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@microsoft.com

Jessica Clarke†
jessica.clarke@cl.cam.ac.uk

Nathaniel Wesley Filardo†
nwf20@cam.ac.uk

Khilan Gudka†
khilan.gudka@cl.cam.ac.uk

Alexandre Joannou†
alexandre.joannou@cl.cam.ac.uk

Ben Laurie¶
benl@google.com

A. Theodore Markettos†
theo.markettos@cl.cam.ac.uk

J. Edward Maste†
emaste@freebsd.org

Alfredo Mazzinghi†
am2419@cam.ac.uk

Edward Tomasz Napierala†
trasz@freebsd.org

Robert M. Norton†
robert.norton@cl.cam.ac.uk

Michael Roe†
michael.roe@cl.cam.ac.uk

Peter Sewell†
peter.sewell@cl.cam.ac.uk

Stacey Son†
sson@me.com

Jonathan Woodruff†
jonwoodruff@gmail.com

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

Abstract

The CHERI architecture allows pointers to be implemented as capabilities (rather than integer virtual addresses) in a manner that is compatible with, and strengthens, the semantics of the C language. In addition to the spatial protections offered by conventional fat pointers, CHERI capabilities offer strong integrity, enforced provenance validity, and access monotonicity. The stronger guarantees of these architectural capabilities must be reconciled with the real-world behavior of operating systems, run-time environments, and applications. When the process model, user-kernel interactions, dynamic linking, and memory management are all considered, we observe that simple derivation of architectural capabilities is insufficient to describe appropriate access to memory. We bridge this conceptual gap with a notional *abstract capability* that describes the accesses that should be allowed at a given point in execution, whether in the kernel or userspace. To investigate this notion at scale, we describe the first adaptation of a full C-language operating system (FreeBSD) with an enterprise database (PostgreSQL) for complete spatial and referential memory safety. We show that awareness of abstract capabilities, coupled with CHERI architectural capabilities, can provide more complete protection, strong compatibility, and acceptable performance overhead compared with the pre-CHERI baseline and software-only approaches. Our observations also have potentially significant implications for other mitigation techniques.

ASPLOS'19, April 13–17, 2019, Providence, RI, USA

ACM ISBN ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304042

CCS Concepts • **Security and privacy → Operating systems security**; *Security in hardware*; • **Software and its engineering** → Maintaining software.

Keywords security, operating systems, hardware, CHERI

ACM Reference Format:
Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. In *Proceedings of 2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS'19).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3297858.3304042

1 Introduction

Conventional architectures and C programming language implementations provide only coarse-grained protection against memory errors. At run time, they represent memory addresses simply as integers, and constrain how they can be used with page-based memory-management units (MMUs). This coarseness is the enabling root cause of large classes of software vulnerabilities, allowing simple bugs (which conventional engineering techniques cannot reliably exclude) to be escalated to loss of data integrity and confidentiality and to arbitrary code execution [41].

The MMUs found in contemporary processors are the result of a long co-evolution with the Multics (and then UNIX) process models [13, 35], to provide process-granularity fault isolation. Programs reside in *virtual memory*, giving separate scopes to the pointers (integers) used by each process. MMUs conflate protection and translation: the granularity of both is one virtual page. When OS kernels act on userspace, e.g., via pointers passed in to system calls, they must act on the correct set of physical pages corresponding to the process.

However, the table-driven approaches of modern MMUs do not scale easily to handle finer-grain protection, e.g., for language-defined objects – often much smaller than a page or not sized in whole pages. Besides, the MMU still does not distinguish virtual addresses from arbitrary integers: while the MMU protects the *structure* of the virtual memory space, the *references* to virtual memory (i.e., integers implementing pointers) are unprotected.

In thinking how we can improve this situation, we are guided by two underlying principles. The first is the *principle of least privilege*, a classic in computer security: greater security can be obtained by minimizing the privileges accessible to running software [37]. The second, newly identified during our work, is the *principle of intentional use*: where a set of privileges is available to a piece of software, an invoked privilege should be selected explicitly rather than implicitly, e.g., by selecting a specific capability with just the required privileges, rather than via an arbitrary table search. The principle of intentional use covers not only minimizing privileges overtly used, but also not discarding valuable information during program compilation. For example, out-of-bounds speculative reads could be avoided if the length of a buffer were available to the hardware.

The application of these principles limits the scope of attacker behavior when exploiting a bug, and, in the context of C-language memory protection, limits the effectiveness of attackers in injecting, manipulating, or abusing pointers in the run-time environment – whether explicit (i.e., declared code or data pointers) or implied (e.g., as used in generated code to implement global variables or return addresses, or by the runtime to implement cross-library control flow). We envisage a conceptual model in which:

- memory accesses are not merely via arbitrary integers (checked against only the process address space), but also require an *abstract capability*, conferring an appropriate set of memory access permissions;
- these abstract capabilities are constructed only by legitimate *provenance chains*[1] of operations, successively reducing permissions from initial maximally permissive capabilities provided at machine reset; and
- code is not given access to excessive capabilities.

Importantly, we aim to provide this across *whole-system* executions, not just within the C-language portion of user processes. This means that we have to capture the many ways that pointer values are constructed and manipulated, including process creation, virtual memory including swapping, system calls, dynamic linking, context switching, signal delivery, debugging, and a host of C-language operations.

Considerable C-language memory-safety research (considered further in Section 7) has explored various software- and hardware-based mitigation techniques, both static and dynamic, that protect the integrity of pointers, constrain control flow, and protect the code and data referenced by pointers [3, 8–12, 17, 22, 23, 25, 26, 28, 32, 33, 38, 42, 48, 50]. This has shown that while enforcing pointer-based protection can be helpful in mitigating bugs, it is also potentially disruptive. Prior work has suffered from a range of practical limitations, including: requiring large changes to existing codebases, using a unique OS or library infrastructure instead of a full POSIX environment, being limited to statically linked code, lacking coverage in run-time libraries or kernels, and incurring high performance costs.

The basic question here is whether it is practical to support a large-scale C-language software stack with strong pointer-based protection (along the lines of the conceptual model above), with only modest changes to existing C codebases, and with reasonable performance cost. *We answer this question affirmatively.* We have adapted a complete C, C++,

[1] By provenance, we mean a series of correct operations like those we describe in [31], not to the attribution of errors as in Bond [5].

and assembly-language software stack, including the open-source FreeBSD OS [30] (nearly 800 UNIX programs and more than 200 libraries including OpenSSH, OpenSSL, and bsnmpd) and PostgreSQL database, to employ ubiquitous capability-based pointer and virtual-address protection.

Our approach implements abstract capabilities using Capability Hardware Enhanced RISC Instructions (CHERI) [48] *architectural capabilities*: these are hardware-implemented run-time capabilities that can be reduced but not forged by software. CHERI capabilities provide *spatial integrity* (a capability cannot be used to access memory outside its intended bounds and privileges) and *referential integrity* (capabilities may be neither forged nor corrupted to alter their bounds).

Two key challenges arise in implementing abstract capabilities. First, CHERI architectural capabilities are expressed in terms of virtual addresses, and have no meaning except in conjunction with a specific virtual-to-physical mapping. Given such a mapping, each capability allows direct access to a specific subset of physical memory. However, these mappings change over time (e.g., when the OS creates a new user process, maps additional memory, alters the backing of a mapping, or context switches to other address spaces). Second, in a real system, pointer values are created in a wide variety of ways, some of which require special intervention to preserve the provenance chain of abstract capabilities, even though the architectural capability chain is broken – for example, when memory is paged out, or during process debugging.

In this paper, we:

- Review the semantics of CHERI capabilities.
- Introduce the concept of an *abstract capability* that grants access to some system resource(s), and discuss its construction and application.
- Describe the adaptation of over 99% of the C-language userspace of a UNIX operating system and enterprise database to employ fine-grained CHERI memory protection throughout userspace, while requiring only minimal source-code modification.
- Extend prior userspace pointer-protection work by enforcing language-defined memory models in the OS kernel through construction and maintenance of abstract capabilities, preventing confused-deputy attacks via the kernel. We consider numerous "edge cases" in OS design often ignored in earlier work in memory protection, such as process startup, dynamic linking, thread-local storage (TLS), signal delivery, management APIs such as (`ioctl`), and debugging, all of which are essential to abstract capabilities.
- Utilize trace-based execution analysis to reconstruct the abstract capabilities of processes, and quantify the increased granularity of architectural capabilities.
- Analyze the impact of pointer integrity, provenance, monotonicity, and spatial protection across UNIX.
- Extend the CHERI ISA to enable generation of more efficient code for dynamically linked programs, and improved compatibility with existing C code.
- Validate, on an FPGA-based platform, that the performance overhead of architectural memory protection is acceptable for mainstream software.
- Discuss changes to the CHERI C compiler based on our experience in compiling and running large amounts of software.

To our knowledge, this is the first complete UNIX system offering ubiquitous spatial and referential memory safety, granting us unique insights into the practicality of C-language protection at scale. Supporting a substantial software stack that includes the entire UNIX OS has forced us to explore many of the dark corners of C programming. However, once the run-time environment has been updated, the vast majority of code can simply be recompiled.

2 CHERI Background

The CHERI architecture adds a new hardware data type suitable to implement strongly protected C-language pointers, namely, the *CHERI capability*. CHERI capabilities extend integer virtual addresses to control access to virtual memory by adjoining **bounds** constraining the range of addresses and **permissions** limiting the use of each capability (e.g., load, store, or instruction fetch). We previously described the initial CHERI architecture [45, 50], demonstrated an efficient compartmentalization framework above it [46, 48], described a C-language compilation mode where all pointers are capabilities [9], and demonstrated the use of those abilities to implement a safer JNI [8]. The architecture enforces the following properties:

Provenance validation ensures that only capabilities derived via valid transformations of valid capabilities using capability instructions can be used.

Capability integrity prevents direct in-memory manipulation of architectural capability *encodings*.

Monotonicity prevents the permissions or bounds associated with a capability from being increased.

These properties collectively imply unforgeability of capabilities. All accesses to virtual memory are via capabilities. Instructions are fetched via the program-counter capability (PCC). CHERI implementations add instructions to support explicit capability-relative load, store, and jump, as well as to manipulate capabilities. Legacy instructions accessing memory via virtual addresses are indirected through a default data capability (DDC) register.

On processor reset, initial global capabilities are made available via registers. Capabilities may be stored from capability registers to memory and loaded back with dedicated instructions. However, IO devices have not been extended to support capabilities; separate action must be taken to preserve the validity of capabilities, for example, if they are

swapped to disk and back. Capabilities in registers may be transformed with further instructions – including duplication (just as one can copy an integer address), arithmetic address manipulation (such as incrementing a pointer through an array), permission reduction (e.g., constructing a read-only capability from a read-write one), and bounds reduction (yielding a capability authorizing access to a smaller span of virtual addresses). Software can employ these features to restrict use of derived pointers – e.g., by narrowing the bounds on a pointer to match an allocation, or to prevent writing via a function pointer. In our prior work [8, 9, 48, 50], reductions in privilege were done via language-level objects or explicit capability instructions, rather than the OS or C-language runtime. This left capabilities to the full address available outside sandboxes, and all kernel interactions were by unbounded and forgeable integer virtual addresses.

In implementation, CHERI extends 64-bit addresses with metadata [47] in both the in-register and in-memory representations, increasing the in-memory size of pointers to 128 bits, plus an out-of-band tag bit.[2] There is one tag bit per capability-sized and capability-aligned region of *physical* memory, to distinguish between data (i.e., integers) and capabilities. This tag bit follows memory contents through the cache hierarchy and into (capability) registers and indicates valid *provenance* of the capability therein. Violations of the architectural capability semantics, including overwriting their representation with (integer) data, will clear the tag, preventing subsequent interpretation as a capability. Tags may not be explicitly set by software; all capabilities are transitively derived from the initial capabilities provided at reset; that is, valid provenance is enforced.

We implemented the CHERI protection model as an extension to the 64-bit MIPS ISA, including ISA-level emulation (Qemu) and hardware (FGPA) implementations. This allows for both fast software simulation and more detailed microarchitectural studies.

The CHERI C compiler supports two modes of operation for pointers: a *hybrid mode* in which only pointers annotated with `__capability` qualifiers become capabilities (unannotated pointers remain integers and are checked w.r.t. the DDC), and a *pure-capability mode* in which all explicit (and implied) pointers are capabilities. Our prior work used the pure-capability mode within sandboxes [8, 48]. By contrast to the present work, these sandboxes utilized only static linking, and provided no (or a limited) system-call layer – limiting adaptability of code and evaluation at scale. Non-sandboxed code retained permissions, via DDC, to the entire user-program address space.

The CheriBSD operating system is an adaptation of the FreeBSD operating system, with added support for CHERI capabilities. In prior work, we implemented only the basic CheriBSD kernel and runtime infrastructure necessary to run a (hybrid-mode) userspace program manipulating capabilities: capability-register context switching, tagged memory, preserving capabilities when copying memory, etc. Outside some limited code in sandboxes, capabilities were used by the compiler only where explicitly annotated, and no implied virtual addresses (e.g., as used in dynamic-linker-implemented Global Offset Tables (GOTs), return addresses, or v-table pointers) were implemented as capabilities.

This paper extends pure-capability support to the full userspace process environment, addressing for the first time topics such as capability interactions with system calls, signals, dynamic linking, and process debugging. Critically, this allows DDC to be assigned a value of NULL, eliminating legacy MIPS loads and stores utilizing DDC implicitly; all memory accesses are performed *intentionally* through explicit capabilities having reduced permissions and narrowed bounds.

3 Abstract capabilities

Our work is based on an abstract conceptual model in which every legitimate memory access is via a deliberately constructed *abstract capability* (following the principle of intentional use), which allows that access but (following the principle of least privilege) should not allow access to unrelated data. Such capabilities are constructed by some legitimate chain of operations rooted at primordial, omnipotent, capabilities.

The abstract capability model is implemented with a subtle combination of architectural capabilities (as provided by the hardware) and the critical systems code involved in managing paging, context switching, linking, memory allocation, and suchlike. The model allows programmers to reason about the effective properties of capabilities without the need to understand the details of each of these issues. This is analogous to the traditional Unix abstract process model, where virtual memory, file descriptors, credentials, threads, and signals combine to form a coherent programming environment in which programmers can reason about processes in relatively simple terms, without knowing all the implementation details.

The main challenge in defining the concept of an abstract capability is the fact that our architectural capabilities are expressed in terms of virtual addresses, and therefore have meaning only in conjunction with a specific interpretation thereof. Given a fixed and *total* virtual-to-physical mapping, each capability would authorize access to a specific subset

[2] Achieving 128-bit pointers requires *compression* of each capability (e.g., including 64-bit position, lower bound, upper bound; permission flags; and other metadata). An alternative implementation, which more directly encodes capabilities, uses 256 bits per pointer (plus the tag bit). Compression exploits commonalities in position and bounds values, but requires that large spans are aligned and sized at larger than byte granularity. Such constraints affect memory allocators and stack layout, which must pad allocation sizes up to ensure that capability references do not overlap.

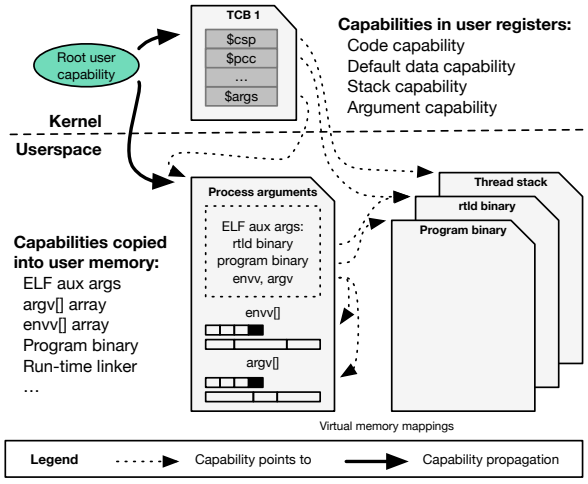

Figure 1. Process creation installs capabilities into both the user-thread register file and initial memory mappings.

of physical memory. However, operating systems actually maintain *partial* mappings, relying on page faults to provide illusory additional physical memory (e.g., by paging to and from disk, zero filling on demand, or copying on writes). Moreover, these mappings are dynamic, as processes request and release memory. We therefore introduce a model in which an abstract capability contains a set of access rights to *abstract memory* and a conceptual *abstract principal ID*. Principal IDs are freshly created for the kernel and each process address space, unique over the entire execution. Abstract physical memory is an unbounded byte array, a source of never-before-used addresses for each successive allocation. The OS is responsible for concretizing this abstract model by implementing it using architectural features. That is, the OS maintains invariants about its virtual-to-physical mappings and paging-related metadata, e.g., to prevent an architectural capability from mistakenly being used to access private abstract memory of another process either though virtual-to-physical aliasing or incorrect paging.

In our experimental system, abstract and architectural capabilities are constructed in the following ways:

CPU reset At hardware reset, maximally permissive architectural capabilities are provided to the boot code.

Kernel startup The kernel deliberately narrows these boot capabilities to ones separately covering userspace, kernel code, and kernel data.

Process address-space creation When a process address space is replaced by `execve`, the kernel establishes new memory mappings for the contents of the address space. It subdivides the previously created userspace capability into one for each mapped object (text, data, stack, arguments, etc). The newly mapped virtual memory maps onto physical memory disjoint from that currently mapped for any other process or the kernel, excepting mappings deliberately shared between processes (including read- or execute-only and copy-on-write pages). Conceptually, we create a fresh principal ID, and the initial user abstract capability encompasses all the physical-memory access rights of those architectural capabilities with regard to the new memory mapping. (See Figure 1.)

Context switching The kernel saves and restores user-thread register capability state, and updates the virtual-to-physical mappings if required. (Figure 2 illustrates this along with swapping and signal handling.)

Swapping Any userspace page and some kernel pages may be swapped out to external storage, which does not preserve tags. The swap subsystem scans evicted pages, recording tags in the swap metadata. When pages are restored, the swap-in code derives a *new* architectural capability from the saved values and an appropriate root capability. This preserves the abstract capability, despite the break in the architectural capability chain.

Signals Signal delivery is similar to context switching, except that the register state is copied to the signal stack for modification. Access to, and manipulation of, saved capability state by the signal handler preserves the architectural capability chain.

System calls When a process makes a system call with a pointer argument (e.g., passing a reference to a buffer), the kernel will use the passed-in capability when dereferencing that pointer (rather than using its own, elevated, authority). We have altered all standard methods of accessing process memory to use an explicit capability (respecting the principle of intentional use). This ensures that the kernel accesses only the memory specified – and authorized – by the user process. Figure 3 illustrates an example path for such a capability from userspace to a `copyin` routine.

Automatic references As references are taken to automatic variables, compiler-generated code derives bounded capabilities to those objects from the stack capability.

Dynamic linking The run-time linker creates subsets of the program and library data capabilities for each global variable, and from code capabilities for each jump destination, and places them in a capability-extended GOT.

PC-relative accesses Values installed in PCC are bounded to shared objects, constraining control flow and limiting potential for arbitrary code execution.

C/C++ function calls At C or C++ function calls and returns, the stack capability is updated to the new stack frame, and both the previous stack frame and return capability are spilled to the stack. The use of capabilities in the return path strongly limits attackers' ability to leverage memory-safety errors by requiring that these capabilities be overwritten with capabilities with appropriate permissions in order to escalate an attack.

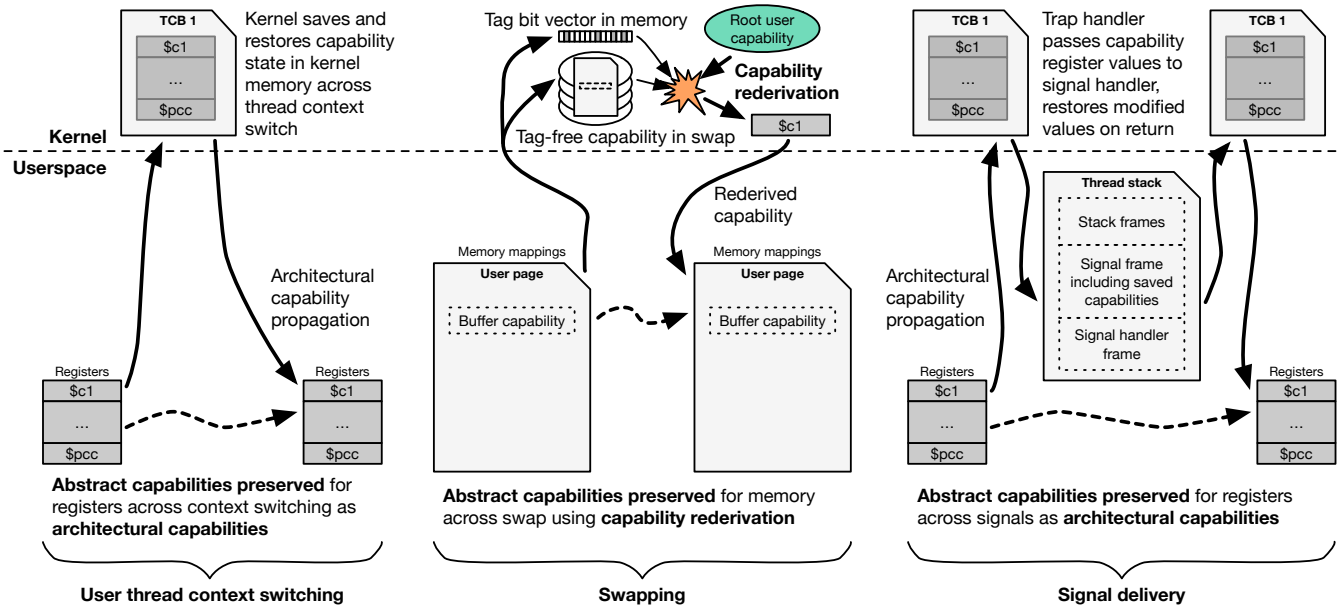

Figure 2. Context switches preserve abstract capabilities using architectural capabilities or capability rederivation.

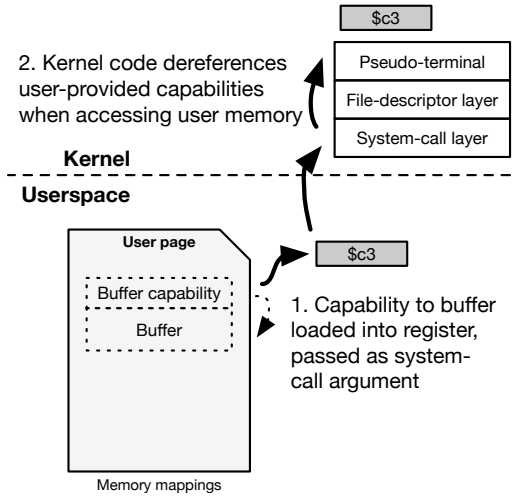

Figure 3. The kernel observes abstract capabilities by accessing user memory only through user capabilities.

C pointer arithmetic C pointer arithmetic manifests as arithmetic on the address contained in the architectural capability, leaving its bounds and permissions unchanged.
Memory allocation Each allocator (e.g., `malloc` and TLS) maintains a set of architectural capabilities to regions allocated by `mmap`, from which it derives narrower responses to requests. Freed capabilities are used to look up internal capabilities and are then discarded.
Pointer propagation Architectural capabilities are maintained across various low-level C idioms including explicit and implied memory copies (e.g., memory sorting and bitwise pointer manipulations used in locking code).
Debugging Two processes are involved in debugging – the debugger and the target – and hence two different principal IDs. Abstract capabilities belong to one or the other, and must not be propagated between them. The debugger process may inspect capabilities from, or inject capabilities into, the target memory or register file; these capabilities are derived from an appropriate extant *target* or root architectural capability.
Legacy loads and stores We prohibit legacy (integer pointer) loads and stores by installing a NULL capability in DDC.

We must ensure not just that the capability used for an access is legitimate and appropriately minimal, but also that the whole set of capabilities available to the code is appropriately minimal, otherwise we would provide weaker mitigation than desired. Most notably, each principal's abstract capability has a disjoint root.

4 Implementation

Our goal is to compile, run, and evaluate the complete C-language userspace of the FreeBSD operating system, compiled such that all dereferenceable pointers and implied virtual addresses are implemented as capabilities having *valid provenance* and *minimized bounds*. The question is then: by how much can we reduce bounds, given the constraints of compatibility with existing code and system-call APIs?

To achieve our goal, we have made changes to the CHERI ISA, the C compiler, the C language runtime, the virtual-memory APIs, and the CheriBSD kernel.

In our prior work [9, 48] on pure-capability C, we derived bounded capabilities on static, global, automatic, and

dynamic allocations from a single, address-space-covering capability. Considered under the lens of the principle of least privilege, this is undesirable. In this work, we set DDC to NULL, and place bounds on PCC and global references derived from capabilities to regions mapped by the kernel in `execve` or `mmap`.

Our implementation extends our prior work on pure-capability code from a sandbox environment [48] with limited POSIX compatibility to a new process ABI, CheriABI. In CheriABI, all pointers are capabilities, and all kernel manipulations of process memory are via explicitly delegated capabilities. In our prior work, the kernel interacted with capabilities via assembly stubs. Our enhanced version of the CheriBSD kernel is a hybrid C program where nearly all interactions with userspace are via explicitly annotated capability pointers. Of the 675 C and 8 assembly language files in our test kernels, 26 were created to support capabilities and 146 required adaptation for capabilities. In the full kernel source, about 750 files were touched. Other than a single file, implementing the CHERI-MIPS specific portions of CheriABI, the changes for CheriABI apply to any CHERI implementation. We continue to support the large suite of "legacy" mips64 userspace applications that adhere to the SysV ABI [44], alongside CheriABI userspace programs.

As part of our baseline we assume that the kernel ensures that the correct physical pages are mapped when accessing user pages from the kernel. This places the kernel within the trusted computing base (TCB) of the process. Our changes to use capabilities for all access to userspace limit the degree to which the kernel can be tricked into performing incorrect accesses. FreeBSD MIPS (and most other architectures) reserves the low portion of the address space for the current user virtual-memory map as an optimization for `copyin` and related functions. We rely on that being correct and, currently, on the kernel using that mapping only via authorized functions.

Starting CheriABI processes with execve CheriABI (and legacy) programs are mapped into the process address space by the `execve` system call, along with command-line arguments, environment, an initial stack, and the run-time linker as shown in Figure 1. Legacy programs store the argument, environment, and ELF auxiliary argument arrays at the top of the stack, adjusting the initial stack pointer appropriately [15, 44]. CheriABI processes have an identical set of arrays, but all pointers are bounded capabilities; the C run-time uses pointers to the arguments and environment in the ELF auxiliary arguments, rather than knowledge of the stack layout. Once loaded, the program drives further changes to the address space (including dynamic loading) via system calls. The program refines both the initial capabilities provided at startup and capabilities to later mappings.

Run-time capability refinement Compiler-generated code sets bounds on references to variables on the stack. These prevent classic stack-based buffer overflows. Dynamic allocation is via a lightly modified version of JEMalloc. We install bounds matching the requested allocation before return, and rederive pointers to the underlying storage in the `free` and `realloc` paths using existing internal interfaces.

Thread local storage We have added a CHERI-compatible TLS implementation modeled on the MIPS implementation. Bounds are per shared-object rather than per variable, to avoid an extra indirection.

Dynamic linking We extended the dynamic linker (*RTLD*) to initialize external symbol references using new dynamic relocations that initialize and bound the capability. Global variables containing pointers are initialized during process startup, as tags are not preserved on disk; this adds overhead comparable to position-independent binaries (commonly used to improve ASLR [20, 43]). We bound function symbols' resolved capabilities to the shared object. While these bounds are not minimal, this preserves the ability of code to use branches in place of jumps between functions. The wide bounds also facilitate the existing practice of referencing global variables using program-counter-relative addressing.

System calls Our implementation transforms the kernel into a hybrid C program where all access to userspace for CheriABI processes is via explicitly annotated capability pointers. System calls use these architectural capabilities to access process memory during a call. When acting on behalf of a CheriABI process, non-capability versions of `copyout` and `copyin` return errors, ensuring that all access to process memory is via explicit capabilities. To reduce the risk of accidental copying of capabilities between userspace and the kernel, we strip tags from copied capabilities unless special interfaces are used. It is usually obvious when these are needed, but `ioctl` and `sysctl` present challenges: it is not easy to tell whether capabilities should be preserved for a given command. Some management interfaces export kernel pointers. Where we have encountered them, we have altered them to expose virtual addresses rather than kernel capabilities. A few system calls take pointers and store them in kernel data structures for later return. In all such cases (including signal handling, asynchronous I/O, and FreeBSD's `kevent`), we have modified the kernel structures to store capabilities and the legacy ABIs to convert pointers into capabilities for storage.

Virtual-address management APIs Programs add memory mappings through three system calls: `sbrk`, `mmap`, and `shmat`; `mmap` and `shmat` allocate (or alter the mappings of) regions of memory, returning a reference to the new allocation. In CheriABI we have altered them to return capabilities that are bounded to the requested allocation length, with permissions derived from the requested page permissions. Both `mmap` and `shmat` allow the caller to specify a target address for the mapping. In the case of `mmap`, the address can be an optional hint or a fixed address for the mapping. With `shmat`, a fixed address is supported. If the fixed address is

a valid capability, we require that it have the vmmap user-defined capability permission. When a capability is passed as the hint argument, the returned capability is derived from it, preserving provenance. In mmap we allow untagged values and capabilities without the vmmap permissions as hints; however, if the caller requests a fixed mapping, we allow it only if it would not replace an existing mapping. We also require the vmmap permission to be present on capabilities passed to munmap and shmdt. This prevents the possibility of replacing the contents of arbitrary memory without a valid capability.

The sbrk API adjusts the heap size. It predates modern APIs such as mmap – and is obsolete. An implementation is possible with some limitations, but few programs require it – so we do not support it in our prototype.

Signal handling CheriABI signal handling is similar to legacy signaling, excepting adjustments for ABI differences in function calls and stack management, and the fact that the return trampoline capability is a tightly bound capability to a read-only shared page mapped by execve.

Dynamic allocations We have altered the FreeBSD malloc implementation (JEMalloc[19]) to return capabilities bounded to the requested size. These allocations are non-executable and have the vmmap permission stripped preventing them from being used to remap memory under management by malloc. We have similarly altered the TLS allocators in libc and RTLD.

Additional changes As we expanded the set of code run under the pure-capability ABI, we found that we needed to extend qsort and other sorting routines to preserve capabilities when swapping array elements. We also replaced use of Berkeley DB as an in-memory hash table, because BDB does not preserve the alignment of stored data. We also addressed a number of issues related to treating integers as pointers: casting pointers through integer types other than intptr_t and expecting to get pointers back; and integer manipulation of pointers to store flags in unused bits, to adjust the alignment of pointers, or to access the virtual address.

Debugging Beyond supporting the ptrace system call, we have extended it to permit reading the values of capability registers and arranged for register values to be stored in core dumps. We have modified GDB to add limited support for capabilities, including dereferencing capability pointers interactively and unwinding stacks. However, there is currently no facility for examining additional properties of capabilities stored in memory, such as validity (tags), bounds, or permissions – as this is not a natural fit for GDB's internals. The user interface does not enforce bounds or permissions, or check the tag, when dereferencing a capability pointer.

Limitations Our implementation has a few limitations. We support nearly all system calls, but have excluded sbrk as a matter of principle, and avoided a small number of administrative interfaces due to the tedium of translating pointer heavy argument structures. Where we have found them necessary, ioctl and sysctl interfaces involving structs containing pointers have been translated, but we have skipped some cases, such as some storage-management interfaces. These limitations are not fundamental, and could be overcome with further engineering effort. The exclusion of sbrk is likely correctable, but the FreeBSD Project shipped the Arm64 and RISC-V ports without it, with few reports of problems; emacs has since removed the requirement for sbrk.

5 Evaluation

We evaluated CheriABI in several dimensions. (1) To demonstrate the completeness of the CheriABI implementation, we ran the FreeBSD and PostgreSQL test suites. (2) We validated performance assumptions with system-call microbenchmarks and whole-application benchmarks. (3) We examined the changes required to support CheriABI across FreeBSD userspace as well as PostgreSQL. (4) To demonstrate the practical value of capability restrictions, we examined a set of memory-safety test cases that show the effectiveness of CHERI protections and relate to memory-safety bugs found in real code. (5) Finally, we used ISA-level traces to reconstruct the abstract capabilities of a process and to study the increased granularity of capabilities in CheriABI programs. Where suitable, we compare with LLVM Address Sanitizer [38]. While the overheads of this software-based sanitizer are significant, it is widely used, primarily as a debugging tool. In an ideal world, we would compare to HWASAN [39] rather than the software implementation, but we do not have an implementation for MIPS.

We benchmark on FPGA, for microarchitectural realism, using a version of CHERI written in Bluespec SystemVerilog, and synthesized for the Stratix IV FPGA at 100MHz. The pipeline is in-order and single-issue, roughly similar to the ARM7TDMI. Our FPGA system has 32-KiB L1 caches and a shared 256-KiB L2 cache, all set-associative, similar to widely shipped CPUs such as many ARM Cortex A53 implementations, although without pre-fetching. Performance and memory scaling are broadly similar to these commercial implementations. Specific performance is subject to the peculiarities of our microarchitecture. For ISA traces and tests, we use a CHERI-extended version of QEMU.

We have benchmarked on 128-bit CHERI, as its lower overheads make it a more realistic candidate for commercial adoption. In an effort to reduce unnecessary differences, benchmark software for CheriABI and MIPS are both compiled with the CHERI compiler (based on LLVM pre-8.0) and use the CheriABI-capable kernel.

5.1 Test suites

The FreeBSD test suite is part of the FreeBSD base system and provides tests for many programs and libraries. The test suite contains over 3500 programs (most of which test many

	Pass	Fail	Skip	Total
FreeBSD MIPS	3501	90	244	3835
FreeBSD CheriABI	3301	122	246	3669
PostgreSQL MIPS	167	0	0	167
PostgreSQL CheriABI	150	16	1	167
libc++ MIPS	5338	29	789	6156
libc++ CheriABI	5333	34	789	6156

Table 1. Test suite results

conditions). We ran the test suite on a mips64 system and CheriABI. The results are summarized in Table 1. In addition to these tests, we use a CheriABI version of CheriBSD under Qemu for daily development.

We also ran the PostgreSQL version 9.6 `pg_regress` test suite. Of the 16 test failures 8 fail because the outputs are sorted in a different order or the test assumes a pointer size of 4 or 8 bytes. One test is failing due to the use of under-aligned pointers, which will trap on CHERI. The remaining failures are returning slightly different results and still need further investigation.

Finally, we also ran the `libc++` tests both for MIPS and CheriABI, and encountered only five additional failures – due a missing runtime library function for atomics – compared to the MIPS baseline.

5.2 Performance

To evaluate the impact of pure-capability compilation, we have run the MiBench benchmarks [21], which are intended to be commercially representative embedded programs. Each is composed of a tight inner loop and spends very little time in the kernel, providing a sampling of performance for pure-capability execution. Additionally, we have run portions of SPEC CPU2006. Figure 4 shows the results of these runs, most of which are well within the noise level for compiler and cache differences. We have excluded a number of bit-manipulation, compression, encryption, and image-processing benchmarks where the separate capability register file in CHERI-MIPS allows the compiler to generate more efficient code in the benchmark kernel (the security-sha benchmark shows an example of this effect).

To determine the worst-case impact of our system-call interface changes, we ran the FreeBSD system call timing benchmarks. Performance impact varies from 3.4% slower for `fork`, to 9.8% faster for `select`. We believe the latter is due to the cost of creating capabilities from four pointer arguments in the CHERI kernel.

As a macro-benchmark, we use the PostgreSQL database `initdb` tool, which sets up a new database. We chose PostgreSQL because it is a large real-world workload written in C. Furthermore, it also makes use of various IPC mechanisms (including sockets, shared memory and semaphores); it is the largest program that we have run dynamically linked so far. Overall, PostgreSQL is only 6.8% slower as a CheriABI binary, even without any changes to reduce the impact of pointer size on structure padding. In contrast to this, compiling the `initdb` binary with Address Sanitizer instrumentation (but without instrumented library dependencies) requires 3.29 times more cycles to complete.

Regarding the performance impact of CheriABI, we discovered that the immediate range of the capability-relative load instruction (CLC) was often too small – leading to expensive accesses to globals. We added a new CLC with larger immediate, allowing most GOT entries to be accessed with a single instruction (as in MIPS). This reduces the code size of most binaries by over 10%, and reduces the `initdb` overhead from 11% to 6.8%.

5.3 Compatibility

CheriABI is almost entirely compatible with the de-facto standard POSIX programming environment used by modern UNIX-like systems. Most programs require no modifications to compile and run successfully. Of the nearly 800 C programs in the FreeBSD source tree, we exclude two management utilities that require compatibility shims to work with CheriABI and the non-CHERI supporting toolchain. Table 2 includes a breakdown of the changes required, summarizing the types of changes and compiler updates to find or address them.

There are three classes of provenance related issues. *Pointer provenance* (**PP**) covers attempts to create a pointer to an object from a pointer to an unrelated object, and attempts to pass pointers over IPC channels. *Integer provenance* (**IP**) relates to the loss of provenance as pointers are cast though integer types other than `uintptr_t`. and *Monotonicity* (**M**) refers to code that assumes it can reach outside object bounds or increase permissions. We have generally found these through debugging.

Pointer shape (**PS**) reflects changes due to the increased size of CHERI capabilities vs. integer virtual addresses. Some objects must be enlarged or more strongly aligned; some struct padding computations required tweaking. Existing alignment warnings help locate many of these.

Pointer as integer (**I**) covers storing sentinel integers in pointers (e.g., `MAP_FAILED` is `(void *)-1`).

There are a number of issues related to manipulating pointers as *virtual addresses* (**VA**). Several are sufficiently common that we have broken them out: *Bit flags* (**BF**) refers to storing flags (e.g., lock status) in the low bits of pointers. *Hashing* (**H**) means computing a hash from a virtual address. *Alignment* (**A**) counts adjusting the alignment of a pointer (e.g., rounding up a `(char *)` to permit storing a pointer). We have added compiler warnings for bitwise math and remainder operations on capabilities. Additionally, we have created a new compiler mode and a supporting `CGetAddr` instruction in which casts of pointers to integers produce the virtual

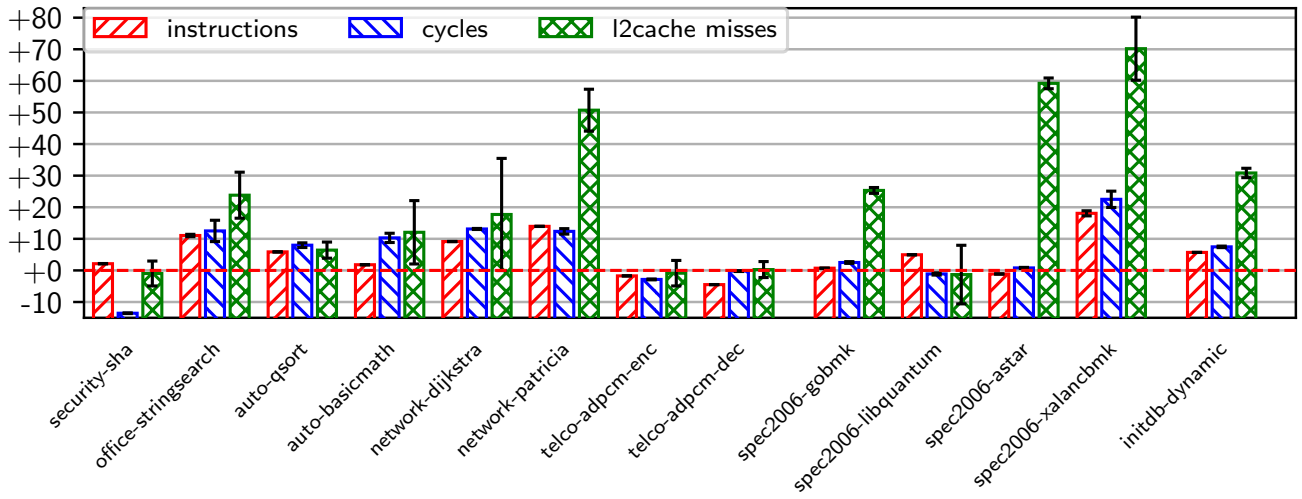

Figure 4. MiBench, SPEC CPU2006, and PostgreSQL benchmark median overheads relative to MIPS baseline. Error bars are interquartile ranges (IQRs).

	PP	IP	M	PS	I	VA	BF	H	A	CC	U
BSD headers	0	8	0	4	2	1	1	0	3	2	0
BSD libraries	5	18	4	19	22	20	11	6	19	42	19
BSD programs	1	11	1	3	13	0	0	0	7	11	2
BSD tests	0	0	0	0	2	0	0	0	2	7	2

Table 2. CheriABI changes. PP: pointer provenance, IP: integer provenance, M: monotonicity, PS: pointer shape, I: pointer as integer, VA: virtual address, BF: bit flags, H: hashing, A: alignment, CC: calling convention, U: unsupported

address (rather than the offset used in prior work). For this paper, we compile CheriBSD in the old mode, but have switched the default to use this mode – based on our experience. This new mode would obviate many of our changes.

The *calling convention* (**CC**) for pure-capability code differs from the MIPS ABI: integer and pointer arguments use different register files, and variadic arguments are always spilled to the stack and passed via a capability. These require correct function prototypes to ensure that values are passed as expected and that we handle variadic arguments directly. The implementation of system calls, including `open` and `syscall`, and a callback API in SunRPC, depends on the overlap in calling conventions on existing architectures. For system calls other than `syscall`, we handle the 'optional' argument as a variadic argument in the C library. In the SunRPC case, programs declare their own callbacks; thus, fixing each one is the only possible solution. We have found numerous bugs due to the variadic calling convention, which poses a compatibility challenge. We have added warnings when calling functions without declared arguments,[3] or converting between variadic and non-variadic function pointers.

The *unsupported* (**U**) label covers an array of things CHERI or CheriABI doesn't support – including XOR on pointers and the `sbrk` system call.

5.4 Memory protection benefit

To evaluate memory safety, we used the BOdiagsuite suite of 291 programs from Kratkiewicz [24] and used by Hardbound [17]. These were intended for testing static analysis tools, but are also useful for dynamic enforcement. The test suite consists of an assortment of C bounds violations, a small number of which use POSIX APIs such as `getcwd` with an incorrect length.

Each program has one variant with no memory-safety errors, and three variants that contain bugs (shown as the column heads in Table 3). **min** has the smallest possible memory safety violation (typically off by one byte); **med** has

[3]Unlike `-Wstrict-prototypes` this warning also allows calls to legacy K&R declarations as long as the declaration with the argument types is visible at the call site. We allow these calls since this style of C function declarations is still very common throughout the FreeBSD source tree.

	min	med	large
mips64	4	8	175
cheriabi	279	289	291
asan	276	286	286

Table 3. BOdiagsuite tests with detected errors (among 291 total tests)

an off-by-8-bytes error; and **large** has an off-by-4096-bytes error. We compiled these variants with optimization disabled, because a number of cases had the violation in code that the optimizer removed (e.g., dead stores to variables not marked `volatile`.) We verified that the variants without memory-safety errors ran correctly as CheriABI and mips64 programs, and then ran all the variants for CheriABI, mips64, and Address Sanitizer [38]. The results are summarized in Table 3. For CheriABI, 279 of the "min" tests aborted with a memory safety violation, while 12 tests contained intra-object memory safety violations. The current CheriABI design does not protect against this, and it cannot be protected without some impact on compatibility [9].

By way of comparison, only 4 mips64 programs fail in the "min" case, and Address Sanitizer fails to catch two cases. Many cases fail in the "large" variants for mips64, but even then 116 do not. Based on these results, CheriABI protection is significantly better than mips64, and slightly better than Address Sanitizer – a scheme with very high overheads (3× stack memory, 12.5% total memory, around 50% performance).

In addition to finding test-suite issues, we have found and fixed dozens of bugs including buffer bounds violations and variadic argument misuse in FreeBSD programs, libraries, and tests. These include: a buffer underrun read in `tcsh` shell history expansion on an empty command line; an out-of-bounds read by the kernel in the FreeBSD DHCP client due to underallocation of the data argument to an `ioctl` call; small buffer overflows in the `ttyname` and `humanize_number` functions and in a test case for the `strvis` function; and many cases in the FreeBSD test suite where `open` was called with a missing permission argument. A recently discovered information leak in kernel management interfaces was partially mitigated by CheriABI. We have fixed these issues in FreeBSD.

5.5 Abstract capability analysis

Because capabilities are explicitly manipulated, we can use an instruction trace (from Qemu) to track capability derivation and use, in order to reconstruct the abstract capability of a process. We have done so for an `openssl s_server` implementation during a client connection set-up and exchange of a small file. `openssl` is a small representative application that exercises the majority of the changes we introduced with

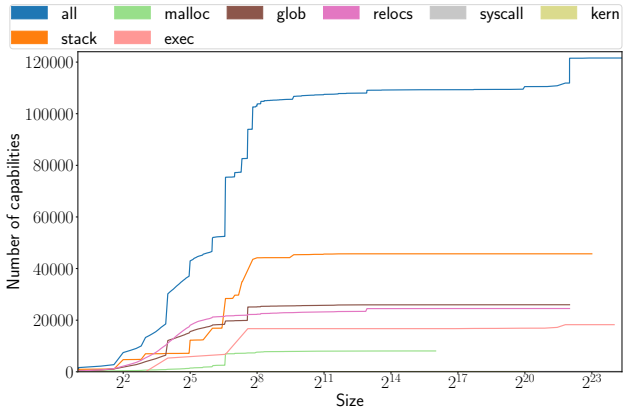

Figure 5. Cumulative sum of the number of capabilities against size of bounds, for different sources of capabilities visible in userspace during a run of openssl s_server involving startup, authentication and a file exchange. (Note that the *kern* and *syscall* lines are present, but virtually indistinguishable from the X-axis.)

CheriABI: it uses thread-local storage, is dynamically linked with multiple libraries, performs considerable memory allocation and pointer manipulation, and exercises system calls. Further, `openssl` is interesting to a history of memory-safety vulnerabilities including Heartbleed [4].

Figure 5 shows the granularity of capabilities from several sources. Each curve in the graph shows the number of capabilities created during the program execution that have bounds size up to the corresponding value on the x axis. Each line represents a different source of capabilities and terminates at the point corresponding to the size of the largest capability found. The baseline for a legacy program would be a vertical line at the maximum user address, because all pointers are implicitly bounded by DDC. As we bound more pointers, the line moves towards the left of the plot, leaving almost no capabilities with large bounds. This is the case in CheriABI `openssl`, where no capability grants access to more than 16MiB of memory, and around 90% grant access to less than 1KiB. Capabilities created from the *stack* capability and *malloc* are well bounded, and permit access to no more than 8MiB of address space: in contrast, pointers in legacy programs could be arbitrarily modified to point to any mapped memory addresses. There are a few capabilities that originate in the kernel and are assigned to the process at startup or returned by system calls. These capabilities point to relatively large objects, such as mapped sections of the executable and are the broadest capabilities in the program. The rest of the broad capabilities are generated as temporary values that are then bounded to a smaller size.

6 Future work

Sub-object and code bounds We do not tighten bounds on references to members of structs, for compatibility with popular patterns such as `container_of`. Most references to struct members could be bounded safely, but the exceptions require exploration and evaluation. Similarly, bounds of code pointers are an area of ongoing research. Tightening the bounds will require eliminating PC-relative addressing from generated code.

Temporal safety CHERI provides the minimum infrastructure required to implement accurate garbage collection and temporally-safe memory reuse: atomic pointer updates and the precise identification of pointers. Because system calls may pass user pointers into the kernel that are then held for an extended duration, new interfaces will be required to expose this information to the garbage collector or heap allocator. Work on a CHERI-aware temporally-safe allocator is ongoing.

Memory APIs The `mmap` API family has significant limitations when implementing least-privilege policies. Combining capabilities, W^X, and JIT compilation will require new interfaces. The `mremap` system call is commonly used by Linux allocators, but is not explored in CheriBSD.

Debugging Existing debuggers encode a flat, integer address space model. More complex models may be necessary for debugging, particularly with modern JIT-based debuggers. Injecting capabilities into debug targets requires further exploration.

Static analysis Our work on compiler warnings to locate code requiring changes for CHERI C is a start, but more complex analysis is likely required to detect provenance bugs such as those common with `realloc` misuse.

Pure-capability kernels Our current, hybrid kernel allows all user-space pointers to be capabilities with appropriate bounds, but most kernel pointers remain unprotected. A pure-capability kernel would increase protection and require a different set of changes.

Formal model of abstract capabilities The abstract capability is currently a purely conceptual model, useful to enable programmers to reason about complex systems. A formalization of the conceptual model is desirable, and we are working on portions of the problem.

Cache studies While our FPGA implementation resembles low-end commercially available hardware, neither our cache hierarchy nor our pipeline resembles a modern super-scalar CPU. A traced-based cache analysis (e.g. using Gem5) would be an appropriate way to address this shortcoming.

Capabilities beyond the CPU CHERI capabilities act on virtual memory and protect access by CPU instructions, but other system components such as DMA devices and IOMMUs also interact with memory. Work on attacking systems with IOMMUs [27] shows the need for strong memory protections beyond the CPU.

7 Related work

CheriABI imposes an abstract capability on the popular POSIX programming environment [2, 29, 30, 40]. This work builds upon the CHERI C environment [9], but substantially extends it across a full real-world OS design.

In the quest to address the array of memory-safety problems in C code [41], there is a long history of memory-protection research adding bounds checking to C programs. The CHERI [50], HardBound [17], and SoftBound [32] research systems add fine-grained memory protection to C. Intel's Memory Protection Extensions [22] provide an example of a commercial hardware implementation of bounds checking – albeit with limited value in many environments due to its fail-open policy. The Address Sanitizer framework [38] aims primarily at temporal safety, but provides some protection for dynamic memory allocations, although with significant performance overhead. Annotation- and proof-driven systems such as CCured [11, 33], Deputy [10], and Checked C [42] attempt to prevent API misuse (which often results in buffer overflows), prove buffer use correct, and insert checks where such proofs prove intractable. Much of this work was inspired by Cyclone's [23] safer (but incompatible) dialect of C. CHERI draws from ideas proposed in the M-Machine [7], particularly object-granularity capabilities protected by in-memory tags (and earlier by PSOS [34]). Unlike M-Machine, CHERI has a stated design goal of source-code and binary compatibility, enabling a transition path from a flat memory model to a pure-capability system. Applications of memory-safety technologies have typically focused on either userspace or kernels, with the majority targeting userspace applications. Notable exceptions are Mondrix [49] (applying Mondriaan to Linux), Safe TinyOS [12] (applying Deputy to TinyOS), and KernelAddressSanitizer[14].

Our use of CHERI capabilities as pointers treats all pointers much as `WILD` pointers in CCured, similar to the treatment in HardBound. Like HardBound, our tags prevent arbitrary overflows from altering pointers. Our approach of increasing the pointer size in-place requires recompilation, but avoids involving the overhead to additional pages beyond those required for tags and the need for transactional memory in HardBound. CheriABI uses CHERI to enforce memory bounds along with stronger protections including integrity, provenance validity, and monotonic access on all pointers in userspace. Memory allocated by the kernel is bounded when returned rather than relying on the runtime to place bounds. In addition to bounding explicit pointers in C source code, we bound implicit pointers including those used for runtime linking. Additionally, CheriABI brings this memory safety into the kernel, requiring the kernel to honor the same protections when accessing userspace. This begins to bridge the memory-safety gap between the kernel and userspace.

Some mitigations aim to limit attacker's ability to jump to arbitrary code locations to limit the impact of attacks such as

return-oriented programming[6]. They include Control-Flow Integrity (CFI) [3], Code-Pointer Integrity (CPI) [25], and Cryptographic Control-Flow Integrity (CCFI) [28] CheriABI limits vectors for such attacks, but does not explicitly model control-flow graphs as in CFI.

Other work on hardware stack protection includes Roessler et al. [36], which utilizes configurable tagged-memory policies similar to Pump [18]. Both are capable of implementing pointer-oriented protection, but have not been applied to complete software stacks.

8 Conclusion

We have demonstrated a complete memory-safe UNIX system that is practical for general use. We support critical real-world technologies such as dynamic linking, and provide protection all the way into the kernel – even to dark corners including `ioctl`.

We have introduced the *abstract capability* as a new abstraction for memory safety in operating systems. The maintenance and enforcement of appropriately minimal abstract capabilities for pure-capability processes is implemented on FreeBSD as CheriABI. Our implementation allows most application software to be recompiled without change, and obtains significant memory safety benefits while preserving acceptable performance – even in a minimally optimized prototype.

Our evaluation of compatibility of existing C code with the pure-capability C-language model, FreeBSD and PostgreSQL test suites, and micro- and macro-benchmarks demonstrates that this approach is feasible. Our use of the BOdiagsuite shows that constraints allow us to catch C-language bugs, and our trace-based abstract capability reconstruction shows that we substantially improve the granularity of pointer bounds.

While preserving support for legacy software, our implementation of CheriABI shows the existence of a path forward from our current run-time foundations set on the shifting sands of integer pointers, to a future where strong referential integrity enforces the principles of least privilege and intentionality even on lowest-level software.

9 Acknowledgments

We thank our colleagues Hesham Almatary, Jonathan Anderson, Ross Anderson, David Brazdil, Ruslan Bukin, Gregory Chadwick, Nirav Dave, Lawrence Esswood, Bob Laddaga, Lucian Paul-Trifu, Colin Rothwell, Linton Salmon, Howie Shrobe, Domagoj Stolfa, Andy Turner, Munraj Vadera, Stu Wagner, Hongyan Xia, Bjoern Zeeb, our shepherd Santosh Nagarakatte, and our anonymous reviewers for their feedback and assistance. This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 ("CTSRD") and HR0011-18-C-0016 ("ECATS"). 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 also acknowledge the EPSRC REMS Programme Grant (EP/K008528/1), the ERC ELVER Advanced Grant (789108), Arm Limited, HP Enterprise, and Google, Inc. Approved for Public Release, Distribution Unlimited.

10 Availability

We have released the CHERI hardware and software stacks, specifications, and manuals, as open source on the CHERI CPU web site [1].

We have published an extended version of this paper with more coverage of implementation details and the various trade-offs made in the process as a technical report [16].

References

[1] CHERI open-source web site. http://www.cheri-cpu.org/. Accessed: 2018-12-16.

[2] The Open Group base specifications issue 7. Technical report, 2016.

[3] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM conference on Computer and Communications Security*. ACM, 2005.

[4] A. Alkazimi and E. B. Fernandez. "heartbleed": A misuse pattern for the openssl implementation of the ssl/tls protocol. In *Proceedings of the 23rd Conference on Pattern Languages of Programs*, PLoP '16, pages 6:1–6:8, USA, 2016. The Hillside Group.

[5] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, K. S. McKinley, M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications - OOPSLA '07*, volume 42, page 405, New York, New York, USA, 2007. ACM Press.

[6] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In P. Syverson and S. Jha, editors, *Proceedings of CCS 2008*, pages 27–38. ACM Press, Oct. 2008.

[7] N. P. Carter, S. W. Keckler, and W. J. Dally. Hardware support for fast capability-based addressing. *SIGPLAN Not.*, 29(11):319–327, Nov. 1994.

[8] D. Chisnall, B. Davis, K. Gudka, D. Brazdil, A. J. J. Woodruff, A. T. Markettos, J. E. Maste, R. Norton, S. Son, M. Roe, S. W. Moore, P. G. Neumann, B. Laurie, and R. N. M. Watson. CHERI JNI: Sinking the Java security model into the C. In *Proceedings of the Twenty Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, New York, NY, USA, 2017. ACM.

[9] D. Chisnall, C. Rothwell, B. Davis, R. N. M. Watson, J. Woodruff, M. Vadera, S. W. Moore, P. G. Neumann, and M. Roe. Beyond the PDP-11: Processor support for a memory-safe C abstract machine. In *Proceedings of the 20th Architectural Support for Programming Languages and Operating Systems*. ACM, 2015.

[10] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *Proceedings of the 16th European Symposium on Programming*, ESOP'07, pages 520–535, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] J. Condit, M. Harren, S. McPeak, G. G. Necula, and W. Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 conference on programming language design and implementation*, pages 232–244,

New York, NY, USA, 2003. ACM.

[12] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 205–218, New York, NY, USA, 2007. ACM.

[13] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the Multics system. In *AFIPS '65 (Fall, part I): Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 185–196, New York, NY, USA, 1965. ACM.

[14] J. Corbet. Software-tag-based KASAN. https://lwn.net/Articles/766768/, September 2018. Accessed: 2018-12-16.

[15] B. Davis. Everything you ever wanted to know about "hello, world"* (*but were afraid to ask.). In *Proceedings of AsiaBSDCon 2017*, AsiaBSDCon 2017, 2017.

[16] 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 (extended version). Technical Report UCAM-CL-TR-932, University of Cambridge, Computer Laboratory, Apr. 2019.

[17] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: architectural support for spatial safety of the C programming language. *SIGARCH Comput. Archit. News*, 36(1):103–114, Mar. 2008.

[18] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, B. C. Pierce, and A. DeHon. Architectural Support for Software-Defined Metadata Processing. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, March 2015.

[19] J. Evans. A scalable concurrent malloc(3) implementation for FreeBSD. In *BSDCan*, 2006.

[20] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*, Feb. 2017.

[21] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[22] Intel Plc. Introduction to Intel® memory protection extensions. http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, July 2013.

[23] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2002. USENIX.

[24] K. Kratkiewicz. Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code. Master's thesis, 2005.

[25] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, Berkeley, CA, USA, 2014. USENIX Association.

[26] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *20th Conference on Computer and Communications Security*. ACM, November 2013.

[27] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. M. Watson. Thunderclap: Exploring vulnerabilities in Operating System IOMMU protection via DMA from untrustworthy peripherals. In *Network and Distributed Systems Security (NDSS) Symposium 2019*, San Diego, USA, Feb. 2019. Internet Society.

[28] A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh. Cryptographically enforced control flow integrity. *arXiv preprint arXiv:1408.1451*, 2014.

[29] M. McKusick, K. Bostic, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, Reading, Massachusetts, 1996.

[30] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Pearson, 2014.

[31] K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell. Exploring C semantics and pointer provenance. In *POPL 2019: Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, Jan. 2019. Proc. ACM Program. Lang. 3, POPL, Article 67.

[32] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2009.

[33] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. *ACM SIGPLAN Notices*, 37(1):128–139, 2002.

[34] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A Provably Secure Operating System: The system, its applications, and proofs. Technical report, Computer Science Laboratory, SRI International, May 1980. 2nd edition, Report CSL-116.

[35] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974.

[36] N. Roessler and A. DeHon. Protecting the stack with metadata policies and tagged hardware. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 1072–1089, 2018.

[37] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[38] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Address-Sanitizer: A fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, 2012. USENIX.

[39] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov. Memory tagging and how it improves c/c++ memory safety. Technical report, February 2018.

[40] W. R. Stevens and S. A. Rago. *Advanced Programming in the UNIX Environment, 3rd Edition*. Addison-Wesley Professional, May 2013.

[41] L. Szekeres, M. Payer, T. Wei, and D. Song. Eternal war in memory. In *IEEE Symposium on Security and Privacy*, 2013.

[42] D. Tarditi. Extending C with bounds safety. Technical report, June 2016.

[43] the PaX Team. Address space layout randomization, 2006.

[44] The Santa Cruz Operation, Inc. System V application binary interface, intel386™ architecture processor supplement (fourth edition). Technical report, 1996.

[45] R. Watson, P. Neumann, J. Woodruff, J. Anderson, R. Anderson, N. Dave, B. Laurie, S. Moore, S. Murdoch, P. Paeps, et al. CHERI: A Research Platform Deconflating Hardware Virtualization and Protection. In *Workshop paper, Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012)*, 2012.

[46] R. N. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera. Fast protection-domain crossing in the cheri capability-system architecture. *IEEE Micro*, 36(5):38–49, Sept. 2016.

[47] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, D. Chisnall, B. Davis, N. W. Filardo, A. Joannou, B. Laurie, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Sewell, S. Son, and H. Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7). Technical Report UCAM-CL-TR-927, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, 2018.

[48] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. s Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, May 2015.

[49] E. Witchel, J. Rhee, and K. Asanović. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[50] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st International Symposium on Computer Architecture*, June 2014.