Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety

Nathaniel Wesley Filardo
Microsoft
Canada

Brett F. Gutstein
University of Cambridge
UK

Jonathan Woodruff
University of Cambridge
UK

Jessica Clarke
University of Cambridge
UK

Peter Rugg
University of Cambridge
UK

Brooks Davis
SRI International
USA

Mark Johnston
University of Cambridge
UK

Robert Norton
Microsoft
UK

David Chisnall
SCI Semiconductor
UK

Simon W. Moore
University of Cambridge
UK

Peter G. Neumann
SRI International
USA

Robert N. M. Watson
University of Cambridge
UK

Abstract

Violations of temporal memory safety ("use after free", "UAF") continue to pose a significant threat to software security. The CHERI capability architecture has shown promise as a technology for C and C++ language reference integrity and spatial memory safety. Building atop CHERI, prior works – CHERIvoke and Cornucopia – have explored adding heap *temporal* safety. The most pressing limitation of Cornucopia was its impractical "stop-the-world" pause times.

We present Cornucopia Reloaded, a re-designed drop-in replacement implementation of CHERI temporal safety, using a novel architectural feature – a per-page capability load barrier, added in Arm's Morello prototype CPU and CHERI-RISC-V – to nearly eliminate application pauses. We analyze the performance of Reloaded as well as Cornucopia and CHERIvoke on Morello, using the CHERI-compatible SPEC CPU2006 INT workloads to assess its impact on batch workloads and using pgbench and gRPC QPS as surrogate interactive workloads. Under Reloaded, applications no longer experience significant revocation-induced stop-the-world periods, without additional wall- or CPU-time cost over Cornucopia and with median 87% of Cornucopia's DRAM traffic overheads across SPEC CPU2006 and < 50% for pgbench.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0385-0/24/04
https://doi.org/10.1145/3620665.3640416

CCS Concepts: • **Software and its engineering** → **Software safety**; • **Security and privacy** → **Operating systems security**; • **Hardware** → *Emerging architectures.*

Keywords: capability revocation, CHERI, temporal safety, use after free

ACM Reference Format:
Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Jessica Clarke, Peter Rugg, Brooks Davis, Mark Johnston, Robert Norton, David Chisnall, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2024. Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3620665.3640416

1 Introduction

Many programming languages offer an object-centric model of memory. New objects, initially unrelated to existing objects, are allocated on demand, used, and then released (implicitly and/or explicitly depending on the language). Lowering the language's model to the underlying architecture, most often built around a coherent, integer-indexed array of memory words, is generally not fully-abstract; it becomes possible to ① confuse integers, object references, and memory indices that do not point to valid objects (such as those used internally by the memory allocator), risking *reference integrity* violations; ② access adjacent objects, reaching beyond the bounds of a referenced object, violating *spatial safety*; and/or ③ access an object after its life ended ("use-after-free", "UAF") or after the underlying memory has been repurposed ("use-after-reallocation", "UAR"), violating *temporal safety*. These affordances beyond the programmer's intent continue to pose a significant threat to software security [17, 35], and

a wide variety of languages, compilation approaches, and runtime strategies have emerged in response.

The CHERI [53] capability architecture, summarized in §2.1, has shown promise as a technology for C and C++ language reference integrity and spatial safety, with overheads acceptable for general-purpose computing [51]. Strategies for C/C++ heap temporal safety atop CHERI have emerged, most notably CHERIvoke [58] and its successor Cornucopia [23], suggesting viability of a *sweeping revocation* approach (§2.2).

Revocation, in this sense, ensures that use-after-free will never alias a different (later) allocation. It does so by *delaying reuse* of freed address space (said to be in *quarantine*; see §2.2.2) until all references to freed objects have been *deleted* from memory. Revocation is similar to garbage collection (GC), in that it makes address space available for subsequent reuse; unlike GC, revocation reclaims only explicitly released memory and does not discover unreferenced objects. That is, while garbage collection uses a graph walk to find and subsequently recycle address space to which no (mutator) references exist, revocation knows, ahead of time, how much address space will be reclaimed and does not need to chase pointers, but merely to test all references in the system. Still, revocation can, as shown by Cornucopia and now this work, use mechanisms inspired by performant collectors (§2.3).

While Cornucopia's aggregate overheads may be tolerable for high-security workloads, its sizable application pause times ("stop-the-world" phases) still likely limit its use to *non-interactive* high-security workloads. Targeting this shortcoming, we exploited recent extensions to the CHERI architecture and built Cornucopia Reloaded, or just Reloaded, a drop-in replacement for Cornucopia's in-kernel component. The key architectural feature is a per-page capability load barrier (§3.2), supporting a fast global enablement (§4.1). Reloaded uses this, in tandem with an improved form of Cornucopia's *capability dirty* tracking (§4.2), to replace Cornucopia's operational core. We analyze its performance as well as (re-implementations of) Cornucopia and CHERIvoke on Morello [10], which has, along with CHERI-RISC-V and the Toooba prototype [44], been extended with this load barrier. We use the CHERI-compatible SPEC CPU2006 INT workloads to assess its impact on batch workloads (§5.1) and use `pgbench` as a representative interactive workload (§5.2).

Our evaluation (§5.4) found that Cornucopia Reloaded achieves its goals: Applications no longer experience significant revocation-induced global stalls; the system incurs no additional wall- or CPU-time cost, relative to Cornucopia; and this new approach uses less bus traffic for revocation. If the modest additional hardware support can be provided, Reloaded should be considered a strict improvement for UAR mitigation. We close with related (§6) and future (§7) work.

2 Background

2.1 CHERI

CHERI [53] is a modern memory capability architecture. It replaces an instruction set's use of integer addresses as pointers with **capabilities**, a distinguished *architectural type*; dereference operations now require an authorizing capability, and not merely an integer memory address. For our purposes, CHERI has three salient features: ① capabilities carry *bounds* information, limiting the range of addresses to which they authorize access, ② capabilities may be constructed only from a superset capability, and ③ software can perfectly distinguish valid capabilities from non-capability data.

Adopting CHERI has consequences above and below the ISA boundary. Below, machinery is required to associate "tags" with memory words, distinguishing well-formed capabilities from mere bit sequences [30]. Above, when compiling C/C++ to a CHERI architecture, all pointers lower to capabilities [55].[1] Operating system, library, and ABI changes must be made [22]. Of particular relevance to this work, a CHERI-enlightened heap memory allocator (`malloc`) will apply bounds to the pointer returned.[2] Assuming correctness of the allocator, these bounds prevent returned pointers from being used to read or write objects elsewhere in memory.[3]

2.1.1 Morello.

Morello [10] is the umbrella term for an experimental CHERI-augmented ARM architecture and its implementation SoC and board. (Specifically, it modifies the 64-bit ARMv8.2 architecture and the Neoverse N1 SoC and reference platform.) The Morello SoC has four cache-coherent cores, clocks at 2.5 GHz, supports 64 GiB of ECC DDR RAM, and has modern high-speed peripheral busses (PCI-e, CCIX, SATA, USB, etc.). That is, it is a modern desktop or server computer, while other CHERI implementations were and are in-FPGA prototypes, with all the attendant caveats. Our evaluation on Morello thus gives the most realistic assessment of upper bounds for costs of establishing heap temporal safety atop CHERI desktop- or server-class systems.

2.2 Global Subset Capability Revocation

Across a general computing system's operation, it will *repurpose* address space and memory to represent a sequence of distinct logical objects. Temporal safety, in its most general

[1] There is also a "hybrid" C/C++ dialect, wherein only explicitly annotated pointers lower to capabilities. The use of this mode is discouraged: juggling two different kinds of pointers, and understanding their interactions, appears rarely worth the effort. We deal exclusively with the "pure capability" dialect and need not distinguish language-level *pointers* and architectural *capabilities*. Capability revocation (§2.2) *could* be used with hybrid mode code, albeit with much weaker safety properties, as with spatial safety.

[2] Globals and stack allocations may also have bounds applied, especially when pointers to these become visible values in the source program and escape the compiler's analysis. Ensuring safety of these references requires cooperation between the compiler, the linker, and the loader.

[3] Of course, if one piece of software holds capabilities to two adjacent regions, it may write to both, but within bounds of the cited capabilities.

form, is assurance that this reuse is not (meaningfully) visible to the software that the system runs.[4] Some systems ensure temporal safety using so-called *linear* or *affine* references, which cannot be copied; this ensures that no aliases exist when the reference is returned, and so the underlying resource is safe to reuse. Rust uses this technique to constrain its mutable references [45, §10.1.13].

Capability systems generally have considered such temporal safety to be a special case of "the **revocation** problem", allowing software to *retract* delegations of access to resources. Traditionally, the desideratum has been *hierarchical* revocation, selectively pruning entire branches of the delegation tree; this necessitates explicitly tracking capability *provenance* (the list of ancestor capabilities from which a capability is derived).[5] Temporal safety can be built atop hierarchical revocation by having the primordial resource allocator revoke its initial return: all aliases will, by assumption, be pruned by revocation, making reuse safe. However, for performance and compatibility with modern (micro)architectures, CHERI does not explicitly track provenance of its copyable, non-indirected capabilities, and so hierarchical revocation would require additional software artifice.

Fortunately, (heap) temporal safety does not require hierarchical revocation; it suffices for the heap allocator to be able to *globally* revoke all references derived from a particular allocation. Moreover, while CHERI does not *track* provenance, there is still an implicit relation: capabilities with smaller bounds must be descended from one with broader bounds. Since a heap allocator will (internally, inaccessible to clients) retain capabilities that span the entire heap, it can demonstrate its progenitor claim to the heap address space also used for a particular allocation as grounds for revocation of the latter. All that remains is to introduce a revocation mechanism receptive to such claims.

2.2.1 CHERIvoke and Cornucopia.

CHERIvoke [58] is a limit study of **sweeping revocation** algorithms that globally revoke selected CHERI capabilities from a process's address space. Cornucopia [23] is an implementation and extension of CHERIvoke; it encompasses an in-kernel revoker subsystem and enlightened user-space allocators, offering heap temporal memory safety within CheriBSD for CheriABI programs [22]. Beyond CHERIvoke, it introduces *concurrent* and *parallel* execution of application threads and the revoker.

2.2.2 Quarantine and Revocation Bitmaps.

Both CHERIvoke and Cornucopia are based around *batched* revocation, as it is prohibitively expensive to sweep all of a process's memory after every free. Between being passed to `free()` and being available for reuse, a region of address space is held in **quarantine**. Address space lingers in quarantine until the allocator can be certain that no external references thereto exist, that is, until after global revocation. Sizing of quarantine (and decisions of *when* revocation will occur) is a matter of user *policy*. The simplest policies are fixed absolute size or fraction of allocated heap memory, as used below.

A pointer to quarantined memory may be dereferenced for an indefinite amount of time, depending on when revocation runs, the pointer's location in the machine, and the details of the revocation algorithm, but certainly not after a subsequent revocation has completed.[6] That is, use-after-free is possible and will continue to access the old object, as if its lifetime extended until revocation; use-after-reallocation is ruled out.

In general, the goal of an attacker, given a use-after-free primitive, is to leverage the allocator against the application or its runtime. Staying strictly within the UAF regime, before reuse, the allocator has not yet aliased the free object, with either another application object or allocator internal state; in CHERIvoke designs, the allocator is permitted to unmap quarantined memory, but "poisoning" or zeroing of freed memory is deferred until reuse.[7] We thus believe the presence of a UAF/UAR gap is a tolerable security/performance tradeoff, but see §7.3 for one approach to closing the gap.

A revocation pass needs to know which capabilities to revoke. CHERIvoke suggests articulating this set by use of a **revocation bitmap** (sometimes "shadow bitmap"). Each capability-sized naturally aligned region of the address space has a corresponding bit in the bitmap (the same density as CHERI tags).[8] A set bit in the bitmap indicates that capabilities pointing to the corresponding address are to be revoked.[9] In Cornucopia, this bitmap is a kernel-provided anonymous

[4] In practice, we focus more on *access* to reused resources through stale references than on ensuring that reuse is *indistinguishable*. In the present work, for example, we tolerate software being able to extract *addresses* from capabilities and notice that address space has been reused, but we ensure that capabilities to old objects are invalidated before the reuse takes place.

[5] This tracking typically uses indirection, with capabilities pointing to their progenitor with (cached) tree ascent on use. Such indirection can be either defined explicitly, as in CAP's tables [37] or seL4's capability derivation tree [21, §2.4.1], or endogenous, as in trusted "caretaker" objects [42, §2.3].

[6] This is akin to the "from" spaces of incremental, moving garbage collectors, which, aside from the complexity of *forwarding pointers*, remain accessible to the application until collection is completed [32].

[7] There are three motivations for such deferral: ① theoretical: it simplifies the claim that object lifetimes have been extended; ② performance: zeroing cannot be relied upon during reuse and so would need to be redone; and ③ implementation: the underlying allocators have not been aware of quarantine and do, indeed, consider the objects' lifetimes extended.

[8] By contrast, a garbage collector may have a "mark" bit, or a few "color" bits, per *object*, rather than per pointer, in memory. The collection algorithm may also allocate mark or color bit(s) within pointer *values*. Neither of these are exactly analogous to the revocation bitmap, though the per-object bits are indeed used as part of object lifecycle management.

[9] As capabilities can point out of bounds, revocation tests the bit corresponding to the capability *base* (lower bound), rather than its *address* (pointer target). The CHERI instruction set ensures that capability bases cannot be taken out of bounds without rendering the capability useless; this property has been formally verified for Morello [12]. An allocation is placed into quarantine by setting *all* of its corresponding revocation bits; any valid capability derived from this allocation will then be subject to revocation.

object in *virtual* memory, written to ("**painted**") by the user program's allocator(s) and read by the kernel.[10]

2.2.3 Revocation Epochs. Processing a batch of revocations is done in a **revocation epoch**. The central guarantee of the revocation process is: all capabilities to memory that is marked in the revocation bitmap as quarantined *prior* to an epoch's start are guaranteed to have been expunged as of the epoch's *end*.[11] For CHERIvoke's stop-the-world, synchronous revocation model, this is almost trivial.[12] However, for Cornucopia's concurrent revocation, it is quite common for memory to be freed *during* a revocation scan. Such memory must be held through the end of *next* epoch, before it can be dequarantined. (During the first epoch, the revoker may have let through a pointer that would be revoked if seen after the update to the bitmap.) Any time after the end of the revoking epoch, memory can be dequarantined.

Cornucopia exposes a publicly readable epoch counter, initialized to zero and incremented prior to the start of every revocation and again after the end. Allocators mark freed memory in the bitmap, read the epoch, and wait for the counter to be advanced at least twice (if initially even) or thrice (if odd), to ensure that at least one revocation has both begun and ended, before reusing that memory.

2.2.4 Capability-Dirty Pages. CHERIvoke observes that many pages within application memory do not hold capabilities. Revocation need not consider such pages, and so Cornucopia sought and found a fairly light-weight way of monitoring capability propagation within an address space. Since at least version 7, CHERI-MIPS offered traps on tagged capability stores on a per virtual page basis [54]. This was intended largely as a *security* feature to prevent the spread of tagged capabilities through certain kinds of inter-process shared memory, such as file mappings.[13] Cornucopia repurposes this mechanism to track which pages hold capabilities at the start of revocation (and so must be visited during revocation) as well as the subset of pages to which capabilities have been written during concurrent revocation (and so must be re-visited once its world is stopped).

2.2.5 Revocation Phases. A direct implementation of CHERIvoke stops the world to let the revoker operate on a snapshot of memory. Cornucopia instead splits each revocation epoch into two phases: ① a concurrent phase, in which a second CPU core visits (and marks clean) all dirty pages, and ② a stop-the-world phase, visiting all pages that were *re-dirtied* during the concurrent phase. Relative to CHERIvoke, this split significantly reduces the wall-clock overheads [23, fig. 6] and pause times [23, fig. 10] involved. SPEC CPU2006 benchmarks with large heaps notably showed 70-90% reduction in pause times.

2.3 Analogous Barriers in Garbage Collection

Reloaded directly builds on insights and techniques from garbage collectors, which have the same primary challenge – to efficiently find, test, and possibly act upon every pointer available to a program – and face the same pressures – to minimize resource requirements and disruption to mutator progress. Let us quickly summarize.

Early garbage collectors [16, 36] employed stop-the-world approaches. Awkwardly long pause times soon resulted in "real-time" or "incremental" collectors [32], interleaving collection with the application and bounding latencies for allocations. Multiprocessor systems drove a need for "concurrent" collectors, running alongside the mutator [8, 14]. In these systems, mutator loads, stores, and/or uses of references are subject to **barriers** to preserve collector invariants. Decades of research into collectors have yielded a plethora of barrier designs [8, 14, 15, 18, 19, 24, 28, 29, 31, 40, 41, 59].

The capability-dirty page tracking common to both Cornucopia and Reloaded (§2.2.4) can be thought of as a *store* barrier, like that of Boehm, Demers, and Shenker [14], but provided by the ISA's capability store instructions and memory management unit. Cornucopia uses this barrier to distinguish between clean, dirty, and *re-dirtied* pages, in further analogy to "card-marking" collectors [56].

Reloaded additionally uses a page-based *load* barrier, like that of Appel, Ellis, and Li [8], but integrated into the capability load instructions, as with Azul's Vega architecture [19]. Load barriers allow collectors to have a simple, powerful invariant: the object graph perceived by the mutator outside of barrier code is *as if* collection has already completed.[14] Since stale references cannot be loaded (nor propagated) by the program, collection has guaranteed progress. Reloaded will build on this invariant to guarantee that any pointer loaded by the application is to an allocation live as of the

[10] Cornucopia provided capability-based access control to the bitmap [23, §A], granting allocators access only to the parts of the bitmap corresponding to their heaps. However, for expedience, both Cornucopia's and our experiments unsafely bypass these controls, easing the use of a *shim*, rather than Cornucopia-aware allocator, to manage revocation. We expect the cost of the shim to exceed the cost of an enlightened allocator's bookkeeping.

[11] In garbage collection, the analogous property, dubbed "snapshot at the beginning", is that the objects whose last reference(s) were dropped prior to a collection's start will be reclaimed during the collection.

[12] If there is only one user of revocation in an address space, it *is* trivial. But, because revocation is *global*, multiple allocators may be able to avoid redundant work by using epochs to observe that some other allocator has triggered revocation [23, §B].

[13] CHERI capabilities are interpreted relative to an address space. In the UNIX-like environment of CheriBSD or Linux, that means that capabilities are only well-defined within a process. Allowing processes to exchange tagged capabilities risks violating the security properties of the capability model, though it is currently tolerated in some circumstances, such as `fork()`-ed processes sharing anonymous memory mappings, for compatibility reasons. Because shared file mappings can spread much more widely, especially between processes with no pretense of commonality in their address space layouts, they are prohibited from carrying tagged capabilities.

[14] Load barriers may also be **self-healing** if they correct memory as it is loaded, committing the mutator's view. Reloaded's load barrier fault handler (§3.2) takes this approach; it cleans memory and re-runs the load instruction.

start of the last epoch (§3.2). Further, its use of load barriers will simplify its use of store barriers (§4.2).

3 Designing a New Revoker

3.1 Motivation: Pause Times and Excess Work

Relative to CHERIvoke, Cornucopia's reduction in wall-clock time overheads comes at the expense of *increasing* the total amount of work done per revocation pass. On the dual-core FPGA CHERI-MIPS system under study, the original publication shows that Cornucopia adds, roughly, ① a 10% increase in total *cycle* cost across most of SPEC CPU2006 [23, fig. 10] and ② a 10-15% *DRAM traffic cost* for memory-intensive workloads (exemplified by `omnetpp` and `xalancbmk`) [23, fig. 7]. In our re-implementation of Cornucopia for Morello (§4.5), we observe 4% cycle and 9% memory bus traffic geomean overheads, with a *33%* traffic overhead for `omnetpp`; these overheads are not merely artifacts of the CHERI-MIPS implementation (see figs. 2 and 4).

Moreover, Cornucopia's stop-the-world pauses were, in memory-intensive workloads, still hundreds of millions of CPU cycles, entire seconds on its test-bed, unlikely acceptable outside of batch processing. On the more sophisticated Morello system, our re-implementation exhibits median pause times as high as 30 milliseconds for these workloads (see fig. 9). *Iterating* the Cornucopia strategy, by using a second concurrent pass through memory, attempting to leave fewer pages in need of cleaning with the world paused, showed very little reduction in pause times [23, fig. 15] and, by definition, would anyway increase total work and DRAM traffic.

All told, Cornucopia demonstrated that concurrent revocation is an attractive idea, but its implementation strategy is otherwise likely a dead end. At the time, we conjectured that another approach, centered around intercepting capability loads (and inspired by the Pauseless GC algorithm [19]), might hold promise [23, §X.C]. The present work quantitatively affirms that intuition via implementation.

3.2 Load Barriers for CHERI Revocation

Cornucopia fundamentally operates by tracking capability *store* operations within a process (recall §2.2.5). Since the application may, at any time, *load* any capability in its address space, the revoker must treat any capability *store* as contaminating its targeted page, necessitating a repeated scan once the world is stopped, even though most applications rarely load, much less copy, dead pointers.[15]

If we could, instead, begin intercepting all of an application's capability load operations, we could provide the illusion that revocation had *already taken place*. That is, every pointer the application loaded could be processed for revocation before the load was allowed. The revoker could, then, entirely in the background, work to catch reality up to this perception, inspecting and erasing revoked capabilities through the entire memory space. Pages stored to during this background work would not need to be swept again: any capability stored must have been verified by the revoker already and so its pointed-to memory must not have been slated for revocation as of the start of this revocation epoch.[16] The application would block awaiting revocation only if another batched revocation became necessary while this background task had not yet finished.

There is a little subtlety to the above story: an application thread may have a to-be-revoked capability in its register file when revocation begins. If we allowed this capability to remain, it would break the invariant that all later capability stores are of capabilities already checked for revocation. Therefore, entry into revocation must still synchronize all application threads and scan their register files (and other non-user-memory hoards of capabilities; see §4.4).[17] At the same time, we must synchronize CPU cores (at least those running application threads) to begin intercepting capability loads. In practice, we implement these two synchronizations as part of a stop-the-world phase at the *start* of revocation; see §4.3. However, as we will see (§5.4), these are orders of magnitude shorter than Cornucopia's stop-the-world phase.

Reloaded thus has a concise central invariant: any pointer held in a user thread's register file did not point into quarantine as of the beginning of the last revocation. Even in the face of concurrent application execution, its measure of inductive progress is similarly simple: pages not yet scanned. By contrast, Cornucopia's invariant must be phrased in terms of pages marked capability dirty, and its progress happens only by stopping the world (recall §2.3).

4 Implementation

We now discuss the central aspects of implementing sweeping revocation, and in particular Cornucopia Reloaded, on Morello. Readers curious for more detail are encouraged to read the relevant chapter of Gutstein's thesis [26, §5]. Our implementations for Morello and CHERI-RISC-V have been merged for the public 23.11 release of CheriBSD (§10).

[15] This is not the *sole* source of inefficiencies, as it is not the sole approximation made, but it is surely the most significant.

[16] Within a given epoch, the application might yet hold, copy, and dereference pointers to memory that has been freed within that epoch. In the C/C++ memory models, the value and use of a pointer to freed memory are undefined; the guarantee of CHERIvoke and Cornucopia is only that, *after* a revocation epoch, pointers to memory freed *prior* to that epoch have been removed. Reloaded draws the same distinction, and only frees from prior epochs are guaranteed to be seen as already revoked *during* revocation.

[17] Analogously, garbage collectors must also be aware of references into the collected heap but held outside the heap. Often, this means synchronizing to gain access to thread registers (or synchronizing to ensure that threads are at "safe points" where their registers are spilled to memory) and specially handling thread *stacks*, references held by foreign code, etc.

4.1 Per-PTE Capability Load Generations

One way to implement the design of §3.2 would be to extend each page table entry (PTE) with a flag that blocks all capability loads. A revocation epoch would begin with clearing this flag on all pages (issuing TLB shootdowns) and would end when all pages have been visited by the revoker and their capability load flags restored. This would unfortunately require updating PTEs *twice* during each epoch.

To remove the initial PTE update, Morello (and CHERI-RISC-V) added, for our use, a new behavior for PTEs: rather than a permission flag, each PTE can be configured to compare one of its bits to a bit in an in-core control register, called the "capability load generation". If these bits do not match, any (valid, tag-asserted) capability loaded from this page will trap.[18] In the steady state, the generation bit in the cores and in all PTEs for the address space match. When revocation begins, only the in-core generation bits are toggled, triggering all subsequent capability loads to trap. Thus, PTEs need be updated only once per epoch.[19]

4.2 Lightweight Per-PTE Capability Dirty Tracking

Modern architectures, including ARM v8 and RISC-V, enable accelerated dirty-tracking for PTEs using an additional "dirty" bit. When moving Cornucopia from CHERI-MIPS to Morello and CHERI-RISC-V, we mirror this optimization for capability stores to enable hardware to track writes that (re)dirty pages during our revocation sweep, and thus must be included in the stop-the-world phase. There is, however, some subtlety here (see §7.4) and our implementation is known to have some slight defects that do not impact the evaluation reported later.[20]

4.3 From Architecture to Software

Like Cornucopia, Reloaded's revocation work is divided into two phases, albeit with quite different behaviors (cf. §2.2.5).

First, with the system in the steady state of all generation bits agreeing, a *stop-the-world* phase synchronizes all cores in an address space. These cores increment their capability load generation bits (and any cores later entering this address space will adopt the incremented value), and the thread register files and kernel capability hoards (see §4.4) associated with this address space are scanned for revoked capabilities. PTEs are not modified. Reloaded's invariant that *no* to-be-revoked capability can be held in a register or loaded from memory is reestablished for the new epoch (see §3.2).

The second phase consists of two simultaneous kinds of work. In the foreground, as the application takes capability load faults, the revoker responds by using the application thread to clean the targeted page and update the PTE. In the background, a revoker thread visits all as-yet unvisited pages and updates the PTE. Some light synchronization is required here, around *updates* to PTEs, but page visits in the same epoch are idempotent, so PTE *reads* are less aggressively interlocked. The background work ensures that revocation terminates and restores the steady state of all load generation bits agreeing. Throughout, Reloaded's invariant holds.

Page scans, both foreground and background, employ heuristics to avoid converting read-only pages to read-write. If the revoker can acquire a page *exclusively* and that page is already writable by userspace, revocation may directly mutate its contents. Otherwise, either the page is not writeable or only *shared* access could be acquired, and the page will be handled as read-only. If a capability on such a page must be revoked, the full page fault machinery will be invoked to upgrade the page to writeable, but if no writes are necessary, the page is put back into service as-is.

A thread taking a load fault will lock that threads's process's page table ("`pmap`") twice. At first, the thread will detect if the local core's TLB is out-of-date with the PTE, which may have already been updated due to completed revocation on another core. But if revocation is indeed required, it proceeds without locks held (because probing the revocation bitmap may trigger further page faults) but with the target page sufficiently quiesced that it cannot, for example, be swapped out or, for a read-write scan, be downgraded to copy-on-write. After revocation, the `pmap` is again locked and the scanned page's entry is idempotently updated to the current load barrier generation. Multiple threads may trigger overlapping revocation visits to the same page: any number of threads may take faults attempting to read the same page, and the background thread may also be reading that page. However, at most one such thread is licensed to write to the page. If a read-only scan overtakes the read-write scan and must write, it will serialize behind the read-write scan when upgrading the page and will not scan again.

For expedience, bulk address space operations, especially cloning for `fork`, are excluded during bulk revocation sweeps (as was also true of Cornucopia).[21] Certain operations, such as unmaps or mapping of regions of zeroed pages, which do not complicate revocation, could be permitted.

[18] It is not strictly necessary to condition the trap on the loaded tag value; one could, instead, trap whenever a capability-width datum were loaded. It is not, however, sufficient to merely *clear* tags, as specified for PTE bits in previous versions of CHERI which were intended prevent cross-address-space capability leakage discussed in footnote 13.

[19] Pages known to not be carrying capabilities ("capability clean") can nevertheless have their generation bits kept up to date, as this simplifies the rest of the system. Yet, we would prefer not to; see §§7.4 and 7.6.

[20] In more detail, the implementation studied herein does not always correctly handle aliased pages, including copy-on-write aliases arising from `fork`. The benchmarks studied herein fortunately do not create the problematic cases, as any aliased memory is not also quarantined. In parallel with publication, our work has been merged to CheriBSD (see §10) and these bugs have either been corrected or acknowledged for correction.

[21] Our concurrent implementations use one system call per revocation phase, invoked by a dedicated thread in userspace. This system call holds the virtual address map "busy" throughout concurrent revocation phases. Yet `fork` can occur between two phases of the same revocation epoch; this is subtle for Reloaded, as the child process must continue to take appropriate load traps.

4.4 Thread Synchronization and Kernel Hoards

One of the largest engineering challenges in fitting CHERIvoke and its successors into CheriBSD has been the free flow of user-space pointers into the kernel. These flows may be ephemeral (lasting only for a single system call, such as `write`) or the kernel may *hoard* the pointer(s), returning them to user-space at a later time; hoarding is popular with asynchronous facilities such as `kqueue` and `aio`. When a thread is context switched off core, pointers in its register file are similarly hoarded. At some point during a revocation epoch, the kernel must scan all the pointers it holds on behalf of a user program, and at no point may it divulge one unchecked by the revoker. For Reloaded, this scan happens in the first, stop-the-world phase, leaving pointer copy-out unaltered. At present, this scan uses an inelegant and invasive approach, stopping all threads associated with the application, completing or aborting all in-progress system calls, and interacting with each hoarding subsystem (including saved register files) with bespoke logic. Better approaches seem possible (see §7.8), but we have not found the engineering time required.

4.5 Reimplementing Cornucopia

For this work, we have largely rewritten the public prototype of Cornucopia's implementation in CheriBSD. Beyond adding the "machine dependent" layer for Morello, we had to contend with the original implementation's somewhat superficial integration with the virtual memory subsystem. Reloaded demands a deeper integration, as it must interpose on every attempt to expose a page to userspace as well as the load generation mismatch faults. As we did not wish to have two completely different revocation codepaths in our implementation, our changes necessarily impacted the Cornucopia functionality as well. Of note, our reimplementation will not skip a mapped page in subsequent epochs when it becomes "capability clean." While this matters in principle, pages becoming clean is not observed to happen in the benchmarks studied here, even when running with our Reloaded implementation, which does attempt to detect such pages. What we call "Cornucopia" below should be understood to be our (imperfect) re-implementation of the algorithm, not the same software as the original paper.

5 Evaluation

Revocation in the style of CHERIvoke has four key overheads: wall-clock time, CPU time, bus accesses, and memory occupancy.[22] We evaluate each of ① Reloaded; ② our re-implementation of Cornucopia; ③ "CHERIvoke", our Cornucopia eschewing its concurrent phase; and ④ "Paint+sync", our in-userspace machinery of quarantine bitmap management but without revocation passes. (*"Paint+sync" does not provide temporal safety*, but it is useful in characterizing overheads.) We use a subset of the SPEC CPU2006 suite to represent CPU-bound batch processing (§5.1). As stand-ins representing latency-sensitive workloads, we use PostgreSQL's `pgbench` (§5.2) and gRPC's QPS (§5.3). The `pgbench` workload is a sequence of many transactions and allows measuring latencies of many small units of work performed against a temporally-safe server. The gRPC QPS benchmark is similar, though does its own aggregation of latencies.

We use `LD_PRELOAD`-ed shims to replace the system heap allocator with `snmalloc` [33] and to select the temporal safety strategy with a modified `mrs` [25]. We configure `mrs` to use Cornucopia's revocation policy: allocation requests made when quarantine exceeds $1/4^{\text{th}}$ of the total heap (or, equivalently, $1/3^{\text{rd}}$ of the *allocated* heap) will trigger revocation, unless less than 8 MiB is in quarantine. For each benchmark, all runs, of both baseline and test conditions, use the same CHERI C/C++ "pure capability" spatially-safe CheriBSD binary; the baseline loads the same `snmalloc` `LD_PRELOAD` shim as the test condition but without `mrs`.

Throughout, our CheriBSD kernel is *not* a "pure capability" program, but rather "hybrid" (recall footnote 1). This has no effect on the *interface* to the kernel and user programs' use thereof, including revocation. We believe the use of a hybrid kernel binary, while generally discouraged, minimizes the impact of some of Morello's microarchitectural quirks [51] on revocation sweep loops; the overheads reported here are both attainable on current Morello boards and likely to reflect overheads seen on a hypothetical successor.

5.1 SPEC CPU2006 INT

Eight of the SPEC CPU2006 [27] integer-math benchmarks compile as pure-capability CHERI C/C++ programs and have been used by past CHERI temporal safety work. These are single-threaded, throughput-oriented applications: metrics of interest are aggregate resource consumption, such as time elapsed or memory occupancy, and not per-event latencies.

Methodology. We consider these programs in their `ref` configurations, and measure a total of twelve executions each across each of our temporal safety strategies.

To mitigate sampling effects, we run four batches of each benchmark, with each batch consisting of a cold boot of a Morello and four executions of the benchmark. This gives us some sampling across any per-boot randomization, and we discard the first run in each batch to suppress system-wide startup transients. To further reduce noise, we partition Morello's four cores. Application threads are always pinned to core 3 and any offloaded revocation thread is pinned to core 2. The remainder of the system runs on cores 0 and 1.

Results. Figure 1 shows that Reloaded performs very similarly to Cornucopia on these SPEC workloads, with modest gains on the highest-overhead cases of `omnetpp` and

[22] Wall-clock is measured directly. Peak memory occupancy is measured by the `rusage` framework. Per-core CPU time and bus accesses (a proxy for DRAM accesses) are reported by system-mode `pmcstat`.

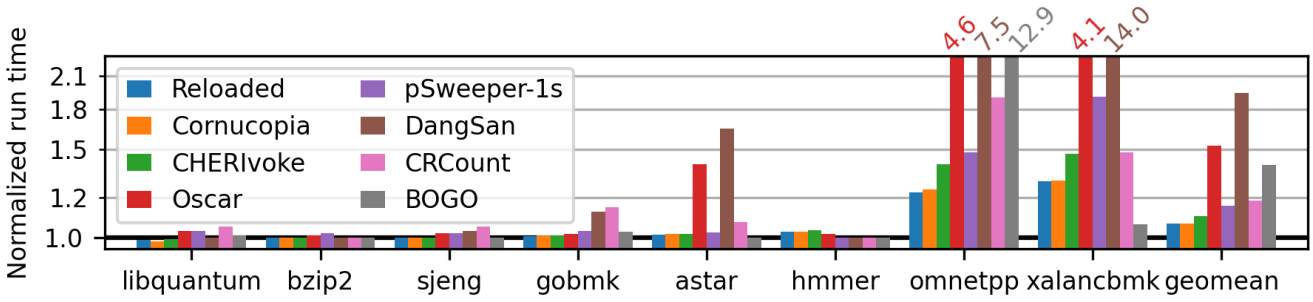

Figure 1. Contrasting wall-clock overheads of Reloaded, Cornucopia, and CHERIvoke against those reported by other published techniques [20, 34, 48, 50, 60]. The Cornucopia and CHERIvoke results reported here are using our re-implementation of the revoker running on Morello and use the CHERI spatially-safe program as the baseline, and so differ from those reported for the earlier implementation on CHERI-MIPS [23, fig. 6]. For programs with multiple workloads (`astar`, `bzip2`, `gobmk`, and `hmmer`), the values shown for CHERI temporal safety schemes are geomeans. The numbers reported for BOGO [60] also factor out their reported cost of spatial safety; other numbers are as reported in their publications. The `bzip2` and `sjeng` benchmarks do not engage revocation and so are excluded from subsequent study.

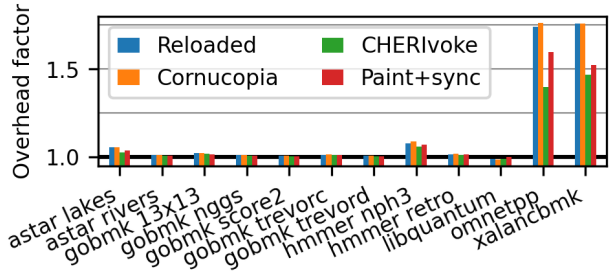

Figure 2. Total CPU-time overheads (both cores) of Reloaded, Cornucopia, CHERIvoke, and asynchronous quarantine management.

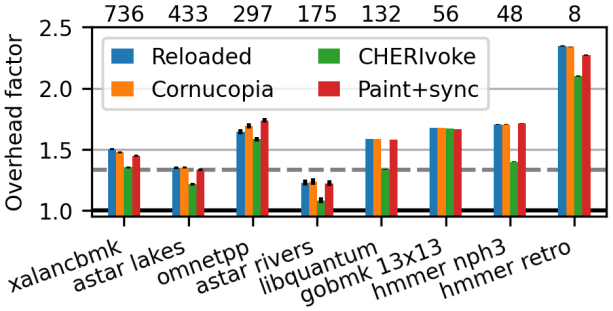

Figure 3. Ratio of averaged peak memory footprint (RSS) between test condition and baseline for a representative subset of benchmarks, sorted descending by the peak RSS of the no-revocation baseline (given above each, in Mibibytes). The general policy target of 33% of the heap in quarantine is shown as a dashed line, but `gobmk` and `hmmer` use so little memory that their revocation behavior is significantly impacted by `mrs`'s default minimum quarantine of 8 MiB.

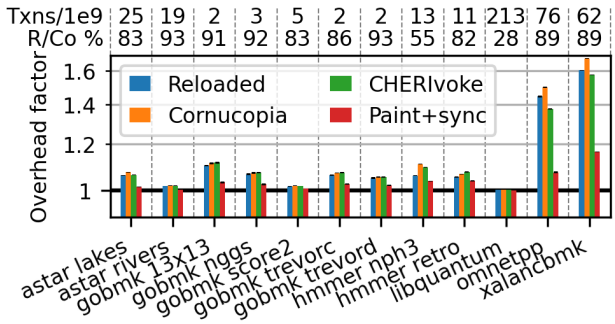

Figure 4. Bus traffic overheads of Reloaded, Cornucopia, and CHERIvoke on Morello. For each benchmark, we also show, above the plot, the mean number of transactions executed by the no-revocation baseline (in billions) and the mean Reloaded traffic as a percentage of Cornucopia mean traffic.

`xalancbmk`. In detail, our worst cases are `xalancbmk`, at 29.4% overhead (down from 29.7% for Cornucopia) and `omnetpp`, at 23.1% (down from 24.8%).[23] All CHERIvoke-based algorithms tested here appear competitive with other published techniques on these workloads.

Figure 2 confirms that Reloaded does not consume more CPU time than Cornucopia, and in some cases, it is modestly cheaper. Tangentially, we speculate that significant differences between "Paint+sync" and "CHERIvoke" are not the result of quarantine per se but largely because `snmalloc`

[23]For perspective, such wall-clock differences, while rare, are not out of the question between different real-world implementations of `malloc`. For example, the SPEC CPU2017 `623_xalancbmk_s` benchmark with the `glibc` allocator ran at approximately *half* the speed relative to running with `snmalloc`, and the same suite's `620_omnetpp_s` benchmark showed a 20% slowdown in the same settings [33, fig. 12].

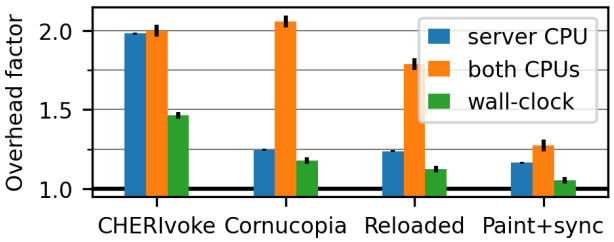

Figure 5. Normalized time overheads for pgbench.

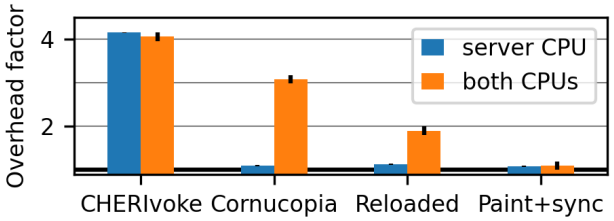

Figure 6. Normalized bus access overheads for pgbench.

behaves differently once quarantine release happens on a second thread.

Figure 3 shows that Reloaded has nearly identical impacts on peak memory usage as Cornucopia. For benchmarks that allocate large heaps and significantly deviate from the expected target of 33% heap in quarantine (`libquantum`, `omnetpp`, and `xalancbmk`), we see that much of the overshoot arises from quarantine, not revocation itself, and that CHERIvoke hews closer to the target. We conclude that these applications free significant amounts of memory while quarantine is still being revoked or otherwise processed.

Figure 4 shows that Reloaded, by not having to *rescan* pages, induces less bus traffic than Cornucopia. The two benchmarks with the highest revocation DRAM overheads both show approximately 11% reduction with Reloaded relative to Cornucopia: `omnetpp` now takes 45% vs. 50%, and `xalancbmk` now takes 60% vs. 68%. The median bus traffic cost of Reloaded relative to Cornucopia is 87% in these benchmarks. While bus traffic reduction was not our primary objective, it is nevertheless a welcome improvement, even if Reloaded's savings here are not reflected in CPU or wall time improvement.

5.2 PostgreSQL pgbench

Here, we seek to assess the impact of these temporal safety technologies on a proxy for *interactive* workloads.

Methodology. We run a pure-capability PostgreSQL [2, 3] server on Morello with our `LD_PRELOAD`-ed shim(s) and run the default workload generator of pgbench [4], without shims, against this server. We use a "scale factor" of 10 and run for 170,000 transactions (which takes about 10 minutes

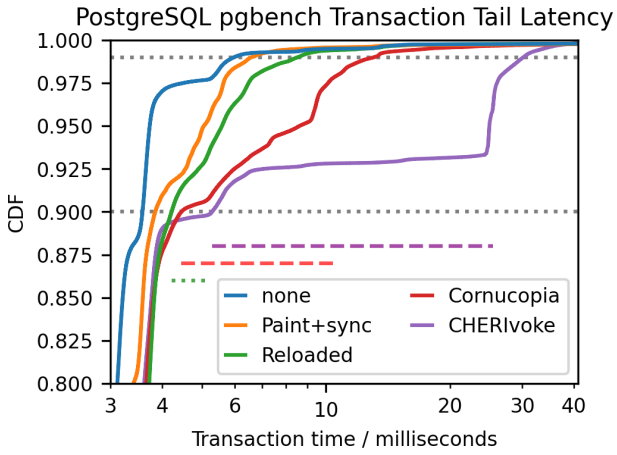

Figure 7. Normalized cumulative distribution function (CDF) of per-transaction execution time of pgbench, showing the fraction of transactions completing in less than the given time, accumulated from four runs of each scenario. 90th and 99th percentiles are marked with dotted grid lines. For comparison, we have approximated (using separate runs, subject to modest *probe effect*) the median per-epoch world-stopped durations for CHERIvoke and Cornucopia, shown as dashed segments, and the median per-epoch *cumulative* time taken to sweep pages by the application thread for Reloaded, shown as the dotted segment. These are shown with their left ends at the corresponding 90th percentiles.

without any temporal safety shim loaded).[24] We use the same quarantine policy and core pinning regime as with the SPEC results above; here, the benchmarked application threads (that is, the PostgreSQL server processes) are pinned to core 3, and any revocation threads are pinned to core 2, while the pgbench client and the rest of the system are on cores 0 and 1. In order to suppress system-wide start-up transients, we initialize the database and perform a 30-second run of pgbench before *restarting* the server for measurements. As invoked here, pgbench performs transactions serially, rather than on an a priori schedule (but see §5.2.1). As such, its results are subject to "coordinated omission" [49], where latency spikes are seen by only a single transaction and do not impact subsequent transactions' reported latencies.

Results. Figure 5 shows that Reloaded offers lower wall-clock and total CPU time overheads than Cornucopia. The overheads imposed on the server thread are nearly identical.

Figure 6 shows that Reloaded incurs less than half the bus traffic overhead of Cornucopia, while only slightly increasing traffic on the application core. This suggests that Cornucopia revisits approximately all pages with the world stopped.

[24] Usually, when benchmarking databases, one would report transactions completed within a fixed period of time. Using a fixed number of transactions makes the workloads more directly comparable to our SPEC CPU results.

Figure 7 captures three significant points: ① All three revocation implementations studied here exhibit similar latency 85th percentiles, and all are only slightly slower than just quarantining memory. ② The strategies begin to differentiate around the 90th percentile, and exhibit stark distinction thereafter. The fully sequential CHERIvoke has its 99th percentile 27 milliseconds slower than the median transaction; Cornucopia's two-pass design lowers its to just under 10, and Reloaded further reduces its to 5.4. The dashed segments of fig. 7 show the median world-stopped times, of 20 milliseconds for CHERIvoke and 6.2 for Cornucopia, respectively, appearing to account for most of the spread between 90th and 99th percentiles. The median sum of trap handling time for Reloaded, of 860 microseconds and shown as the dotted segment, less obviously corresponds with a change in its cumulative distribution function (CDF). ③ Reloaded shows only modest latency increase over its prerequisite overheads ("Paint+sync") until around the 98th percentile.

The pgbench server peak RSS was dominated by "autovacuum" background workers, not the worker processing our pgbench transactions. However, the worker heaps are fairly small, with a mean of 22 MiB allocated (for Reloaded; 21 for Cornucopia, 19 for CHERIvoke); the resulting small mean quarantine sizes (5.6 MiB for Reloaded, 5.3 for Cornucopia, and 4.8 for CHERIvoke) are under 4% of the worker's RSS.

Discussion. This workload is not (steadily) CPU bound; without sweeping revocation, we measure the server thread on core for only a median 290 of the median 598 wall-clock seconds elapsed. Two notable consequences follow: ① CPU overheads, even restricting to the PostgreSQL server thread, can be significantly larger than elapsed runtime overheads, as the server process can "expand" into this already-present inter-transaction idle time. ② Stop-the-world phases, in particular, can be "hidden" in these idle intervals. This helps to explain why, although CHERIvoke has a pronounced "corner" in fig. 7, comparable to its median pause time (purple, upper dashes), Cornucopia sees no corner directly comparable to *its* median pause time (lower, red dashes).

Last, as said, the Reloaded strategy tracks quite closely to its prerequisite overheads. This suggests that, at least for interactive workloads, changes to quarantine representation and management may have a more significant effect on transaction latency than refinements to fault handling.

This workload differs significantly from those of §5.1 in its revocation *rate*: approximately once every 17 transactions, or over 14 times per second, with Reloaded. We defer further contrastive analysis to §5.5.

5.2.1 Latency vs. Throughput.

We can also run pgbench such that it imposes an *a priori* schedule on its transactions (its `--rate` option). For some schedules, we reran the benchmark with Reloaded and otherwise as above. Per-transaction latencies, ignoring schedule lag, in millisecons, are as shown

	Latency Percentile:	50	90	95	99	99.9
Tx/sec	100	3.15	5.14	6.28	12.8	32.4
	150	3.12	5.12	6.35	12.5	43.9
	250	3.06	4.13	5.49	8.72	68.6
	unscheduled	3.15	4.22	5.59	8.55	69.6

Table 1. Latency percentiles, in milliseconds, for various fixed rate schedules of pgbench.

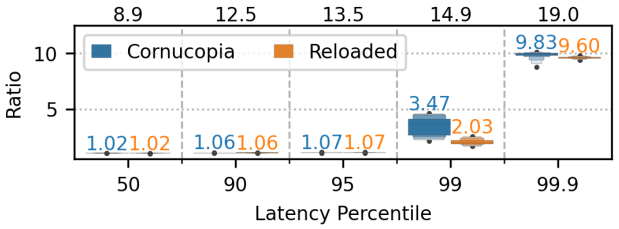

Figure 8. Boxplot of latency percentiles for gRPC's QPS benchmark, using the scenario of §5.3, normalized by their respective percentile's no-revocation mean latency (given, in milliseconds, above the chart). Means of each distribution are shown above.

in table 1. As expected, the long tail 99.9th percentile decreases with lower throughput. However, this comes with unexpected *increases* in short tail latencies (90th - 99th percentiles). This effect is also seen without revocation; further investigation is necessary.

5.3 gRPC QPS

Methodology. To further explore the impact of revocation on latency-sensitive workloads, we run a gRPC 1.48.1 QPS [1] client-server workload. As with pgbench, we shim the server process with revocation and leave the client unmodified.

To focus on the effects of revocation, we use a custom gRPC "scenario": transport security is disabled, each of the client and server is a single process with two threads, both client and server are asynchronous and apply throughput-focused tuning to a minimal stack, each client thread opens 10 gRPC "channels", and each channel is permitted to have 4 outstanding messages. The client will send messages to the server and await responses, while measuring throughput and latency percentiles, for 30 seconds, after a 5 second warmup.

Because QPS workers are multi-threaded, we pin the server process to CPU cores 2 and 3, but do not attempt to pin the background revocation thread; that is, revocation more directly competes with foreground work in this workload. The client, and the rest of the system, is restricted to cores 0 and 1. To mitigate sampling effects, we collect data for each revocation strategy from three batches of four runs, with each batch run after a cold boot and beginning with an additional, discarded, run. Unlike the earlier experiments, all data collection took place on the same Morello machine.

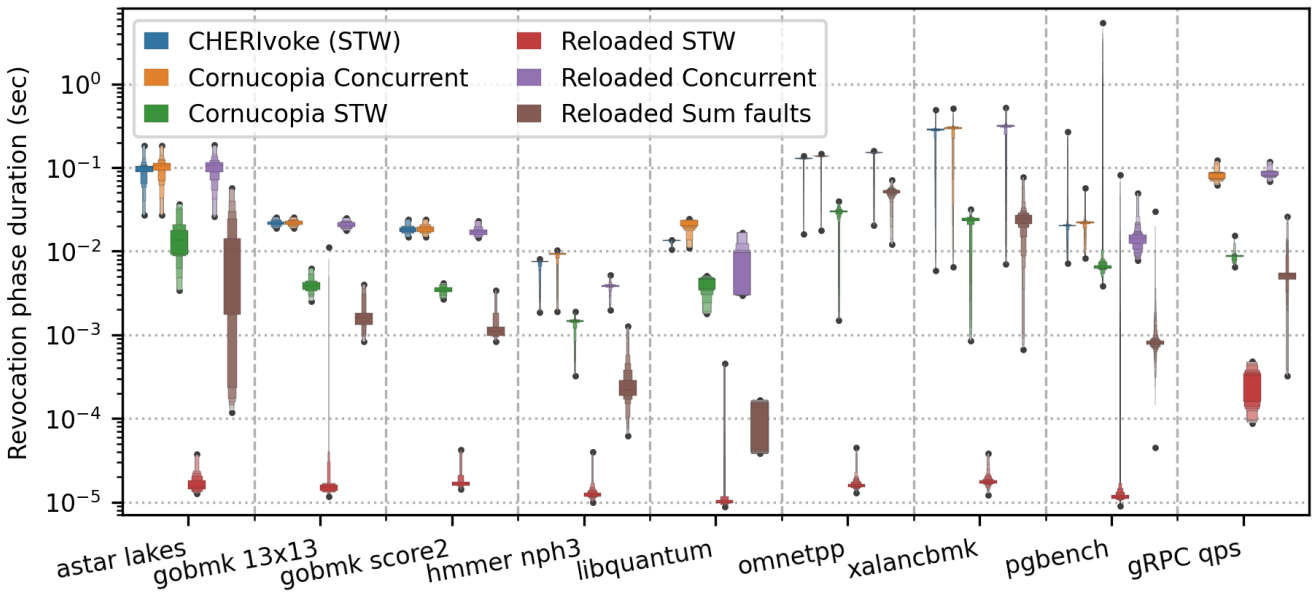

Figure 9. A representative subset of benchmarks' revocation phase times. For each benchmark, we show (left to right) a boxplot summarizing the distribution, across all runs and all revocations, of time taken by CHERIvoke's single (world-stopped) phase (in blue), the concurrent (orange) and world-stopped (green) phases of Cornucopia, the world-stopped (red) and concurrent (violet) phases of Reloaded, and all Reloaded faults in the application thread during the concurrent phase (*cumulative* per revocation, brown). The black dots show extremal values. gRPC QPS triggers a bug in CHERIvoke, so that bar is absent.

Results. Reloaded induces a 12.82% ± 0.16% reduction in QPS throughput, not statistically significantly different from Cornucopia's 12.88% ± 0.16%.[25] Figure 8 shows the selection of latency statistics that the benchmark accumulates. Similarly to fig. 7, we see that either temporal safety strategy introduces modest increases to observed 50%, 90%, and 95% latency percentiles (resp.). At 99%, Reloaded doubles (2.0 ± 0.3) latency relative to the baseline (adding an additional 15 milliseconds, mean), which compares favorably to Cornucopia's more than tripling (3.5 ± 0.9; or adding 37 milliseconds, mean). At 99.9%, both revocation strategies impose long tail latencies of nearly ten times those of the baseline (9.6 ± 0.1 for Reloaded and 9.9 ± 0.4 for Cornucopia).

Using separate runs of the benchmark with more detailed probing, we can estimate Cornucopia's mean stop-the-world time to be 8.7 ± .8 milliseconds, and Reloaded's 0.3 ± 0.1. Reloaded's cumulative fault handling time shows high variance across revocations, at 6 ± 3 milliseconds, with a measured range from 0.3 to 39. Revocation's concurrent phases take 86±6 milliseconds for Reloaded or 81±8 for Cornucopia.

[25]Because of an undiagnosed bug in our implementation apparently involving signal handling, we are unable to obtain CHERIvoke results for this experiment. As the focus here is on the concurrent revocation schemes, we have chosen to exclude it from dicussion rather than debug the issue.

Discussion. Reloaded and Cornucopia perform almost identically up through the 95th percentile, adding very little latency relative to the baseline. This suggests that, as with pgbench, differences in this region are dominated by the costs of quarantining memory; in particular Reloaded's faults do not add significant cost to every transaction. At 99%, we see impacts of revocation and a differentiation of strategies' impact. In particular, Reloaded's mean impact is less than Cornucopia's measured *minimum*. At 99.9%, we see pathological behavior that was not observed with pgbench, where this percentile showed essentially no difference from baseline. Here, we see some transactions stalled across *two* revocation epochs! We suspect that two factors are at play: ① the "background" revoker threads are contending for CPU time, and will, when revocation is active, use their entire preemptive quantum (see §7.7), and ② the `mrs` shim blocks an allocation or free operation if its quarantine is over twice full (but see §7.2; future policies may differ). Nevertheless, Reloaded still performs marginally better than Cornucopia.

5.4 Revocation Phase Timing

Figure 9 visualizes the impact of revocation on interactive workloads from another angle; we display the time spent in different phases of each revocation strategy, including the critical stop-the-world phase. To reduce sampling effects, each benchmark here was run at least four times on at least two physical Morello machines. These times are subject to

a minor *probe effect* from the more detailed accounting necessary to obtain them, relative to the overheads reported earlier.

First, we can validate Cornucopia's claim that its world-stopped phases (third from the left) are on the order of a tenth as long as its concurrent phase (recall [23, fig. 10]), even given the dramatic differences between CHERI-MIPS and Morello. We can also, again, see that Cornucopia and Reloaded both do more aggregate work than CHERIvoke, but that the vast majority of this work is done in the background.

Reloaded's stop-the-world phase is almost always very brief for single-threaded workloads, on the order of tens of microseconds. In workloads with large amounts of memory (`astar`, `omnetpp`, `xalancbmk`, `pgbench`), this can be *three or more orders of magnitude* smaller than Cornucopia's stop the world phase. In many workloads, even the *cumulative* time spent handling Reloaded's on-demand faults is significantly smaller than Cornucopia's stop-the-world phase. In `pgbench`, even if the entirety of this time is borne by one transaction's processing, Reloaded will still have less of an effect on that transaction than one suspended by Cornucopia's stop-the-world phase. The exceptions are pointer-chase-heavy workloads (`astar`, `omnetpp`, `xalancbmk`); however, even these applications see that impact spread out across the concurrent phase rather than all at once.

For the multi-threaded gRPC workload, we see that the stop-the-world phase of Reloaded has a median of 323 microseconds. This increased time is likely inter-core synchronization, as gRPC will have both cores busy. Still, this is over an order of magnitude smaller than Cornucopia's.

5.4.1 Outliers. There are a few outlier points in this graph. One run of `gobmk 13x13` may have seen one context switch during a Reloaded STW phase, which took just over ten milliseconds. Eight `pgbench` Reloaded STW phases took longer than ten milliseconds and 92 Cornucopia STW phases took longer than a second; we suspect this is some pathological interaction between quiescing application thread(s) and system calls in progress. One early epoch in all `libquantum` Reloaded runs seems to consistently take around half a millisecond; this, too, may be a result of interaction with a system-call in flight. FreeBSD's `thread_single()` mechanism used to quiesce application threads is primarily used in rare, expensive operations (`fork`, `exec`, `exit`, debug tracing, etc.), and approaches more tuned for revocation seem plausible, at the cost of some engineering effort (see §7.8).

5.5 Revocation Rates

A significant difference between the workloads we have studied is the *rate* at which they perform revocation. Table 2 collects relevant statistics for Reloaded under a representative set of benchmarks. We see that the most RSS-heavy SPEC workloads, namely `xalancbmk`, `astar lakes`, and `omnetpp`, cycle gigabytes of address space through the allocator while having mean heap allocation footprints of hundreds of megabytes. These programs perform, on average, less than one revocation per second. By contrast, the PostgreSQL server process attending to `pgbench` cycles nearly as much address space as `xalancbmk` while having a mean heap of around 4% the size. This workload performs an average of nearly fifteen revocations per second! This helps explain the discrepancy between fig. 4, where 60% bus overhead was the worst case, and fig. 6, where 96% was the *best* case.

Benchmark	Mean Alloc (MiB)	Sum Freed (GiB)	F:A	Revocations	Rev. / sec.
`xalancbmk`	625	66.9	110	426	0.572
`astar lakes`	235	3.36	14.7	39	0.150
`omnetpp`	365	73.8	207	827	0.880
`hmmer nph3`	49.3	2.06	42.8	168	1.45
`hmmer retro`	20.4	0.579	29.0	117	0.481
`gobmk trevord`	124	0.212	1.75	7	0.0623
`pgbench`	23.0	55.1	2534	10072	14.8
gRPC QPS	340	4.65	14.0	54	1.54

Table 2. Reloaded revocation rate statistics for a representative subset of benchmarks, using the same detailed reporting runs as in §5.4: the mean allocated heap size (sampled at each revocation), the mean sum of all quarantined bytes per run (accumulated at each revocation), the ratio of the above two numbers, the mean count of revocations performed (per run), and mean revocations per run wall-clock time. All gRPC revocations are assumed to happen during the 35 wall-clock seconds of useful work done by the worker process.

5.6 The More Subtle Costs of Concurrency

Across SPEC, in fig. 4, we see that Reloaded's concurrent behavior can require moderately more (`omnetpp`, `xalancbmk`) to slightly less (`gobmk`, `hmmer`) memory traffic than the fully-sequential CHERIvoke (though it always uses less than Cornucopia; recall §5.1). For PostgreSQL, in fig. 6, Reloaded has dramatically lower overheads. CHERIvoke's sequential scan effectively flushes the cache, whereas Reloaded's load faults often warm the cache with useful application data, leaving a second core with an independent cache to scan in the background. Conversely, traffic may increase for pointer-chase-heavy workloads, such as `Omnetpp` and `xalancbmk`, may have sparse page utilization such that the revoker's page scans may contend with useful application data.

More generally, this suggests future avenues of exploration, including non-temporal loads for the page scan and/or bitmap probes, or "run-ahead" during revocation page faults to reduce the total number of faults. If hardware offload engines should be designed to perform the revocation sweep, cache contention with the application cores should be a primary design consideration.

6 Related Work

6.1 Memory Coloring

Some mainstream architectures have added pointer and memory "coloring" in an effort to address memory safety woes and accelerate instrumentation-based technologies like Address Sanitizer [46]. Prominent examples include SPARC's SSM or ADI [38] and, more recently, Arm's Memory Tagging Extension (MTE) [9, §D10]. These technologies color both memory and pointers by adding a few bits of metadata to small granules of physical memory and carving out an equal number of bits from pointer addresses. When dereferencing a pointer, the colors of the pointer and memory can be checked to match. With strict enforcement, this can be leveraged to provide deterministic defense against overflow into adjacent objects, and can additionally provide stochastic defense against UAF, UAR, and double-free, by recoloring memory per use.

Such machinery generally comes with two caveats:

1. Memory colors must be kept secret from adversaries capable of forging pointers (akin to ASLR's entropy).
2. Strict trapping of violating stores requires a read-modify-write of memory, so coloring architectures optionally track stores "imprecisely," merely flagging violations for later analysis.

Microsoft Security Response Center [13] and Google [47] have published reports analyzing the security offerings of these schemes. Broadly speaking, performance concerns mean production software will use imprecise tracking for at-scale *auditing*, with precise tracking used during development or to collect crashes *after* detection in production. This strategy relies on *surveillance* (and *reactive* patching) to deter vulnerability exploitation. By contrast, Cornucopia Reloaded, like CHERI and all CHERIvoke descendants, aims to be a deterministic, *proactive*, *fail-stop* defense. Nevertheless, combining the two holds exciting promise; see §7.3.

6.2 Unmapped Memory

While `snmalloc` and the embedded allocators in the CheriBSD C runtime never relinquish address space via `munmap()`, some `mmap()` consumers do (e.g, a program that repeatedly maps files in order to copy them). This introduces opportunities for both intra- and inter-allocator UAF and UAR. We have implemented (but do not evaluate in this paper) a solution to this problem in two parts:

1. Capabilities returned by `mmap()` are backed by a *reservation* [11]. When part of the reservation is unmapped via `munmap()`, the addresses are backed by guard mappings until the entire reservation is unmapped. This ensures that holes in the reservation can not be filled by subsequent `mmap()`s and create UAF issues.[26]
2. When reservations are completely unmapped, they are placed in quarantine. We have extended Cornucopia and Reloaded's sweep infrastructure to search for and revoke capabilities referencing quarantined mappings. Only after a revocation pass do we free the unmapped reservation.

Together, these changes eliminate gaps in CHERIvoke and Cornucopia's UAF and UAR protections.

6.3 CHERIoT Temporal Safety

The CHERIoT embedded platform [6, 7] features heap temporal safety via an adaptation of Reloaded to a MMU-less system.[27] It, too, batches revocations, bitmaps quarantine, and operates by interposing on capability loads. CHERIoT, however, leverages the small size of the systems it targets to eliminate the distinction between UAF and UAR; freed objects become inaccessible immediately. As a result, details of revocation batching and epochs are much less visible to the client.

The perceived immediacy of revocation is accomplished by having the CHERIoT Capability Load instruction directly probe the revocation bitmap to *filter* capabilities: a capability to a revoked object has its tag cleared on its way into the register file,[28] without traps or software intervention. To make this tolerable, CHERIoT's revocation bitmap is ① defined by the processor *architecture*; ② microarchitecturally situated in a "tightly coupled memory" next to the CPU core pipeline, accessible with low latency bounds; and ③ physically indexed, as the architecture lacks virtual memory.

Closing the UAF/UAR gap allows the CHERIoT allocator to use in-band metadata for its quarantine and free lists, without additional defensive measures [39, 43], as freed words are not accessible by clients. By contrast, Cornucopia and Reloaded have to behave as though quarantined objects were still allocated, using out-of-band lists to track quarantine.

Given the tight coupling, CHERIoT's Ibex implementation uses a small, cycle-stealing, pipelined hardware revocation state machine rather than a software thread [7, §3.3.2]. This engine can, in its steady state, test one capability for revocation every cycle. At a modest 20 MHz, the demonstration platform's 512 KiB of RAM takes just over 3 milliseconds, less than an idle time quantum, to sweep.

[26] Reservations also serve a *spatial safety* role, as they are padded as required by CHERI capability bounds compression [57]. This padding is initially backed by guard pages, as if it had been unmapped immediately, and prevents `mmap()` from creating spatial aliases.

[27] Despite the chronology of publication, much of the design and implementation of Reloaded predates CHERIoT.

[28] This load barrier is *not* self-healing; recall footnote 14.

7 Future Work

7.1 Concurrent Background Revocation

Cornucopia and Reloaded both use a single thread for all their background revocation work. It should be relatively straightforward to split this work between multiple threads, enabling multiple cores to accelerate revocation. This would allow revocation sweeps to complete faster and could be attractive on systems with multiple temporally safe processes. In fact, we imagine eliminating the current per-process background thread in favor of making the revocation system call asynchronous, backed by a shared pool of background, in-kernel worker threads.

7.2 Quarantine Policy Tuning

The single revocation policy in the `mrs` shims used herein is not particularly tuned. First, it may be that the ratio of quarantine to allocated heap chosen is not ideal, or not ideal for some workloads. Second, more subtle considerations also apply. For example, `mrs`'s internal list of quarantined objects is double-buffered to permit `free` operations while revocation is in progress. If, however, an *allocation* request arrives and this second quarantine buffer is also over policy, `mrs` blocks and waits for revocation to finish. This, too, may not be ideal behavior. Thirdly, `mrs` makes no effort to detect and *unmap* entire pages that are quarantined. Such unmapped pages could be discounted in quarantine, giving significant reduction in the need to run the revoker [23, §VIII.A].

7.3 Composing CHERI and Memory Coloring

A *non-orthogonal* composition of CHERI and memory coloring (recall §6.1) could bring an order of magnitude improvement to revocation overheads, while also closing the UAF-UAR gap, simplifying allocator data structures, justifying a faster store behavior for memory colors, and exposing enough information for a hardware revocation engine (as with CHERIoT; recall §6.3). Concisely, this composition would move in-pointer color bits under CHERI's integrity protection and would require *authority* to re-color memory. When the heap allocator sets bounds on an object, it can also fix the color of the capability returned; using its elevated authority to the heap, the allocator can recolor memory as part of `free()` and promptly prevent stale access. As the color space is finite, revocation is still required, but only when a given span of memory has exhausted all of its colors. Quarantine, and the pressure to revoke, thus grows at a rate inversely proportional to the number of colors available.

After an allocation is freed, capabilities to it are permanently useless: either they have the wrong color or they have been revoked. As such, clients will never be able to read or write through such capabilities, and, in particular, cannot read their own writes to see whether they happened or not. Thus, we are justified in *discarding* stores through mis-colored capabilities, rather than reporting traps, without sacrificing security. Further, we may also revoke mis-colored capabilities whenever they are found, even if their target address space is not quarantined. This test is completely architectural, making it well-suited for DMA accelerators.

7.4 Revised Capability-Dirty Tracking

The architectural machinery underpinning capability-dirty tracking emerged from an experimental *security* feature in CHERI-MIPS, meant to constrain the flow of capabilities in the system on a per *virtual* page basis. While it works in its new role, it is awkward: we are attempting to measure a property of the *data* on a physical page through its (potentially many) virtual aliases. Marking a page as needing the revoker's attention is straightforward: if any alias would permit a capability store without trapping, the page must be visited during revocation. Unfortunately, detecting that a page no longer holds capabilities is quite challenging, involving multiple rounds of PTE updates and careful synchronization of software metadata. We would prefer *physically-indexed* control information. While most architectures have not, historically, had such structures, the push towards confidential computing has given rise to plausible candidates, such as the Granule Protection Table in Arm's RME [9, §D9] or the Reverse Map Table in AMD's SEV-SNP [5].

7.5 Relaxed Capability Tag Coherence

Reloaded's use of load barriers, and the concomitant invariant that applications can propagate only checked capabilities, opens the door to tolerating one-sided error in capability-dirty tracking. In stark contrast to Cornucopia, Reloaded need not be informed of capability stores *during* revocation. In fact, Reloaded can operate on a view of capability tags potentially *as stale as* the start of the current revocation epoch, so long as its updates to memory remain properly atomic and sensitive to later stores. If we can efficiently achieve a global view of tags at revocation's start (say, by writing tags back to memory), we may be able to significantly reduce cache coherency traffic associated with probing for the presence of capabilities in memory.

7.6 Revised PTE Capability Load Control

The per-PTE-based control of capability loads and generations given in §4.1 is less than ideal. Because faults are triggered only in the case of generation bit mismatches, capability-clean pages that are permissive of capability stores are awkward: we must keep the generation bit up to date even though no capabilities can (yet) be loaded from this page. Failure to do so risks a load generation trap outside of revocation or, worse, a capability escaping the attention of the revoker. We are therefore faced with an unpleasant decision: either always keep such pages' generations up to date on every revocation scan (unnecessarily taking the `pmap` lock) or completely disallow capability stores to clean pages

(requiring at least that transition *from* clean *broadcast* the update to all aliases).

To address this awkwardness, we propose adding a PTE configuration in which capability loads *always trap*;[29] such pages must be considered by the revoker, but their contents may be skipped while the physical page remains clean and their generation bits do not need to be updated. Traps triggered by this PTE disposition can be quickly resolved by replacing the PTE with one that contains the current load generation (to be maintained until the page is clean).

7.7 Revocation's Time Quantum and Priority

The investigation of gRPC's *multi-threaded* workload (§5.3) highlights another source of tail latency: preemptive multitasking with the "background" revoker thread. The other workloads have had the luxury of a spare core onto which to delegate this work, but for this case, the revoker more obviously competed with the application for CPU time. At the moment, the background revoker thread is merely another `pthread` thread in the system; it is quite likely that tuning (down) the preemptive quantum for this thread and/or lowering its priority would improve tail latencies.

7.8 Thread Synchronization and Kernel Hoards

As mentioned in §4.4, our machinery for addressing pointers held ephemerally or hoarded by the kernel on behalf of a user program is crude. The necessity to wait for system calls to complete or abort and the need to interact with (the locks within) each hoarding subsystem create long tails of latency for revocation phases. In some cases, these tails span many orders of magnitude, which may pose a challenge to revocation's use in even soft real-time systems. Two thrusts of engineering effort may be able to reduce these tails:

1. Allowing system calls that do not take pointers or whose pointer arguments are not revoked and are to flat data (without pointers); these can continue in parallel with revocation.
2. Rewriting hoarding subsystems to use a generic indirection layer, with all hoarded pointers stored on pages subject to revocation load traps. A variety of mechanisms are available to ensure that the copying and/or use of capabilities loaded from this layer are atomic with respect to revocation.

8 Conclusion

We have demonstrated a new approach to global CHERI capability revocation with better performance and lower latencies than the prior state of the art. This expands the set of workloads that can benefit from deterministic heap UAF mitigation built atop this mechanism. In particular, we have shown that Reloaded's approach offers dramatically improved tail latencies for request-based workloads, as exemplified by `pgbench`, while also improving overheads experienced by batch workloads. We have outlined next steps, ranging from software engineering to novel architecture, to continue this work.

[29] An "all tagged loads trap" configuration is specified in CHERI-RISC-V, but is, we believe, not yet completely implemented [52, §4.3.12].

9 Acknowledgements

We are deeply indebted to Alfredo Mazzinghi for his help with the gRPC benchmark. The CHERI and Morello efforts are possible only with the collaboration of a community of people far too numerous to name here, but we thank the giants upon whose shoulders we stand. Our anonymous reviewers and our shepherd, Martin Maas, provided excellent feedback, and we tried to improve the work correspondingly.

This work was supported by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 105694, and by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-18-C-0016 ("ECATS") and Contract No. HR0011-23-C-0031 ("MTSS"). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA). Distribution Statement A. Approved for public release: distribution is unlimited. For the purpose of open access, the authors have applied a Creative Commons Attribution (CC BY) license to the accepted version of this manuscript.

10 Availability

We have incorporated a variant of Reloaded using the `mrs` shim and a lightly modified `jemalloc` into the 23.11 release of CheriBSD. It is available at https://www.CheriBSD.org and can be used on shipping prototype Arm Morello hardware.

References

[1] Benchmarking | grpc. URL: https://grpc.io/docs/guides/benchmarking/.

[2] PostgreSQL. URL: https://www.postgresql.org/.

[3] PostgreSQL 9.6 CHERI port. URL: https://github.com/CTSRD-CHERI/postgres/tree/96-cheri.

[4] PostgreSQL pgbench benchmark. URL: https://www.postgresql.org/docs/9.6/pgbench.html.

[5] Advanced Micro Devices, Inc. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more, January 2020. URL: https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[6] Saar Amar, Tony Chen, David Chisnall, Felix Domke, Nathaniel Filardo, Kunyan Liu, Robert Norton-Wright, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. Cheriot: Rethinking security for low-cost embedded systems. Technical Report MSR-TR-2023-6, Microsoft, February 2023. URL: https://www.microsoft.com/en-us/research/publication/cheriot-rethinking-security-for-low-cost-embedded-systems/.

[7] Saar Amar, David Chisnall, Tony Chen, Nathaniel Filardo Wesley, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. CHERIoT: Complete memory safety for embedded devices. In *proceedings of the 56th IEEE/ACM International Symposium on Microarchitecture.* Association for Computing

Machinery, Oct 2023. doi:10.1145/3613424.3614266.

[8] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. *SIGPLAN Not.*, 23(7):11–20, jun 1988. doi:10.1145/960116.53992.

[9] Arm Inc. Arm architecture reference manual for a-profile architecture. URL: https://developer.arm.com/documentation/ddi0487/latest/.

[10] Arm Inc. Morello program. URL: https://www.arm.com/architecture/cpu/morello.

[11] Arm Limited. Morello pure capability kernel user Linux ABI specification, August 2023. URL: https://git.morello-project.org/morello/kernel/linux/-/wikis/Morello-pure-capability-kernel-user-Linux-ABI-specification.

[12] Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. Verified security for the Morello capability-enhanced prototype Arm architecture. Technical Report UCAM-CL-TR-959, University of Cambridge, Computer Laboratory, September 2021. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-959.pdf, doi:10.48456/tr-959.

[13] Joe Bialek, Ken Johnson, Matt Miller, and Tony Chen. Security analysis of memory tagging, 2020. URL: https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20memory%20tagging.pdf.

[14] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, page 157–164, New York, NY, USA, 1991. Association for Computing Machinery. doi:10.1145/113445.113459.

[15] A. M. Cheadle, A. J. Field, S. Marlow, S. L. Peyton Jones, and R. L. While. Non-Stop Haskell. *SIGPLAN Not.*, 35(9):257–267, sep 2000. doi:10.1145/357766.351265.

[16] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, nov 1970. doi:10.1145/362790.362798.

[17] The Chromium Team. Memory safety. URL: https://www.chromium.org/Home/chromium-security/memory-safety.

[18] Austin Clements and Rick Hudson. Proposal: Eliminate STW stack re-scanning, October 2016. URL: https://github.com/golang/proposal/blob/master/design/17503-eliminate-rescan.md.

[19] Cliff Click, Gil Tene, and Michael Wolf. The pauseless GC algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, page 46–56, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1064979.1064988.

[20] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A practical Page-Permissions-Based scheme for thwarting dangling pointers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 815–832, Vancouver, BC, August 2017. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/dang.

[21] Data61 Trustworthy Systems Team. sel4 reference manual version 12.1.0. URL: https://sel4.systems/Info/Docs/seL4-manual-latest.pdf.

[22] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. CheriABI: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 379–393, New York, NY, USA, 2019. Association for Computing Machinery. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201904-asplos-cheriabi.pdf, doi:10.1145/3297858.3304042.

[23] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal safety for CHERI heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1507–1524, Los Alamitos, CA, USA, 5 2020. IEEE Computer Society. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2020oakland-cornucopia.pdf, doi:10.1109/SP40000.2020.00098.

[24] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2972206.2972210.

[25] Brett Gutstein. CHERI Malloc Revocation Shim. URL: https://github.com/CTSRD-CHERI/mrs.

[26] Brett Gutstein. Memory safety with CHERI capabilities: security analysis, language interpreters, and heap temporal safety. Technical Report UCAM-CL-TR-975, University of Cambridge, Computer Laboratory, November 2022. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-975.pdf, doi:10.48456/tr-975.

[27] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4), September 2006.

[28] Urs Hölzle. A fast write barrier for generational garbage collectors. In *Proceedings of the OOPSLA'93 Workshop on Garbage Collection*, October 1993. URL: https://bibliography.selflanguage.org/write-barrier.html.

[29] Lorenz Huelsbergen and Phil Winterbottom. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, page 166–175, New York, NY, USA, 1998. Association for Computing Machinery. doi:10.1145/286860.286878.

[30] 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 *2017 IEEE International Conference on Computer Design (ICCD)*, pages 641–648, November 2017. doi:10.1109/ICCD.2017.112.

[31] Mark Stuart Johnstone. Non-compacting memory allocation and real-time garbage collection, December 1997. URL: https://proquest.com/docview/304374180.

[32] Henry G. Baker Jr. List processing in real time on a serial computer. Technical Report 139, MIT Artificial Intelligence Laboratory, 1977. URL: https://dspace.mit.edu/handle/1721.1/41976.

[33] Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J. Parkinson, Alex Shamis, Christoph M. Wintersteiger, and David Chisnall. Snmalloc: A message passing allocator. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2019, pages 122–135, New York, NY, USA, 2019. ACM. doi:10.1145/3315573.3329980.

[34] Daiping Liu, Mingwei Zhang, and Haining Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1635–1648, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3243734.3243826.

[35] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape, February 2019. URL: https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-

%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf.

[36] M. L. Minsky. A LISP garbage collector algorithm using serial secondary storage. Technical Report AIM-058, MIT Artificial Intelligence Lab, December 1963. URL: https://dspace.mit.edu/handle/1721.1/6080.

[37] R. M. Needham and R. D.H. Walker. The Cambridge CAP Computer and its protection system. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP '77, page 1–10, New York, NY, USA, 1977. Association for Computing Machinery. doi:10.1145/800214.806541.

[38] Oracle Inc. Hardware-assisted checking using Silicon Secured Memory (SSM), 2015. URL: https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html.

[39] Matthew J. Parkinson. Hardening snmalloc. URL: https://github.com/microsoft/snmalloc/tree/9d4466093a7c42e4fe43e032aeca356674d6e55c/docs/security.

[40] Pekka P. Pirinen. Barrier techniques for incremental tracing. In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, page 20–25, New York, NY, USA, 1998. Association for Computing Machinery. doi:10.1145/286860.286863.

[41] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. *SIGPLAN Not.*, 43(6):33–44, jun 2008. doi:10.1145/1379022.1375587.

[42] David D. Redell. *Naming and Protection in Extendable Operating Systems*. PhD thesis, University of California, Berkeley, September 1974. URL: https://dspace.mit.edu/handle/1721.1/149438.

[43] Chris Rohlf. Isolation alloc heap allocator security feature comparison. URL: https://github.com/struct/isoalloc/blob/947f3bc91c3a61e3ec85bff8f43034d2a59c897d/SECURITY_COMPARISON.MD.

[44] Peter David Rugg. Efficient spatial and temporal safety for microcontrollers and application-class processors. Technical Report UCAM-CL-TR-984, University of Cambridge, Computer Laboratory, July 2023. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-984.pdf, doi:10.48456/tr-984.

[45] *The Rust Reference*. 1.71.0 edition. URL: https://doc.rust-lang.org/1.71.0/reference.

[46] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC 2012*, 2012. URL: https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker.

[47] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory Tagging and how it improves C/C++ memory safety. URL: http://arxiv.org/abs/1802.09517, arXiv:1802.09517.

[48] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. CRCount: Pointer invalidation with reference counting to mitigate use-after-free in legacy c/c++. NDSS '19. Internet Society, 2019. doi:10.14722/ndss.2019.23541.

[49] Gil Tene. How NOT to Measure Latency. In *Low Latency Summit*, 2013. A version of this presentation is available at https://www.infoq.com/presentations/latency-pitfalls/.

[50] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. DangSan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 405–419, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3064176.3064211.

[51] Robert N. M. Watson, Jessica Clarke, Peter Sewell, Jonathan Woodruff, Simon W. Moore, Graeme Barnes, Richard Grisenthwaite, Kathryn Stacer, Silviu Baranga, and Alexander Richardson. Early performance results from the prototype Morello microarchitecture. Technical Report UCAM-CL-TR-986, University of Cambridge, Computer Laboratory, September 2023. doi:10.48456/tr-986.

[52] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9). Technical Report UCAM-CL-TR-987, University of Cambridge, Computer Laboratory, September 2023. doi:10.48456/tr-987.

[53] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, oct 2020. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf, doi:10.48456/tr-951.

[54] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7). Technical Report UCAM-CL-TR-927, University of Cambridge, Computer Laboratory, 10 2018. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf, doi:10.48456/tr-927.

[55] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, and Peter G. Neumann. CHERI C/C++ Programming Guide. Technical Report UCAM-CL-TR-947, University of Cambridge, Computer Laboratory, June 2020. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf, doi:10.48456/tr-947.

[56] P. R. Wilson and T. G. Moher. A "card-marking" scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *SIGPLAN Not.*, 24(5):87–92, may 1989. doi:10.1145/66068.66077.

[57] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert Norton-Wright, Thomas Bauereiss, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. Cheri concentrate: Practical compressed capabilities. *IEEE Transactions on Computers*, April 2019. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2019tc-cheri-concentrate.pdf.

[58] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (IEEE MICRO 2019)*, MICRO-52, Columbus, Ohio, USA, 10 2019. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201910micro-cheri-temporal-safety.pdf, doi:10.1145/3352460.3358288.

[59] Albert Mingkun Yang and Tobias Wrigstad. Deep dive into zgc: A modern garbage collector in openjdk. *ACM Trans. Program. Lang. Syst.*, 44(4), sep 2022. doi:10.1145/3538532.

[60] Tong Zhang, Dongyoon Lee, and Changhee Jung. Bogo: Buy spatial memory safety, get temporal memory safety (almost) free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 631–644, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3297858.3304017.