Technical Report

UCAM-CL-TR-932
ISSN 1476-2986

Number 932

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

CheriABI: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment

Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James 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, Jonathan Woodruff

January 2019

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report extends our paper of the same title published at ASPLOS 2019 with a focus on implementation details of interest to operating system and compiler developers.

Approved for public release; distribution is unlimited. Sponsored 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”), as part of the DARPA CRASH, and SSITH research programs. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government. Additional support was received from St John’s College Cambridge, the Google SOAAP Focused Research Award, a Google Chrome University Research Program Award, the RCUK’s Horizon Digital Economy Research Hub Grant (EP/G065802/1), the EPSRC REMS Programme Grant (EP/K008528/1), the EPSRC Impact Acceleration Account (EP/K503757/1), the ERC Advanced Grant ELVER (789108), the Isaac Newton Trust, the UK Higher Education Innovation Fund (HEIF), Thales E-Security, Microsoft Research Cambridge, Arm Limited, Google DeepMind, and HP Enterprise.

ISSN 1476-2986

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.

This technical report extends our paper of the same title published at ASPLOS 2019 [17] with a focus on implementation details of interest to operating system and compiler developers.

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 [44].

The MMUs found in contemporary processors are the result of a long co-evolution with the Multics (and then UNIX) process models [14, 38], 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 [40]. 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 without expensive and disruptive lookups.

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 [35, 36, 12, 11, 13, 3, 18, 28, 31, 29, 24, 55, 10, 52, 45, 9, 25, 41]. 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++, and assembly-language software stack, including the open-source FreeBSD OS [33] (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.

[1] By provenance, we mean a series of correct operations like those we describe in [34], not to the attribution of errors as in Bond [6].

Our approach implements abstract capabilities using Capability Hardware Enhanced RISC Instructions (CHERI) [52] *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 [48, 55], demonstrated an efficient compartmentalization framework above it [52, 49], described a C-language compilation mode where all pointers are capabilities [10], and demonstrated the use of those abilities to implement a safer JNI [9]. 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*. Specifically, any direct store to memory containing a capability not via a capability store instruction renders the stored capability invalid.

Monotonicity prevents the permissions or bounds associated with a capability from being increased. E.g. attempting to increase the length of a capability.

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 persistent storage and later restored. 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 [55, 10, 52, 9], 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 [51] in both the in-register and in-memory representations, increasing the in-memory size of pointers to 128 bits [54], plus an out-of-band tag bit.[2] There is one tag bit per capability-sized and capability-aligned re-

[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 con-

gion 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 [52, 9]. 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. With the exception of thread-local-storage, data pointers are restricted to C-language objects.

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.

straints affect memory allocators and stack layout, which must pad allocation sizes up to ensure that capability references do not overlap.

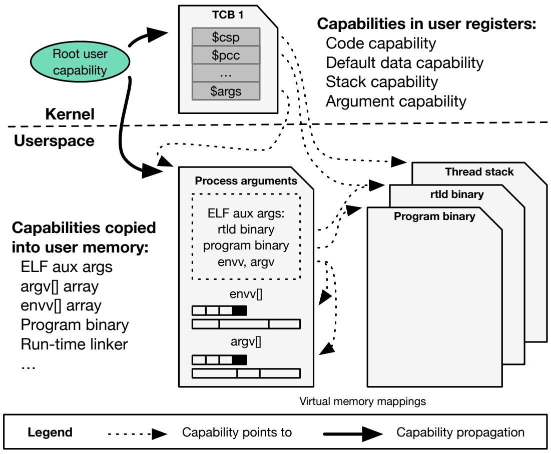

Figure 1: Process creation installs capabilities into both the user-thread register file and initial memory mappings.

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 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 storage, 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

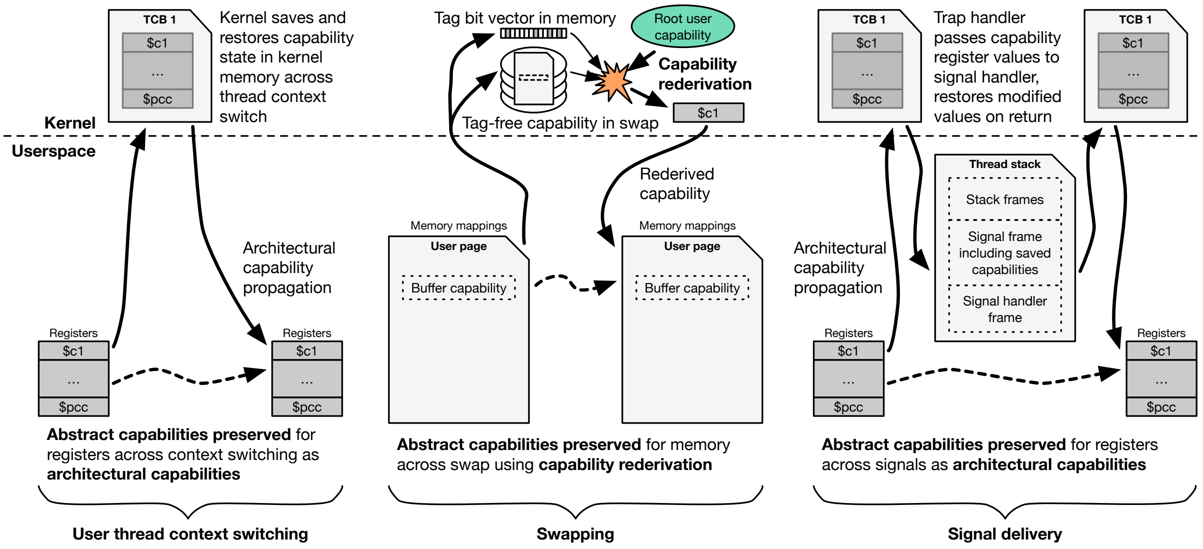
TCB 1
$c1
...
$pcc
Kernel saves and restores capability state in kernel memory across thread context switch
Kernel
Userspace
Architectural capability propagation
Registers
$c1
...
$pcc
Registers
$c1
...
$pcc
Abstract capabilities preserved for registers across context switching as architectural capabilities
User thread context switching
Tag bit vector in memory
Root user capability
Capability rederivation
Tag-free capability in swap
$c1
Rederived capability
Memory mappings
User page
Buffer capability
Memory mappings
User page
Buffer capability
Abstract capabilities preserved for memory across swap using capability rederivation
Swapping
TCB 1
$c1
...
$pcc
Trap handler passes capability register values to signal handler, restores modified values on return
TCB 1
$c1
...
$pcc
Thread stack
Stack frames
Signal frame including saved capabilities
Signal handler frame
Architectural capability propagation
Registers
$c1
...
$pcc
Registers
$c1
...
$pcc
Abstract capabilities preserved for registers across signals as architectural capabilities
Signal delivery

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

the boot code. In CHERI-MIPS, DDC and PPC contain identical, omnipotent capabilities at reset time. An architecture with fewer compatibility constraints could take a more aggressive approach, e.g. making it impossible to encode a capability with both write and execute permissions.

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

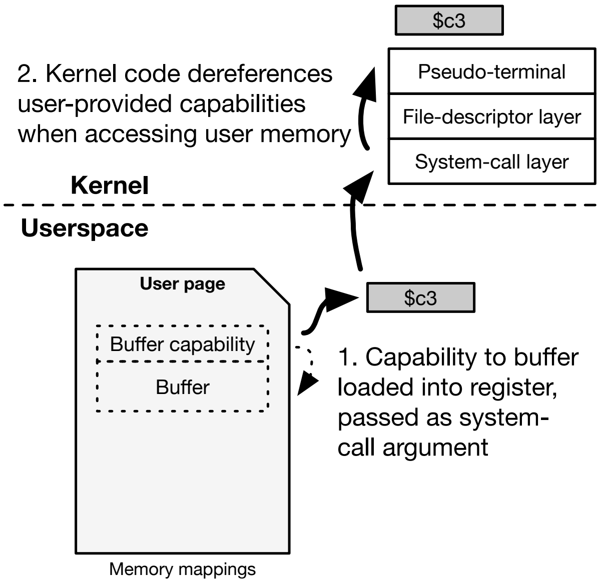
$c3
Pseudo-terminal
File-descriptor layer
System-call layer
2. Kernel code dereferences user-provided capabilities when accessing user memory
Kernel
Userspace
User page
Buffer capability
Buffer
Memory mappings
$c3
1. Capability to buffer loaded into register, passed as system-call argument

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

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.

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. Every MIPS load/store is indirected and bounds-checked using DDC; therefore, installing a NULL DDC will result in a trap being raised whenever such a load/store is used.

Shared memory Shared memory is a key form of IPC on POSIX systems. While doing so safely can be tricky, some applications do share pointers between processes after careful arrangement to ensure that the shared pointers are valid in both address spaces (either due to a parent-child relationship via `fork` or due to mapping the same contents in the same place) or used only in one process. Without a flow control mechanism of some sort, it would be easy for colluding processing to violate the integrity of the abstract capability; thus, we currently disable storing or loading capabilites in shared memory.

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. Specifically, while principals may have memory mapped at the same virtual address, they do not share access to physical pages (with a few explicit exceptions involving the signal trampoline and programmer controlled shaped memory).

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 [52, 10] 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 [52] with limited POSIX compatibility to a new process Application Binary Interface

(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 inline 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.[3] 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" MIPS userspace applications that adhere to the SysV ABI [47], 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 (as does the fact that the kernel must have the ability to create capabilities for new processes). 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.

4.1 Implementation tradeoffs

In any large-scale systems project, implementation tradeoffs are inevitable. With CheriABI we faced a number of choices, including:

- Method of handling system call capability arguments.
- Kernel capability programming model (inline assembly, hybrid C, or pure-capability C).
- Making CheriABI the default ABI vs a compatibility ABI.

These options intertwine with each other. We discuss these options and the evolution of our implementation as our thinking evolved.

A spectrum of options exists for implementing system calls that take capability arguments. At one end, requiring the least work by the kernel developers, language annotations or alternative system-call stubs can be used to call into the unmodified system-call vector with pointer arguments being integers. At the other end, a pure-capability kernel would handle capability system-call arguments directly. In the middle, capability pointers can be converted to integer virtual addresses and passed to otherwise unmodified kernel functions, with or without validation of their range and permissions. Across this spectrum, more work yields more precise enforcement of abstract capability properties. Table 4.1 summarizes some interesting points on the spectrum.

When we started work on CheriABI, the kernel was a legacy (MIPS) C program with some assembly stubs to manipulate capabilities. Before we started work, we rejected a declaration

[3] The primary exception is in the code the copies argument and environment strings to the stack during `execve`. This is correctable with some refactoring.

	Kernel implements user memory model	Kernel acts via	Issues
Userspace wrappers	none	Virtual Address	Wrappers are subject to races
Pointer translation	none	Virtual Address	Capabilities bypassed
Capability validation	tag, length, perms	Virtual address and Capability	Annotations must be correct & Confused deputy issues
Capabilities in kernel	all	Capability	Non-capability ABIs require kernel changes

Table 1: Spectrum of system-call implementation options

annotation model, due to the principle that declarations should not be modified if compatibility is to be maintained with existing – often poor – programming practices, and the fact that it would not avoid the need to translate structures containing pointers – and thus would still require kernel modifications or large userspace stubs duplication kernel code. We similarly rejected the system-call stub model we had used in CHERI JNI [9], because that model can enforce capability restrictions only if the caller is inside a sandbox and unable to make system calls directly. The stub and proxy model can work, but it requires expensive copying semantics to prevent concurrency vulnerabilities [50]. Due to a desire to avoid widespread modifications to the kernel, our first system call implementation used a verification based approach where generated code checked that the capabilities passed to system calls had appropriate bounds and permissions. This approach had serious drawbacks. The most common issue was that capability arguments without associated lengths (usually file paths) could not be checked at system-call entry, because that would introduce a time-of-check to time-of-use vulnerability. Avoiding this by copying at entry would have been enormously disruptive, so we initially skipped the checks. Programming with assembly macros was also awkward. After a rejection forced us to consider the work in new light, we took advantage of the fact that C compiler support for hybrid code was significantly advanced from our start in 2015, and switched away from the validation model to the model described in this paper, where capabilities are carried down the call stack and are first class citizens in the kernel. In doing so, we converted the kernel to a hybrid program, eliminating the use of inline assembly macros in favor of annotated capabilities and compiler provided `__builtin_*` functions.

It's worth noting that the approach of checking and translating at system call entry had one major advantage. It took a few months of less than full-time effort to get basic pure-capability binaries up and running while touching few files. This was aided by the fact that FreeBSD generates many aspects of system-call tables using a script, so additional code generation could be slotted in with relative ease.

By contrast, making a hybrid kernel was fairly invasive, and each system call needed to be handled manually. Additionally, all code that handled user pointers needed to be modified to accept capabilities.[4] Further, code that handles objects containing capabilities needed to be modified to use annotated versions of the objects. Because we wanted to keep legacy binaries working, this required us to perform transformations to unify the type of storage used between legacy and CheriABI. In practice, this meant converting legacy virtual-address-based pointers into capabilities. One complexity of these conversions is the practice of passing sentinels in place of pointers. For example, the `sigaction` system call takes an action that is `SIG_DFL` , `SIG_IGN`, or a signal handler function pointer. We do not want to turn `SIG_IGN` (defined as `((__sighandler_t *)1))` into a valid pointer to the virtual address `1` in the thread's DDC. Instead we want it to be an untagged value of `1` (e.g. `(char *)NULL + 1`). As a pragmatic compromise, we observed that sentinel values are generally in the kernel address space (often represented as negative numbers) or within the first 4KiB of address space – which is unmapped in all sensible ABIs to prevent NULL-pointer dereference bugs – and thus create any such capabilities as offsets from the `NULL` capability rather than from the thread's DDC. This conversion is performed by explicit macros in the kernel.

This leads into the third tradeoff: making CheriABI the default ABI or a compatibility ABI. In our initial model, it made a great deal of sense to make CheriABI a compatibility layer. Most capability interaction occurred at the system call boundary so it was straightforward to start from the 32-bit compatibility layer and convert it into a CheriABI layer. This allowed the system to keep working with legacy binaries allowing us to debug from within the running system rather than having to make `init` work before any other program. As we transitioned to a hybrid kernel this became awkward. Where legacy object types were shared between kernel and userspace (e.g., `struct iovec`, `struct sigevent`, and `union sigval`), we had to add a new type that could hold capabilities and modify both legacy and CheriABI support code to use the new types. An alternative approach we are currently exploring is to implement a 64-bit compatibility ABI and transitioning CheriABI to the default ABI. We are doing is this work in the context of a port from a hybrid kernel to a pure-capability kernel.

4.1.1 Considerations for memcpy

The `memcpy` function must preserve capabilities in C objects that are copied. Since `memcpy` does not know what is in the bytes being copied, it must conservatively preserve capabilities, at a minimum when both source and destination have capability alignment. Due to the possible use of `memcpy` for partial structure copying (and for better efficiency on large copies), we have chosen to implement copying of capabilities for any capability aligned subset of the source and destination buffers. Arguably, this leads to over-preservation in some cases, but the C language aliasing rules make it hard to do better. For example, we could avoid copying capabilities for arrays whose elements are less then the size of a capability, but must still preserve them for any array of byte alignment since such an array can be promoted to any alignment via a cast.

[4] We found cases that required modification through a number of mechanisms, primarily adding annotations to interfaces and searching for consumers, falling backed to letting the compiler help us propagate annotations down to the point of use. The `__user` annotations used in Linux might provide some value here, but they annotate the storage (e.g. `char __user * buf`) rather than the pointer (e.g. `char * __capability buf`) so they are not directly usable.

4.2 Limited precision of 128-bit capabilities

With compressed (128-bit) capabilities, placing bounds on capabilities poses some complications. Large, but oddly sized allocations must be rounded up so that returned capabilities have a base and length that are precisely *representable*. In our original implementation, we used a compressed capability format where the base and length could differ in at most 20 consecutive bits. Given that most `malloc` implementations round up any allocation over a page in size (4KiB here), this means that any allocation under 4GiB ($2^{(20+12)}$) will be bounded to the underlying allocation. Larger allocations at page granularity are certainly possible, but we did not address the issue. Post publication we have switched to a new capability format where capabilities covering more than 2^{13} bytes have only 10 bits of precision. This will require significantly more rounding up. Some allocators make this easy (e.g. `malloc` returning more than requested is normal, but stack allocations may require run-time stack alignment).

4.3 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 runtime 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 [16, 47]. CheriABI processes have a similar set of arrays, the main difference being that all pointers are bounded capabilities. We have currently left the contents of arrays alone including carrying some unnecessary baggage such as stack canary initialization values (not needed with strong, compiler-enforced stack bounds). We have altered the C run-time to take a pointer to the ELF auxiliary argument array and extended it to contain pointers to the argument and environment arrays (`argv` and `environ`) as well as argument and environment counts (`argc` and `envc`), rather than using knowledge of the stack layout and walking off the end of the environment array to find the auxiliary argument vector.

In addition to these arrays, the signal trampoline and the `ps_strings` structure sit at the very top of the stack area. On most FreeBSD architectures the signal trampoline is located in a special page that is shared between processes of the same ABI. This was not the case with FreeBSD-MIPS due to limitations in the virtual memory subsystem on TLBs for certain low-end MIPS CPUs and instead the signal trampoline was installed on the the stack. As our evaluation platforms do not suffer from the underlying TLB limitations, we enabled shared-page support in both BERI and Malta (Qemu) as well as CHERI configurations allowing us to keep the executable signal trampoline separate from the stack. On platforms where the TLB can protect against execution, this separation is critical as it prevents the stack from needing execute permissions. MIPS is not such an architecture, but the separation seemed advisable.

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.

4.4 Run-time capability refinement

Capabilities used throughout the lifetime of the program are further bounded by the compiler and the runtime linker (or C startup code for statically linked binaries).

Refining stack capabilities Compiler-generated code sets bounds on references to variables on the stack. These prevent classic stack-based buffer overflows. Dynamic allocation (e.g., `malloc` is discussed later in this section).

Refining heap capabilities All capabilities for objects on the heap are created by `malloc()`. See section 4.10 for the modifications that need to be made to `malloc` so that each heap object is tightly bounded and has bounds that do not overlap with other objects. If an application uses a custom allocator, or further subdivides allocations performed by `malloc()`, it will also need to be modified in the same way. It is worth noting that some sub-allocators use patterns that result in the storage being unusable for capabilities. For example, the SQLite allocator returns a pointer to a region aligned to a `size_t` due to allocation size metadata being stored at the beginning of the block. Since `size_t` is 8-bytes and capabilities require 16-byte alignment, capabilities can not be stored.

Refining global capabilities Capabilities for global variables (such **`const char`**`*` string constants, etc.) are set-up at program startup by the runtime linker.[5] This ensures that a pointer to the string *`"Hello,␣World!"`* is represented by a capability of length 14 (spanning the string data and the terminating NULL character). Section 4.6 explains how this can be done in more detail.

4.5 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, but this is a marked improvement over the original, which provided no isolation of TLS segments.

The legacy MIPS implementation is a TLS Variant I [20] implementation where the read-hardware-register instruction is used to retrieve a virtual address pointing to the current thread's TLS object from a reserved hardware register. On systems that do not support the instruction, the trap handler emulates it. We extended CHERI-MIPS to include a capability read-hardware-register instruction and return a capability to the TLS object from a new reserved hardware capability register, rather than having to maintain access to all TLS objects via DDC.

The legacy MIPS implementation adds a bias of `0x7000` to this virtual address to allow for more efficient use of the 16-bit signed immediate range of its load instructions. To ensure the TLS object capability remains representable, we removed this bias for CheriABI, though a smaller bias could be re-introduced based on the capability format.

Many of the optimizations used by models other than General Dynamic rely on adding a (possibly computed) offset to a pointer, with varying ways in which the offset is computed and

[5] In statically linked binaries we still need to initialize these capabilities and therefore the subset of the runtime linker that performs this initialization is embedded into each static binary as part of the C startup code

initial pointer obtained. These would no longer be possible with tight bounds. Local Dynamic would become identical to General Dynamic, but as with the existing Initial Exec and Local Exec models, TLS accesses from executables could still be optimized to some extent by inlining the accesses to the TLS object.

One possible design for tight bounds would be to modify the Dynamic Thread Vector (*DTV*) such that each slot is a capability for a specific symbol, like a thread-local GOT. However, in the presence of `dlclose` this could lead to significant fragmentation of the DTV. An alternative scheme would be to keep one slot per module in the DTV, with each module's capability then indirecting through another vector of that module's symbols. This would avoid fragmentation, but at the cost of an extra level of indirection per TLS lookup. In both cases there would no longer be a static TLS block after the Thread Control Block.

4.6 Dynamic linking

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 [46, 21]). We extended the dynamic linker (*RTLD*) to initialize external symbol references using new dynamic relocations that initialize and bound the capability. We use a new ELF relocation `R_MIPS_CHERI_CAPABILITY` for external symbol references; for local (non-preemptible) symbols, we use a special section (*__cap_relocs*) that contains information about the capabilities to be initialized. Although using two different mechanisms for the same goal may not seem ideal, there is one advantage. Statically linked programs contain only local symbols, allowing us to share the code to process the *__cap_relocs* section. While some operating systems support `dlopen()` in static binaries (and therefore embed all the ELF relocation processing code), this is not the case on FreeBSD. Therefore, we used a different mechanism for capability initialization in static binaries. When we added support for dynamic linking, we chose to reuse the existing code for local symbols and only treat references that require a symbol lookup differently.

In the MIPS n64 ABI references to global variables and functions are made by loading the virtual addresses of the symbol from a table in the binary, called the GOT (Global Offset Table). For the pure-capability ABI we changed this table to contain tightly bounded capabilities instead. This ensures that taking the address of a global variable in C (`&my_global`) will give a capability spanning only the object and will not grant access to adjacent variables. In order to add bounds for global variables, we parse the ELF symbol table and use the `st_size` field in the `Elf_Sym` structure as the size of global variables. The `st_size` value is emitted by the compiler and will correspond to the size of the C/C++ declaration[6] and for functions it is the size of all instructions in that function. However, we only use exact bounds for data symbols and bound function symbols to the containing shared object's code segment.[7] While these bounds are not minimal, this preserves the ability of code to use branches in place of jumps

[6] We discovered that some C programs loop over ELF sections using the linker-generated `__start<name >` and `__stop_<name>` symbols. This initially caused bounds violations at run-time since the `st_size` value that the static linker emits was zero and therefore resulted in a zero-length capability. To fix this problem, we modified the static linker to use the section size for linker-generated symbols.

[7] We have an experimental compiler and linker mode that will bound code pointers to only the current function, but do not use this by default.

between functions. The wide bounds also facilitate the existing practice of referencing global variables using program-counter-relative addressing. The same bounds logic is also used when looking up a symbol at run-time with the `dlsym()` API.

4.7 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 the address space of a CheriABI process, non-capability versions of `copyout` and `copyin` return errors, ensuring that all access to process memory is via explicit capabilities. Some of the tradeoffs in converting to explicit capabilities are discussed above.

A few system calls take pointers and, rather than dereferencing them, 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. Some of the tradeoffs involved are discussed above.

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. Our capability-copying versions `copyincap`, `copyoutcap`, etc are the same as the ordinary versions except that they preserve capabilities when copying between appropriately sized and aligned regions. 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. FreeBSD's `ioctl` (as opposed to Linux's) centralizes the copying in and out of data passed to `ioctl` calls. This has the upside that there are fewer opportunities for error, but the downside that we must always use `copyincap` and `copyoutcap` because we can not tell from the 32-bit `ioctl` command if a given object might contain capabilities. It may be possible to add a bit indicating the presence of pointers in future `ioctl` commands, but compatibility would present challenges. Handling `ioctl` commands which contain pointers requires either that the kernel use the pure-capability object natively or that each command be audited by hand. Additionally, if the size encoded in the ioctl command is `0` then a pointer to the raw system call argument is passed down and is some cases is use as an argument to `copyin` or `copyout`. The `sysctl` interface is somewhat easier to handle in this regard. Copying to or from userspace occurs in the `sysctl` handler; thus, the types being copied are known (most importantly, it is known if the data being copied contains pointers). Unfortunately, the code to do so is hard to find; all copy sites need to be audited, because the object names do not appear in declaration macros. For one use case where exporting a capability made sense, we have extended the `sysctl` interface to preserve capabilities if the `sysctl` definition is annotated appropriately. Some management interfaces export kernel pointers. Where we have encountered them, we have altered them to expose virtual addresses rather than kernel capabilities. More work is likely required in this area.

4.8 Virtual-address management APIs

POSIX programs add or extend memory mappings through three system calls: `sbrk`, `mmap`, and `shmat`. The `mmap` and `shmat` system calls allocate (or alter the mappings of) regions of memory, returning a reference to the new allocation. The `sbrk` API adjusts the heap size; adding or removing mappings beyond the `bss` segment of the program binary.

4.8.1 `mmap` and `shmat`

While the interfaces differ, `mmap` and `shmat` system calls share many considerations with regard to maintaining and preserving the integrity of the abstract capability so we discuss them together. In CheriABI we have altered both to return capabilities that are bounded to the requested allocation length, with permissions derived from the requested page permissions.

The first question that arises when implementing `mmap` or `shmat` is, where does the return capability come from? In our implementation, we create a capability covering the potential process address space during `execve` and attach it to the thread structure[8] (propagated from the parent thread when a new thread is created.) In the common case of allocation of previously unallocated space, we use this per-thread capability to derive a capability to return to the user. One significant exception is that 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.[9] These restrictions are necessary to ensure that one thread can not change the mapping out from under another thread (gaining a valid capability to that memory in the process). Without this, the abstract capability is not sound.

In the common case, the returned capability has permissions derived from **OR** of zero or more of `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC` values passed to the `prot` argument of `mmap` **AND**ed with the permissions of the per-thread capability. These permission include – as appropriate – load, store, and execute permissions as well as a user permission, `vmmap`, which we use to control the ability to make future alterations to a mapping via a capability to it. Where the user passes a capability we follow the same policy except that we use the passed capability. An astute reader might notice that this would break the somewhat common pattern (e.g., when establishing a VM address space) of reserving a region of memory with `PROT_NONE` and then mapping things into in portions of it (e.g., a kernel binary.) We resolve this problem by extending the `mmap` API with bits indicating the maximum permission the mapping could have and returning a capability with those permissions. This requires a simple change to users of a pattern that can usually be spotted by searching for `PROT_NONE` use. Another pattern where

[8] We attach the capability to the thread structure (vs. the process structure) on the theory that application developers might choose to limit which threads can allocate from certain regions of memory. To support this use case, we provide `sysarch` commands to query and reduce the bounds and permissions of the per-thread capability. We have not yet explored such models.

[9] In CheriBSD, we do this by implying the FreeBSD-specific `MAP_EXCL` flag.

JIT engines map a region writable and then convert it to executable will also require alteration. We believe these small changes to low-level code, which colors well outside the lines of ISO C, are well within the range of reasonable changes.[10]

Compressed capabilities pose precision challenges for mmap and shmat, similar to those in malloc, discussed below. However, there are some differences. On the simplifying side, allocations are always page sized, so only fairly large allocations are impacted. On the complicating side, the mmap / munmap API works differently from the malloc / free API, in that while passing values other then those returned from malloc to free is disallowed, munmap can unmap any previously allocated page or range of pages, while unmaps always take an explicit length. This means that free can (and must) look up the size of the allocation in internal data structures, but munmap must rely on the caller's notation of the length. Because the mmap interface provides no way to return the length actually mapped, if an allocation is rounded up for representability, it will not be freed when the programmer calls munmap. We currently reject unrepresentable allocation requests, requiring the programmer to round their requests up. Alternatives are available, such as adding special padding to the end that can be unmapped if the entire requested mapping is unmapped. We have not yet explored such an implementation. In this regard, the interface of shmat is better designed. The shmat system call takes a shared memory id allocated by shmget and shmdt looks up the length in the kernel shared memory object to unmap pages.

In addition to the need to round up sizes, compressed capabilities require rounded up alignments for representability. In general, the kernel can just do the right thing, checking that MAP_FIXED requests are properly aligned and requesting sufficiently aligned mappings from the VM system in other cases. Because FreeBSD has an existing mmap extension for requesting both explicit minimum alignments and architecture independent super-page alignments, we used this in our implementation and extended it with a flag for the user to request representable alignment from legacy binaries (e.g. to support mapping memory sandbox address spaces).

4.8.2 sbrk

The sbrk interface predates modern APIs such as mmap – and is obsolete. Implementations are possible with some limitations, but few programs require it – so we do not support it in our prototype.

The interface of sbrk poses a number of challenges. As a reminder, its signature is **void** * sbrk(**int** incr); and it takes a signed integer increment and adjusts the set of mapped pages after (and immediately adjacent to) the bss segment up or down as required. It returns a pointer to the so-called *break*. Based on our experience removing it from the FreeBSD RISC-V and AArch64 ports, the vast majority of uses are not even for memory allocations. Instead, consumers call sbrk(0) in an attempt to track heap usage. On systems where malloc is built atop sbrk, this works, but with modern allocators based on mmap the result never changes and is useless. A few systems (including GNU binutils) use it to implement internal allocators. In almost all cases, we have found that these systems supported a mmap based allocator already

[10] The FreeBSD project is in the process of adopting the PROT_MAX() implementation to complement and extend W^X on even non-CHERI systems. By providing a way to express the maximum permission a page can have, this change allows most pages to be mapped so that they can never become executable.

(usually due to adding support for Windows at some point in the past), but some prefer the sbrk implementation.[11] Finally, and infamously, the Emacs editor's unexec process depends on sbrk as an allocator and upon assorted related baggage, including the end symbol, to pre-compile Lisp code and dump it to disk as a new executable. Needless to say, if that image is full of CHERI capabilities, it will not be possible to run without some additional infrastructure to re-initialize those values. Fortunately, modern versions of Emacs have provided alternatives to the un-portable and unsafe unexec due to ongoing maintenance issues in both Emacs and glibc.

For consumers wanting to measure memory use, a version that returns NULL will be as useful and accurate as the current implementation in practice. For allocator use of sbrk, a simple version with a static pool should be straight forward. It seems likely that consumers will assume that they can access any part of the heap from the pointer returned by sbrk, so an implementation would likely want to return a capability to the whole pseudo-heap. Consumers who need to access the whole binary as well as the heap are unlikely to see any benefit from CheriABI, and should continue targeting the legacy ABI.

4.9 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 rather than a pointer to the top of the stack. The variety of ways signal handlers propagate values and function pointers poses compatibility challenges for the legacy ABI. In particular, it is difficult to know what to extract from a **union** sigval when converting it to the kernel type, because only the caller knows which part of the union was used. The use of sentinel values also adds complications and requires careful use of casts to ensure the same values are generated between pure-capability and hybrid code.

The main extension to the signal ABI was to extend ucontext_t with a pointer to capability registers. As FreeBSD has added new architectures (recently AArch64 and RISC-V), we have ensured that ucontext_t is appropriately padded to allow extension.

4.10 Dynamic allocations

Dynamic allocations use a lightly modified version of JEMalloc, FreeBSD's standard malloc. 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. In addition to bounding the allocations, we strip the vmmap tag from returned pointers to prevent the protections or mappings of the underlying memory from being changed by the process. In the current version of JEMalloc, metadata is allocated separately from storage and located in internal data structures, using the allocation's virtual address as a key. This model makes it easy to retrieve metadata and to access the original capability returned by mmap, e.g., when realloc extends an allocation in place. In versions of JEMalloc prior to 5.0, metadata was

[11] GNU binutils has support for mmap, but prefers sbrk extremely aggressively. It will happily declare a prototype for sbrk if one is not defined and use it over mmap so long as the symbol resolves.

stored near to the allocation, and found by manipulating the pointer. This required a way to obtain a pointer to the memory holding the metadata, without accessing the metadata. We originally stored a copy of a capability to the whole address space for this purpose, but safer schemes are certainly possible where allocators follow this pattern. This issue, coupled with the long history of exploitation of malloc metadata overwrites [5, 23], suggests that the use of local metadata storage in mallocs should be relegated to the past.

In our initial JEMalloc implementation, we made setting bounds on allocations optional and controllable at run-time as a debugging aid. In practice, we never used this feature. Besides, it came with a small, but measurable performance cost, so we have converted the run-time check to a compile-time option (in the C standard library).

We have similarly altered the TLS allocators in libc and RTLD. These are trivial memory allocators with power-of-two buckets.

4.11 Swapping

The current implementation of capability swapping extends the always-in-core swap metadata structure with an array of tag bits. This conservative approach avoids new allocation on the swap path, at the cost of a significant size increase. A better approach would include dynamically sized side-car storage, which would hold a compressed tag array. Our work on tag caching [26] demonstrates that the potential space savings are significant based on real-world capability distribution.

4.12 `printf` and `scanf`

Furthermore, we found that code parsing `printf`/`scanf` format strings and fetching variadic arguments was often broken[12]. Some cases (e.g. the minimal `printf` used by `libthr` and RTLD) would use `va_arg(ap, unsigned long)` instead of `intptr_t` to read the argument for `%p`. In CheriABI all variadic arguments are passed on the stack. Due to this implementation choice, reading an argument with the wrong size would then result in the remaining arguments being read from the wrong offset from the stack. This would then cause either wrong output for integers or CHERI traps when dereferencing an invalid capability for `%s`. While a register-based variadic calling convention could avoid this kind of bug (at least for the first few variadic arguments), we believe the stack-based variadic calling convention has security benefits. Using the stack instead of registers allows us to tightly bound the variadic argument block, and would therefore cause a bounds violation when the variadic function reads too many arguments instead of reading an uninitialized value from the next argument registers.

Due to the reality that `printf` debugging is inevitable, we extended the `%p` format to accept the # modifier and print a human-readable decoding of capabilities. The C standard would allow us to make this the default, but we deemed that too confusing and likely to cause problems in utilities that do things like printing kernel addresses using `%p`. It is an unfortunate feature of the C standard that the use of `%#p` is undefined rather than implementation defined. It's

[12] The FreeBSD source tree contains at least five implementation of `printf`/`scanf`-like format string parsing code that needed changes. We also discovered that PostgreSQL and NGINX would use their own version of `printf` (which do not handle `%p` correctly for CHERI) when being cross-compiled.

worth noting that we don't support creating tagged capabilities in scanf and make no effort to support the seeming requirement of the standard to allow pointers to round trip through printf and scanf.[13] We deem this feature undesirable, and have not seen code that relies on it.

4.13 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 (heapsort and mergesort) to preserve capabilities when swapping array elements that might be (e.g.) structures containing pointers. We also replaced use of Berkeley DB as an in-memory hash table, with uthash.h, because BDB does not preserve the alignment of stored data, and thus does not preserve the integrity of stored capabilities. Specifically, it assumes that any object being serialized can be stored by unaligned byte-by-byte copies and preserved alignment no larger than 2-bytes. 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.

These changes are discussed in more detail in section 5.3.

4.14 Debugging

Beyond supporting the ptrace system call under the pure-capability ABI, we have extended the operating system's debugging facilities to support capabilities. These extensions include an additional ptrace operation to read the values of capability registers from a target thread as well as a new per-thread core dump note containing the value of each thread's capability registers. However, there is currently no facility for examining tags stored in memory either via ptrace or in core dumps.

We have modified GDB to add limited support for capabilities, including dereferencing capability pointers interactively, annotating bounds and permissions on capability pointers, and unwinding stacks. GDB has limited support for tags on capability registers used in single-stepping, but is otherwise unaware of tags. Since GDB's model of memory access is centered on bare virtual addresses, the user interface does not enforce bounds or permissions, or check the tag when dereferencing a capability pointer. Finally, we have not yet attempted to compile and run GDB under the pure-capability ABI, though we have used a MIPS version of GDB to examine live pure-capability ABI processes via ptrace.

One interesting aspect of debuggers is that they have the ability to write to memory in the target address space. In some cases, developers want to inject pointers. With capabilities as pointers, ptrace will need to be extended to manufacture and inject capabilities. We have envisioned two models for this. In the first model, a capability description to something in the user address space is constructed, and the kernel creates that pointer from the user process capability. In the second, the capability is created by specifying a set of manipulations to perform on a specific capability in the process. The latter would ensure monotonicity (but not intentionality) within the target, while the former more closely models current debugger

[13] Implementations of this are surely possible with limitations, but would almost certainly require unbounded storage for escaped pointers in the standard library.

behavior. We have not yet explored these options. As a result, GDB is currently unable to modify pointers.

4.15 Limitations

Our implementation has a few limitations. We support nearly all system calls, but have excluded `sbrk` as a matter of principle, and have 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. However, 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 has 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.

- To demonstrate the completeness of the CheriABI implementation, we ran the FreeBSD, PostgreSQL, and LLVM libc++ test suites.
- We validated performance assumptions with system-call micro-benchmarks, synthetic micro-benchmarks, and whole-application benchmarks.
- We examined the changes required to support CheriABI across FreeBSD userspace as well as PostgreSQL.
- 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.
- 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 [41]. 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 [42] 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

	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 2: Test suite results

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.

In an effort to reduce unnecessary differences, benchmark software for CheriABI and MIPS are both compiled with the CHERI compiler (based on LLVM 8.0) and use the CheriABI-capable kernel. In particular, this ensures that the more efficent memcpy employing capability registers is used in both cases.

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 conditions). We ran the test suite on a mips64 system and CheriABI. The results are summarized in Table 2. 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 to 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 [22], 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

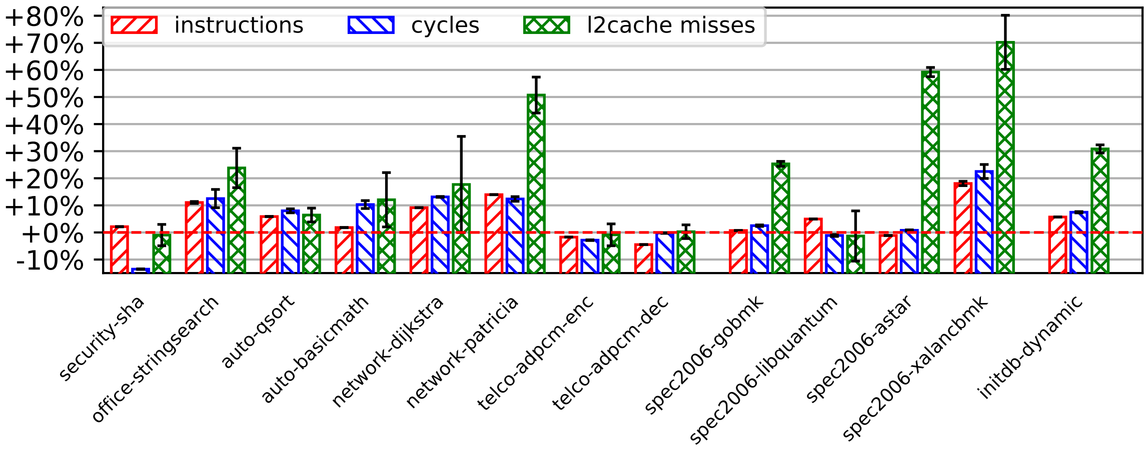

Figure 4: MiBench, SPEC CPU2006, and PostgreSQL benchmark median overheads relative to MIPS baseline. Error bars are interquartile ranges (IQRs).

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%. Since submitting the final ASPLOS 2019 paper we have further improved performance with optimizations such as refined stack bounds (see section 4.4), improved use of the NULL register and other small compiler changes. Therefore, these performance numbers are noticeably better than previously published pure-capability CHERI results and in many cases even require fewer cycles than legacy MIPS code – even though the increased pointer size results in more L2-Cache misses.

In the conference version of this paper, the compiler used a naive approach for setting stack bounds: every use of a stack allocation (represented by an **`alloca`** instruction in the LLVM IR) would instead use the result of a call to the `llvm.cheri.cap.bounds.set()` intrinsic,

	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 3: 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

which would return a correctly bounded capability. However, we discovered that this approach was causing significantly increased stack-frame sizes for CHERI compared to MIPS. This was caused by these bounded capabilities being spilled to the stack and reloaded, instead of being re-derived from the stack capability. Instead, we have since changed the code-generation to allow re-deriving the bounded capability from the stack pointer, which has noticeably reduced the size of the code. Furthermore, the compiler will no longer create a bounded capability for loads and stores to objects at fixed offsets from the current stack frame, and instead will perform the operations relative to the stack pointer.

In `JSC::Lexer::lex()`, a frequently used function in QtWebkit, these two optimizations reduced the CheriABI stack-frame size from 3232 to 672 bytes and the number of instructions from 28320 to 27581, by no longer spilling and reloading of bounded capabilities to the stack.

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 3 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, or attempts to pass pointers over IPC channels. An example of the former can be seen in listing 1. *Integer provenance* (**IP**) relates to the loss of provenance as pointers are cast though integer types other than `uintptr_t` and can be seen in listing 2. *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. A somewhat common incorrect assumption is that `sizeof(void *)` can be used to compute the number of bits of available virtual address space. A more correct assumption for conventional systems would be that `sizeof(size_t)` is directly related to the size of the

```
        if ((nstrings = realloc(we->we_strings, we->we_nbytes)) ==
           NULL) {
                error = WRDE_NOSPACE;
                goto cleanup;
        }
        for (i = 0; i < vofs; i++)
-               if (we->we_wordv[i] != NULL)
-                       we->we_wordv[i] += nstrings - we->we_strings
   ;
+               if (we->we_wordv[i] != NULL) {
+                       we->we_wordv[i] = nstrings +
+                           (we->we_wordv[i] - we->we_strings);
+               }
        we->we_strings = nstrings;
```

Listing 1: Typical example of a pointer provenance bug. Here array of strings is extended with `realloc` and the pointers are updated. The old code updated the old pointers rather than deriving new pointers. This example is from `lib/libc/gen/wordexp.c`.

```
  */
-#define __UNCONST(a)    ((void *)(unsigned long)(const void *)(a))
+#if !__has_feature(capabilities)
+#define __UNCONST(a)    ((void *)(__uintptr_t)(const void *)(a))
+#else
+#define __UNCONST(a)    ((void *)(__uintcap_t)(const void *
   __capability)(a))
+#endif
```

Listing 2: Example of an integer provenance change, casting through `intptr_t` or `intcap_t` as appropriate when removing **`const`** in a macro. This change was made in `lib/libnetbsd/sys/cdefs.h`.

address space.

Pointer as integer (**I**) covers storing sentinel integers or integer data 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) as seen in listing 3. 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 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.

```
 label_done:
         if (metadata_thp_madvise()) {
                 /* Set NOHUGEPAGE after unmap to avoid kernel defrag
                    . */
-                assert(((uintptr_t)addr & HUGEPAGE_MASK) == 0 &&
+                assert(__builtin_is_aligned(addr, HUGEPAGE) &&
                     (size & HUGEPAGE_MASK) == 0);
                 pages_nohuge(addr, size);
         }
```

Listing 3: An example of pointer alignment from JEmalloc

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. The variadic calling convention has allowed us to find numerous bugs, but it does pose compatibility issues. We have added warnings defaulting to errors when calling functions without declared arguments,[14] or converting between variadic and non-variadic function pointers.

The *unsupported* (**U**) label covers an array of things CHERI or CheriABI does not 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 [27] and used by Hardbound [18]. 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 4).

- **min** has the smallest possible memory safety violation (typically off by one byte);
- **med** has an off-by-8-bytes error; and
- **large** has an off-by-4096-bytes error.

[14] 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 4: BOdiagsuite tests with detected errors (among 291 total tests)

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 [41]. The results are summarized in Table 4. 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 [10].

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 setup and exchange of a small file. `openssl` is a small representative application that exercises the majority of the changes we introduced with 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 due 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 capabil-

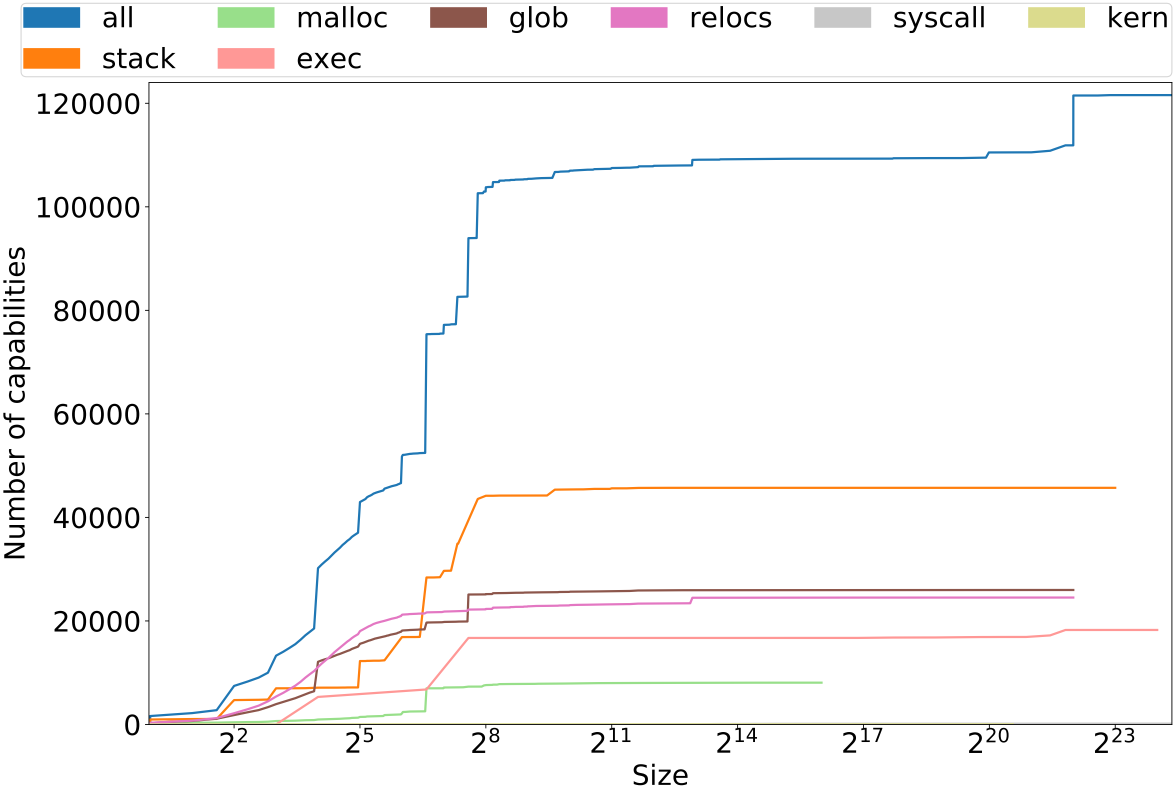

Figure 5: Cumulative sum of the number of capabilities against size of bounds, for different sources of capabilities visible in userspace during a run of openssl s_server involving startup, authentication and a file exchange. (Note that the *kern* and *syscall* lines are present, but virtually indistinguishable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

ities 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 remaining 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 including `container_of`. Old implementations of flexible array members – in which the last member of a structure is declared as having length of `[1]` rather than having an unspecified length (`[]`) – may present another complication. Likewise, the use of structs as headers without a flexible array member may pose some compatibility issues. That being said, most references to struct members could be bounded safely. Exceptions require further 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.

We have not yet examined these trade-offs in detail.

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.

Quantify the benefits of memory safety While it is self-evident that narrower bounds on pointers reduce the impact of bugs and complicate attempts to leverage flow control, we do not have a way to quantify how much harder reduced bounds make things for the attacker. Similarly, we do not have an easy way to evaluate the value of a given mitigation. For example, we could easily alter the stack capability on function entry to exclude directly access to the stack before the current function, requiring a pointer to be restored from the stack before accessing the caller's stack, but this would have non-zero cost and not fundamentally alter the set of reachable memory (the whole stack would still be accessible via a chain of loads). How can we decide which such changes are worthwhile?

Memory APIs The mmap API family has significant limitations when implementing least-privilege policies. Combining capabilities, WˆX, and JIT compilation in an optimal manner will require new interfaces. We believe that a model separating address space reservation from mapping and retrieving pointers to reservations could help resolve the mismatches between the mmap API and modern programming and mitigation models.

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. Likewise, the ability to detect patterns common to custom allocators (which would benefit from explicit CHERI support) would be useful.

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. We hypothesize that while the set of change will be significant, they will be more concentrated and easier to maintain then the current hybrid model.

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 including models of components such as linkers.

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 [30] shows the need for strong memory protections beyond the CPU.

7 Related work

CheriABI imposes an abstract capability on the popular POSIX programming environment [32, 33, 43, 2]. This work builds upon the CHERI C environment [10], 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 [44], there is a long history of memory-protection research adding bounds checking to C programs. The CHERI [55], HardBound [18], and SoftBound [35] research systems add fine-grained memory protection to C. Intel's Memory Protection Extensions [24] 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 [41] 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 [36, 12], Deputy [11], and Checked C [45] 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 [25] safer (but incompatible) dialect of C. CHERI draws from ideas proposed in the M-Machine [8], particularly object-granularity capabilities protected by in-memory tags (and earlier by PSOS [37]). Unlike M-Machine, CHERI has a stated design goal of source-code compatibility (so existing software benefits when recompiled) and binary compatibility (so existing compiled code continues to work), 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 [53] (applying Mondriaan to Linux), Safe TinyOS [13] (applying Deputy to TinyOS), and KernelAddressSanitizer[15].

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 of 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[7]. They include Control-Flow Integrity (CFI) [3], Code-Pointer Integrity (CPI) [28], and Cryptographic Control-Flow Integrity (CCFI) [31] 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. [39], which utilizes configurable tagged-memory policies similar to Pump [19]. 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, Steven Murdoch, 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 website [1].

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] Anonymous. Once upon a free()... *Phrack Magazine*, 11, 2001.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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). In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, New York, NY, USA, 2019. ACM.

[18] 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.

[19] 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.

[20] U. Drepper. ELF handling for thread-local storage. December 2005.

[21] 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.

[22] 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.

[23] huku. Exploiting dlmalloc frees in 2009. *Phrack Magazine*, 13, 2009.

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

[25] 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.

[26] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. M. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. Son, and A. T. Markettos. Efficient tagged memory. In *Proceedings of the 2017 IEEE 35th International Conference on Computer Design (ICCD)*, November 2017.

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

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

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

[34] 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.

[35] 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.

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

[37] 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.

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

[39] 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.

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

[41] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: 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.

[42] 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.

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

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

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

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

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

[48] 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.

[49] 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.

[50] R. N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *WOOT Workshop*, Gaithersburg, Maryland, 2007. USENIX Security. http://www.watson.org/ robert/2007woot/20070806-woot-concurrency.pdf.

[51] 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. Rugg, 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.

[52] 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.

[53] 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.

[54] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. Norton, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, A. T. Markettos, M. Roe, P. G. Neumann, R. N. M. Watson, and S. W. Moore. CHERI concentrate: Practical compressed capabilities. *IEEE Transactions on Computers*. Forthcoming.

[55] 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.