Early CHERI Capabilities

- ISCA 2014 paper: Revisiting RISC in the Age of Risk
- 256b capability with bounds and permissions but no address!

![Figure 1: Memory capability](image)

- Separate capability register file in capability coprocessor

![Figure 2: BERI pipeline with capability coprocessor](image)

- Too big (cache footprint)
- Expensive to implement
- Difficult to use
Minimising the Cost of Capabilities

• Include an address in the capability so that all pointers can be turned into capabilities
  • This simplified the software model
• Compress capabilities into a 128-bit format
  • Reduces data-cache footprint + extra DRAM bandwidth
  • ARM were particularly concerned about power from extra DRAM traffic
• On CHERI-RISC-V we extend the integer register file to hold capabilities
  • Reduces the amount of rename logic required, forwarding logic, register space, etc.
Capability Compression

Capabilities encode three 64-bit fields (plus permissions, etc.):

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

- Larger objects require greater alignment
- Address must be “near” the Top and Bottom
CHERI Concentrate: Practical Compressed Capabilities

Jonathan Woodruff \textsuperscript{©}, Alexandre Joannou, \textit{Member, IEEE}, Hongyan Xia \textsuperscript{©}, Anthony Fox, Robert M. Norton \textsuperscript{©}, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, \textit{Fellow, IEEE}, Robert N. M. Watson, and Simon W. Moore \textsuperscript{©}, \textit{Senior Member, IEEE}

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 existing best capability-bounds format, by inferring the most significant bit of the Top field and by encoding the exponent such that provides maximum precision for small objects, spending bits to encode an exponent only for objects, spending bits to encode an exponent only for larger and less common objects.

- Published in IEEE Transactions on Computers, October 2019
- Has all the maths and formal proof in it…
Tips on Implementing CHERI Concentrate

• CHERI concentrate doesn’t require much logic, but it isn’t easy to implement so that it is small and correct
  • We used formal verification (symbolic proof) to check key invariant properties
  • Conventional testing is insufficient because the state space is large and there are many corner cases
• Tip: use our library of base functions and **DO NOT write your own!**
  https://github.com/CTSRD-CHERI/cheri-cap-lib
• C library to support software that needs to manipulate capabilities:
  https://github.com/CTSRD-CHERI/cheri-compressed-cap
Tagging Capabilities

• Capabilities have a hidden validity tag
  • In registers and memory
• Tag bit is critical to security
  • Conventional operations (arith, memory) clear the tag
  • Only capability instructions preserve the tag and guarantee monotonic decrease in rights
• One hidden bit per 128-bits avoids using other integrity measures (no crypto needed…)

Propagating tags from registers to DRAM

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

Temporal and Spatial Hits vs. Line Size
for Earley-Boyer, 256KiB tag cache, 8-way set associative

- 1-byte line of tags covers 128B of data
- 64-byte line of tags covers 8KiB block of data
Hierarchical Tag Compression

- Size tag cache line length to 64-byte DDR4 burst transfer size
  ⇒ one line covers tags for 8KiB of memory (128-bit capabilities)
- Many lines don’t contain tags (code, large blocks of data, disk cache, etc.)
  - So handling tag sparseness is important
- Only want to pay for tagging when needed
Tag Compression

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

![Diagram showing 2-level tag table structure]

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

Tags for a 8KiB of data

leaf table

root table

64 bytes

...
Tag Cache Optimisations

- Hierarchical compression
  Root-level bit can eliminate a leaf-level group

- Silent write elimination
  Don’t mark tag cache line dirty if not modified

- Empty line fabrication/invalidation
  Create line in the cache when leaf cache-line gets its first tag, invalidate without writeback when leaf cache-line becomes clear
Benchmarks in Hardware

DRAM Traffic Overhead in FPGA Implementation

Note: MiBench overheads with tag compression are approximately zero

Almost zero overhead with tag compression
CHERI-RISC-V SOFTWARE STACK
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)
What are CHERI’s implications for software?

- Efficient fine-grained architectural memory protection enforces:
  - Provenance validity: Q: Where do pointers come from?
  - Integrity: Q: How do pointers move in practice?
  - Bounds, permissions: Q: What rights should pointers carry?
  - Monotonicity: Q: Can real software play by these rules?

- Scalable fine-grained software compartmentalization

Q: Can we construct isolation and controlled communication using integrity, provenance, bounds, permissions, and monotonicity?

Q: Can sealed capabilities, controlled non-monotonicity, and capability-based sharing enable safe, efficient compartmentalization?
CHERI C/C++ MEMORY PROTECTION
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)
CHERI-based pure-capability process memory

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

```c
struct timezone tz;

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

```riscv
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)
    ld a0, 8(sp)
    ld ra, 24(sp)
    addi sp, sp, 32
    ret
```

```cheri
get_unix_time_cheririscv:
    cincoffset csp, csp, -32
    csc cra, 16(csp)
    cincoffset ca0, csp, 0
    csetbounds ca0, ca0, 16
.LBB0_1:
    auipc cal, %captab_pcrel_hi(tz)
    clcal1, %pcrel_lo(.LBB0_1)(cal)
.LBB0_2:
    auipcc cal2, %captab_pcrel_hi(gettimeofday)
    clca2, %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
CheriBSD: A pure-capability operating system

• Complete memory- and pointer-safe FreeBSD C/C++ kernel + userspace
  • **OS kernel**: Core OS kernel, filesystems, networking, device drivers, …
  • **System libraries**: `crt/csu`, `ld-elf.so`, `libc`, `zlib`, `libxml`, `libssl`, …
  • **System tools and daemons**: `echo`, `sh`, `ls`, `openssl`, `ssh`, `sshd`, …
  • **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()`
CheriBSD 22.12 (December 2022)

~100MLoC of spatially memory-safe C/C++:
- FreeBSD UNIX kernel w/all drivers
- FreeBSD userspace: libraries, tools, and key daemons (e.g., OpenSSH)
- OpenGL, Wayland display server
- Plasma, KDE base applications including Dolphin, Okular, Konsole
- 9K CheriABI (memory-safe) packages

Also shipped in December 2022 with:
- aarch64 CHERI/Morello-aware GDB
- 20K aarch64 (legacy) packages
- Experimental library compartmentalization

Shipping in June 2023 (we hope):
- Heap temporal memory safety (w/Microsoft)
- CHERI-enabled hypervisor
- Memory-safe Google’s Chromium
## 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><strong>Kernel</strong></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><strong>Userspace</strong></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
C/C++ compatibility: WebKit - JSC 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>JSC-C</td>
<td>3368</td>
<td>148</td>
<td>4.4</td>
<td>550k</td>
<td>2217</td>
<td>0.40</td>
</tr>
<tr>
<td>JSC-JIT</td>
<td>3368</td>
<td>339</td>
<td>10.1</td>
<td>550k</td>
<td>7581</td>
<td>1.38</td>
</tr>
</tbody>
</table>

Notes:
- JSC-C is a port of the C-language JavaScriptCore interpreter backend
- JSC-JIT includes support for a meta-assembly language interpreter and JIT compiler
- Runs SunSpider JavaScript benchmarks to completion
- Language runtimes represent worst-case in compatibility for CHERI
  - Porting assembly interpreter and JIT compiler requires targeting new encodings
- Changes reported here did not target diff minimization
  - Prioritized debugging and multiple configurations (including integer offsets into bounded JS heap) for performance and security evaluation
  - Some changes may not be required with modern CHERI compiler
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§  david.chisnall@cl.cam.ac.uk
Jessica Clarke‡  jessica.clarke@cl.cam.ac.uk
Nathaniel Vessey-Reddy‡  nathaniel.vessey-reddy@cl.cam.ac.uk

* University of Cambridge, Cambridge, UK
† Microsoft Research, Cambridge, UK
‡ Ararat River Consulting, Walnut Creek, CA, United States
§ Google Inc., London, UK

• 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


*University of Cambridge, Cambridge, UK; †SRI International, Menlo Park, CA, USA; §Microsoft Research, Cambridge, UK; ‡Ararat River Consulting, Walnut Creek, CA, USA

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 addressing these violations. However, a new threat arises as the CHERI capability system is exposed to external, untrusted code that is allowed to access to program memory. To address this threat, we introduce Cornucopia, an architectural extension to CHERI that combines capabilities, temporal memory safety, and tag controllers to provide advanced memory protection.

While use-after-free heap vulnerabilities are ultimately due to application misuse of the malloc() and free() interface, complete sanitization of the vast "heap" of C and 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
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

Security Research & Defense / By MSRC Team / October 14, 2020 /
Memory Corruption, Memory Safety, Secure Development, Security Research

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/
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.
- **Active area of research; early prototype available for co-processes**

Prototype to appear in CheriBSD 22.10

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
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 desktop 3-month study: Key outcomes

One person in 3 months:

- Ported 6 million lines of C/C++ code compiled for memory safety; modest dynamic testing
- Three compartmentalization case studies in Qt/KDE

Evaluation results:

- 0.026% LoC modification rate across full corpus for memory safety
- 73.8% mitigation rate across full corpus, using memory safety and compartmentalization

http://www.capabilitieslimited.co.uk/pdfs/20210917-capltd-cheri-desktop-report-version1-FINAL.pdf
Grand challenge in progress: Google Chromium

- Google Chrome, Microsoft Edge, Microsoft Teams, Electron, …
- “The real thing”:
  - 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
  - Performance critical components
- Current state
  - In-progress adaptation to memory-safe C/C++, with 98% compiling
  - Current technical challenge: V8 runtime
  - “Just one last bug needs to be fixed in V8” (ask again in April)
- Pilot jointly funded by UKRI, Google
Where to learn more?

• Project web pages:

• An Introduction to CHERI, Technical Report UCAM-CL-TR-941, Computer Laboratory, September 2019

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


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
EXPLOITATION PATHS
CHERI research and development timeline

**Years 1-2:** Research platform, prototype architecture

**Year 2:** Hybrid C/OS model, compartment model

**Year 4:** Efficiency, CheriABI/C/C++/linker, ARMv8-A

**Year 8-9:** RISC-V, temporal safety, formal proof

Over 150 researcher years of effort by Cambridge & SRI
Many engineer years by Arm

LAW 2010: Capabilities revisited

RESolve 2012: Hybrid MMU/capability model

Oct. 2011: Capability microkernel runs sandbox on FPGA

Jul. 2012: LLVM generates CHERI code

Nov. 2011: FPGA tablet + CHERI-specific microkernel

Oct. 2010: CTSRD project begins work

Nov. 2012: Sandboxed code on CheriBSD; live FPGA-base Trojan mitigation demo

Sep. 2014: MIT LLVM team live Heartbleed mitigation demo

Jun. 2012: CheriBSD capability context switching

July 2019: CheriBSD temporal

Jul. 2010: CTSRD proposal submitted

Sept. 2015: CheriABI pure-capability processor

Oct. 2019: ESP32 microcontroller

July 2018: CHERI-ARM

POPL 2019: C pointer provenance

ASPD 2019: Pure-capability UNIX userspace

MICRO 2019: Temporal memory-safety feasibility study

IEEE TCS 2019: Compressed capabilities

ICCD 2018: CheriRTOS, 32-bit ISAs
Bridging the commercialisation chasm

Proven Technology

Fundamental Principles

University

Research in the Lab

Simulation/Prototyping

Industry

Real World

Fundamental Principles

Proven Technology

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

Arm Morello CHERIoT

Simulation/Prototyping

CHRIOt talk from Microsoft later today
UK Industry Strategy Challenge Fund: Digital Security by Design

• £90M UK gov. funding, >£200M 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)
  • Technology Access Program in 4th round (https://www.dsbd.tech/technology-access-programme/)
  • Ongoing collaboration reviewing and distilling {essential, desirable, experimental} CHERI features for use in SoC
  • Science designed allowed: Support multiple architectural design choices for software-based evaluation once fabricated
  • 2020 emulation models; 2021 “Morello” board delivery
Morello Demonstrator Board

https://www.arm.com/architecture/cpu/morello
Open Source Stack: Research and Deployment

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

Project URL:
http://cheri-cpu.org/

links to:
https://www.cl.cam.ac.uk/research/security/ctsrd/
Also:
http://CheriBSD.org
Open-Source CHERI-RISC-V Cores Implemented

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

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

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

• CHERI provides the hardware with more semantic knowledge of what the programmer intended
  • Toward the principle of intentional use
• Efficient pointer integrity and bounds checking
  • Eliminates buffer overflow/over-read attacks (finally!)
• Provide scalable, efficient compartmentalisation
  • Allows the principle of least privilege to be exploited to mitigate known and unknown attacks
  • Large performance improvement over process-based compartmentalisation
• Working with industry & open-source community to deploy the technology
  • Thanks to sponsors: DARPA, ARM, Google, EPSRC, ESRC, HEIF, Isaac Newton Trust, Thales E-Security, HP Labs