Cornucopia: Temporal Safety for CHERI Heaps

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[†], Robert N. M. Watson[*]
[*]University of Cambridge, Cambridge, UK; [†]SRI International, Menlo Park, CA, USA;
[§]Microsoft Research, Cambridge, UK; [‡]Ararat River Consulting, Walnut Creek, CA, USA

***Abstract*—Use-after-free violations of temporal memory safety continue to plague software systems, underpinning many high-impact exploits. The CHERI capability system shows great promise in achieving C and C++ language *spatial memory safety*, preventing out-of-bounds accesses. Enforcing language-level *temporal safety* on CHERI requires *capability revocation*, traditionally achieved either via table lookups (avoided for performance in the CHERI design) or by identifying capabilities in memory to revoke them (similar to a garbage-collector sweep). CHERIvoke, a prior feasibility study, suggested that CHERI's tagged capabilities could make this latter strategy viable, but modeled only architectural limits and did not consider the full implementation or evaluation of the approach.**

Cornucopia is a lightweight capability revocation system for CHERI that implements non-probabilistic C/C++ temporal memory safety for standard heap allocations. It extends the CheriBSD virtual-memory subsystem to track capability flow through memory and provides a concurrent kernel-resident revocation service that is amenable to multi-processor and hardware acceleration. We demonstrate an average overhead of less than 2% and a worst-case of 8.9% for concurrent revocation on compatible SPEC CPU2006 benchmarks on a multi-core CHERI CPU on FPGA, and we validate Cornucopia against the Juliet test suite's corpus of temporally unsafe programs. We test its compatibility with a large corpus of C programs by using a revoking allocator as the system allocator while booting multi-user CheriBSD. Cornucopia is a viable strategy for always-on temporal heap memory safety, suitable for production environments.

I. Introduction

Memory allocators hold a special position within software systems: they govern the object abstraction over memory. Specifically, they allocate regions of heap memory to store language-level objects (such as C-language structs), possibly reusing address space previously occupied by objects no longer in use. Explicitly managed memory allocators – those requiring calls to `free()` rather than utilizing techniques such as garbage collection – open the door to programmer-introduced bugs that violate **heap temporal safety**, by allowing heap objects to *alias*. In programs with these "**use-after-free**" flaws, more accurately termed **use-after-reallocation**, a heap object can be accessed erroneously after it has been `free()`-d and its underlying memory has been reused for some other purpose, such as storing a different object or allocator metadata. These aliased accesses may leak information, damage allocator metadata, or corrupt application data. Use-after-free flaws are prolific in C and C++ programs and have been exploited extensively in real-world systems [11, 46].

While use-after-free heap vulnerabilities are ultimately due to application misuse of the `malloc()` and `free()` interface, complete sanitization of the vast legacy C code base, or even of highly maintained security-critical programs, has proved infeasible. Instead, we turn our attention to the allocator itself, and seek to robustly *mitigate* temporal-safety bugs. Many attempts have been made to mitigate temporal-safety vulnerabilities in existing architectures (discussed in §IX), and they have demonstrated that temporal safety is not possible for today's computers without overheads above 5%, a threshold that has been identified as necessary for universal deployment [46].

CHERI (§II-B) is a promising extension for general-purpose architectures that can replace integer pointers with unforgeable capability pointers. It has recently been adopted by Arm for their Morello prototype processor, SoC, and board [20]. These *capability* pointers enforce spatial safety and are *tagged*, allowing them to be reliably identified in memory, consequently solving one of the major challenges to C temporal-safety systems. CHERIvoke [52] (§II-C and §II-D) proposed an algorithm for temporal safety in CHERI C and C++ using *sweeping revocation*, and modeled key aspects on x86 machines to characterize its performance.

We present Cornucopia, a *practical design and implementation* of the algorithm for sweeping capability revocation that was proposed and modeled in the CHERIvoke paper. Cornucopia is implemented in CheriBSD, a CHERI-aware fork of FreeBSD supporting the CheriABI [13] spatially and referentially safe process environment. Cornucopia consists of an in-kernel service (§IV-A), controlled via shared memory (§IV-B) and new system calls (§IV-C), that uses architectural assistance (§IV-D) to instantiate the CHERIvoke sweep (§IV-E); and of changes to heap allocators for expressing revocation requests to the kernel (§V). Cornucopia also extends the initial CHERIvoke algorithm by introducing *concurrent* revocation, resulting in lower wall-clock overheads.

A. Contributions

In this paper, we:

- Demonstrate how the CHERI architecture, CheriBSD operating system, and CHERI C/C++ toolchain compose to ensure that language-level pointers, implemented as capabilities, can be identified *exactly* via sweeping. This identification facilitates *non-probabilistic* temporal safety for the heap (§II-B, §II-C, and §II-D).

DOI 10.1109/SP40000.2020.00098

- Implement the CHERIvoke algorithm in CheriBSD on the CHERI-MIPS processor [51].
- Extend the CHERIvoke sweeping revocation algorithm to support (1) concurrent sweeping revocation that can be performed in parallel with application threads (§III); (2) key cases in sweeping revocation beyond the user address space, including user capabilities in register files and kernel structures (§IV-A); (3) virtual-memory techniques that facilitate tracking the spread of capabilities to efficiently prune pages from sweeping passes (§IV-D); and (4) asynchronous revocation that enables multiple allocators to safely and efficiently share kernel-managed sweeping resources (Appendices A and B).
- Adapt `dlmalloc` and `snmalloc` to use Cornucopia and develop an allocator-agnostic wrapper that can augment any existing allocator with temporal safety, facilitating an exploration of allocator designs and their implications for Cornucopia temporal safety (§V).
- Evaluate Cornucopia-related OS and allocator changes on an FPGA hardware prototype with respect to a broad range of benchmarks, security-evaluation test cases, and general-purpose applications including a full Unix-like OS. We demonstrate substantially greater efficiency and security than prior work, which validates Cornucopia's architectural viability and practicality (§VII and §VIII).

B. Threat Model

We imagine that a non-malicious application with a potentially flawed implementation is exposed to an adversary who manipulates program input in an attempt to trigger heap object confusion and cause malicious program behavior. This adversary can directly invoke arbitrary sequences of `malloc()` and `free()` and can perform arbitrary loads and stores within these allocations (of both data and capabilities; see §II-B). This roughly corresponds to real-world situations in which an adversary can indirectly influence the heap activity of a target program, such as an adversary-controlled webpage being executed by a target JavaScript runtime or an adversary-controlled process using system calls to influence the heap of a target kernel.

Cornucopia is designed to prevent the adversary from using an old capability to access a region of the heap that has been reallocated. Allocators do not reallocate a region of memory until all capabilities to it have been revoked. Attempts to access a `free()`-d heap memory object will succeed and see a preserved version of the object until revocation; after revocation, they will result in a CHERI trap. In this way, Cornucopia prevents heap aliasing and delivers temporal safety.

We assume that the allocator (and compiler) correctly enforces *spatial* safety by setting bounds on allocated capabilities that are architecturally guaranteed to be monotonically nonincreasing (see §II-B). The adversary is thus able to reduce bounds on capabilities given to them by `malloc()` but not to increase them beyond the bounds of the referenced memory object. We further assume that the allocator is hardened against malformed input to `free()`, such as invalid capabilities or capabilities that do not point to the start of a legitimate allocation, so that the attacker cannot deallocate objects they do not own or otherwise cause internal allocator corruption. We assume that the allocator zeroes memory before it is reallocated, so that adversaries cannot exploit uninitialized reads of heap memory objects. Designing or augmenting memory allocators to meet these assumptions on CHERI platforms is discussed in detail in §V.

Questions of memory-object lifetime and temporal safety also apply to the stack. We follow much of the existing literature [1, 12, 27] in focusing solely on the heap. Heap temporal safety violations are much more commonly exploited [12]. Moreover, stack-allocated object pointers are amenable to static escape analysis [12, 49] and possibly-escaping objects can be relegated to a heap.

C. Non-Goals

Cornucopia is intended for high-performance exploit mitigation, and is not a sanitizer [45] for use as a debugging aid. Sanitizers often aspire to catch even subtle or benign bugs and use probabilistic mechanisms to improve performance. Their performance and security properties make them unsuitable for deployment as a mitigation in production environments. For example, AddressSanitizer [39], the most prolific run-time sanitizer, is probabilistic, and therefore likely to be circumvented in a directed attack, and it incurs high performance overheads, even with hardware assistance [40]. In contrast, Cornucopia absolutely enforces security-critical properties, and it does not rely on any probabilistic mechanisms; it achieves performance by sacrificing the detection of harmless use-after-free accesses that do not yet alias new objects.

Cornucopia prevents old capabilities from being used to access memory but cannot track integers, notably addresses, derived from CHERI capabilities. Cornucopia even preserves the addresses within a revoked capability. These aspects create opportunities for integer confusion issues; for example, a map keying on an object's address would continue to return any previously associated data even after address reuse. Using capabilities whenever possible reduces these opportunities.

While the design of Cornucopia extends to guarding against address-space reuse across `munmap()` calls, we have not yet implemented such protections (but see §X-B). The primary allocators of our study (`dlmalloc` and `snmalloc`) do not release address space.

We do not attempt to detect or mitigate bugs within the allocator, kernel, or any other part of the trusted runtime. The kernel freely manipulates the virtual-memory map, and may access all physical memory. The allocator freely arranges memory into allocations, sets bounds on returned capabilities, and retains access to allocated memory. We assume that these components are constructed in good faith and can be made trustworthy.

II. Background

A. Temporal Safety

Traditionally, a system is said to be **temporally safe** if it does not allow a memory object to be accessed beyond its

declared lifetime. For an explicitly managed C heap, in which a memory object is considered live between when it is allocated by `malloc()` and freed by `free()`, violations of temporal safety have been equated with *use-after-free* flaws, which allow a dead (`free()`-d) heap object to be accessed via a *stale* (or *dangling*) reference. However, these flaws are only exploitable when the memory underlying the dead heap object has been reallocated, for example to store allocator-internal metadata or a new heap object, causing the old and new allocations to *alias*. Thus, a system that prevents this aliasing (i.e., one that prevents *use-after-reallocation* flaws) is also temporally safe.

Heap-memory-object aliasing opens the door to exploits that can result in information leakage, data injection, and control-flow hijacking [46]. As a typical example of the latter, a dangling reference to the dead object might be used to mutate bytes interpreted as a code pointer in the aliased live object.

B. CHERI Capabilities and Spatial Safety

Traditional instruction-set architectures use machine-word pointers to address memory. This makes pointers indistinguishable from integers in execution, making it impossible to determine what memory may be referenced on a finer granularity than the entire address space: any word value can be constructed and used as an address.

CHERI [47] extends traditional instruction-set architectures with a new architectural data type: *unforgeable* and *bounded capabilities*, which can replace machine-word references to memory. CHERI capabilities allow hardware to efficiently enforce bounds atomically on every memory access. CHERI capabilities include permissions, which may restrict their use in load, store, instruction fetch, and other accesses. Unforgeability is enforced using out-of-band tags on each capability-sized memory location; capabilities cannot be arbitrarily fabricated but must be derived monotonically, i.e., from valid capabilities to equal or larger regions with a superset of desired permissions.

CheriABI [13] is a process environment (with attendant runtime code and compiler target) that uses CHERI architectures to express every pointer in C and C++ programs as a capability, completely eliminating raw machine-word addressing and enforcing ubiquitous *spatial safety*. Under CheriABI, every pointer returned from a heap allocator is bounded by the size of the allocation, and references through that pointer can never reach addresses not included in that allocation. Unlike machine-word pointer programs, a CheriABI program can be trivially inspected dynamically at run time to determine the complete set of virtual addresses that may ever be referenced by that program (without acquiring new capabilities from a more privileged source).

CHERI's unforgeability and spatial safety limit an attacker's flexibility, as integers cannot be used as capabilities (pointers), and data pointers cannot be used as code pointers. Despite CheriABI's strong spatial safety, attackers nevertheless can leverage temporal-safety vulnerabilities to corrupt heap allocations with *existing* capabilities.

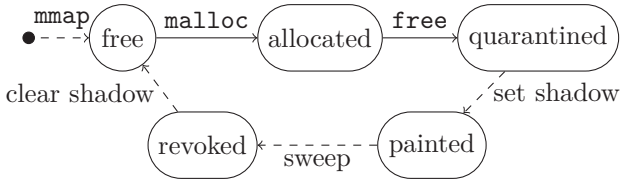

Fig. 1. Address lifecycle in CHERIvoke. Solid edges represent transitions made on behalf of the application; dashed edges represent transitions made within the allocator.

C. Capability Revocation

Revocation, the act of *retracting* granted authority, is a key design concern in any capability system. Historically, a popular approach has been indirection, making each capability access indirect through protected tables, providing a single point for revocation of a capability and those derived from it [38]. However, as capabilities are used far more often than they are revoked, this indirection is most often an unnecessary expense.

In the interest of RISC-style high-performance execution, CHERI does not utilize indirection tables, but uses tagged memory to protect capabilities distributed throughout program memory and register files. This direct capability pointer design matches the typical execution style of C-language programs, which allows pointers to spread throughout program memory and does not require implicit indirection on memory access. However, this model presents challenges for efficient revocation, as capabilities requiring revocation must be found. As CHERI capabilities do not overtly track their provenance, this search must involve *sweeping* through memory.

D. CHERIvoke Algorithm for Sweeping Revocation

CHERIvoke [52] defines a **sweeping revocation** algorithm atop CHERI, periodically scanning all application memory to identify and remove capabilities that authorize access to `free()`-d memory. Fig. 1 shows its address lifecycle; the two central aspects of its design are:

1) A **quarantine** for `free()`-d memory, implemented in the memory allocator, used to defer revocation until a threshold of unusable memory has accumulated.
2) A **shadow bitmap**, updated by allocators and read by the sweeping revoker, with one bit for each word of memory, set if that word is in quarantine.

CHERIvoke can revoke arbitrary subsets of the address space in a single sweep. Its performance should be weakly sensitive only to memory layout and the number of `free()`-d regions being reclaimed, as it sweeps the entire address space. The *architectural* nature of pointers means that CHERIvoke implementations can be oblivious to any language-level types; casts, including to and from C's `void *` and `uintptr_t`, preserve the architectural tags. CHERIvoke also proposed two techniques designed to accelerate memory sweeps: (1) page-level tracking of capabilities using an architectural **capability dirty** bit, and (2) a `CLoadTags` instruction to load capability tags without loading data into caches.

The CHERIvoke experiments left a number of questions unanswered. Performance of the algorithm was predicted using x86 machines, but the algorithm was not implemented on

CHERI hardware, precluding validation against security test suites and full evaluation of the `CLoadTags` optimizations. The CHERIvoke simulation also neglected to consider capabilities beyond user memory (§IV-A) and did not define a software interface to revocation (§IV-B and §IV-C). Finally, while the CHERIvoke *algorithm* has the potential to be run concurrently with program execution, the experiment was limited to predicting sequential performance, leaving the possibility of a significant further reduction in overhead (§IV-D).

III. Cornucopia Overview

Cornucopia is a cooperative effort between a kernel service and user-space allocators. The kernel allocates shadow-space memory and performs sweeping revocation; user-space allocators manage buffering of quarantined allocations, painting the shadow memory, and calling into the kernel to perform revocation.

Cornucopia elevates the CHERIvoke sweeping revoker into the kernel for both correctness (§IV-A) and performance (§IV-D). The shadow bitmap is provided by the kernel as a single reserved region of virtual address space, covering all possible user addresses. The bitmap is the central communication channel from user-space to the revoker; it is *written* by user allocators and *read* by the kernel revoker. For a single act of revocation:

1) The allocator sets every bit in the shadow bitmap corresponding to the words in allocations that have been `free()`-d and are now in quarantine.
2) The kernel revoker sweeps through memory, consulting the shadow bitmap for each capability that is discovered, to determine whether it points into the quarantine.
3) When revocation is complete, the allocator clears the shadow bits of address space now exiting quarantine.

Cornucopia generalizes CHERIvoke's stop-the-world sweep, and implements a *mostly concurrent* multi-sweep kernel revoker. The revoker initially sweeps memory while the application can continue to run, and completes revocation by stopping the world to sweep only those pages that have accepted capability writes during the initial sweep (§IV-D). (Objects `free()`-d during concurrent revocation must wait for the *next* revocation to finish; see Appendix B.)

Cornucopia leaves the policy and work of quarantining to user heap allocators. Heap allocators come in a variety of designs and implementations as a result of many decades of focused performance optimization effort. Delayed reuse, as required by CHERIvoke, is indeed sometimes antithetical to the design of existing allocators, and represents a fundamental change to the optimization criteria. We explore the impact of *wrapping* a Cornucopia shim around three allocators (§VII) and also explore *integration* into two allocators (§VIII and Appendices A and B).

IV. Kernel Revocation Service

Several key functions of Cornucopia are implemented as a FreeBSD kernel service. CheriABI code generally executes according to the principle of least privilege and is not able to access memory for which it does not have references. In contrast, revocation requires extraordinary privilege to identify and modify capabilities from all of the user address space, from register files, and even from kernel structures. Privilege is also required to ensure that the shadow bitmap is mapped at a fixed offset in the address space that it represents to enable efficient lookups. These mapping guarantees are most naturally maintained by the kernel. This avoids a new highly-privileged CheriABI user space service, and preserves the kernel's role in address-space mapping.

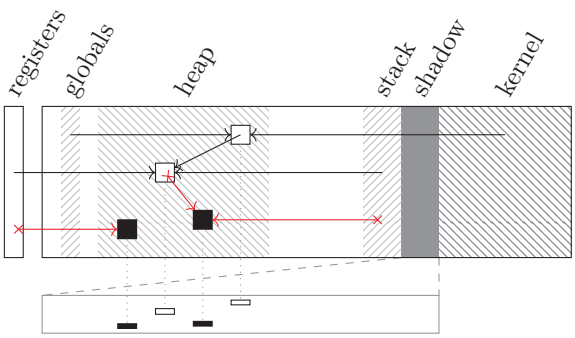

Fig. 2. The shadow bitmap within the process's address space (not to scale) and its correspondence with the rest of the address space. White boxes within the heap represent live allocations with clear shadow bits, while black regions represent `free()`-d space with set shadow bits. Pointers to allocations may be held in registers and in heap, stack, global, and kernel memory. Pointers to `free()`-d address space (red, ×) must be revoked before the referenced space is reused.

A. Beyond User-Space Memory

CHERIvoke's simulation considered the spread of pointers only within user-space memory. However, in order to completely rule out temporal-safety violations, the revoker must reach application pointers that have spread beyond user-space memory. Specifically, the revoker must scan the register files of each thread and must also consider pointers passed to the kernel. Most of these are ephemeral, associated with instances of typical system calls such as `read()`. Our revoker does not scan kernel memory, as capabilities from multiple address spaces appear and are not easily re-associated with their shadow bitmaps. Instead, during revocation, an application's threads will all be held at the system call boundary, ensuring that no system calls are in progress and all such ephemeral capabilities reside in the trap frame, not elsewhere in the kernel. The kernel does, however, hold (or *hoard*) user-space pointers in its own data structures, past system-call invocation. Most often, these are "cookies," merely returned to the user program in response to some event (e.g., `kqueue` triggers) and not directly used by the kernel, but there are more complicated cases, such as asynchronous I/O (`aio`) tasks. Our revoker has specialized code for each hoarding subsystem; while most merely revoke as if the pointer were in user memory, the `aio` handler also cancels associated requests.

B. Maintaining the Shadow Bitmap

Our prototype implementation of Cornucopia reserves a contiguous region of the address space for the shadow bitmap, as shown in Fig. 2, with one bit representing each 16-byte word

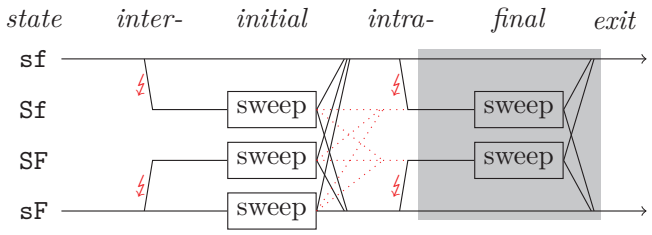

Fig. 3. The capability-dirty states for a page undergoing concurrent revocation, as time advances to the right. The four horizontal bands represent page states, with `s` and `f` designating capability "sweep-clean" and "full-clean" respectively, while `S` and `F` designate corresponding dirty states. State transitions marked with ↯ are cap-dirtying stores. The "sweep" boxes and associated transitions represent the action of the sweeper; dotted lines indicate the effect of application capability stores concurrent with the sweeper. The shaded region runs with the world stopped, which prevents concurrent stores.

of user memory. Despite the current system revoking only heap objects, the shadow-map region is sufficient to represent all valid user addresses.

The user-space allocator paints bits in the shadow bitmap representing words of `free()`-d, quarantined memory (typically on `free()`, but any time prior to revocation suffices). User-space requests a revocation sweep from the kernel, and the kernel service looks up each capability it encounters in the shadow bitmap to determine if it references `free()`-d memory. When the kernel sweep is complete, the user-space allocator zeroes bits of the shadow bitmap representing the words of memory that can now be reallocated to live objects.

For simplicity, we can provide user-space with a capability pointer to the entire shadow bitmap in response to a system call. More principled designs would aim to compartmentalize rights to the shadow bitmap. Where there may be multiple, possibly unrelated, allocators within the same program, each allocator should gain access only to those regions of the shadow space corresponding to address space under that allocator's control; absent control of the address space, access to the shadow bitmap should not be possible. We have implemented such a mechanism and made use of it in our experimental allocator integrations, but not in our wrapper; details are deferred to Appendix A.

C. Kernel Revocation API

Our kernel exports a `caprevoke()` system call that uses the calling thread to sweep the process's memory and kernel-hoarded capabilities. The basic form of this call promises that all memory will be swept and all pending revocations will be performed synchronously before the call returns. While the calling thread is blocked for the duration of the scan, the scan may be otherwise mostly concurrent with other threads of the application, and so allocators can achieve mostly-concurrent revocation using a dedicated worker thread for revocation, as we explore below. Complete revocation requires eventually stopping all application threads to sweep their register files as well as all pages that experienced new capability writes during the initial scan. However, the kernel takes pains to minimize this "stop the world" period.

D. Capability-Dirty Pages and Concurrent Revocation

Two of the high-level optimizations we employ to reduce the performance overhead of sweeping revocation are (1) bypassing pages that are known not to contain capabilities and (2) running the sweep mostly concurrently with the application being swept. Both of these optimizations involve tracking the flow of capabilities through the system, which is made possible by the CHERI-MIPS processor's ability to trap on stores of valid capabilities to specified pages of the virtual address space [52]. By convention, pages that will trap on stores of capability pointers are called *cap-clean*, and pages that allow such stores are called *cap-dirty*. CheriBSD has previously used this mechanism to prevent the propagation of capability pointers to certain pages, e.g. memory-mapped files. Cornucopia refines the use of this mechanism to track pages that are

1) **full-clean**, which the revoker has found to be devoid of capabilities during a revocation pass, and
2) **sweep-clean**, which have not been the target of capability stores since they were last visited by the revoker.

The kernel service for Cornucopia maintains both a full-clean flag and a sweep-clean flag per virtual page.

1) Bypassing Pages

To track capability flow, our Cornucopia kernel modifications initially arrange for capability stores to all pages to cause traps. When handling this trap, the kernel clears the sweep-clean flag for the page and remaps it to henceforth permit stores. During revocation, the revoker examines all pages that have the sweep-clean flag clear (and also examines all pages that have the full-clean flag clear). For each such page, it first sets the sweep-clean flag and arranges for subsequent capability stores to trap then examines and revokes capabilities as appropriate. If it does not encounter any capabilities that it leaves unrevoked on the page, it sets the full-clean flag. Under this scheme, if a revoker encounters a page that is both full-clean and sweep-clean, that page is guaranteed to be devoid of capabilities and can be bypassed. Note that the full-clean flag is only updated after a revocation pass; a page devoid of valid capabilities may thus incorrectly be marked as not full-clean; this causes slightly more work for the revoker but does not affect correctness.

2) Concurrent Revocation

To minimize the amount of time spent with all application threads paused, we take inspiration from live migration [8] and "Mostly Parallel Garbage Collection" [4] and split revocation into two sweeps, which we call *initial* and *final*. The initial sweep can run concurrently with any other application thread. It visits pages by setting the sweep-clean flag, revoking appropriate capabilities, and possibly setting the full-clean flag, as described above. The final sweep requires stopping the threads associated with the current program, synchronizing and/or flushing the caches of all cores involved in running said program, sweeping thread register files, and ransacking kernel hoards of user capabilities. However, it need visit only those pages that have been dirtied since the previous sweep (i.e., it can skip pages that are sweep-clean). The possible flows of a page through revocation are summarized in Fig. 3.

```
1  foreach page in userspace_pages
2    if (is_sweep_clean(page) && is_full_clean(page))
3      next page // skip completely clean
4    if (!initial_pass && is_sweep_clean(page))
5      next page // revisit only sweep-dirty
6
7    // Assuming 128B cache lines & 16B caps, tags ≤ 0xFF.
8    tags = CLoadTags(page)
9    for (line = page; line < page + 4096; line += 128)
10     next_tags = CLoadTags(line + 128)
11     if (next_tags) prefetch(line + 128)
12     // Loop at most 8 times as tags ≤ 0xFF.
13     for (ptr = line; tags != 0; tags >>= 1, ptr += 16)
14       if (tags & 1)
15         cap = load(ptr)
16         if (shadow_bitmap(cap.base)) revoke(cap, ptr)
17     tags = next_tags
```

Listing 1. Pseudo-code for the sweep of user-space memory.

At present, our revoker can run concurrently with application threads, but only one thread at a time can actually perform revocation sweeps. Future work might investigate how the revoker would benefit from internal parallelism by allowing a single revocation sweep to be carried out in parallel by multiple cores. On a smaller scale, revocation passes are also amenable to vectorization and data-level parallelism more generally.

E. The Revoker's Inner Loop

The revoker's memory-scanning inner loop is shown in Listing 1, iterating through pages, cache lines, and capabilities with filters to avoid work at each level to save time, DRAM bandwidth, and cache contention.

On the page level, the full-clean and sweep-clean flags allow the revoker to skip pages that certainly do not contain capabilities to dead objects. For the cache line level, the capability iteration loop will exit when `tags` is zero so that a line with no tags will not be inspected at all. On the capability level, each bit of `tags` is checked before loading a capability and indexing the shadow bitmap.

The `CLoadTags` instruction performs a *non-temporal load* of the capability tag bits of each cache line; that is, `CLoadTags` will respond with tags stored either in tag caches or the centralized tag controller, but will not perturb data caches. Furthermore, the result of `CLoadTags` is used to prefetch data lines that hold capabilities, improving performance. Our cache implementation treats prefetched lines as non-temporal by storing them in a fixed cache *way* of the L2 cache to reduce contention. As a result, our loop writes and over-writes lines from this single cache way when pulling in capability-bearing lines from the address space. Shadow-bitmap accesses are cached as standard data, as we expect to benefit from both temporal and spatial locality – although these are likely to contend with application accesses when sweeping concurrently.

In the concurrent revocation loop, the result from `CLoadTags` is used to authoritatively identify capabilities in a cache line, even though the tags may be stale and cause the sweep to miss a capability. However, the capability store that caused `tags` to become stale will have marked the page sweep-dirty, causing the page to be revisited.

We test the *base* of each capability against the shadow bitmap rather than the *address*. While the current address of any capability is allowed to wander beyond the bounds of the allocation due to pointer arithmetic (even though capabilities cannot be dereferenced out-of-bounds), CHERI guarantees that the base will lie within the original allocation, and will reliably be detected using shadow-bitmap lookup.

If the capability under test must be revoked, an atomic compare-and-swap (CAS) safely replaces it with a tag-cleared version. In initial sweeps, a CAS failure causes the revoker to mark the page sweep-dirty and defer processing of that page until the next sweep. In the final, stop-the-world sweep there will be no CAS failures.

V. User-Space Allocator

Having explored the kernel component of Cornucopia, we turn our attention to the design of Cornucopia user-space heap allocators. Because Cornucopia's temporal safety relies on memory allocators that enforce spatial safety and are hardened against malformed inputs to `free()`, we first discuss the process of making memory allocators CHERI-aware and spatially safe. We then describe the changes necessary to incorporate temporal safety with Cornucopia. These latter changes can be made mostly independently of the base allocator, so we introduce the design of a *Cornucopia wrapper* that allows CHERI-aware allocators to leverage Cornucopia with minimal modification.

A. CHERI-Aware Allocators

Making allocators CHERI-aware involves replacing all pointers with bounded CHERI capabilities, which has a number of implications for allocator design [13]. Most directly, capabilities given out by `malloc()` and accepted by `free()` are tightly bounded to enforce spatial safety for heap memory objects. This means that allocators cannot use pointer arithmetic on capabilities accepted by `free()` to access heap metadata outside of the freed object, as is standard practice in some existing allocators. Instead, CHERI-aware allocators internally maintain capabilities from `mmap()` (which have a special software permission bit, `VMMAP`, set, indicating their provenance) to all of the memory they manage, including metadata. The allocator uses these to derive more tightly bounded capabilities for `malloc()`'s return values and to access allocator metadata in `free()`. Capabilities returned to an application by `malloc()` must have the `VMMAP` permission cleared: in order to support the correct functioning of userspace allocators, `VMMAP`-bearing capabilities are not subject to revocation.

To provide a robust foundation for spatial safety, CHERI-aware allocators must also be hardened against invalid inputs to `free()`. Specifically, capabilities passed to `free()` must be tagged, have appropriate permissions, and point legitimately (within bounds) to the beginning of a non-free region of memory that was previously returned by `malloc()`. Otherwise, an adversary might be able to deallocate objects they do not own by passing untagged capabilities or capabilities with out-of-bounds addresses to `free()`; alternatively, they might be able to corrupt the allocator's internal state by `free()`-ing capabilities with modified bases or addresses. Corruption of allocator-internal state can result in capabilities being returned by `malloc()` with bounds that violate desired safety properties.

For a CHERI-aware allocator, we stipulate that the `free()` function must not accept capabilities that point to the middle of an allocation or untagged or inappropriately permissioned capabilities.

We modified our CHERI-aware memory allocators to perform these checks. Validation of tags and permissions is straightforward, and the exact method for validating that a capability points to the beginning of an allocation depends on the allocator's implementation. In slab allocators, such as `snmalloc`, `free()` first checks that the offset of the passed-in capability is zero. This guarantees that the capability's address is not out of bounds, and `snmalloc` does not return capabilities with non-zero offset, so the only possible valid offset is zero. `free()` then checks that the base (address) of the capability is both within a slab and at a position consistent with the beginning of an allocation. In allocators that mix object sizes within an arena, such as `dlmalloc`, `free()` may validate the passed-in capability against a copy of the original capability stored in metadata *outside* the bounds of the allocation.

Some existing allocators, like `snmalloc`, already have the option to be built with checks that validate `free()`-d pointers' pointing to the beginning of an allocation. Informal testing suggested that such checks add a 1-2% cycle overhead on our evaluation platform.

B. Adding Temporal Safety with Cornucopia

A few modifications are necessary to turn a spatially safe CHERI-aware allocator into a temporally safe Cornucopia allocator. When memory objects are `free()`-d, they must be put into a quarantine state until the next revocation pass has completed. The allocator must also keep track of the total amount of quarantined memory, and once the quarantine size passes some threshold, the allocator must engage the kernel's revocation service by making the `caprevoke()` system call as described above. The allocator must manage the shadow bitmap, ensuring that regions corresponding to quarantined memory are set before the revocation pass and cleared afterwards. Once the quarantined memory has been revoked, it returns to the free state and can be safely reallocated.

For a memory allocator to be safe in the face of uninitialized reads on the heap, it must zero memory before reallocation. This issue has long been understood and appears in the Common Criteria as Residual Information Protection [10]. This is orthogonal to techniques to prevent heap aliasing; many existing temporal-safety studies have not evaluated how their techniques compose with memory zeroing. We focus on the cost of revocation, but do measure the cost of zeroing in §VII-A and §VII-C.

Key design choices in Cornucopia allocators include the quarantine threshold value, the data structures used to track quarantined memory objects, and whether it is possible to coalesce objects in quarantine to reduce the work involved in returning them to the free state. The optimal choices (as well as the effect of quarantining in general) can depend on the design of the base CHERI-aware allocator, although a single policy implemented as a universal wrapper is sufficient in most cases.

C. A Wrapper for Temporal Safety

CHERI-aware allocators can be augmented with Cornucopia in a minimally invasive way. Our temporally safe allocator-wrapper is easy to use, offers full protection, and can be configured in many ways to explore the design space of temporally safe Cornucopia allocators. It incurs only slight inefficiencies relative to integrating Cornucopia temporal safety directly into an allocator.

The Cornucopia wrapper is a shared library that exports allocation functions including `malloc()` and `new` and deallocation functions including `free()` and `delete`. When the wrapper is preloaded by the run-time linker, its exported functions call the corresponding functions of the underlying allocator and perform the operations necessary to provide Cornucopia temporal safety. For simplicity, we will focus our discussion on `malloc()` and `free()`, as other allocation and deallocation functions are similar.

The only demand the Cornucopia wrapper makes on underlying CHERI-aware allocators is that they export a function called `malloc_underlying_allocation()` that takes in a capability corresponding to some allocation, validates that it is not not malformed (as described in §V-A), and returns a capability whose bounds match those that were given for the capability in the original call to `malloc()`. This information allows the wrapper to track the quarantine size correctly and to revoke references to any part of the original allocation; the bounds of the capability passed to the wrapper's `free()` function could have been shrunk by the application.

The wrapper's `malloc()` function calls the underlying allocator's `malloc()`, increments a counter that tracks the total heap size by the size of the returned capability, and returns the capability. The wrapper's `free()` function validates the passed-in capability using `malloc_underlying_allocation()`, increments a counter that tracks the quarantine size by the size of the returned capability, then adds the returned capability to the quarantine list, which is implemented as a linked list of arrays that the wrapper manages using `mmap()`.

The wrapper checks whether to initiate a revocation pass on calls to `malloc()`. When a revocation pass is initiated, the wrapper iterates through the quarantine list and paints the shadow bitmap appropriately, calls the `caprevoke()` system call, iterates through the quarantine list to clear the shadow bitmap and free the quarantined allocations back to the underlying allocator, subtracts the quarantine size from the heap size, and resets the quarantine size. Operations that interact with the shadow bitmap use lightweight synchronization to guarantee correctness even in multi-threaded scenarios. Revocation can be performed asynchronously in an offloaded thread or synchronously in the application thread. If revocation is offloaded and performed asynchronously, the wrapper limits the work that must be performed in the application thread. While asynchronous revocation is in progress, the application thread can continue working and filling up a second quarantine.

The application thread blocks if this second quarantine passes the revocation threshold while asynchronous revocation is still in progress.

One cost of using the Cornucopia wrapper relative to direct allocator integration is that operations to validate or look up metadata for each freed capability must be performed twice: once by `malloc_underlying_allocation()` and once when the capability is freed back to the underlying allocator. In an integrated design, the allocator might avoid this duplicated work. Another cost is that data structures for storing the quarantine list must be maintained externally to the allocator, whereas an integrated design might reuse existing allocator metadata to track the quarantine list. Despite these costs, the wrapper is a useful way to easily deploy temporal safety and evaluate the design space.

VI. Experimental Setup

A. System Configuration

All benchmarks are run on a dual-core configuration of the 64-bit CHERI-MIPS processor [51] synthesized for the Stratix IV FPGA at 50 MHz. The pipelines are in-order and single-issue, roughly similar to the ARM7TDMI. The system has 32-KiB per-core L1 I and D caches, and a 256-KiB L2 cache shared by both cores. All caches are set-associative in a configuration matching widely-shipped ARM Cortex A53 configurations. Performance and memory scaling are broadly similar to these commercial implementations. CHERI is configured for 128-bit capabilities [50].

Experiments are run on the SPEC CPU2006 [21] suite; while our headline figures come from the *ref* set, more complex overhead breakdowns were performed on *test*. We ran all benchmarks that executed under CheriABI [13] on the CHERI-MIPS FPGA (all benchmarks that were compatible with FreeBSD MIPS also ran under CheriABI). We set the quarantine limit to 25% of the total heap.

B. Underlying allocators

We use CHERI-aware forks of the existing `dlmalloc`, `jemalloc`, and `snmalloc` allocators. We investigate both wrapped and integrated versions of `dlmalloc` and `snmalloc`, and a wrapped version of `jemalloc`. Where only one result is presented, `snmalloc` is used since it is the fastest unmodified allocator. `dlmalloc` [26] is a coalescing free list allocator that was the default allocator for many Linux distributions. It is no longer considered state-of-the-art, but it remains in broad use. Its constant-time coalescing algorithm naturally aggregates `free()`-d memory into contiguous `free()`-d regions. This opens opportunities for releasing physical pages that are entirely in quarantine to be reused by the system, which we explore in §VIII-A. `jemalloc` [16] is a state-of-the-art slab allocator that was created for FreeBSD but is now widely used, including in Android. It optimizes for multi-threaded use by maintaining per-thread caches of recently `free()`-d memory that can be reallocated without costly synchronization. `snmalloc` [28] is a new allocator design from Microsoft Research intended for heavily multi-threaded workloads; it uses dedicated, per-thread heap allocators, each of which maintain slab-like arenas, and uses message-passing for cross-thread work.

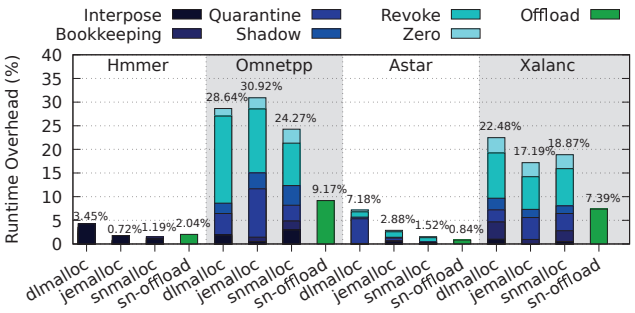

Fig. 4. Run-time overheads for revocation using a generic wrapper around `dlmalloc`, `jemalloc`, or `snmalloc` on the **worst**-performing, CHERI-compatible SPEC CPU2006 `test` workloads. The baseline is pure-capability execution without temporal safety. The fastest case is `snmalloc` offloading revocation to a separate thread. The offload case performs revocation without memory zeroing.

VII. Evaluation

Here we evaluate the Cornucopia wrapper. We see that it performs well across many different allocators even on the most challenging benchmarks in the SPEC CPU2006 suite, with costs mainly coming from the revocation sweep and quarantine bookkeeping. Cornucopia improves significantly on results from the literature, at 5.8% overhead using stop-the-world revocation, and less than 2% overhead when offloaded. Re-evaluated on our FPGA system, the Boehm Garbage Collector and AddressSanitizer face 19× and 31× larger overheads, respectively.

A. Decomposing Overheads

Fig. 4 shows the sources of the cycle overheads of three allocators behind our Cornucopia wrapper. This study used the test workloads of the four **worst** performing of the eight CHERI-compatible SPEC CPU2006 integer benchmarks, and set the quarantine size to 25% of the heap. We decompose the overheads of Cornucopia into: (1) the wrapper interposing on the allocator (2) bookkeeping: validating wrapper function inputs (3) quarantining: delayed memory reuse (4) managing the shadow bitmap (5) sweeping revocation: *temporal safety* (6) zeroing memory before allocation. Finally, we include the overhead for offloading revocation to a dedicated thread under `snmalloc`, which had the best absolute performance of the three allocators.

We observe an interesting variation in performance overheads between `jemalloc` and `snmalloc`. `jemalloc` and `snmalloc` achieve very similar performance on the baseline (while `dlmalloc` is notably slower in allocation-intensive workloads), but `jemalloc` develops a far greater overhead for quarantining allocations and `free()`-ing them in batches. This is due to the fast-path in `jemalloc` being the thread-local reuse of recently `free()`-d memory. `free()`-ing memory in large batches disrupts this optimization, causing `jemalloc` to resort to its slower path far more often. `snmalloc` uses entire thread-exclusive slabs for its fast path, which is much less disturbed by batched reuse. This allows `snmalloc` to hold the overhead due to the quarantine buffer below 8.2% in the worst case.

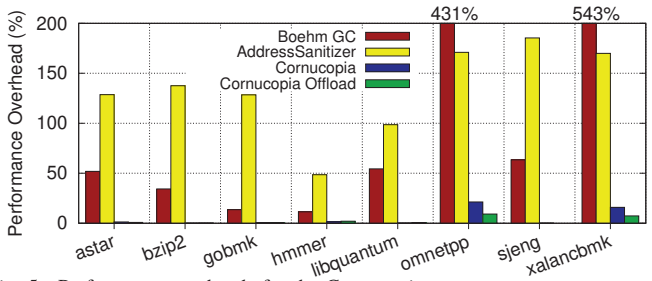

Fig. 5. Performance overheads for the Cornucopia `snmalloc` wrapper versus Boehm GC and AddressSanitizer executing on the MIPS FPGA platform. The baseline for Cornucopia is CheriABI, and the baseline for the others is MIPS. MIPS and CheriABI are generally within 5% of each other [13]. All benchmarks are run in their `test` configuration.

The worst case for `snmalloc` was the `omnetpp` benchmark, which has an offloaded overhead of 9.2%. If the kernel sweeping revocation passes were completely independent of the application, we could expect the performance with an offload thread to approximate the 8.2% cost of quarantine. Offloaded revocation incurs 1 percentage point additional overhead, indicating a reasonably small final sweep with program threads stopped (see §VII-C and Fig. 10).

Sequential memory zeroing atop a revoking `snmalloc` incurs an additional overhead of less than 3% for the benchmarks of Fig. 4. Most of our subsequent measurements will focus on revocation and will run without zeroing.

B. Cornucopia versus Boehm GC and AddressSanitizer

In Fig. 5, we compare the Cornucopia-wrapper-with-`snmalloc` results against MIPS ports of AddressSanitizer [39] and Boehm GC [4] executing the CHERI-compatible benchmarks on our FPGA system. While the purpose and design of Cornucopia differs significantly from these tools, the performance comparison is instructive.

AddressSanitizer's primary performance overhead is its memory-validity checks that scale with data access, not with deallocation. As a result its highest overheads do not coincide with those of Boehm GC and Cornucopia, but it maintains a relatively predictable overhead of around 100%. This overhead is higher than reported on x86, which appears to be due to the shadow-map checking instructions bloating code due to the low instruction density of the MIPS ISA, combined with the scalar, in-order core being unable to execute these instructions in parallel. The overhead of AddressSanitizer on our MIPS implementation on FPGA has a geometric mean of 31× the overhead of single-threaded Cornucopia, and of 56× when offloaded.

Boehm GC overheads scale with deallocation and memory size, similarly to Cornucopia, but the work that must be done to infer memory management without reliable pointer identification or quarantine metadata is formidable. The Boehm GC has a geometric mean of 19× the overhead of single-threaded Cornucopia, and of 37× with offload.

While these three tools are related, they have been designed with different motivations. AddressSanitizer intends to efficiently detect memory safety bugs with high probability, and so prioritizes detection latency, while Cornucopia and Boehm GC intend to prevent illegal behavior. Garbage collectors expend effort to automate memory management by inferring freed memory, while Cornucopia (and C in general) leverages manual management for performance.

While the CHERI architecture could accelerate either of these algorithms, particularly Boehm GC, it is clear that these types of algorithms cannot approach the efficiency of Cornucopia without hardware support, as both must do significantly more work.

C. General Overheads

Fig. 6 shows the performance overhead of Cornucopia, using `snmalloc` on SPEC CPU2006 in *`ref`* mode, in comparison with numbers reported by software systems in the literature [12, 25, 29]. The worst-case benchmark is `omnetpp`, which has a sequential overhead of 26.2% and an offloaded overhead of 8.9%. These benchmarks yield a geometric mean sequential overhead of 5.8% and only 1.9% with offload. By comparison, other techniques in the literature suffer significantly higher average performance overheads: evaluated on the same subset of benchmarks that run on our MIPS system, we see overheads of 45% for DangSan, which stores lists of pointer locations, 16% for pSweeper [29], which stores a single list of pointer locations and offloads revocation, 52% for Oscar, which uses the page table to revoke allocations, 18% for CRCount [41], which uses reference counting, and 68% for BOGO [54], which is built on top of Intel MPX. These techniques are also unable to give reliable memory consumption guarantees [52], and so suffer significantly in both the average and worst cases. While we should note that the numbers reported are run on vastly different platforms from ours, the great discrepancy in overheads demonstrates the scale of advantage that CHERI can provide for temporal safety when combined with an efficient algorithm.

Fig. 7 shows the increase in DRAM traffic, both with and without offloading. The resulting DRAM overheads are similar in magnitude to sequential performance overhead, and are higher for the offload case due to two-phase sweeping and due to L2 cache contention during the sweep. As described in §IV-E, the sweep makes some effort to reduce cache disruption to the main program, but disruption is nevertheless apparent in concurrent results.

Fig. 8 shows the rate of sweeps performed by each application. The number of sweeps during each benchmark varies based on the rate that memory is `free()`-d. As offloading introduces nondeterminism, the total space `free()`-d during a sweep may be sufficient to fill this additional buffer; when this happens, the benchmark blocks and waits for the sweep to finish. As this increases the size of the heap, the application subsequently performs fewer sweeps. `astar` alone reliably triggers this behavior, while `omnetpp` and `xalancbmk` do so only occasionally, and the remainder, never. Fig. 8 shows the most common case for each benchmark with error bars showing observed ranges.

Fig. 9 charts the memory pages used by each benchmark process with Cornucopia protection. Our revocation sweep is triggered when 25% of the heap is in quarantine so

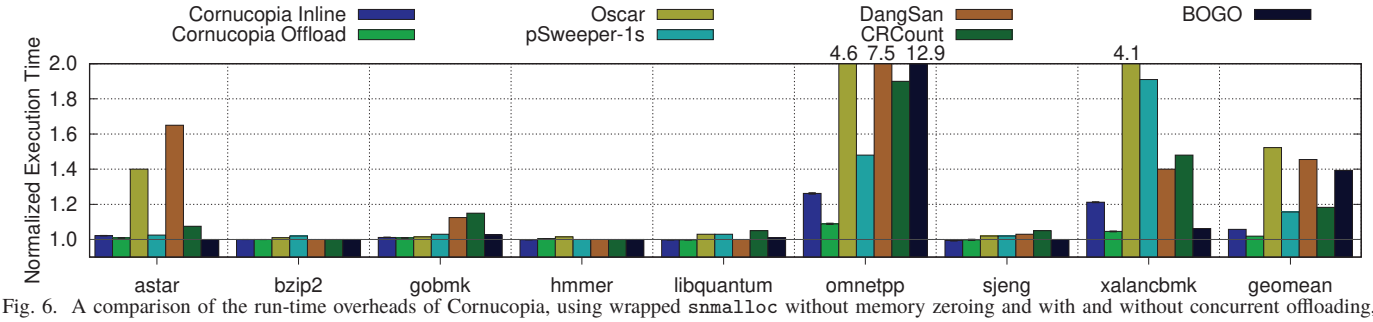

Fig. 6. A comparison of the run-time overheads of Cornucopia, using wrapped `snmalloc` without memory zeroing and with and without concurrent offloading, against other techniques in the literature [12, 29, 25, 41, 54]. Cornucopia results are from the SPEC CPU2006 `ref` set. Error bars are the standard error normalized by the baseline mean and, here, are always less than half a percent.

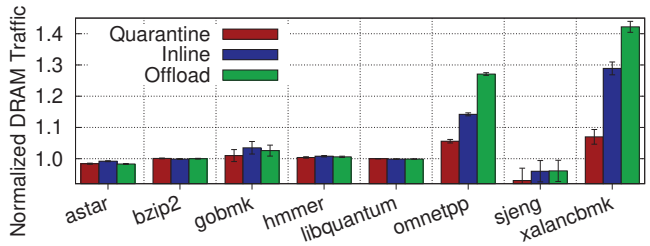

Fig. 7. DRAM traffic overheads for Cornucopia, both single-threaded and in its concurrent offloaded form. Benchmarks used `ref` workloads.

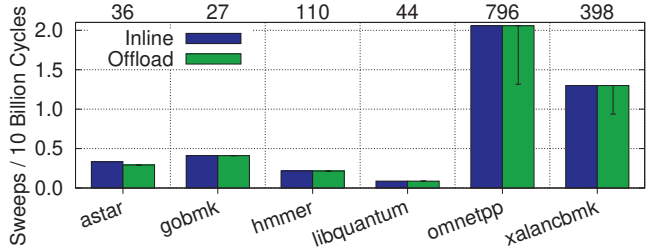

Fig. 8. Frequency of revocation sweeps for Cornucopia: sweeps over *baseline* `ref` mean cycle counts. Above the chart is the count of revocation sweeps performed for each benchmark's inline configuration. `bzip2` and `sjeng` do not sweep.

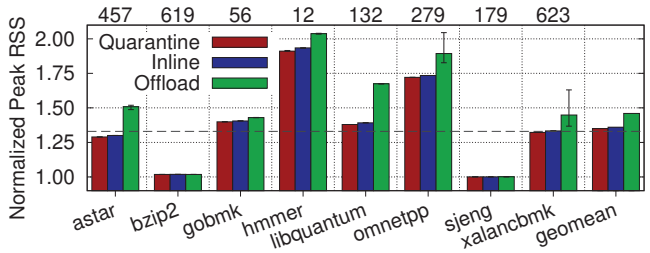

Fig. 9. Mean peak memory footprint (RSS) relative to baseline mean (shown above each benchmark, in MiB), on SPEC `ref` workloads. Error bars show observed extremal values. The policy target of 33% is shown, dashed. `bzip2` and `sjeng` do not sweep.

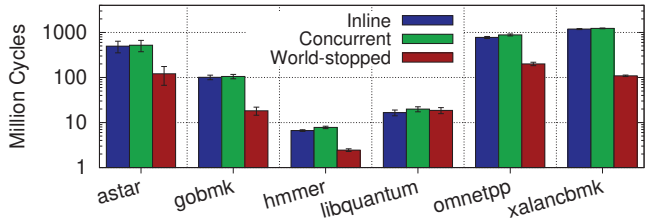

Fig. 10. Mean (and std. dev.) time per revocation pass. When offloading, the concurrent pass is slightly longer than the non-offload sweep, but the second, world-stopped pass is dramatically shorter.

the target memory overhead is 33%. The single-threaded performance generally tracks this target with a geometric mean 36%, and offloaded revocation has a slightly higher memory overhead due to memory continuing to be quarantined during revocation. `hmmer` and `omnetpp` have an unusually high memory overhead, which we tentatively attribute to fragmentation due to interaction between quarantine and slab-based allocation: insufficient page reuse between slabs leaves unused pages in multiple slabs. Further, the generic quarantine metadata management used by our wrapper is inefficient for modern allocators such as `snmalloc`, where data layout choices in the allocator itself would allow us to more efficiently store secure metadata.

Fig. 10 shows the time taken by an average revocation sweep for each benchmark. We observe that (1) two-phase revocation is working as intended, reducing pause times to 28% (geometric mean) of the non-offloaded sweep time, and (2) the pause time is 19% (geometric mean) of the total time taken across both initial and final phases.

Last, while we leave detailed analysis to future work, we can nevertheless informally quantify the costs of memory zeroing. On our worst-case benchmarks of Fig. 4, but now in their `ref` configurations, we observe a few points of note. (1) Zeroing on transition from quarantined to free adds up to 1.0% cycle overhead (on `omnetpp`, relative to the *non-revoking* baseline) but adds a *substantial* 21.6% DRAM access overhead (`xalancbmk`). (2) Zeroing on allocation, which takes place on the allocating (i.e., benchmark) thread, is more expensive, cycle-wise, with an observed maximum of 3.3% (`omnetpp`) but a substantially lower maximum DRAM access overhead of 1.1% (`xalancbmk`). We expect future optimizations, such as being able to release quarantined pages (§VIII-A), to lower these overheads.

D. Security

We used the Juliet Test Suite [6, 44] of C/C++ programs to evaluate Cornucopia's effectiveness in mitigating temporal-safety attacks using the wrapper implementation, testing against the classes that are relevant to temporal safety: CWE 415, Double Free; and CWE 416, Use After Free.

Each pair of good (bug-free) and bad (bug-known) executables was run on the CHERI-MIPS FPGA using CheriBSD's version of `jemalloc` that supports spatial safety, and was also run using the Cornucopia wrapper that adds temporal safety. To test security in the limit, the Cornucopia wrapper was configured with a quarantine size of zero. This caused revocation to

be performed on every call to `free()` so that temporal-safety violations immediately manifest as CHERI exceptions. For all 818 test cases in CWE 415, and all 393 in CWE 416, the good configuration executed successfully, while the bad configuration executed successfully with the unwrapped version of `jemalloc` (indicating that a temporal safety violation was allowed to occur) and terminated abnormally with the wrapped version (indicating that the temporal-safety violation was mitigated). For use-after-free tests, the bad configurations caused CHERI exceptions by attempting to use a revoked capability. For double-free tests, the bad configurations attempted to free a revoked capability, and we configured the wrapper to exit abnormally in this case.

For comparison, the test executables were also run on an x86 host with and without AddressSanitizer [39]. The results were the same: for all test cases, the good executables executed successfully in both configurations, while the bad executables executed successfully without AddressSanitizer but resulted in an error with AddressSanitizer. To evaluate our hardiness against malformed inputs to `free()`, we evaluated a wrapped version of `snmalloc` against tests for CWE 761, Free of Pointer Not at Start of Buffer, and confirmed that it correctly detected malformed inputs, matching the behavior of AddressSanitizer.

VIII. Additional Experiments

While the Cornucopia wrapper represents the most straightforward approach to implementing Cornucopia in a user-space allocator, deep modification of the allocator can enable further performance improvements and safer engineering principles. We have explored several paths while integrating Cornucopia into `snmalloc` and `dlmalloc`:

- extending `dlmalloc` to coalesce quarantined regions to enable early reuse of physical pages (§VIII-A),
- using an integrated `dlmalloc` as the system allocator for CheriBSD and booting to multi-user mode (§VIII-B),
- enforcing controlled access to the shadow bitmap to safely share the revocation service between multiple allocators in an address space (Appendix A),
- extending Cornucopia to enable *lightweight synchronization* between different allocators (Appendix B), and
- comparing Cornucopia to the *modeled* overheads predicted by CHERIvoke (Appendix C).
- lowering pause times by sweeping more (Appendix D).

A. Cornucopia `dlmalloc`: Fast Page Invalidaton

We adapted CHERIvoke's experimental `dlmalloc` to use CHERI bounds and to perform Cornucopia revocation. The `dlmalloc` allocator naturally coalesces `free()`-d allocations, and our adaptation of `dlmalloc` capitalizes on this mechanism to coalesce allocations in quarantine and to discover when a `free()` call results in entire pages being quarantined. In such circumstances, our `dlmalloc` advises the kernel that such pages are no longer needed, so that the kernel may reuse the *physical* memory. Our allocator can be configured not to count these pages against the quarantine threshold in order to reduce sweeping frequency.

Fig. 11 shows such reductions in the amount of sweeping work for a benchmark that replays traces of `malloc()`-s and

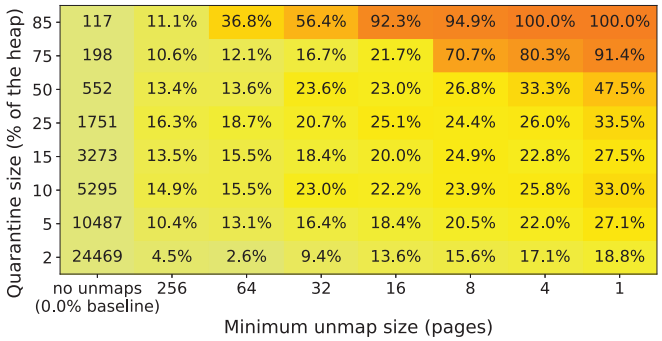

Fig. 11. Percentage decrease in number of revocation sweeps due to coalescing and unmapping quarantined memory when replaying a trace of Chromium allocations. Lower minimum unmap size (X-axis) means higher run-time overhead due to more frequent `munmap()` system calls. Higher quarantine percentage-size (Y-axis) means approx. a factor of $\frac{y}{100-y}$ higher memory overhead. The left-most column shows the absolute number of revocation sweeps that is the baseline for each row. The percentage reductions directly translate to reductions in the total run-time overhead of sweeping – higher is better.

`free()`-s as recorded from a Chromium workload simulating multi-tabbed browsing. In the simulation, up to 64 webpages were assigned per tab and every tab loaded some of them in a round, one webpage immediately after the other, tab after tab. The number of tabs was randomly adjusted up or down between 1 – 32 once a mean number of webpages were loaded across all tabs. 800 webpages were loaded in total, over HTTP and from a local filesystem. The set of webpages was a selection of WebKit's Layout Tests and simple documentation webpages, that are not JavaScript intensive (as the V8 JS engine has its own heap). Chromium was configured to use our allocator for the subsystems that support it and to run in a single process. With the protections afforded by CheriABI and Cornucopia, it may be possible for Chromium to maintain the required protection and fault isolation in this configuration.

We find that a quarantine threshold policy of 25% allows a 33.5% reduction in the number of sweeps by coalescing and unmapping at a single-page granularity. Larger quarantines are able to avoid sweeping altogether, as quarantined memory entirely coalesces into contiguous pages, despite over 50 GiB being allocated across the trace.

Notifying the kernel about quarantined pages incurs the cost of system calls to manipulate virtual-memory structures, and this cost scales with granularity. Fig. 11 shows that a single-page granularity achieves the greatest reduction in sweeps, but a practical design may choose a multi-page granularity or a batch system call to amortize the cost of virtual-memory manipulation.

B. Cornucopia From Boot

We have booted CheriBSD to multi-user login and a shell prompt using a Cornucopia-integrated `dlmalloc` as the `libc` `malloc()` implementation and with the "bootstrapping" allocators inside `rtld` and for thread-local storage also aware of revocation. These latter allocators force a revocation only in the rare event that they must request more memory from the kernel; otherwise, they *passively* reclaim memory using the epoch counters of Appendix B.

When `dlmalloc`'s quarantine was set to the default policy, the machine booted successfully and no issues were observed. When the quarantine was eliminated, so that every `free()` triggered a revocation, the system was slow, but short-lived use-after-free bugs were observed in `sh` and `cron`. Triaging these bugs remains work in progress. To debug these issues, as well as bugs within Cornucopia itself, we use standard systems-level debugging tools: the CHERI-MIPS processor can generate instruction traces and `gdb` works within CheriBSD.

IX. Related Work

A. Nullification

Cornucopia is not the first technique to use pointer nullification to eliminate use-after-free attacks. DangNull [27] and FreeSentry [53] store stacks of use locations with each allocation, to allow nullification on deallocation. DangSan [25] optimizes this approach to deal with the complexities of multithreaded workloads. PSweeper [29] combines these into one large unified list of pointers, which it continuously sweeps through on another thread. Cornucopia can achieve much lower memory and performance overheads than these techniques because pointers are architectural features [23] that do not need to be separately logged – instead, we can efficiently sweep through memory.

B. Debugging Sanitizers

Many sanitizers are available to assist in debugging by identifying the use of `free()`-d memory. Perhaps the most well-known is AddressSanitizer [39], which instruments loads and stores by looking them up in a shadow poison region, set to mark the edges of accessible regions as well as unallocated space. Cornucopia uses a similar shadow, but reduces overheads by removing the need to instrument memory accesses.

These debugging sanitizers have also seen improvements from hardware support. In hardware, techniques such as SPARC ADI [36, 24] and Arm MTE [18] use limited wrap-around versioning of regions and pointers to help identify, but not prevent, use-after-free (but see §X-E). Hardware-assisted AddressSanitizer [40] uses the top few pointer bits, ignored by memory accesses on AArch64, to store version numbers.

C. Probabilistic Reuse Techniques

Rather than eliminating use-after-free attacks, some allocators have reduced overheads by probabilistically delaying reuse of memory, to force an attacker to reallocate large amounts of data to successfully reallocate the victim's space. This approach is taken by FreeGuard [42], DieHard [3], and DieHarder [34]. Cling [2] uses this same strategy, along with more permissive short-term reuse within a call-site and allocation size, to reduce overheads.

D. Combining Spatial and Temporal Safety

In this paper, we observe that the spatial safety properties of CHERI give both performance benefits and stronger guarantees to temporal-safety mechanisms. Other work has exploited the same property: for example, BOGO [54] builds temporal safety atop the Intel MPX mechanism [22]. However, their different algorithmic choices in the allocator result in larger overheads of 1.6× for SPEC CPU2006.

Watchdog [30] uses labeling in hardware to provide spatial and temporal safety. Software techniques for temporal and spatial safety combined include MemSafe [43], CETS [31], SoftBound [32], FailSafeC [35], and CCured [33].

CHERIvoke [52] observed that buffering `free()`-d memory in quarantine can greatly amortize the work of invalidating dangling pointers to render address space suitable for reallocation. We extend this methodology to augment full allocators and detail the techniques necessary for implementation on a real CHERI system, including the complexities of multiple different production allocators, interaction with the kernel, and concurrent execution.

E. Garbage Collection

Garbage collectors [5], though often more expensive than manual deallocation [52], are a method of protecting against use-after-free errors by preventing the programmer from freeing data. It is instructive to compare the CHERIvoke algorithm used by Cornucopia to the mark and sweep phases of the standard garbage collector model. CHERIvoke performs a sweep without the mark phase, naturally decreasing overhead. Furthermore, explicit deallocation allows CHERIvoke to schedule sweeps to maintain a specified memory overhead without wasting effort when sweeps are not required. Finally, the CHERIvoke algorithm does not suffer memory leaks due to residual references, as it does not rely on the existence of references to determine the liveness of objects, but actively eliminates such references.

Project Snowflake [37] optimizes relative to garbage collection for high-level languages by using safe manual memory management for a proportion of allocations and also using compaction and relocation to improve performance. While C does not allow these forms of relocation (as they violate commonly used techniques involving visible pointer addresses), the Cornucopia service and CHERI primitives could further accelerate systems for higher-level languages. ManagedC/TruffleC [19] execute C using the Java virtual machine to inherit the memory safety guarantees of Java, including bounds-checked structures and garbage collection before memory reuse. CRCount [41] emulates a tagged pointer shadow in software similar to CHERI's hardware implementation, and uses this to build a reference-counting technique to prevent use-after-free.

F. Page Protection Mechanisms

Some systems page permissions to prevent use-after-free attacks by allocating each address only once, and allocating only a single object per page. One example of such a tool is Electric Fence [15]. Dhurjati and Adve [14] extend this idea by allowing reuse of physical addresses; specifically, by unmapping the unique virtual address of each object and mapping multiple virtual addresses to the same page. Oscar [12] further extends the idea by removing the need for compiler support. Cornucopia allows the use of a similar technique for large allocations (§VIII-A), but avoids the high TLB pressure of such techniques for small objects by allowing reuse of old virtual addresses post-revocation.

X. Future Work

A. Distinguishing Authority and Information Flows

In this paper we have reported the overhead of allocators returning "uninitialized" memory and allocators returning fully zeroed memory. One could accelerate memory reinitialization by distinguishing between authority leakage and information leakage: when memory is left uninitialized on reallocation, information is leaked; if this memory also contains capabilities, authority is also leaked. A `CClearTags` counterpart of `CLoadTags` could minimize memory traffic and cache impact of closing authority leaks.

B. Address-Space Reclamation

The kernel cannot safely reissue virtual address space that has previously been `munmap()`-ed, as it cannot be certain that the new capability returned from `mmap()` is the only reference to that memory in the address space. We have designed, but not implemented, extensions to Cornucopia to safely facilitate such reuse.

C. Further Reducing Pause Times

While Cornucopia's concurrent sweep greatly reduces application pause times, they remain quite high for some workloads (recall Fig. 10), and much more could be done to ensure interactive applications experience minimal interruption. Given proper architectural assistance, sweeping revocation could allow the application to run but trap on any read of a page that has yet to be swept [9]. The revoker would then sweep only that page (and possibly those nearby), reducing application pause durations.

D. Deferred Page Sweeping

Efficient support for swapping pages to disk requires the ability to defer page sweeping for out-of-memory capability-bearing pages. To process historic pages, we need to replay the sweep of that page using the bitmap as of each sweep since that page was removed from memory, which may require storing (compressed) copies of older bitmaps.

E. Data Tagging Opportunities

In recent years, proposals such as ARM MTE [18] and SPARC M7's ADI [24, 36] have emerged. These color both memory and pointers, and upon dereference, require that the pointer and target colors match. Due to small color spaces and permissive recoloring, these offer *probabilistic* protection against accidental misuse. However, in composition with CHERI (e.g., [47, §D.9]), such memory coloring may significantly strengthen and accelerate Cornucopia.

F. Revocation of Other Identifiers

In CHERI, (virtual) memory addresses are but one kind of identifier protected with architectural capabilities. Other spaces explicitly instantiated in the architecture are *object types* [7, 47] and *compartment identifiers* [48, 47]. Sweeping revocation can be extended to reclaim these as well.

G. Hierarchical Revocation

CHERIvoke and Cornucopia achieve only a limited form of capability revocation, as they distinguish only between capabilities of the TCB (in the kernel or bearing `VMMAP`) and those held by the rest of an application. New (architectural) mechanisms must be developed to enable revocation to selectively undo delegations outside the TCB.

XI. Conclusion

We have presented Cornucopia, a system for heap temporal-safety built atop CHERI's spatial safety and capability model that is faster and stronger than other known techniques. We demonstrated that optimizations to sweeping revocation can bring overheads below 2% on average, suggesting that temporal safety can be enabled by default for C and C++ programs on future CHERI platforms.

XII. Acknowledgements

We thank our colleagues Hesham Almatary, Jonathan Anderson, Ross Anderson, David Brazdil, Ruslan Bukin, Gregory Chadwick, Nirav Dave, Lawrence Esswood, Tim Harris, Robert Kovacsics, Bob Laddaga, Ben Laurie, J. Edward Maste, Alan Mujumdar, Prashanth Munkdur, Steven J. Murdoch, Philip Paeps, Keith Rebello, Colin Rothwell, Peter Rugg, Hassen Saidi, Linton Salmon, Howie Shrobe, Domagoj Stolfa, Andy Turner, Munraj Vadera, Stu Wagner, Hongyan Xia, and Bjoern Zeeb. This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 ("CTSRD") and HR0011-18-C-0016 ("ECATS"). The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. We also acknowledge the EPSRC REMS Programme Grant (EP/K008528/1), the ABP Grant (EP/P020011/1), the ERC ELVER Advanced Grant (789108), the Gates Cambridge Trust, Arm Limited, HP Enterprise, and Google, Inc. Approved for Public Release, Distribution Unlimited.

Additional data relating to this paper can be found at https://doi.org/10.17863/CAM.51028.

References

[1] S. Ainsworth and T. M. Jones. "MarkUs: Drop-in Use-After-Free Prevention for Low-Level Languages". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, May 2020, pp. 860–860. URL: https://doi.ieeecomputersociety.org/10.1109/SP.2020.00058.

[2] Periklis Akritidis. "Cling: A Memory Allocator to Mitigate Dangling Pointers". In: USENIX Security'10. USENIX Association, 2010, pp. 12–12. URL: https://www.usenix.org/legacy/events/sec10/tech/full_papers/Akritidis.pdf.

[3] Emery D. Berger and Benjamin G. Zorn. "DieHard: Probabilistic Memory Safety for Unsafe Languages". In: PLDI '06. ACM, 2006, pp. 158–168. DOI: 10.1145/1133981.1134000.

[4] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. "Mostly Parallel Garbage Collection". In: PLDI '91. ACM, 1991, pp. 157–164. DOI: 10.1145/113445.113459.

[5] Hans-Juergen Boehm and Mark Weiser. "Garbage Collection in an Uncooperative Environment". In: *Softw. Pract. Exper.* 18.9 (Sept. 1988), pp. 807–820. DOI: 10.1002/spe.4380180902.

[6] Tim Boland and Paul E. Black. "Juliet 1.1 C/C++ and Java Test Suite". In: *IEEE Computer* 45.10 (Oct. 2012), pp. 88–90. DOI: 10.1109/MC.2012.345.

[7] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N. M. Watson. "CHERI JNI: Sinking the Java Security Model into the C". In: ASPLOS '17. ACM, 2017, pp. 569–583. DOI: 10.1145/3037697.3037725.

[8] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. "Live Migration of Virtual Machines". In: NSDI'05. USENIX Association, 2005, pp. 273–286. URL: https://www.usenix.org/legacy/event/nsdi05/tech/full_papers/clark/clark.pdf.

[9] Cliff Click, Gil Tene, and Michael Wolf. "The Pauseless GC Algorithm". In: VEE '05. ACM, 2005, pp. 46–56. DOI: 10.1145/1064979.1064988.

[10] *Common Criteria for Information Technology Security Evaluation. » Part 2: Security functional components*. Apr. 2017.

[11] The MITRE Corporation. *CWE-416: Use After Free*. 2018. URL: https://cwe.mitre.org/data/definitions/416.html.

[12] 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)*. USENIX Association, 2017, pp. 815–832. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/dang.

[13] 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: ASPLOS '19. ACM, 2019, pp. 379–393. DOI: 10.1145/3297858.3304042.

[14] Dinakar Dhurjati and Vikram Adve. "Efficiently Detecting All Dangling Pointer Uses in Production Servers". In: DSN '06. IEEE, June 2006, pp. 269–280. DOI: 10.1109/DSN.2006.31.

[15] *Electric Fence*. 2015. URL: https://elinux.org/index.php?title=Electric_Fence.

[16] Jason Evans. "A Scalable Concurrent malloc(3) Implementation for FreeBSD". In: BSDCan. Apr. 16, 2006. URL: https://www.bsdcan.org/2006/papers/jemalloc.pdf.

[17] Keir Fraser. *Practical lock-freedom*. Tech. rep. UCAM-CL-TR-579. University of Cambridge Computer Laboratory, Feb. 2004. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

[18] Matthew Gretton-Dann. *Arm A-Profile Architecture Developments 2018: Armv8.5-A*. 2018. URL: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv85a.

[19] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. "Memory-safe Execution of C on a Java VM". In: *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*. ACM. 2015, pp. 16–27.

[20] Richard Grisenthwaite. *A Safer Digital Future, By Design*. URL: https://www.arm.com/blogs/blueprint/digital-security-by-design.

[21] John L. Henning. "SPEC CPU2006 Benchmark Descriptions". In: *SIGARCH Comput. Archit. News* 34.4 (Sept. 2006).

[22] Intel. *Introduction to Intel® Memory Protection Extensions*. July 16, 2013. URL: https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions.

[23] 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)*. Nov. 2017, pp. 641–648. DOI: 10.1109/ICCD.2017.112.

[24] G. K. Konstadinidis, H. P. Li, F. Schumacher, V. Krishnaswamy, H. Cho, S. Dash, R. P. Masleid, C. Zheng, Y. D. Lin, P. Loewenstein, H. Park, V. Srinivasan, D. Huang, C. Hwang, W. Hsu, C. McAllister, J. Brooks, H. Pham, S. Turullols, Y. Yanggong, R. Golla, A. P. Smith, and A. Vahidsafa. "SPARC M7: A 20 nm 32-Core 64 MB L3 Cache Processor". In: *IEEE Journal of Solid-State Circuits* 51.1 (2016).

[25] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. "DangSan: Scalable Use-after-free Detection". In: *EuroSys*. 2017.

[26] Doug Lea. *A Memory Allocator*. 2000. URL: http://gee.cs.oswego.edu/dl/html/malloc.html.

[27] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. "Preventing Use-after-Free with Dangling Pointers Nullification". In: NDSS '15. Internet Society, 2015. DOI: 10.14722/ndss.2015.23238.

[28] 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: ISMM 2019. ACM, 2019, pp. 122–135. DOI: 10.1145/3315573.3329980.

[29] Daiping Liu, Mingwei Zhang, and Haining Wang. "A Robust and Efficient Defense Against Use-after-Free Exploits via Concurrent Pointer Sweeping". In: CCS '18. ACM, 2018, pp. 1635–1648. DOI: 10.1145/3243734.3243826.

[30] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. "Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety". In: ISCA '12. IEEE Computer Society, June 2012, pp. 189–200. DOI: 10.1109/ISCA.2012.6237017.

[31] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. "CETS: Compiler Enforced Temporal Safety for C". In: ISMM '10. ACM, 2010, pp. 31–40. DOI: 10.1145/1806651.1806657.

[32] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C". In: PLDI '09. ACM, 2009, pp. 245–258. DOI: 10.1145/1542476.1542504.

[33] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. "CCured: Type-safe Retrofitting of Legacy Software". In: *ACM Trans. Program. Lang. Syst.* 27.3 (May 2005).

[34] Gene Novark and Emery D. Berger. "DieHarder: Securing the Heap". In: CCS '10. ACM, 2010, pp. 573–584. DOI: 10.1145/1866307.1866371.

[35] Yutaka Oiwa. "Implementation of the Memory-Safe Full ANSI-C Compiler". In: PLDI '09. ACM, 2009, pp. 259–269. DOI: 10.1145/1542476.1542505.

[36] *Oracle's SPARC T7 and SPARC M7 Server Architecture*. Oracle. Aug. 2016.

[37] Matthew Parkinson, Kapil Vaswani, Manuel Costa, Pantazis Deligiannis, Aaron Blankstein, Dylan McDermott, Jonathan Balkind, and Dimitrios Vytiniotis. *Project Snowflake: Non-blocking safe manual memory management in .NET*. Tech. rep. July 2017. URL: https://www.microsoft.com/en-us/research/publication/project-snowflake-non-blocking-safe-manual-memory-management-net/.

[38] David D. Redell. *Naming And Protection In Extendable Operating Systems*. Tech. rep. MAC TR-140. Massachusetts Institute of Technology, 1974.

[39] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: USENIX ATC'12. USENIX Association, 2012, pp. 309–318. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[40] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. "Memory Tagging and How It Improves C/C++ Memory Safety". In: (Feb. 26, 2018). URL: http://arxiv.org/abs/1802.09517.

[41] 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++". In: NDSS '19. Internet Society, 2019. DOI: 10.14722/ndss.2019.23541.

[42] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. "FreeGuard: A Faster Secure Heap Allocator". In: CCS '17 (2017), pp. 2389–2403. DOI: 10.1145/3133956.3133957.

[43] Matthew S. Simpson and Rajeev K. Barua. "MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime". In: SCAM '10. IEEE, Sept. 2010, pp. 199–208. DOI: 10.1109/SCAM.2010.15.

[44] *Software Assurance Reference Dataset :: Around the Software Assurance Reference Dataset*. URL: https://samate.nist.gov/SRD/around.php#juliet_documents.

[45] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. "SoK: Sanitizing for Security". In: SP '19. IEEE, May 20–22, 2019, pp. 1275–1295. DOI: 10.1109/SP.2019.00010.

[46] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory". In: SP '13. IEEE Computer Society, 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[47] 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 Sewell, Stacey Son, and Hongyan Xia. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7)*. Tech. rep. UCAM-CL-TR-927. University of Cambridge, Computer Laboratory, 2018. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.html.

[48] Robert N. M. Watson, Jonathan Woodruff, Michael Roe, Simon W. Moore, and Peter G. Neumann. *Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks*. Tech. rep. UCAM-CL-TR-916. University of Cambridge, Computer Laboratory, Feb. 2018. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-916.pdf.

[49] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. *Oxide: The Essence of Rust*. 2019. URL: https://arxiv.org/abs/1903.00982.

[50] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert Norton, Thomas Bauereiss, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. "CHERI Concentrate: Practical Compressed Capabili-

ties". In: *IEEE Transactions on Computers* (2019). DOI: 10.1109/TC.2019.2914037.

[51] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. "The CHERI capability model: Revisiting RISC in an age of risk". In: *Proceedings of the 41st International Symposium on Computer Architecture (ISCA 2014)*. June 2014.

[52] 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: MICRO '52. ACM, 2019, pp. 545–557. DOI: 10.1145/3352460.3358288.

[53] Yves Younan. "FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers". In: NDSS '15. Internet Society, 2015. DOI: 10.14722/ndss.2015.23190.

[54] Tong Zhang, Dongyoon Lee, and Changhee Jung. "BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free". In: ASPLOS '19. ACM, 2019, pp. 631–644. DOI: 10.1145/3297858.3304017.

Appendix

A. Shadow-Space Access Control

The wrapper design and implementation explored throughout this paper assumes that `malloc()` is the only allocator, and it obtains access to the entire shadow bitmap. To support compartmentalized applications or those with multiple allocators, we must redesign Cornucopia's kernel interface to enable *selective* access to the shadow space.

To revoke pointers to a given allocation, user-space must set the shadow bits corresponding to that allocation's memory before requesting revocation from the kernel, and clear them after revocation and before the memory is reused; recall Fig. 1. We observe that, while user-space must write to the shadow bitmap, unmediated writes to this structure are able to break temporal-safety guarantees. For example, a spurious clear of bits in the shadow bitmap could result in capabilities to quarantined memory surviving the revocation sweep. Therefore, we must carefully control access to the shadow bitmap within an application and must not share access between distrusting sandboxes.

In order to support safe sharing of address space between distrusting allocators, we have implemented a kernel API to mediate control of the shadow bitmap space. Our kernel API builds on CHERI's *spatial* protection to enforce a privilege-minimizing policy: access to a region of the shadow bitmap is predicated on presenting a capability to the corresponding memory. The effect is that an allocator (or any user of the `mmap()` API, more generally) can access the shadows of its mappings but not those of other allocators.

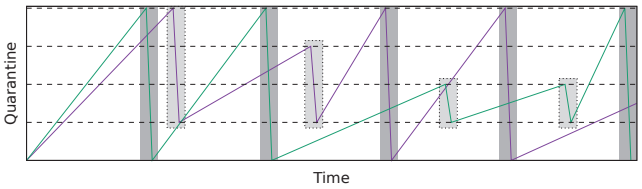

Fig. 12. A schematic example of two independent, Cornucopia-aware allocators running within a single address space. Here, both divide their quarantines into four segments and label each full segment with the *epoch* at time of fill. Each allocator fully drains its quarantine when all its segments are filled, which happens five times in this scenario (indicated by the vertical bars). Four times, indicated by dotted rectangles, an allocator fills a quarantine segment and discovers that existing segments were filled prior to the last revocation (i.e., in an earlier epoch) and so its quarantine is just the new segment.

CheriBSD, as part of CheriABI [13], also infers privilege of user-space code from capabilities presented to system calls, but defines an additional layer of privilege. As mentioned in §V-A, CheriBSD uses one of the capability *permission bits* left available for software interpretation, `VMMAP`, to confer rights to manage address space. The kernel returns a capability with this permission bit set whenever a mapping at a *new* address is created and subsequently checks that this bit is present on the authorizing capability whenever changes to the memory map are requested. CHERI-aware allocators strip this permission on capabilities returned to the program by `malloc()`.

We have replaced the generic whole-shadow access mechanism in Cornucopia with one that requires user-space to present a `VMMAP`-bearing capability and returns a capability only to that capability's shadow. This allows an allocator to prevent any other code from accessing the shadow bitmap of its heap. Our extensions of `snmalloc` and `dlmalloc` were not unduly burdened by managing disjoint capabilities to the shadow bitmap. Modern allocators must maintain metadata for multiple slabs of disjoint address space returned from `mmap()`, and we found it natural to expand this metadata to hold these bitmap capabilities.

B. Revocation Epochs and Segmented Quarantines

In addition to supporting safe sharing of address space between allocators as described in Appendix A, Cornucopia supports efficient *work-sharing* by enabling non-cooperating allocators to share revocation sweeps. That is, *all* allocators in a process benefit from a revocation pass performed by *any* allocator. In particular, by the time a revocation is performed by one allocator, other allocators may already have expressed (parts of) their quarantines in the shadow bits. The revocation performed will take those bits into consideration, and allocators need not perform additional unnecessary revocations.

Towards this end, we use a variant of epoch-based reclamation [17] and introduce the notion of a *revocation epoch*: a per-address-space count of the number of revocations performed. Allocators can exploit the published epoch counters by using a **segmented quarantine** structure. If a set of quarantined allocations are labeled with the epoch counter observed *after* setting the shadow bits, the observation of a sufficiently larger epoch counter later implies that a revocation sweep has revoked this set.

This design scales to any number of allocators with completely independent quarantine and revocation policies. In practice, only the allocator subject to the highest `free()` load is likely to discover a need to revoke, while allocators responding to lighter `free()`-ing loads will almost always find that their oldest quarantine segments are already revoked and so can clear their shadow bits and reuse their memory immediately.

Our integration of Cornucopia into `snmalloc` uses this epoch mechanism. Each of its per-thread allocators maintains a segmented quarantine, and threaded programs do indeed behave as expected with shared revocation work. Figure 12 shows a schematic interaction between two allocators with varying workloads that benefit from one another's revocations.

1) Epoch Counters and Memory Ordering

Cornucopia in fact exposes *two* epoch counters to user-space. The **enqueue** epoch counter is suitable for labeling quarantine segments as they become full, while the **dequeue** epoch counter should be used for testing revokedness of segments.

Between revocations, the enqueue counter is ahead of the dequeue value by *two* increments. When a revocation begins, the enqueue counter is incremented, and a release fence performed, before memory is swept (implicitly performing an acquire fence as part of the kernel's locking). When a revocation finishes, the enqueue counter is again incremented and the dequeue counter is advanced by two (and followed by an implicit release fence). A quarantine segment is revoked if its label is less than or equal to the dequeue counter: the staggered advance of the enqueue counter ensures that a segment remains in quarantine until a revocation both begins and ends after its labeling. (Counter values are 64-bit, and so for the prototype described here, we ignore wrap-around. In the future, we envision guarding our counters with *capabilities* whose revocation also indicates revokedness of a segment; this will moot any concerns of wrap-around.) The `caprevoke()` system call of §IV-C takes an optional epoch counter argument and will quickly return if a segment labeled by this epoch is already revoked by the time the kernel has entered its critical section. An allocator passing the epoch of its eldest quarantine segment will ensure that there is no TOCTTOU gap and avoids useless back-to-back revocations.

When labeling a segment of the quarantine with the enqueue epoch counter, we must be sure that the load of the counter happens after all cores would see the updates to the bitmap. This requires that the allocator perform an *acquire-and-release* fence after populating the bitmap and before loading the counter. This holds even if the allocator itself is fully serialized, because, while the interaction with the revocation bitmaps happens within the allocator's critical section, it is otherwise independent of other allocators' interactions with revocation.

When releasing a segment of the quarantine back to the free pools, a traditional allocator using locks will likely require no additional fencing. So long as an implicit *release*, such as exiting the critical section, separates clearing the shadow bitmap from derivation of the non-`VMMAP`-bearing capability returned to the client, all cores will perceive the shadows of these allocated objects to be clear before the allocator has

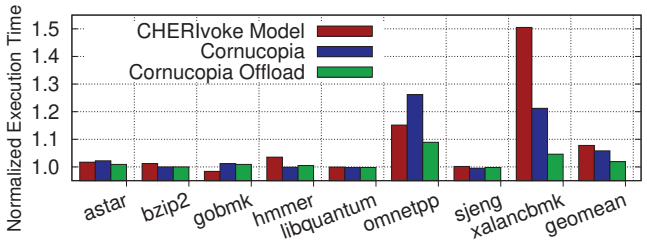

Fig. 13. CHERIvoke's modeled overheads vs the Cornucopia implementation. All benchmarks are run in their `ref` configuration.

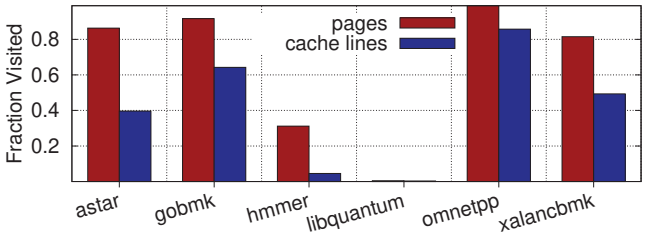

Fig. 14. Proportion of memory swept by the revoker, at both page and cache-line granularity. The cap-clean flag allows elision of page sweeps, and the `CLoadTags` allows elision of cache line loads.

constructed a revoke-able capability. The *data dependence* between the capability loaded by the revoker and the location looked up in the shadow bitmap is sufficient to ensure that no additional interlocking with the revoker is required: if the revoker perceives a capability to reused address space, the fence implies that it will perceive zeros in the shadow (unless they have become set again).

However, for allocators that do not enter and exit critical sections as part of their routine operation, such as `snmalloc`, it is necessary that explicit fences be inserted. `snmalloc`'s case is especially interesting, as it has three different behaviors for de-quarantining address space. (1) Large objects' addresses are immediately published to a global queue, and so require a release fence between clearing shadow bits and said publication. (2) Addresses belonging to other threads will be queued in this thread's "remote cache." This cache will be drained when it fills above some threshold; a release fence must be performed before such draining, but insertions into the cache need not fence. (3) Other addresses, belonging to the current thread, will not be considered for reuse until after the de-quarantining operation; here, a single release fence can cover any number of de-quarantined objects. Having performed this single fence, this thread is justified in immediately reusing any of its de-quarantined address space.

C. Correspondence with CHERIvoke Model

Cornucopia uses the CHERIvoke algorithm, which was originally characterized on modern out-of-order x86 processors [52]. Fig. 13 shows the results of the CHERIvoke modeling experiment against the measured overheads of the Cornucopia implementation. The results of the CHERIvoke study roughly agree with the scale of overheads we have measured for Cornucopia, despite vast differences in microarchitectural sophistication, indicating that the sweeping algorithm scales similarly to general performance. The biggest divergence is in `xalancbmk` where half of the overhead in the CHERIvoke

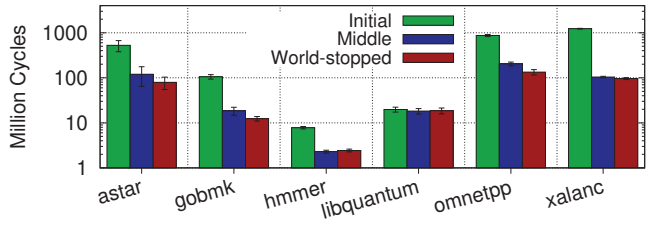

Fig. 15. Mean (and std. dev.) time per revocation phase, with revocation broken down into *three* phases. Compare Fig. 10.

experiment was from allocator layout, which varies significantly between `dlmalloc` and `snmalloc`, which we used for these Cornucopia results.

Fig. 14 shows the rates at which Cornucopia can take advantage of cap-clean and `CLoadTags` to save work. These results qualitatively confirm CHERIvoke's predictions (Fig. 8a of [52]), but we observe broadly elevated frequency of capabilities, likely due to CHERI's increased pointer size.

The sequential performance overhead of Cornucopia is lower than the overhead predicted by CHERIvoke: while CHERIvoke described using `CLoadTags` instructions (§IV-E) to avoid accessing cache lines without capabilities, it could not evaluate its performance effect. Cornucopia's implementation on CHERI benefits from `CLoadTags` for workloads such as `xalancbmk` that have irregular pointer distributions. Concurrent sweeping using an offload thread further improves over predicted performance by allowing continued execution during a sweep, achieving a geometric mean overhead of less than $1/3$ of the modeled overhead.

D. Adding Another Incremental Pass

One way to attempt to reduce the large pause times observed in §VII-C would be to introduce one or more "middle" phases to revocation, between the initial sweep and the world-stopped sweep. These middle phases could be, like the final sweep, *incremental*, considering only sweep-dirty pages, and could, like the initial sweep, be run concurrently with other application threads. Ideally, the number of pages left sweep-dirty after such a middle phase is smaller than the number of sweep-dirty pages at its start, as the application has less time to dirty pages than it did during the previous phase.

We re-run the experiments of §VII-C with a wrapper that makes *three* `caprevoke()` calls per revocation. While the results, shown in Fig. 15, suggest that our intuition is correct and that middle phase(s) could reduce pause times, the gain is perhaps less than might be desired. Moreover, this multiple-phase design requires revisiting pages yet again. That is, they necessarily increase both cycle and, worse, memory access overheads.