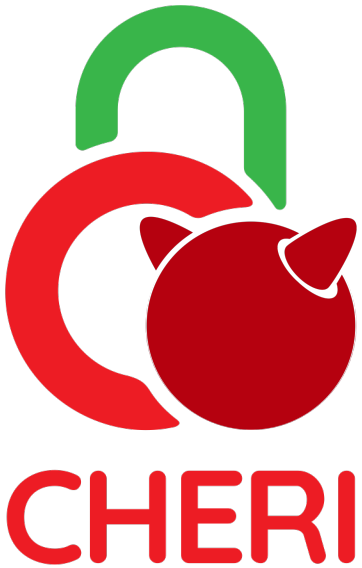
Bringing memory safety to BSD with CHERI

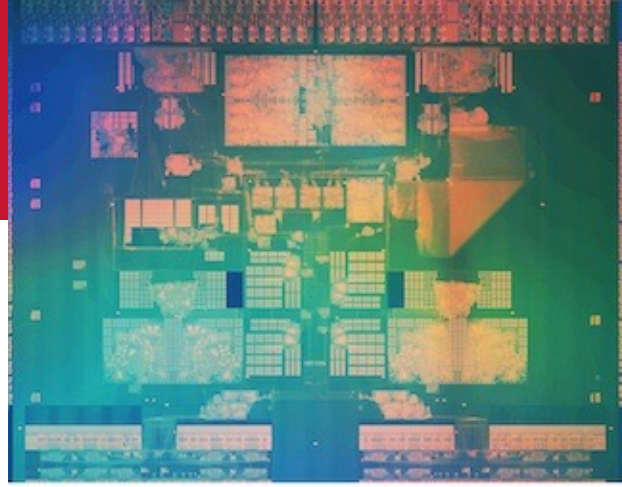
AsiaBSDCon 2026, Taipei Taiwan



Brooks Davis

Capabilities Limited

What is CHERI

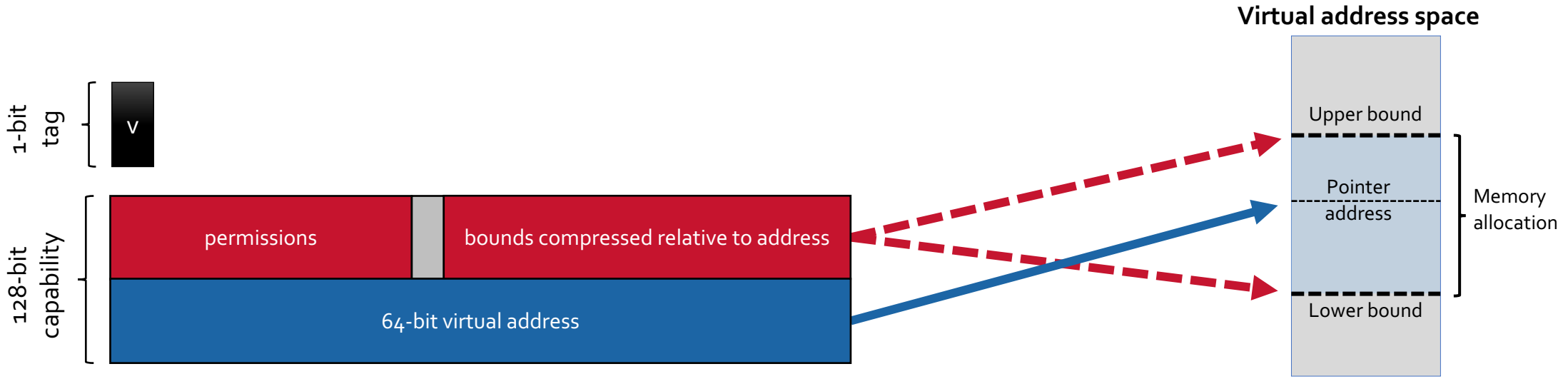


Morello chip – 7nm quad-core multi-GHz Arm processor and SoC with CHERI extensions, Arm, 2022.

- 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
- There are now **multiple industrial implementations** (Arm, Microsoft, Codasip, SCI, SeqAI, …)

CHERI

CHERI 128-bit capabilities

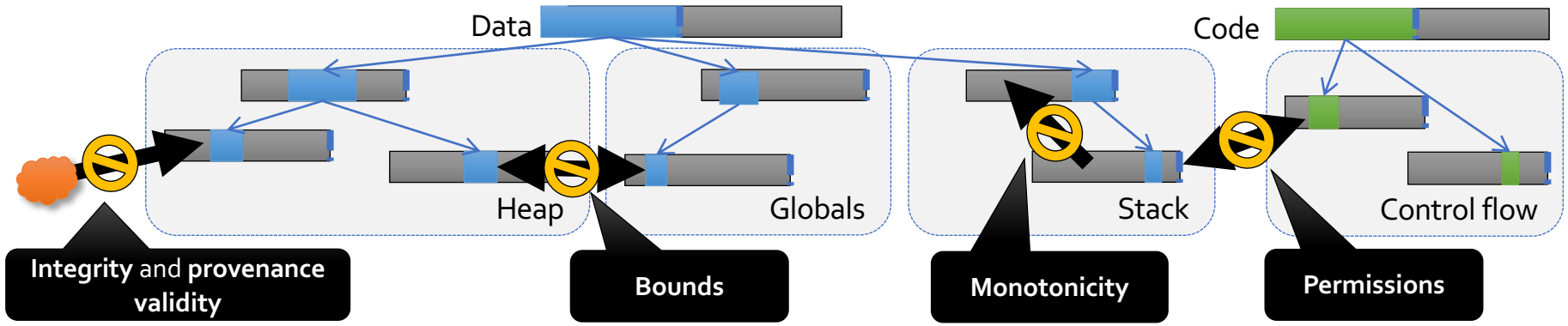


- **Capabilities** extend **integer memory addresses**
- **Metadata** (bounds, permissions, …) control how they may be used
- **Guarded manipulation** controls how capabilities may be manipulated; e.g., **provenance validity** and **monotonicity**
- **Tags** protect capability integrity/derivation in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

Hardware guarantees correct usage

CHERI

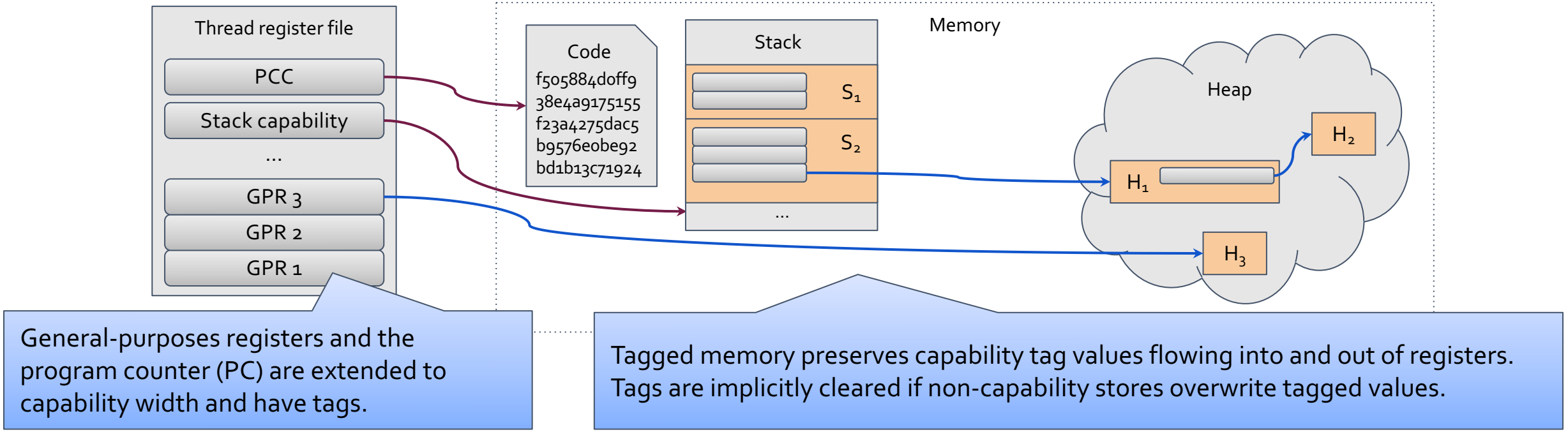
CHERI enforces protection semantics for pointers



- **Integrity** and **provenance validity** ensure that valid pointers are derived from other valid pointers via valid transformations; **invalid pointers cannot be used**
- **Bounds** prevent pointers from being manipulated to access the wrong object
- **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 compartmentalisation**

CHERI

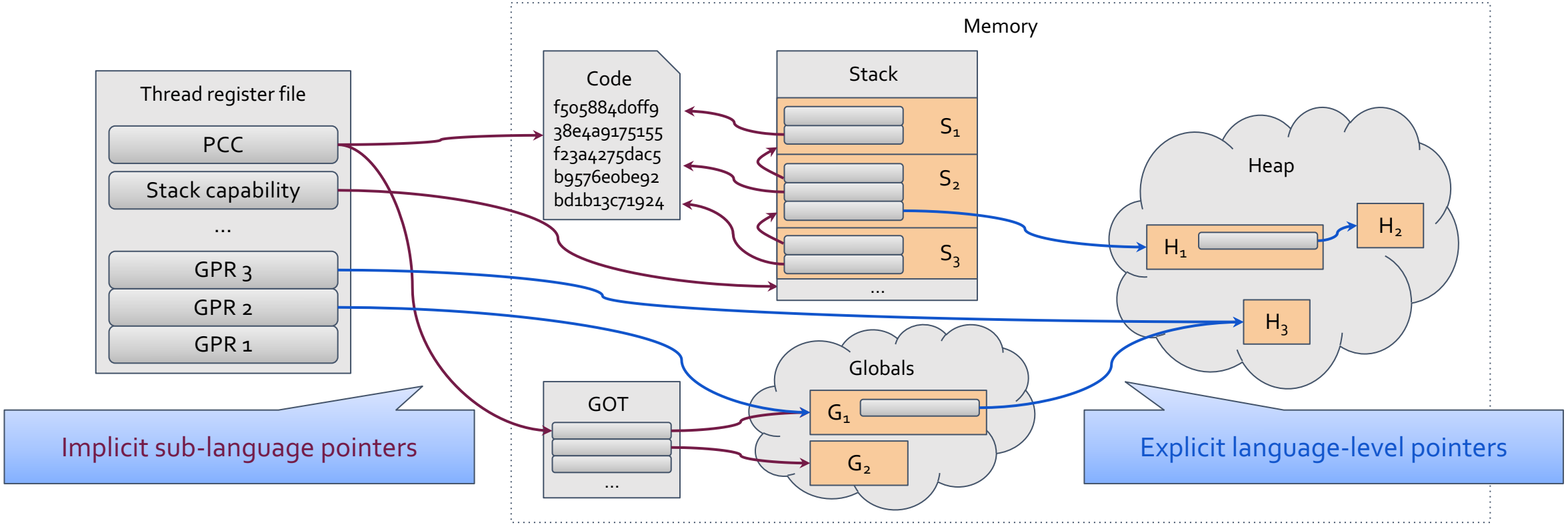
Implementing C/C++ memory safety with CHERI (1/2)



- Whereas conventional C/C++ use **integer pointers**, CHERI C/C++ uses **capability pointers**
  - C/C++ data types are laid out to hold wider capability-width, capability-aligned pointers
  - Code generation is pointer-aware and uses explicit load/store-capability instructions
- Software TCBs restrict capability bounds and permissions as code runs
- Hardware continuously enforces capability protections on all pointer manipulation and use

CHERI

Implementing C/C++ memory safety with CHERI (2/2)

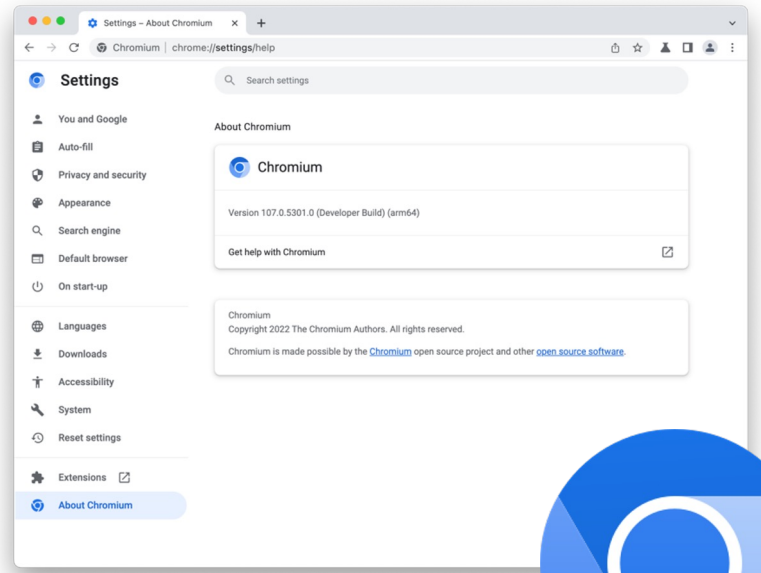


- Protections are applied to all pointer types in compiled code and the run-time environment:
  - **Implicit sub-language pointers** such as GOT entries, stack pointers, return addresses
  - **Explicit language-level pointers** to globals, stack and heap allocations, functions
- Software TCBs refine **bounds** and **permissions** during execution
  - E.g., the mmap() system call, run-time linker, heap allocator, and stack allocator

CHERI

Grand challenge application: Google Chromium

- Foundation for Google Chrome, Microsoft Edge, Microsoft Teams, Electron, …
  - ~**27MLoC** base source code with (at least) **7** compilers, of which **6** are Just-In-Time compilers (JITs)
  - ~**47MLoC** including >**760** library dependencies
  - V8, an intimidatingly real runtime for JavaScript and WASM **(~2MLoC)**
  - Code from diverse origins and in idiomatic C and C++
  - Vast wealth of past vulnerabilities to use in evaluation (~1 critical memory-safety vulnerability every 2 days in 2023)
  - Performance critical components, especially V8
- Chromium able to browse increasingly complex web sites
  - Roughly 18 staff months of effort so far; close to functional browsing
  - .. but .. compare to size of **100-member Chromium security team**!
  - Chromium base (27MLoC) **~0.045%** LoC change; V8 (2MLoC) **~0.8%** LoC change
- Pilot project funded by UKRI, Google, DARPA



CHERI

Chromium: CVE-2023-4863 ("BLASTPASS")

CHERI deterministically mitigates this Chromium vulnerability without any awareness about the nature, location, or origin of the vulnerability during development.

This memory-safety vulnerability was discovered "in the wild" following targeted attacks against victims in the US using NSO Group's Pegasus:

- **Naturally occurring vulnerability** in Google's libwebp image library
  - Heap-memory buffer overflow exploitable for remote arbitrary code execution
  - Undiscovered for years despite fuzzing due to complexity of Huffman coding logic
- Affected Chrome, Edge, and WebKit
  - 1st-party code for Google
  - 3rd-party for Apple and Microsoft
  - Zero interaction exploitation of Apple iOS
- No prior awareness of this CVE in our work
- 0% LoC changes to webp for use on CHERI

CHERI

AIxCC nginx evaluation with CHERI

DARPA AI Cyber Challenge (AIxCC) applies AI to vulnerability discovery and patching

- Reintroduced past vulnerabilities / created new vulnerabilities, in open-source software – Challenge Problems (CPs)

One such problem adds vulnerabilities to the open-source nginx web server

- Already running with CHERI spatial/temporal safety following DSTL study
- AIxCC added 14 memory-safety Challenge Problem Vulnerabilities (CPVs)
- ASAN reproduction trigger provided for each CPV

The AIxCC team didn't know about CHERI, and the CHERI team didn't know about the AIxCC nginx work – so this is a great science experiment:

- **Query:** How would CHERI fare in mitigating vulnerabilities selected without regard to the protection approach?
- **Methodology:** Adversarial analysis (e.g., exploit aarch64 version, and attempt exploitation for CHERI version)

CHERI

CPV					
1	heap-buffer-overflow	CVE-2009-2629	yes	yes	
2	heap-buffer-overflow	Synthetic	yes	yes	
3	heap-buffer-overflow	Synthetic	yes	yes	
4	heap-buffer-overflow	Synthetic	yes	yes	
5	SEGV	Synthetic	yes *	yes *	Unlikely to be directly exploitable for RCE on aarch64
8	heap-buffer-overflow	Synthetic	yes	yes	
9	heap-use-after-free	Synthetic	no	yes	Mitigation argument required code-specific analysis as revocation is deferred while quarantining
10	double-free	Synthetic	yes **	yes	
11	heap-use-after-free	Synthetic	no	yes	Mitigation argument required code-specific analysis as revocation is deferred while quarantining; Aliasing blocked but some information leak still
~~12~~	~~heap-buffer-overflow~~	~~Synthetic~~	-	-	Not implemented on FreeBSD
13	SEGV	Synthetic	yes *	yes *	Unlikely to be directly exploitable for RCE on aarch64
14	global-buffer-overflow	Synthetic	yes	yes	
15	SEGV	Synthetic	yes	yes	Sub-object bounds provide better protection
17	heap-use-after-free	Synthetic	no ***	yes	Mitigation argument required code-specific analysis

* On Morello, page faults take priority over tag faults for NULL pointer dereferences, but both are checked architecturally

** Double frees are safely disregarded in CheriBSD 24.05; from CheriBSD 25.03 onwards, they will trap immediately

*** Use-after-free is followed by a double free that does trap; see also **

CHERI

Bonus discoveries

					Notes
CPV 15	-	Spatial safety vulnerability is a sub-object overflow	yes	yes	ASAN aborts because the provided trigger overflows the full heap allocation, but in fact we believe this vulnerability can be triggered by overflowing only the sub-object
BCPV 1	-	Linear buffer overflow	yes	yes	Occurs after a double free in CPV 10 in which ASAN had stopped execution, but also reachable via other inputs
BCPV 2	-	Null byte poisoning	yes	yes	One-byte overflow overwrites string terminator in AIxCC added feature

- Using reproduction triggers for two of the CPVs, we identified:
  - CPV15: A sub-object overflow could be exploited using input that would not have caused ASAN to fault
  - BCPV 1: An additional (and likely accidental) exploitable spatial-safety vulnerability (CPV 10)
  - BCPV 2: An additional (and likely accidental) one-byte overflow in an AIxCC added feature
- These shed light on the differences between ASAN and CHERI notions of memory safety and bug finding vs vulnerability mitigation, as well as the risks even expert developers experience when working with C/C++

CHERI

CHERI and legacy applications side-by-side

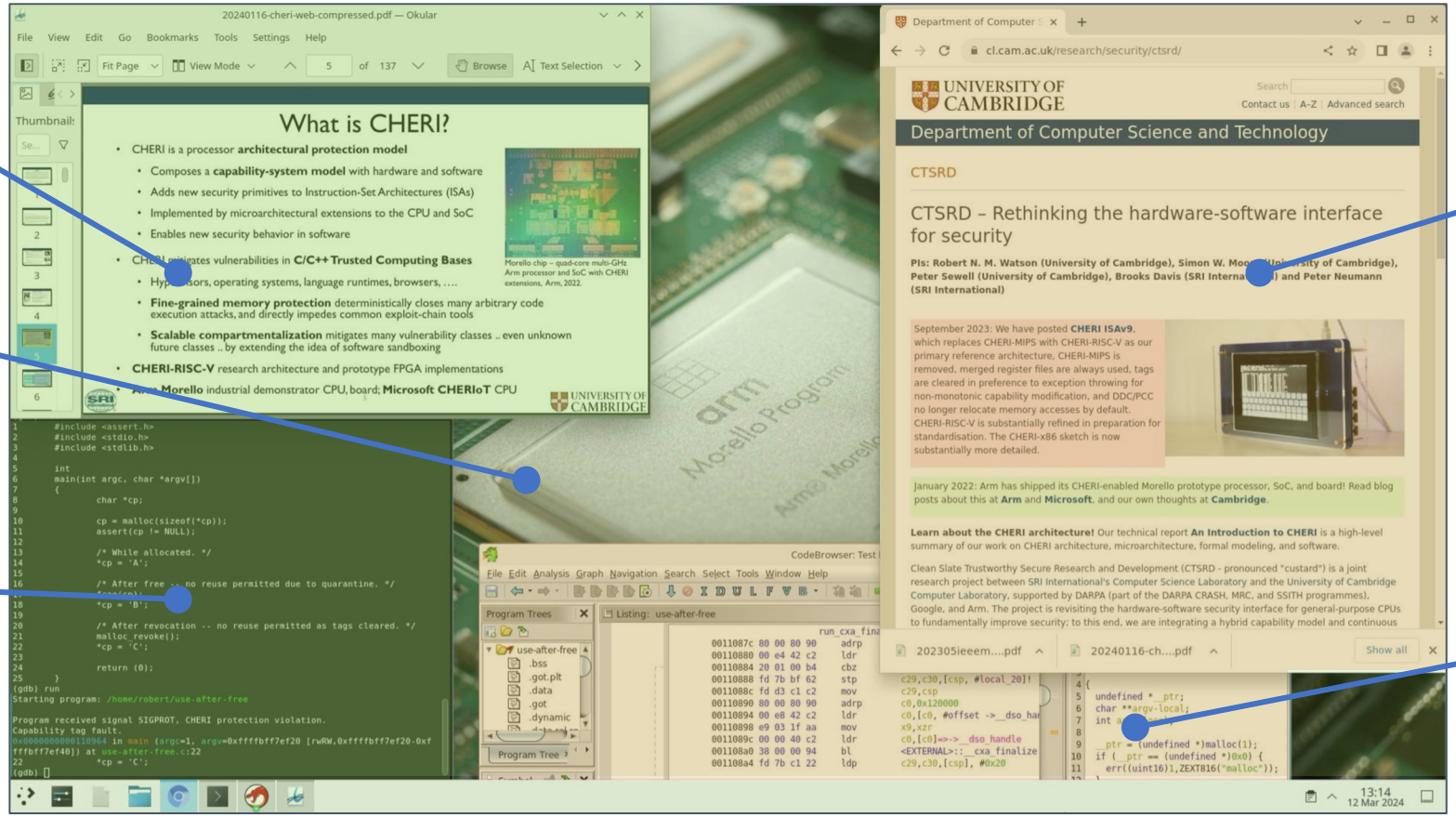
Enables incremental application migration to memory safety

Memory-safe PDF viewer

Memory-safe desktop environment

Memory-safe terminal window and commands

Memory-safe OS kernel



Legacy 64-bit Arm Chromium browser

Legacy 64-bit Arm JVM

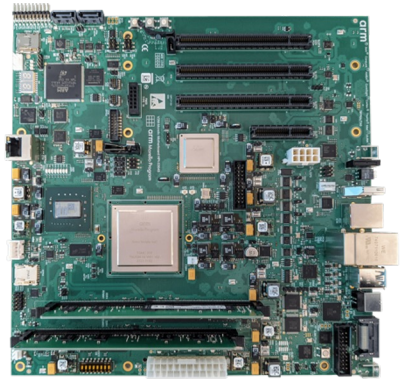
Memory-safe OS kernel and libraries

CHERI

12

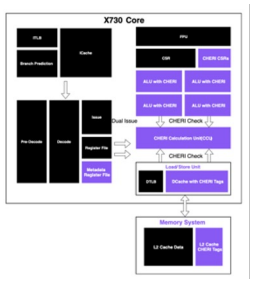
CHERI demonstrated at a range of scales

Mobile devices, data centers



Arm Morello application core + SoC, based on Neoverse N1*
64-bit Arm-A baseline ISA
Multicore, MMU-enabled, out-of-order core 2.5GHz
CHERI-adapted FreeBSD, Linux, seL4, VxWorks OSes

Automotive, embedded, high-end IoT



Codasip X730 application core, based on A730
32/64-bit RISC-V baseline ISA
Dual-issue, pipelined, with MMU
CHERI-adapted FreeBSD, Linux, seL4 OSes

IoT, roots of trust

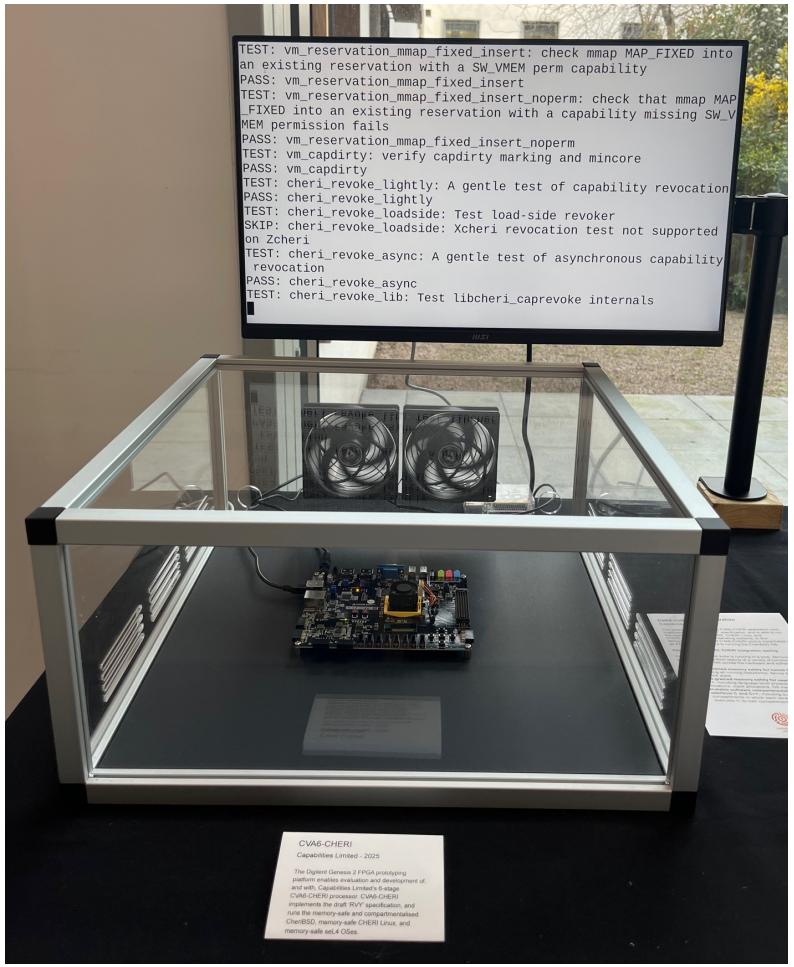
Microsoft CHERIoT Ibex microcontroller
32-bit RISC-V baseline ISA
3-stage pipeline, no MMU, 200-300MHz
CHERIoT RTOS embedded OS

CHERI

* Supported by Innovate UK as part of Digital Security by Design (DSbD)

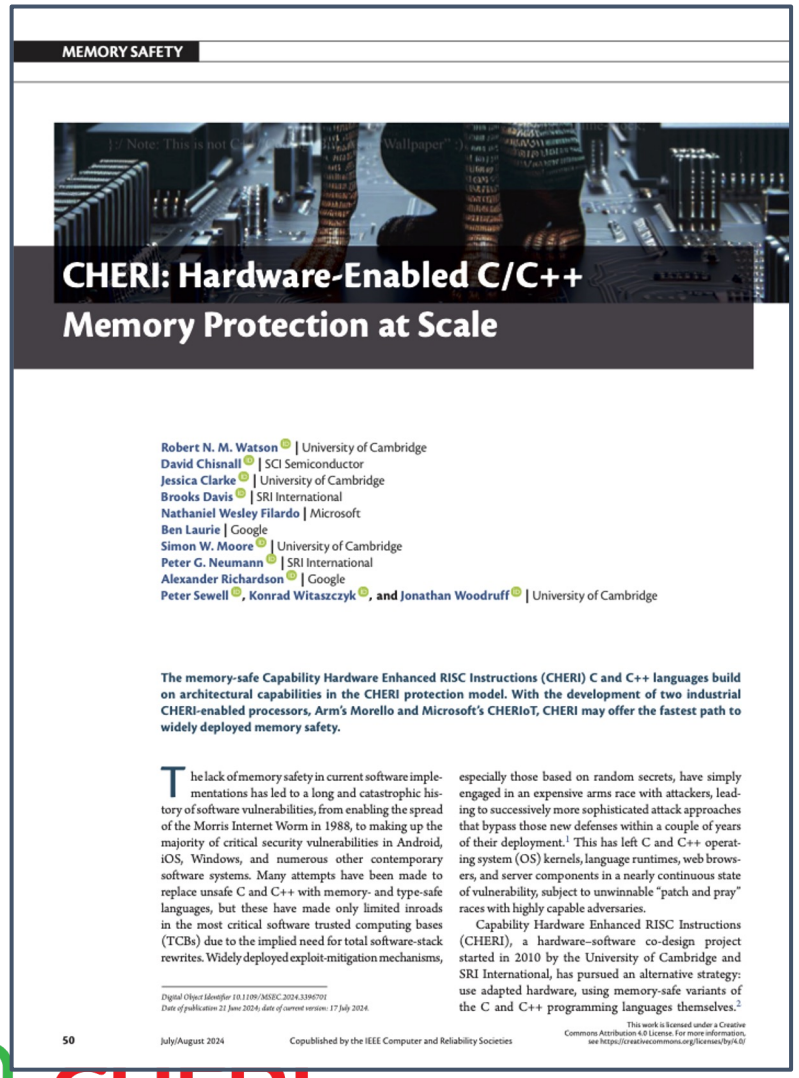
CVA6-CHERI: Open-source RV64Y application core

- Base CVA6 is an open-source application core
  - OpenHW Foundation (Eclipse Group)
  - Under active development (incl. by Thales)
  - Production core written in SystemVerilog
  - 6-stage pipeline, dual-issue, MMU, multicore
- Extending CVA6 with CHERI RV64Y
  - Sufficient DV to allow CheriBSD to boot and run benchmarks on FPGA
  - Initial PPA improvements for CHERI
  - Multi-year R&D and upstreaming plan
- Initial prototyping project funded by the UK government
- New Innovate UK-funded joint COSMIC project with lowRISC and Oxford University to productionize as an open-source Secure Enclave product with formal verification



CHERI

Learning more about CHERI



MEMORY SAFETY

CHERI: Hardware-Enabled C/C++ Memory Protection at Scale

Robert N. M. Watson | University of Cambridge
David Chisnall | SCI Semiconductor
Jessica Clarke | University of Cambridge
Brooks Davis | SRI International
Nathaniel Wesley Filardo | Microsoft
Ben Laurie | Google
Simon W. Moore | University of Cambridge
Peter G. Neumann | SRI International
Alexander Richardson | Google
Peter Sewell, Konrad Witaszczyk, and Jonathan Woodruff | University of Cambridge

The memory-safe Capability Hardware Enhanced RISC Instructions (CHERI) C and C++ languages build on architectural capabilities in the CHERI protection model. With the development of two industrial CHERI-enabled processors, Arm's Morello and Microsoft's CHERIoT, CHERI may offer the fastest path to widely deployed memory safety.

The lack of memory safety in current software implementations has led to a long and catastrophic history of software vulnerabilities, from enabling the spread of the Morris Internet Worm in 1988, to making up the majority of critical security vulnerabilities in Android, iOS, Windows, and numerous other contemporary software systems. Many attempts have been made to replace unsafe C and C++ with memory- and type-safe languages, but these have made only limited inroads in the most critical software trusted computing bases (TCBs) due to the implied need for total software-stack rewrites. Widely deployed exploit-mitigation mechanisms, especially those based on random secrets, have simply engaged in an expensive arms race with attackers, leading to successively more sophisticated attack approaches that bypass those new defenses within a couple of years of their deployment.[1] This has left C and C++ operating system (OS) kernels, language runtimes, web browsers, and server components in a nearly continuous state of vulnerability, subject to unwinnable "patch and pray" races with highly capable adversaries.

Capability Hardware Enhanced RISC Instructions (CHERI), a hardware–software co-design project started in 2010 by the University of Cambridge and SRI International, has pursued an alternative strategy: use adapted hardware, using memory-safe variants of the C and C++ programming languages themselves.[2]

Digital Object Identifier 10.1109/MSEC.2024.3396701
Date of publication 21 June 2024; date of current version: 17 July 2024.

- Visit cheri.cst.cam.ac.uk
- Read our article in the 2024 special issue of IEEE Security and Privacy Magazine:

  CHERI: Hardware-Enabled C/C++ Memory Protection at Scale

- See our technical reports for great detail:

  Introduction to CHERI

  CHERI C/C++ Programming Guide

  CHERI ISA Specification

- And see our research papers on everything from microarchitectural implementations of tagged memory to the implications of memory safety for the UNIX process model

Robert N.M. Watson, David Chisnall, Jessica Clarke, Brooks Davis, Nathaniel Wesley Filardo, Ben Laurie, Simon W. Moore, Peter G. Neumann, Alexander Richardson, Peter Sewell, Konrad Witaszczyk, and Jonathan Woodruff. CHERI: Hardware-Enabled C/C++ Memory Protection at Scale. IEEE Security & Privacy, vol. 22, no. 04, pp. 50-61, July-August 2024.

CHERI

Upstreaming CHERI support

Let's ship FreeBSD 16 with CHERI support!

Upstreaming CHERI support to FreeBSD is supported by Innovate UK and the Department for Science, Innovation and Technology for the adoption and diffusion of CHERI technology under project 10168042 ("CheriBSD feature extraction, maturity, and testing")

CHERI

SRI UNIVERSITY OF CAMBRIDGE

Why upstream?

- FreeBSD needs a memory safety story
  - Virtually all OSes lack memory safety today
  - … but that won't remain acceptable forever (government regulations, etc)
  - Rewriting it all isn't a practical answer (besides, many of us like C)

- Reduce adoption risk in products (vs integrating a fork)
- Provide a production OS with proven features (vs the CheriBSD research OS)
- Changes generally, improves code quality
  - Most changes clarify programmer intent
  - Testing with CHERI finds general bugs, some exploitable
- CheriBSD gives us a lead on CHERI adoption - Let's keep it!

CHERI

What we're upstreaming

- RV64Y (risc64c) – RISC-V CHERI standard (Tier 2)
  - Hardware on the way
- Arm Morello – Prototype armv8.2 + CHERI (Tier 3)
  - Prototype hardware exists, allows direct development, good demos
- Focus on pure-capability CHERI C/C++ programming
  - Most secure and least disruptive to the source tree
  - Sub-object bounds to follow (causes modest disruption, but seems worth it)
- Kernel support leads userspace

CHERI

Timeline of patch production

- September 2025 - <now>: pure-capability kernel support*
  - Goals: spatially safe kernel supporting FreeBSD and CheriBSD userspace
- April 2026 – September 2026: pure-capability userspace support*
  - Spatially safe userspace on FreeBSD/CHERI kernel
- October 2026 – February 2027: linker-based compartmentalization†
  - Enables stronger, finer grained isolation of components
- March 2027 – June 2027: Heap temporal safety†
  - Eliminates use-after-reallocation temporal safety issues

* Critical functionality with low adoption risk

† More disruptive changes with higher adoption risk

CHERI

Progress to date

```
root@freebsd-aarch64:~ # uname -a
FreeBSD freebsd-aarch64 16.0-CURRENT FreeBSD 16.0-CURRENT #28 ca-cheri-n284486-472d6582ec6e-
dirty: Fri Mar 20 02:39:27 GMT 2026 bed22@baume.cl.cam.ac.uk:/home/bed22/cheri/build/freebsd-
aarch64-build/home/bed22/cheri/ca-cheri/arm64.aarch64/sys/GENERIC-MORELLO arm64
root@freebsd-aarch64:~ # sysctl kern.supported_archs
kern.supported_archs: aarch64c aarch64
```

- Booting to multi-user on Morello with unmodified FreeBSD userspace (aarch64)
  - Pure-capability process ABI in progress
  - RISC-V support coming soon
    - Landing final RV64Y ISA support in CheriBSD causing a fair bit of churn at the moment
- ~250 commits in current WIP patch
- 484 files changed, 39761 insertions(+), 3777 deletions(-)
  - Generated portion: 12 files changed, 15752 insertions(+), 14 deletions(-)

CHERI

Tidbit: CHERI and programmer intent

- Capability systems such as CHERI implement two key 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
- New APIs to make it clear when we do or don't propagate capabilities
  - `copy{in,out}` does not. `copy{in,out}ptr` does.
  - `memcpy` does. `memcpy_data` does not.
  - …

CHERI

Tidbit: freebsd64 (legacy 64-bit compatibility)

- Inspired by freebsd32 (legacy 32-bit compatibility)
  - … but different
- Under freebsd32, lot of sizes change
  - Pointers
  - `long` → (`size_t`, `uintptr_t`, `long`)
  - Sometimes `time_t` (x86 only)
  - Special handling for `off_t` (split across registers, maybe aligned)
  - See also: "So you want to add a syscall?"
- Under freebsd64, pointers change size and nature
  - Other types remain the same
- New wrappers needed to pass capabilities to implementations

CHERI

Tidbit: freebsd64 (example wrapper)

```
 int
 sys_exterrctl(struct thread *td, struct exterrctl_args *uap)
+{
+       return (kern_exterrctl(td, uap->op, uap->flags, uap->ptr));
+}
+
+int
+kern_exterrctl(struct thread *td, u_int op, u_int flags, void *ptr)
 {
        uint32_t ver;
        int error;

-       if ((uap->flags & ~(EXTERRCTLF_FORCE)) != 0)
+       if ((flags & ~(EXTERRCTLF_FORCE)) != 0)
                return (EINVAL);
…
```

CHERI

Tidbit: freebsd64 (example wrapper call)

```
int
freebsd64_exterrctl(struct thread *td, struct freebsd64_exterrctl_args *uap)
{
        return (kern_exterrctl(td, uap->op, uap->flags,
            USER_PTR(uap->ptr, sizeof(struct uexterror))));
}
```

- Same as `sys_exterrctl`, except we create a (possibly bounded) capability with the USER_PTR macro

CHERI

Tidbit: networking ioctls (struct ifreq)

- ioctl commands are declared as a group, number, and type (encoded as size):

```
#define     SIOCGIFMAC      _IOWR('i', 38, struct ifreq)
```

- Usually, if the type contains pointers or other things requiring adaptation, the size changes
  - `struct ifreq` contains pointers in a union, but the size does not change between 32-bit and 64-bit (the union in 16-bytes)!
    - Special handling still required so we use process ABI (`SV_ILP32`)
  - On CHERI, the size does change!
    - There are lots of ioctls using `struct ifreq`
    - Helper macro: `CASE_IOC_IFREQ`
      - "`case SIOCGIFMAC:`" becomes "`case CASE_IOC_IFREQ(SIOCGIFMAC):`"
    - Accessors like `ifr_data_get_ptr()`

CHERI

git strategy

- Fork of the freebsd-src repo under the CHERI Alliance github
  - https://github.com/cheri-alliance/freebsd-src
- Maintain **rebased** branch(es) of upstreaming candidates
  - `main` tracks upstream FreeBSD
  - `main-ci` adds CI from The Capable Hub
  - `cheri` contains patches adding CHERI
    - *Warning: patch readiness varies!*
- Patches or patch groups to be submitted via Phabricator or GitHub pull requests as appropriate
- Some patches arriving directly
  - https://github.com/freebsd/freebsd-src/pull/2068

CHERI

Please get involved!

- We aim to make FreeBSD/CHERI the reference POSIX-like OS for CHERI
  - We're broadly confident in our approach, but more eyes will help make it better
- Most reviews in phabricator (reviews.freebsd.org) tagged with `#cheri`
  - Commits tagged with "Effort: CHERI upstreaming"
- Occasional pull requests for example https://github.com/freebsd/freebsd-src/pull/2068

- Run a different BSD? Consider adding CHERI support!

CHERI

Q&A