Bringing memory safety to BSD with CHERI

Brooks Davis
Capabilities Limited
brooks@capabilitieslimited.co.uk

Abstract – After a decade and a half of academic and industrial research, products using CHERI to provide hardware enforced, deterministic memory safety are making their way to market. CheriBSD, a research fork of FreeBSD, has been the primary vehicle for operating system research on supporting CHERI's memory safety and compartmentalization potential. We are now bringing the most complete and stable work from CheriBSD to FreeBSD with the goal of shipping FreeBSD 16 with spatial and heap temporal memory safety. This talk will motivate this upstreaming, explain our plan, and cover progress to date. We hope to update the FreeBSD community on our progress and encourage other BSDs to learn from our adoption of CHERI.

I. INTRODUCTION

Like all widely used operating systems, FreeBSD's kernel and userspace are written in assembly, C, and C++ all of which are memory unsafe. Even when writing software in a memory-safe language, system calls, foreign function interfaces (FFIs), and often the language runtime itself mean these interfaces pose a risk. Decades of work to improve C and C++ codebases have raised the bar, but ultimately made little difference in the rate of discovery and exploitation of memory safety vulnerabilities leading to arbitrary code execution.

CHERI is an architectural security extension that together with the compiler and runtime support brings hardware enforced spatial and temporal memory safety to C and C++ codebases with modest levels of modification. Further, CHERI enables compartmentalization at extremely fine granularity, permitting orders of magnitude more compartments per-application than with current virtualization based techniques.

SRI and the University of Cambridge have developed CHERI over the past 15 years with increasing industrial involvement ranging from Arm's Morello prototype as part of the Digital Security by Design (DSbD) program in the UK, Microsoft's open-source CHERIoT microcontroller and RTOS, and the standardization of CHERI extensions to RISC-V as the RV32Y and RV64Y base instruction sets lead by Google, The University of Cambridge, and Codasip. Multiple implementations of RV64Y are approaching readiness for tapeout and we expect to see design wins in the near future.

The time is now right to bring CHERI support to FreeBSD and Capabilities Limited has funding from Innovate UK to extract patches from our existing CheriBSD research OS and integrate them into FreeBSD. We aim to add support for spatial safety in the kernel and userspace, linker-based compartmentalization in userspace, and heap temporal safety in userspace in time for them to be included in the FreeBSD 16.0 release.

We are working to upstream support for both RV64Y and Morello for timing related reasons. While Morello is a prototype with a limited run of hardware, it's here today and able to run a large and complex software stack. By contrast, RV64Y will be standardized shortly and several tapeouts are likely in the near future. The first application cores will be at the lower end and may not end up in designs where running large software such as the Clang compiler or the Chromium web browser makes sense and it's extremely useful to have a normal software development environment available when porting other software.

II. What is CHERI?

CHERI is an architectural security extension that brings memory safety and lightweight compartmentalization to C and C++ trusted computing bases while enhancing the underpinnings of memory safe languages including Java, JavaScript, Rust, and Python. CHERI extends instruction sets (RISC-V and Arm today) adding a new hardware type, the CHERI capability which extends integer pointers with metadata including bounds, permissions, and an out-of-band validity tag. In CHERI systems, all memory accesses are performed via capabilities, either explicitly via new capability aware instructions or implicitly via a default data capability which allows existing software to run unmodified. CHERI capabilities enforce province and monotonicity in that any capability must be derived from another valid capability and any operation on a capability must not increase its bounds or

permissions. All capabilities in a CHERI system are derived from one or more root capabilities made available to the operating system at boot time (and the boot loader at reset time).

We have implemented variants of C and C++ which use CHERI capabilities in place of all pointers both explicitly and implicitly (return addresses, GOTs, etc.). To implement CHERI C/C++ the compiler, operating system, and runtime cooperate to place bounds (and occasionally permissions) on all memory objects.

On architectures FreeBSD cares about, CHERI capabilities expand 64-bit integer pointers into 129-bit capabilities. 128-bits of that are directly accessible in memory and the tag bit is preserved by capability aware load and store instruction as well as instructions that perform valid manipulations in registers. Astute readers might have noticed that 64-bits of metadata isn't enough to place precise bounds on an object in a 64-bit address space. Bounds are in fact compressed relative to the address in a manner similar to floating point where bounds get coarser as allocations get larger. In order to support real world software, the address is allowed to stray somewhat out of bounds, but must return before the capability is used to access memory and must not stay so far that it would point to something else. This combines to pose some representability requirements on pointers. First, memory (and virtual address space) allocations must be padded such that they do not overlap with other allocations. This aligns with typical memory allocator patterns and is not extremely disruptive for `malloc` and the like. Second, addresses never be taken too far out of bounds. This is undefined behavior in C so good practice anyway, but some patterns take an address far out of bounds and then bring it back in and need to be addressed.

Well written C and C++ software usually requires few or no modifications to support CHERI, but lower level code tends to require more changes than higher level code. Operating systems, language runtimes, and web browsers require significant modifications while modern C++ programs or well written C like OpenSSH often require none.

For a more detailed introduction to CHERI, see our paper *CHERI: Hardware-enabled C/C++ memory protection at scale.*

For more information on programming with CHERI C and C++, see the *CHERI C/C++ Programming Guide* (https://ctsrd-cheri.github.io/cheri-c-programming).

III. What is CheriBSD?

CheriBSD is a research fork of FreeBSD developed since around 2012 to support CHERI. We initially added support for CHERI as an extension to the MIPS64 ISA with later work adding support for RISC-V and Arm's Morello. CheriBSD has evolved with CHERI and has played an integral role in CHERI's development. For example the first versions of CheriBSD worked with assembler support, but no compiler support for CHERI capabilities. Later versions had quite primitive support for integer manipulations of pointers when those pointers were CHERI capabilities. This continues to have minor impacts on code structure as we sometimes choose the quickest conversion from inline assembly macros to direct C support or compiler builtins.

To enable the exploration of different ideas, CheriBSD supports a variety of features that are unlikely to make it into production ISAs. For example Arm's Morello prototype implemented several different domain transition mechanisms which can support compartmentalization because they were all microarchitecturally feasible, but it was unclear which ones were most appropriate from a software perspective. By contrast, RISC-V has endeavored to pick the most minimal set of functionality to keep implementation size small and reserve bits in the capability format for later use.

CheriBSD also implements support of less mature ideas such as co-process compartmentalization which uses CHERI protections to allow multiple processes to share a single address space. This allows rapid domain transitions and low overhead sharing of resources while still maintaining much of the familiar process model and its affordances around privileged operations such as filesystem and device access. The general idea is sound, but we're not yet clear that our prototypes of domain transition are ready to upstream and our ideas around resource sharing would benefit from some refinement.

All of this means that while CheriBSD has a number of features such as spatial memory safety that are highly robust, others aren't suitable for industrial deployment. We are working to extract the most production-ready features for upstreaming while maturing ones we're convinced are broadly good ideas, but need some refinement before upstreaming.

IV. FreeBSD and CHERI

Applying CHERI to FreeBSD requires changes to a number of areas. In some cases that is necessary for things to work at all (e.g., pointer provenance and monotonicity must be preserved). In others, we have made changes to avoid propagating capabilities by accident or to prevent errors further downstream from becoming exploitable. This section covers some of these changes and the impacts they have on programmers.

A. CHERI and intentionality

CHERI systems implement two principles:

- The *principle of least privilege* which is the idea that only those privileges required for an operation should be granted to the entity performing it. This typically manifests as narrowing of bounds and reducing emissions on capabilities.
- The *principle of intentional use* (commonly referred to as *intentionality*) which is the idea that where multiple sources of privilege are available, the source which most precisely grants the required permissions should be used. By making their intent clear, the programmer allows the hardware (and lower levels of software) to follow the programmer's intent.

In traditional C with integer pointers it's possible to get away with imprecise operations where things like pointer provenance is unclear. These cases must be adjusted in CHERI C programs to ensure the programmer's intent is clear and pointer provenance can be preserved. For example, consider the expression `a = b + c;` where all variables are of type `uintptr_t`. It's unclear if which of `b` or `c` should provide the provenance for `a`. By default, the compiler assumes the left most (`b`) provides the provenance, but warns so the code can be cleared. If this is accurate, then casting `c` to `ptraddr_t` ensures that provenance comes from `b`. In other cases, `uintptr_t` wasn't the right type and other types should be adjusted. See the CHERI C/C++ Programmers Guide for more examples and explanations.

In other cases, we need to change APIs to more accurately express intent. This often becomes a tradeoff between source compatibility and intentionality or least privilege. For example, `memcpy` must preserve pointer provenance because it is used (including implicitly by the compiler) in cases where it is unknown if provenance should be preserved, but we often do know that we do not want to preserve provenance because a buffer only contains data. Similarly, the vast majority of locations (>90% in CheriBSD) copying to or from the kernel handle data that does not contain pointers and thus we should avoid preserving provenance when we don't want to. To support this we have introduced new APIs to express these cases. Some of these APIs are shown in the table below.

Original	**preserving**	**non-preserving**
`memcpy`	`memcpy`	`memcpy_data`
`memmove`	`memmove`	`memmove_data`
`bcopy`	`bcopy`	`bcopy_data`
`copyin`	`copyinptr`	`copyin`
`copyout`	`copyoutptr`	`copyout`
`pagecopy`	`pagecopy`	`pagecopy_cleartags`
`pmap_copy_pages`	`pmap_copy_page_tags`	`pmap_copy_page`

In an ideal world we'd probably pair `copyinptr` with a `copyindata` and phase out `copyin` (and similar for `copyout`), but that seems too disruptive for now. We've typically favored changing the side that is least disruptive rather than forcing all locations to change.

Another place we control the flow of pointers is in copies via uio(9). We have extended `enum uio_rw` with two more values: `UIO_READ_PTR` and `UIO_WRITE_PTR`. The traditional `UIO_READ` and `UIO_WRITE` values cause data to be copied without preserving provenance (e.g., the expected result for storage). The new values preserve pointer provenance in cases where it is required such as the CAM target code.

B. CheriABI -- A pure-capability process environment

CheriABI is a process environment where all pointers are CHERI capabilities and the default data capability is NULL so all access to memory must be via explicit capabilities derived from bounded capabilities proved by the kernel. The compiler, runtime, and kernel strive to support a least-privilege model; each section virtual address space is allocated by the kernel at startup or returned in response to a `mmap`, `shmat`, or (rarely) `ioctl` call. Pointer arguments to system calls are likewise capabilities and the kernel preserves these capabilities throughout the stack to the hardware in nearly all cases.[1]

CheriABI necessitated some changes to the virtual memory system and the APIs to manage the process virtual address space. After the process was created, there were historically two not entirely compatible ways to manage the virtual address space. Originally, there was `sbrk`[2]. It adjusted the amount of virtual address above the program binary that is accessible, growing toward the stack. Its association with the top of the program binary (used by some software such as older versions of emacs) as well as the way it grows over time is contrary to a least-privilege model. It is also not thread safe. As a result we decided not to support it.

The `mmap` model, as used in most, but not all cases is reasonably compatible with a least-privilege model. Typically an area of address space is allocated and filled with default contents (a file, demand filled zeroed pages, etc.) and then updated with different permissions and or contents as needed (for example, space is reserved for a dynamic library and then backed by mapping parts of the file, zeroed pages, or guard pages with appropriate permission for different areas (read-only, read-write, read-execute, none). The `mmap` API has the downside of conflating address space reservation and backing of the reservation. We imposed some additional structure on that model by making the initial mapping create a `reservation` in the kernel which corresponds to the returned capability (including adding padding as required for representability) and remains in place, in its entirety, until all pages have been unmapped. Only then is the address space returned to the system. Further manipulation must be performed a capability to the reservation and operations may not span reservations. There is no mechanism to glue two reservations together so programs that need large contiguous areas must allocate all the address space they might need up front or relocate later. For more information about the reservation model, see the talk *Address space reservations: Re-thinking address space management for pointer provenance*.

For more information on CheriABI, see our paper *CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment*.

C. Legacy ABI compatibility

Porting software to CHERI C/C++ is often straightforward, but large complex software can be time consuming to port (for example, the Chromium web browser took about 18 person-months to port to the point of basic functionality[3]) As a result it has been and continues to be imperative that we support legacy ABIs with integer pointers. Fortunately, FreeBSD makes this relatively easy with our strong compatibility framework. In CheriBSD we have added a `freebsd64` layer beside `freebsd32`. While we derived `freebsd64` from `freebsd32`, the problems they have to solve are in fact quite different.

In `freebsd32` on a 64-bit system, system calls need compatibility shims if they take `signed long` arguments (requiring sign extension), 64-bit integer arguments (split between two registers and possibly aligned) or pointers to objects containing pointers, `long` values, or `time_t` (which change size). In many ways `freebsd64` is simpler because 64-bit arguments and long arguments require no handling. Similarly, `long` and `time_t` values in objects don't impact size and thus don't require special handling. CHERI systems

[1]The main exception revolves around optimizations using `vm_fault_quick_hold_pages` to allow direct access to memory. In these cases, capability bounds and permissions are checked to ensure that the access is appropriate before the page is mapped.

[2] Spelled `break` when Unix was written in the B language

[3] While this might seem like a lot of work, the Chromium security team has on the order of 100 full time staff so relatively speaking it's small for porting over 40 million lines of code.

do add a new requirement that any system call that takes pointer arguments requires a shim to create capabilities from the integers arguments passed by userspace. Likewise, it's not sufficient to simply cast a `uint64_t` representing a pointer in a structure to `void *` to turn it into a capability.

This required both a mechanism to create such capabilities and a place to pass them. To that end we have added over 100 `kern_<sys>` and `user_<sys>` functions to interpose between the system call and the implementation. We've tended to error on the side of imposition because we've implemented a lot of ABI compatibility layers (CheriABI was originally a compatibility layer and we've since added freebsd64) and found code duplication and `sys_<sys>` between `freebsd*_<sys>` annoying.

To support creating arguments to call these wrappers, we have macros that take either a provenance-free pointer type as appears in syscall argument structures or an integer address and turn them into a capability relative to the default data capability of the current thread. Most of these macros take a length or count to allow bounds to be placed on the resulting capability. They include:

Macro	**Cap address**	**Cap length**	**Applies to**
`USER_PTR(ptr, len)`	`ptr`	`len`	ptr, int
`USER_PTR_OBJ(objp)`	`objp`	`sizeof(*objp)`	ptr
`USER_PTR_ARRAY(objp, cnt)`	`objp`	`sizeof(*objp) * cnt`	ptr
`USER_PTR_PATH(path)`	`path`	`MAXPATHLEN`	ptr, int
`USER_PTR_STR(strp)`	`strp`	DDC bounds	ptr, int
`USER_PTR_ADDR(ptr)`	`ptr`	DDC bounds	ptr, int
`USER_PTR_UNBOUND(ptr)`	`ptr`	DDC bounds	ptr, int

Generally speaking, the most specific macro should be used. Because legacy ABIs are unaware of capabilities, bounds may not be precise and may alias with other allocations. Bounds are provided on a best effort basis and provide some protection against kernel bugs, but often trust input from user space like any traditional integer pointer system.

Another type related issue is that some code achieved compatibility between 32-bit and 64-bit architectures by using `uint64_t` or similar in place of pointer types. This is common in Linux APIs and also used in OpenZFS. These integer types can't hold CHERI capabilities so we need to expand them and now need to provide compatibility code. We have added new types (`int64ptr_t` and `uint64_ptr_t`) to support these transitions. These types are 64-bit integers unless they need to be larger to hold a pointer in the current ABI.

V. The upstreaming process thus far

Our project timeline has us producing patches to FreeBSD for CHERI functionally on the following timeframe:

- September 2025 – March 2026: pure-capability kernel support (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 for heap allocations)

As of the end of January 2026, we have a kernel that compiles for CHERI, but is untested and known to be missing some functionality. We're in the process of filling in holes and breaking up the largest chunk of the patch (virtual memory system changes and process startup) into more bite sized pieces.

We have begun actively upstreaming in a few areas:

- New types in support of intentionality, ABI compatibility, and correctness
- Migrating to using compiler builtins to adjust the alignment of pointers and sizes
- Cleanups of hardcoded or otherwise questionable sizes in a variety of bits of code that would also need definitions for CHERI

We will soon start landing changes to virtual memory APIs as well as compatibility shims.

Changes related to this project (both funded by Innovate UK and by DARPA/AFRL) are tagged with the commit trailer `Effort: CHERI upstreaming`.

VI. Conclusions

We're working to bring memory safety to FreeBSD's legacy C/C++ codebase. This work will improve FreeBSD's overall code quality while making FreeBSD the reference implementation of CHERI support on a POSIX OS (a crown currently held by CheriBSD). Beyond improving FreeBSD we hope to provide a template for other BSDs and POSIX-like operating systems to implement CHERI support.

VII. Acknowledgements

The current work on 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”).

Distribution Statement A: Approved for public release; distribution is unlimited. Sponsored in part by DARPA and the Air Force Research Laboratory, under Contract FA8750-10-C-0237 (“CTSRD”); with additional support from Contract FA8750-11-C-0249 (“MRC2”), Contract HR0011-18-C-0016 (“ECATS”); Contract FA8650-18-C-7809 (“CIFV”); Contract HR0011-22-C-0110 (“ETC”), Contract HR0011-23-C-0031 (“MTSS”); and Contract FA8750-24-C-B047 (“DEC”) as part of the DARPA CRASH, MRC, SSITH, and CPM 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, either expressed or implied, of the U.S. Department of Defense or the U.S. government.

This work was supported in part by the Innovate U.K. DSbD Technology Platform Prototype Project 105694 as well as Innovate U.K. Project 107145 and Project 10027440. This project received funding from the European Research Council under the European Union’s Horizon 2020 research and innovation program (Grant 789108). We further acknowledge the EPSRC REMS Program (Grant EP/K008528/1) and the EPSRC CHaOS (Grant EP/V000292/1). Additional support was received from Arm, Google, and Microsoft.

VIII. References

T. Aird, H. Almatary, A. A. Garcia, J. Baldwin, P. Buxton, D. Chisnall, J. Clarke, B. Davis, N. W. Filardo, F. A. Fuchs, T. Hutt, A. Joannou, M. Kaiser, T. Kurd, B. Laurie, M. van der Maas, M. Malenko, A. T. Markettos, D. McKay, J. Melling, S. Menefy, S. W. Moore, P. G. Neumann, R. Norton, A. Richardson, M. Roe, P. Rugg, P. Sewell, C. Shaw, R. Tura, R. N. M. Watson, T. Wenman, J. Woodruff, and J. Z. Yu. *RISC-V Specification for CHERI Extensions (v0.9.6.1)*. Dec. 2025. https://github.com/riscv/riscv-cheri/releases/tag/v0.9.6.1

R. N. M. Watson, D. Chisnall, J. Clarke, B. Davis, N. W. Filardo, B. Laurie, S. W. Moore, P. G. Neumann, A. Richardson, P. Sewell, K. Witaszczyk, and J. Woodruff. *CHERI: Hardware-enabled C/C++ memory protection at scale.* IEEE Security & Privacy, 22(4):50–61, 2024. https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/20240419-ieeesp-cheri-memory-safety.pdf

B. Davis. *Address space reservations: Re-thinking address space management for pointer provenance*. BSDCan 2024, Ottawa, Canada. 2024.
Talk: https://www.youtube.com/watch?v=1VVVgOIZMp8
Slides:
https://indico.bsdcan.org/event/1/contributions/14/attachments/5/5/address-space-reservations.pdf)

B. Davis, R. N. M. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka, A. Joannou, B. Laurie, A. T. Markettos, J. E. Maste, A. Mazzinghi, E. T. Napierala, R. M. Norton, M. Roe, P. Sewell, S. Son, and J. Woodruff. *CheriABI: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment.* In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, page 379–393, New York, NY, USA, 2019. Association for Computing Machinery. https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201904-asplos-cheriabi.pdf