CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety

Hongyan Xia[*][†]
University of Cambridge

Jonathan Woodruff [*][†]
University of Cambridge

Sam Ainsworth[*][†]
University of Cambridge

Nathaniel W. Filardo[*]
University of Cambridge

Michael Roe[*]
University of Cambridge

Alexander Richardson[*]
University of Cambridge

Peter Rugg[*]
University of Cambridge

Peter G. Neumann
SRI International

Simon W. Moore[*]
University of Cambridge

Robert N. M. Watson[*]
University of Cambridge

Timothy M. Jones[*]
University of Cambridge

ABSTRACT

A lack of temporal safety in low-level languages has led to an epidemic of use-after-free exploits. These have surpassed in number and severity even the infamous buffer-overflow exploits violating spatial safety. Capability addressing can directly enforce *spatial* safety for the C language by enforcing bounds on pointers and by rendering pointers unforgeable. Nevertheless, an efficient solution for strong *temporal* memory safety remains elusive.

CHERI is an architectural extension to provide hardware capability addressing that is seeing significant commercial and open-source interest. We show that CHERI capabilities can be used as a foundation to enable low-cost heap temporal safety by facilitating out-of-date pointer revocation, as capabilities enable precise and efficient identification and invalidation of pointers, even when using unsafe languages such as C. We develop CHERIvoke, a technique for deterministic and fast sweeping revocation to enforce temporal safety on CHERI systems. CHERIvoke quarantines freed data before periodically using a small shadow map to revoke all dangling pointers in a single sweep of memory, and provides a tunable trade-off between performance and heap growth. We evaluate the performance of such a system using high-performance x86 processors, and further analytically examine its primary overheads. When configured with a heap-size overhead of 25%, we find that CHERIvoke achieves an average execution-time overhead of under 5%, far below the overheads associated with traditional garbage collection, revocation, or page-table systems.

[*]Email: {firstname.lastname}@cl.cam.ac.uk

[†]These authors contributed equally to this paper, and are named in reverse alphabetical order.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Security and privacy** → *Systems security*; *Security in hardware*; *Software and application security*.

KEYWORDS

temporal safety, use-after-free, architecture, security

1 INTRODUCTION

Large codebases written in low-level languages have been plagued by violations of temporal safety. A typical temporal-safety violation consists of a pointer to a deallocated object being mistakenly reused by the programmer. Such a *use-after-free* temporal-safety violation, in combination with other program behaviour, can result in a security vulnerability. Temporal-safety vulnerabilities can allow attackers manipulating data inputs to achieve full control of a program [8, 11], or even the entire system [46]. Indeed, this form of attack was recently found to be more common [26, 30] than buffer-overflow attacks that result from spatial-safety violations. Future computer systems require much stronger enforcement of both spatial and temporal memory safety.

Recent research has shown that spatial safety can be guaranteed at low cost by using architectural extensions, such as CHERI [44, 45], a hardware capability architecture that is influencing the direction of industry [19]. CHERI replaces pointers with unforgeable, architecturally identifiable references (capabilities) that convey not only the current address, but also the full range that is legally accessible through that reference. In this paper, we show that CHERI further allows us to achieve low-cost temporal safety for the heap in low-level languages, such as C and C++, with only minor architectural changes. By contrast, legacy architectures do not allow fine-grained

temporal safety of untrusted programs, as they can neither eliminate the possibility of retaining or fabricating references to freed memory nor distinguish dangling pointers from innocuous data.

We have designed CHERIvoke, a technique providing temporal safety for memory allocators on top of hardware capabilities, complete with minor architectural extensions for performance. CHERIvoke delays reallocation of memory, holding manually freed objects in a *quarantine buffer* until performing a sweep of memory to remove all references to these objects. An implementation of CHERIvoke thus prevents reallocation of memory that may still be addressable by references available to the program. Furthermore, CHERIvoke specifies a data structure for describing the freed memory locations using a shadow map wherein one bit represents a 16-byte allocation granule. This shadow map enables a single sweep to revoke access to an arbitrary number of memory locations, regardless of heap layout.

CHERIvoke provides fast, uncircumventable temporal guarantees: memory cannot be addressed without a capability, and all references to a region can be found as each capability contains full bounds information. In contrast to other temporal-safety systems for C [12, 27, 41], pointers cannot be hidden from the system. CHERIvoke also has a variety of useful performance properties: memory overhead can be capped due to active revocation (vs garbage collection), and regions with no capabilities can be entirely skipped based on hardware-tag metadata.

We evaluate the performance of CHERIvoke by reproducing its behaviour on a modern x86 system to account for state-of-the-art memory subsytems, and demonstrate overheads far lower than in previous work [12, 27, 41]. CHERIvoke can achieve strong temporal memory safety for the heap at an average of 4.7% runtime overhead (and a maximum of 51%) at the cost of 25% increase in heap size across SPEC CPU2006 benchmarks. Sweeping-time overhead is determined by the application's pointer density and the rate at which memory is freed, rather than more complicating factors such as the number of frees, number of loads, or allocation strategy. Further, this cost can be deterministically traded for increased heap overhead. Unlike pure software approaches, full memory safety for low-level languages is practical and efficient using CHERIvoke, and its overheads are predictable and intuitive to understand. The contributions of this paper are:

- The case for temporal safety built on top of tagged CHERI capability pointers.
- An algorithm for CHERI temporal safety that uses buffered revocation to achieve predictable costs that are substantially lower than previous techniques.
- An evaluation of this algorithm on a state-of-the-art memory subsystem.
- Lightweight CHERI extensions to optimise sweeping memory to identify capabilities.

2 BACKGROUND

2.1 Temporal-Safety Violations

The C memory model presents to the programmer a view of memory consisting of a set of objects. C allows the programmer to manipulate pointers to these objects and perform arithmetic on these pointers. As a result it is possible for programmers to mistakenly
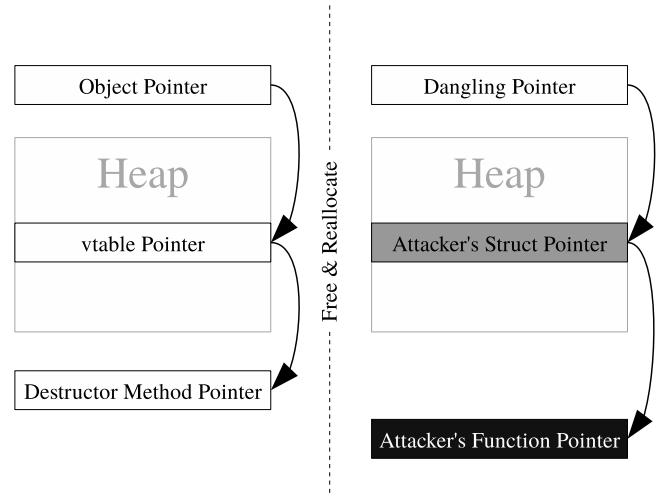
Figure 1: A use-after-free attack which overwrites a reallocated vtable pointer to reference attacker function pointers.

violate the memory model of the language such that a reference to one object may actually reach a different object. Objects in the C memory model are distinct from one another in both space and time. That is, two objects may be distinguished from one another by occupying disparate addresses in memory, or by existing at different times during the program's execution. C implementations have traditionally allowed violations of both of these boundaries, dubbed *spatial* and *temporal* safety respectively. CHERI can naturally enforce spatial safety by attaching bounds to pointers such that no manipulation of a reference to one object can cause it to reach another object. However, CHERI does not naturally defend against temporal-safety violations that arise from using a pointer after the program has asked for the object to be freed.

Accidental reuse of objects past their point of deallocation is common in low-level languages such as C and C++. These violations of temporal safety can result in security vulnerabilities, whereby an attacker can manipulate memory reached through a dangling pointer, causing it to point to a different object. This routinely allows attackers the flexibility to fully compromise computer systems.

An illustrative temporal-safety violation for C++ is depicted in figure 1. Here, delete is called on an object, which jumps to the destructor from the object's vtable which will free the object. Though the object is now notionally deleted, a pointer to the object's old location in memory is still accessible and now becomes a *dangling pointer*. This memory is then reallocated by the program to an object that holds external data input that has come from the attacker. An accidental second call to delete on the dangling pointer will now jump to an address of the attacker's choosing, ceding control over the process, and, if the vulnerability is within kernel mode, the entire system.

Besides pointer corruption, data corruption can change program execution [10], for example, to alter administrator checks to gain control of a program.

The above scenarios would commonly be classified as *use-after-free* vulnerabilities [11], but, more accurately, these are examples of *use-after-reallocation* attacks. Attacks that take advantage of
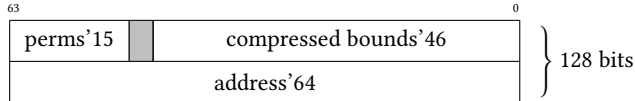
Figure 2: Bit representation of a CHERI-128 capability.

reallocation are most dangerous from a security perspective as they allow an attacker to take advantage of the mismatch in memory interpretation to gain influence over execution, particularly if one of the interpretations includes user-supplied data. By comparison, use-after-free before reallocation does not allow manipulation of a different object's data and, while erroneous, rarely results in security vulnerabilities. The enforcement of use-after-reallocation rather than strict use-after-free allows us the flexibility to batch revocations to achieve reasonable performance [27].

There is a class of use-after-free attacks that do not require reallocation, but corrupt allocator metadata that has been stored in freed memory. These vulnerabilities are solved relatively inexpensively by careful placement of metadata, as in BIBOP designs [16, 39, 40] and need not be addressed by revocation.

2.2 CHERI Capabilities

CHERI is an instruction-set extension [45] that requires addressing memory through unforgeable, bounded references called *capabilities* after the classic concept from computer science [14]. CHERI capabilities embed protected metadata to each pointer word, typically extending pointers to 128-bits for a 64-bit address space, or 64-bits for a 32-bit address space. As shown in figure 2, protected metadata includes the bounds of the object referenced and permissions granted by this reference. To enforce *monotonicity* of access rights, capability instructions do not allow the bounds of a capability to be enlarged.[1] To enforce *unforgeability* of capabilities, each capability word is protected by a 1-bit tag [22] that distinguishes a capability from arbitrary data. This tag is cleared on a non-capability write, preventing that word from being used as a capability. As a result, the virtual addresses accessible to a program are limited to those authorised by capabilities in the register file and reachable capabilities in (transitively) authorised regions of memory.

Prior work has described CheriABI [13], a new application binary interface for C and C++ programs under a CHERI-aware branch of FreeBSD. Programs compiled to CheriABI use capabilities for every reference, achieving spatial safety (against attacks such as buffer overflows) for all references, including the stack, heap, and globals, at an overhead that is typically less than 10%, even for pointer-heavy applications.

A primary benefit of the CHERI architecture is that the set of memory locations accessible to the program is entirely encoded in the memory state. Tags uniquely identify capability pointers, and these capabilities entirely define the range of memory they can reference. This structure facilitates precise pointer identification, eliminating both false negatives and false positives; inspection of memory cannot miss "hidden" pointers and cannot mistake data

for a pointer. This visibility at the architectural level enables a temporal-safety system that is both strong and high performance.

A secondary benefit of the CHERI architecture is its strong spatial safety, providing *object allocators* with the ability to bound returned pointers and, thereby, ensure that every object is accessed only within its bounds. For example, cross-object buffer-overflow attacks are impossible in C programs compiled to CheriABI, when linked with a correct, bounds-setting allocator. CHERIvoke uses this ability to ensure that each application-held capability with authority to access the heap has authority to *exactly one* heap object, so that *object* lifetimes imply *capability* lifetimes.

Prior work on CHERI has analysed the performance of tag storage [22]. Tag performance can have a major effect on pointer inspection, particularly if tags are read separately from their associated data in order to avoid loading untagged non-pointer data. CHERI prototypes store capability tags in a hierarchical table in conventional DRAM, and introduce a tag cache to reduce additional DRAM traffic. This tag cache achieves very high hit rates, while separation of tags and data facilitates efficient tag inspection without loading all associated data.
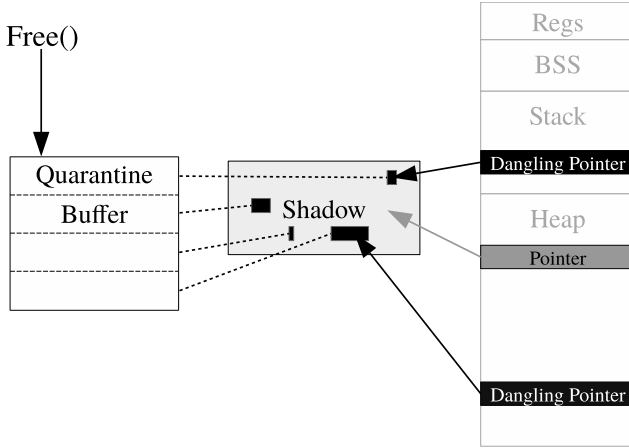
2.3 Threat Model

Our threat model assumes a non-malicious programmer who has inadvertently created a local program with a use-after-free vulnerability, and a malicious external attacker able to influence its behaviour – for example, via I/O over a network socket. By manipulating the vulnerable program, the attacker can utilise dangling pointers caused by this use-after-free vulnerability, to induce reads and/or writes via both the prior and current pointers to that memory. This allows an attacker broad scope for exploitative data corruption and control-flow attacks [9, 47], particularly where user-supplied data is confused for trusted data or function pointers.

Our aim is to remove dangling pointers to address-space regions before they are reallocated. This strategy addresses a critical set of exploit techniques relating to manipulation of data through different object pointers to the same memory. As with other techniques in the literature [27, 41], CHERIvoke does not address a broader category of temporal-safety violations, such as use of uninitialised data [28] or information leakage between prior and current allocations, and should be used alongside orthogonal low-cost protection mechanisms [29] for this purpose.

CHERIvoke could also be extended to address stronger CHERI threat models, such as software compartmentalisation, in which the local programmer may also be malicious [42]. This requires ensuring that shared memory referenced by two mutually distrusting compartments could not be improperly freed by either compartment. We do not address more sophisticated guarantees required by such use cases in this paper.

3 CHERIVOKE

We propose CHERIvoke as a technique to enforce temporal safety using CHERI by revoking access to freed memory before allowing reallocation. To revoke a capability is to remove all copies and all derivatives of that capability from a program. While this could be done on every free, CHERIvoke periodically performs bulk revocation to reduce overhead. This is achieved by holding manually freed

---

[1]At CPU power-on, the register file is initialised with *omnipotent* capabilities, bearing all permissions to all words of memory. Every capability created during the system's execution traces its provenance to these; there are no architectural operations that derive a tagged word exclusively from untagged inputs, and, for all derivations, the result bounds are no larger than those of a tagged input.

Free()

Quarantine
Buffer

Shadow

Regs

BSS

Stack

Dangling Pointer

Heap

Pointer

Dangling Pointer

Figure 3: Deallocations are kept in a quarantine buffer before they are revoked. Revocation is implemented efficiently by using a small shadow map of the heap that marks deallocated regions in quarantine. Memory and registers are swept using this shadow map to identify any dangling pointers. After the sweep, CHERIvoke clears the shadow map and moves quarantined locations into the free list for reallocation.

heap memory in a quarantine buffer until CHERIvoke has swept through program memory to clear tags on all capability references to quarantined memory, as shown in figure 3.

3.1 Quarantine Buffer

In order to prevent use-after-reallocation attacks, an allocator must not reissue freed address space until it has ensured that there are no remaining references to this memory in memory segments available to its caller. When allocations are freed, our allocator does not immediately return addresses to the reallocatable state, but places them in a quarantine buffer. When this quarantine buffer is full, we sweep all memory that could contain references to the heap and invalidate any capability reference that points to any region in the quarantine buffer. After the sweep, all quarantined addresses are returned to a free, reallocatable state.

In order to maintain a consistent memory overhead, this buffer can be set to a fixed proportion of heap size. For example, we may initiate a revocation sweep when the quarantined data has reached $\frac{1}{4}$ the size of the rest of the heap. The quarantine-buffer size can be scaled to trade off memory overhead for runtime overhead, increasing or reducing sweeping frequency.

3.2 Revocation Shadow Map

To achieve reliable, high performance regardless of application, the sweeping procedure should ideally be deterministic and independent of heap layout. We achieve these properties by maintaining revocation metadata in a *revocation shadow map*. For each allocation granule, which we choose to be 16 bytes of memory to match the default in dlmalloc [25], we allocate 1 bit in a shadow map; this shadow space occupies less than 1% of the heap. Before a sweep, for all allocations in the quarantine buffer, we "paint" the bits of the shadow map corresponding to the allocation granules to indicate that references to this memory should be revoked in the sweep.

The actual sweeping procedure performs a lookup in the shadow map using the base of each capability to detect if it is pointing into a revoked object.[2]

This shadow-map scheme allows fast, flat index lookup for testing each capability reference during a sweep, and is deterministic in its instruction count. As the shadow map is significantly smaller than the heap itself, and accesses to it are highly likely to be both temporally and spatially local, the shadow-map working set will typically fit in the last-level cache, and accesses to it should not limit DRAM bandwidth available to the primary sweep.

Most importantly, this shadow-space strategy allows revocation of all quarantined address space in a single sweep, with the result that sweeping frequency depends purely on the free rate of the application (in MB/s) and the size of the quarantine buffer, and not on heap layout. This ensures predictable and reliable performance for all applications.

3.3 Sweeping Procedure

A revocation sweep must cover all memory that could contain capability references to the heap. This includes the heap itself, the stack, register files, and global segments (such as .data and .bss). This sweep is the primary overhead in CHERIvoke. The sweep needs to be fast, and should aim to fully utilise the DRAM bandwidth of the system, requiring a highly optimised inner loop. While we limit our investigation to the efficiency of software implementations of this loop in the evaluation section, it would be reasonable to extend direct memory access (DMA) engines or digital signal processors (DSPs) in the system to perform this loop at bus speed and without CPU involvement.

In software, this inner loop consists of the following code:

```
 1  for(uintptr_t* x=MIN_ADDR; x<MAX_ADDR; x++) {
 2      uintptr_t capword = *x;
 3      if(is_capability(capword)) {
 4          capword >>= 4; // 16-byte alloc granule
 5          // Get the byte from the shadow space.
 6          char shadowbyte = shadowbyte_get(capword);
 7          // Get the bit index.
 8          int bitIdx = capword & 0x7;
 9          if(shadowbyte & (1<<bitIdx)) {
10              // Pointing at freed memory.
11              // Invalidate the capability.
12              *x = 0;
13          }
14      }
15  }
```

One issue in this circumstance involves two data-dependent branches, including the data-dependent store at the end. Here, the branch predictor will often predict them in the wrong direction. This means that this inner-loop should be carefully implemented to use conditional execution or conditional-move instructions (rather than true branching), to achieve the highest possible performance in a software-only implementation. However, even with accurate speculation, the loop can easily end up compute bound, despite the large number of memory accesses. To ensure we are not compute bound, we have implemented our model loop using Intel AVX2 vector extensions along with software pipelining. Vector extensions in an actual CHERI implementation must be able to read capability

---

[2]We can be sure that any heap capability will have a base within the original allocation, because the bounds of a capability can never be enlarged, only restricted, and the CHERIvoke allocator sets the bounds of its returns to match the requested allocation.
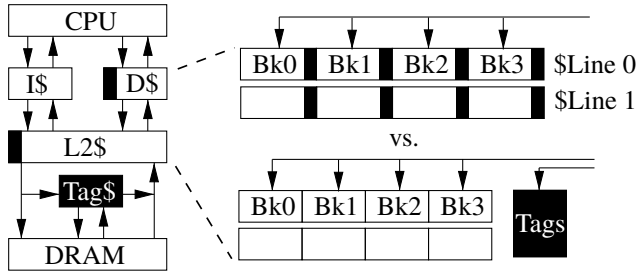
Figure 4: The implementation of `CLoadTags` requires moving tags from the data banks to tag metadata.

tags, but need not directly dereference the capabilities themselves, since the sweeping loop only looks up the pointer values in the shadow map.

3.4 New Hardware Support

A CHERI capability system tracks the presence of capability references in hardware and can therefore facilitate a sweep that inspects only genuine capability pointers. This gives us an opportunity to optimise the sweeping procedure: as capability state is an architectural feature, we can avoid sweeping through entire regions of memory that are pointer free, fundamentally decreasing the amount of work that must be done.

However, to check whether a memory word is tagged (i.e. contains a valid capability), the current CHERI ISA requires a load of the full capability word and tag into a register followed by the `CGetTag` instruction to query the tag bit. Use of this mechanism requires loading all data into caches, despite (as we measure later in table 2) fewer than a quarter of cache lines holding pointers in many applications. To implement CHERIvoke efficiently, we should directly exploit tag metadata to eliminate non-capability data from the sweep to save DRAM bandwidth and power, and to increase performance. We propose two new architectural assists atop CHERI's existing, spatial-safety-focused specification [45]. In section 6.3, we show that these significantly reduce DRAM traffic and time consumed by sweeping revocation.

3.4.1 `CLoadTags`. We introduce a new instruction, `CLoadTags`, to the CHERI architecture, which directly loads tag bits without loading the data from the given address. If `CLoadTags` returns a zero, this cache line can be skipped in the sweep because it contains no capabilities, thus avoiding DRAM traffic for this line.

The implementation of `CLoadTags` requires extensive integration with the memory hierarchy. First, we require a new memory-request type that loads only the tags of a cache line. Such requests require support in the L1 and L2 caches, as well as the tag controller. Furthermore, the L1 and L2 caches needed to be modified to be able to report all tags for a cache line in a single lookup. In the CHERI-MIPS implementation [45], cache lines are stored across four banks, so four cycles are required to read the entirety of the cache line. Storing capability tags with data rules out a single-cycle response to a `CLoadTags` bus request. We therefore implement CHERI caches that store capability tags in a tag metadata block for each line, as shown in figure 4.

Any cache where the line is held will respond to a `CLoadTags` bus request. If the `CLoadTags` request misses in all data caches, the tag controller will respond with only the tags of that line without fetching the corresponding data from DRAM. As this response contains only the tags of a cache line, it is inconvenient to cache the result in intervening caches; as a result we approximate streaming semantics for `CLoadTags` requests. Conveniently, this instruction is likely to be used only when sweeping memory, and caching its response is unlikely to be helpful. Future microarchitectures might consider prefetching data for a cache line when `CLoadTags` returns a non-zero result from the tag cache.

3.4.2 Page-table capability dirty (`PTE CapDirty`) bits. At a coarser scale, we repurpose a flag from the existing CHERI-MIPS page-table entries to avoid sweeping entire pages that do not contain capabilities. This flag is similar to a traditional dirty flag in page-table entries, although it specifically records the presence of valid capability writes in a page.[3] If `CapDirty` indicates that a page is clean, a store of a word tagged as a capability will throw an exception, allowing the operating system to record the presence of capabilities in that page by marking `CapDirty` in that page-table entry. Clean pages will not contain capabilities and need not be scanned during a sweep. As with the traditional `dirty` flag in page-table entries, some architectures may maintain `PTE CapDirty` entirely in hardware. This approach has false positives, as clearing all capabilities in a page will not reset `CapDirty`, though the page can be marked clean again if found to be without capabilities on the next sweep. However, our preliminary evaluation finds that the false-positive rate is negligible for all the benchmarks we evaluate, as the use of a page generally determines whether it can and does hold capabilities; we rarely encounter pages that alternate between holding capabilities and holding none.

3.5 Opportunities for Parallelism

Our description of CHERIvoke so far has described sweeping as part of application execution, that is, a program is paused while the sweep occurs. However, sweeping revocation can be made independent of execution and can run alongside the execution of the program. In addition, the sweep procedure itself is embarrassingly parallel. The shared revocation shadow map is read-only during the sweep, and pages to sweep can be distributed between independent threads. For this reason, it is not unreasonable to expect that even a pur- software sweeping routine could realistically saturate the full DRAM bandwidth of a system.

Shadow-map maintenance also has convenient concurrency properties. Updating the shadow map of different memory chunks in the quarantine buffer may occur in parallel, though care must be taken to prevent race conditions on bit masks within the same word. Painting the shadow map may use vector instructions, but is unlikely to require this level of optimisation, as explored in section 6.1.2.

Our evaluation framework using the x86 architecture does not allow a meaningful measurement of concurrent revocation, so we do not explore the implications of parallelism further in this paper.

---

[3]The capability-store-inhibit bit, *S*, is only lightly used in existing CHERI software. Its sole use is reflecting *static* properties of kernel-managed objects, e.g. preventing capability stores to shared memory segments (potentially violating capability provenance within an address space) or direct mappings of file pages (because the file system is not capable of storing tags).

3.6 Role of Allocator

CHERIvoke must invalidate all references to quarantined memory available to the program. Nevertheless, the allocator itself must hold references to heap memory, including quarantined memory, if it is to later reallocate memory to the program. CHERIvoke counts the allocator as part of the *trusted computing base* (TCB), and relies on the allocator to enforce temporal safety. Indeed, the definition of temporal safety itself is derived from allocator state. In order to preserve allocator references, CHERIvoke must distinguish between pointers held by the allocator and pointers issued to the program. While there are several plausible mechanisms for this preservation, one simple option is for the allocator to always use whole-heap-spanning capabilities whose bases are never quarantined.

3.7 Protection Guarantees

CHERIvoke enforces temporal safety for heap allocations only. Heap allocations have proven to be the most dangerous and common source of temporal-safety exploits [7], and stack exploits can be prevented using other techniques, such as escape analysis [12]. Strictly, CHERIvoke prevents *use-after-reallocation* rather than *use-after-free*, as the program still holds references to quarantined memory until a revocation sweep. Nevertheless, CHERIvoke guarantees that an allocated object can be accessed only through references derived from the latest allocation of that memory. While CHERI could facilitate strict *use-after-free* for debugging if a sweep was performed on every free, CHERIvoke is designed to enforce temporal safety for deployed user programs. This subset of heap temporal memory safety provides protection from the vast majority of exploitable bugs while taking advantage of buffering to achieve reasonable performance [27].

3.8 Summary

CHERIvoke is a technique for temporal safety on architectures with CHERI support. Dangling pointers can be revoked by sweeping through an application's memory, to remove references to deallocated locations stored within a quarantine space. We can do this because CHERI uniquely distinguishes pointer capability locations at the architectural level, along with the valid ranges to which a capability can point. Revocation can be implemented efficiently by using a shadow map to indicate invalid capability pointers; with the addition of new hardware support, `PTE CapDirty` bits and `CLoadTags` instructions, we can limit the memory that needs to be swept to include only cache lines that contain pointers.

CHERIvoke's quarantine buffer with shadow-map strategy achieves overheads that we now show are far lower than existing systems in practice, and further, can be easily understood and accounted for.

4 CHERI BENEFITS

CHERIvoke relies on the CHERI capability architecture to provide precise pointer identification, spatial enforcement, and efficient pointer-location metadata. These mechanisms enable the properties discussed below.

4.1 Efficient and Precise Revocation

In programs compiled to CHERI's pure capability mode, all pointers are tagged as capabilities to distinguish them from data. CHERI therefore eliminates conservative pointer classification that causes integers to be misclassified as pointers in garbage collection [6] and other techniques [12]. Conversely, clearing the tag of a capability on revocation completely prevents its use for referencing memory.

Furthermore, CHERIvoke relies on CHERI bounds enforcement to ensure that capabilities to the heap are easily attributed to exactly one allocation. Specifically, the base of any heap capability must remain within the original allocation, even as the pointer address can wander out of bounds. This relies on the property that the bounds of capabilities cannot be expanded, and that there is no mechanism to "fuse" adjacent objects into one capability, which could then reference multiple allocations with different lifetimes.

As CHERI identifies references with certainty and associates them uniquely to allocations, even a simple system can correctly invalidate references to quarantined memory knowing that it will not affect the behaviour of a correct program.

4.2 Full Memory Safety

CHERI capabilities provide spatial safety and unforgeability: that is, all memory accesses must be within the bounds of their allocation, and capabilities to other allocations cannot be fabricated. Based on these properties, CHERIvoke can completely prevent access to heap allocations after revocation, even in the face of adversarial programs. As capabilities are easily identified and trivially associated with their original allocation, we can reliably identify dangling pointers, and pointers can never be hidden from CHERIvoke without destroying their ability to ever reference memory.

CHERIvoke thus completely prevents even adversarial programs from accessing deallocated memory after a revocation sweep. Consequently, the temporal-security guarantees in the presence of capabilities are significantly stronger than in previous work [26, 27, 41].

4.3 Efficient Pointer Search

CHERI's architecturally visible capability tags not only enable identification of pointer words, but can even detect the presence of pointers in memory. Optimised tag storage in current CHERI prototypes enables `CLoadTags` to eliminate non-pointer data, reducing work by limiting the revocation sweep to regions that may contain dangling pointers.

5 EXPERIMENTAL SETUP

5.1 Systems

The systems we use in our evaluation are shown in table 1. In addition to evaluation of our hardware extensions on the CHERI FPGA platform [45], we have designed experiments to evaluate CHERIvoke revocation on a modern x86-64 machine to establish performance expectations for a wide deployment of mature CHERI implementations. Memory-sweeping performance depends heavily on the microarchitecture. These experiments allow us to characterise revocation using state-of-the-art memory systems, vector extensions, and out-of-order superscalar hardware. We simulate the existence of capabilities in these experiments using conservative pointer estimation, as used by garbage collectors [6], considering any 64-bit integer that is a valid virtual address to be a pointer. Evaluating on a mature x86 platform also provides higher application coverage, as the current CHERI prototype implements the 64-bit

System	Specification
x86-64	Intel Core i7-7820HK CPU, 2.9GHz, 4 cores 8 threads, 8MiB LLC, 14–18 stage out-of-order superscalar pipeline, AVX2 support, 16GiB DDR4 2400, FreeBSD 12.0
CHERI	Stratix IV FPGA, 100MHz, single core, 256KiB LLC, 6-stage in-order scalar pipeline, 1GiB DDR2

Table 1: System setup for processors used in the evaluation.

MIPS instruction set that lacks ports for many applications and benchmarks. Because the toolset for CHERI is based on FreeBSD, we also run FreeBSD on our x86 system to ensure uniformity, though results apply to any operating system.

To measure the impact of our new hardware additions, we extend a 64-bit CHERI core and cache subsystem to implement the `CLoadTags` instruction, and add `PTE CapDirty` support to our prototype operating system. To measure their impact on performance, we perform revocation sweeps on the CHERI FPGA implementation over application memory dumps taken from our x86 system, allowing us to measure data elimination for applications that are not yet able to execute natively on the CHERI-MIPS architecture.

5.2 dlmalloc_cherivoke

We have implemented `dlmalloc_cherivoke` as an extension of `dlmalloc` [25], a classic allocator that remains in wide use. This modified allocator maintains a quarantine buffer proportional to heap size, and also maintains the corresponding shadow map. Calls to `free()` insert allocations into a quarantine buffer that uses the `dlmalloc` constant-time algorithm for aggregating contiguous allocations. When a certain proportion of the heap is in quarantine, `dlmalloc_cherivoke` logs a simulated sweep event and returns all chunks in the quarantine buffer to the internal free list. As a result of aggregation, the number of internal frees may be much smaller than the number of frees without quarantine.

To implement the shadow map, each `mmap()` call is accompanied by a smaller mapping at a fixed transform from the original allocation. This allows the sweeping procedure to index the shadow map for any heap allocation, by shifting the pointer by a fixed amount and adding to the base of the shadow map. As `dlmalloc` aligns allocations to at least 16-byte boundaries (128 bits), each shadow map is $\frac{1}{128}$ of the primary allocation. When a region is unmapped, its corresponding shadow map is also unmapped.

Besides shadow-map allocation, `dlmalloc_cherivoke` delays shadow-space operations until a simulated sweep is triggered. Before a sweep event, we traverse the quarantined chunks in the buffer and set shadow-map bits for each. After a sweep event, these bits are cleared. We have optimised the shadow-map painting procedure such that large and aligned contiguous regions use byte, half-word, word, and double-word store instructions when possible, rather than setting individual bits.

5.3 Sweeping Cost

`dlmalloc_cherivoke` evaluates all overheads besides the revocation sweep itself. In order to accurately model a CHERI revocation

loop, pointers must be architecturally visible. Simulating this visibility requires memory state to be preprocessed, which prevents accurate performance modeling during execution. To capture memory state, we dump the core image periodically when the quarantine buffer is full and a sweep would have been triggered. We preprocess the memory image to identify all virtual addresses that lie within regions of the core dump, and zero all non-pointer words. This allows a test against zero to simulate the ability to test the capability tag in a true CHERI system. The core dump also preserves the revocation shadow map, which is used during the sweep.

We simulate a sweep that uses `PTE CapDirty` optimisations (section 3.4.2) to eliminate non-pointer data at a page granularity, but that does not use the `CLoadTags` instruction. While page elimination can be modelled sufficiently on a standard microarchitecture, `CLoadTags` is difficult to model due to its interaction with tagged memory and a tag cache. As a result, our performance numbers are a pessimistic estimation of the full optimisations possible on CHERI. Our sweep procedure simulates a system API that returns an array of pages that could contain capabilities[4] according to `PTE CapDirty` flags.

To evaluate the overall cost, we perform revocation sweeps on ten sample core dumps from across each application's execution.[5] We then multiply the average sweep time by the total number of sweep events to derive the total sweeping cost for that execution.

5.4 Benchmarks

To evaluate CHERIvoke, we are interested in both worst-case and average-case overhead. To do this, we evaluate on benchmarks taken mostly from SPEC CPU2006 [21], in line with other papers in the literature [12, 27, 41]. The subset we evaluate includes the three most allocation-intensive workloads [41]: dealII, omnetpp, and xalancbmk. We also include all other SPEC CPU2006 benchmarks that would compile under the 64-bit FreeBSD setup necessary to use our current CHERI infrastructure: astar, bzip2, gobmk, h264ref, hmmer, lbm, libquantum, mcf, milc, povray, sjeng, soplex, and sphinx3. In each case, we evaluate on the reference input. We further add ffmpeg, which has a larger allocation throughput than any SPEC benchmark and is useful to more fully account for worst-case application behaviour. We take the average of 5 runs for each benchmark.

6 EVALUATION

The overall observed overhead of CHERIvoke is shown in figure 5, compared with other temporal-safety techniques in the literature [6, 12, 27, 41] that do not make use of CHERI capabilities. For a target 25% heap storage overhead in the quarantine buffer, we achieve an average 4.7% execution time and 12.5% total memory overhead. This significantly outperforms any other technique. Further, CHERIvoke performs far more reliably, with only 1.51× and 1.35× maximum runtime and memory overheads. CHERIvoke

---

[4] A similar API, `GetWriteWatch()`, is implemented in Windows to return the list of pages that have been written since last reset to accelerate garbage collection and language runtimes [23].

[5] Collecting more than ten core dumps per application increased evaluation time but was not found to improve the accuracy of results. Sweep time for each core dump is averaged over 20 sweeps.

(a) Execution Time

(b) Memory. The dashed line shows CHERIvoke's default quarantine size at 25% of the heap.
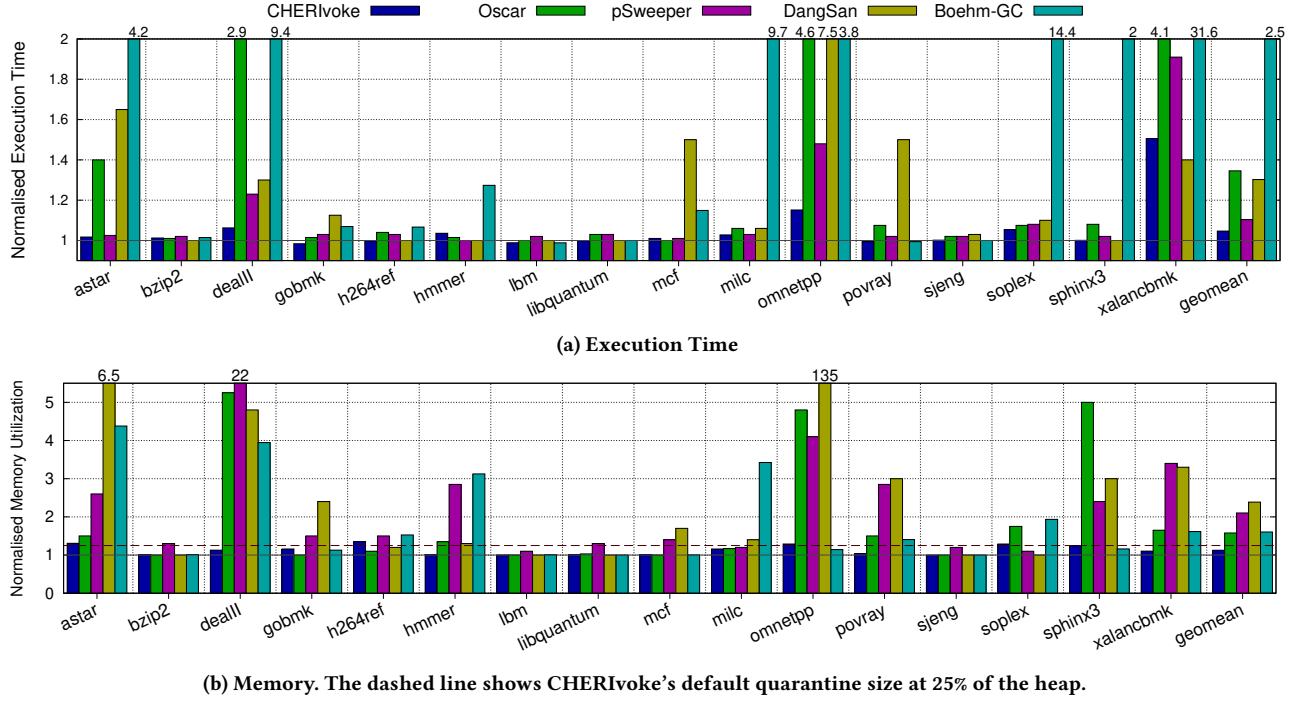
Figure 5: Overheads for CHERIvoke, compared with results reported by other state-of-the-art techniques.
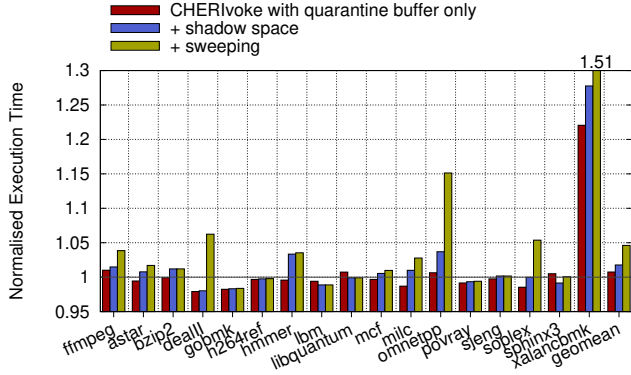

Figure 6: Decomposition of run-time overheads of CHERIvoke, with the default 25% heap overhead.

has significantly more predictable behaviour regardless of workload, as its sweeping technique suffers none of the worst cases encountered by more complex temporal-safety schemes: overheads are proportional to memory freed and pointer density, rather than pointer movement, number of frees, loads per second, or memory layout. In addition, CHERI enforces the strongest safety guarantees: construction of references to freed memory is impossible by any means after revocation.

6.1 Breakdown of Overheads

Figure 6 shows overheads for successively adding constituent parts of CHERIvoke, beginning with quarantining freed memory, adding shadow-map maintenance, and, finally, full-memory sweeps. While

memory sweeping is usually the dominant overhead, we discover notable exceptions that are discussed below.

6.1.1 Quarantine buffer. Many existing allocators, including `dl-malloc`, attempt to reuse freed memory as quickly as possible to improve cache performance. `dlmalloc_cherivoke`, however, introduces a quarantine buffer where freed memory is detained, missing the opportunity to reuse cached memory.

The quarantine buffer has negligible impact on most benchmarks. For xalancbmk, however, the quarantine buffer increases execution time by 22%. Performance counters confirm that instruction count only grows by 3%, but level-2 cache misses grow by 50%. While favorable deallocation patterns allow us to simply move to fresh, unquarantined cache lines, xalancbmk has a combination of small allocations, a high allocation throughput (nearly 1 million per second according to table 2), and *temporal fragmentation*. Temporal fragmentation occurs when objects with very different lifetimes are interspersed on the heap, leaving holes of quarantined memory in cache lines that are still in use. This suggests that a CHERIvoke memory allocator might attempt to group objects of similar lifetime. Nevertheless, we discover in section 6.4 that increasing the quarantine-buffer size consistently improves cache performance for xalancbmk.

The quarantine buffer actually improves performance in most of the benchmarks. One reason for this is batching and aggregating calls to free. DealII, for example, has 630,000 calls to free per second (see table 2), constituting a significant amount of execution time. `dlmalloc_cherivoke` quarantines these allocations at typically less than half the execution time of a real free. If these freed regions aggregate well, many fewer free operations will be performed when the quarantine buffer is drained than would have been performed

Benchmark	Pages with pointers	Free rate (MiB/s)	Frees (thousands/s)
ffmpeg	4%	**1268**	44
astar	**62%**	24	27
bzip2	0%	0	≈ 0
dealII	**70%**	40	**498**
gobmk	**54%**	1	1
h264ref	9%	3	1
hmmer	4%	17	12
lbm	0%	5	≈ 0
libquantum	1%	5	≈ 0
mcf	46%	53	≈ 0
milc	3%	224	≈ 0
omnetpp	**95%**	175	**1027**
povray	19%	1	17
sjeng	24%	0	≈ 0
soplex	23%	287	2
sphinx3	18%	33	30
xalancbmk	**86%**	371	**811**

Table 2: Deallocation metadata from applications.

on demand. While this effect is minor, many of the benchmarks that gain advantage from the quarantine buffer do not experience a net overhead for full temporal safety.

6.1.2 Shadow-map maintenance. CHERIvoke also requires maintenance of the revocation shadow map (the second bar in figure 6). While the size of the shadow map is small compared to the heap itself, and the quarantined portion is even smaller, the overhead of painting is hard to predict due to sensitivity towards the alignment and size of allocations. Nevertheless, the net impact of shadow-space maintenance is minor for all applications benchmarked.

6.1.3 Sweeping overhead. Where CHERIvoke has significant execution time overhead, the largest cost is in memory sweeping. Of the four benchmarks in figure 6 that have overheads beyond 5%, dealII, omnetpp and soplex are dominated by sweeping overhead and xalancbmk is a special case, as discussed above. Sweeping cost is predictable and can be described mathematically.

A memory sweep will be initiated when the amount of memory freed reaches the current size of the quarantine buffer, and thus the frequency of sweeping is directly proportional to the *QuarantineSize* and the *FreeRate* (in MB/s). This relation allows us to analytically derive an estimation for runtime overhead for a single-threaded implementation:

$$RuntimeOverhead \approx \frac{FreeRate \cdot PointerDensity}{ScanRate \cdot QuarantineFraction}$$

This equation assumes that we must only sweep the proportion of memory that contains pointers, *PointerDensity*, which is at a page granularity for this experiment. This equation also assumes the quarantine-buffer size to be a fixed proportion of the total memory space (rather than of the heap), which is a rough approximation if the heap is large. Nevertheless, this equation provides an intuitive model for the cost of sweeping using CHERIvoke.

The numerator, (*FreeRate · PointerDensity*), constitutes an application-specific cost factor. A low throughput for frees, or conversely a low pointer density, will result in a low sweeping cost for CHERIvoke. In the denominator, the *ScanRate* is a function of the memory bandwidth of the system and the efficiency of the sweeping loop, and *QuarantineFraction* is a tunable property to balance performance and memory consumption.
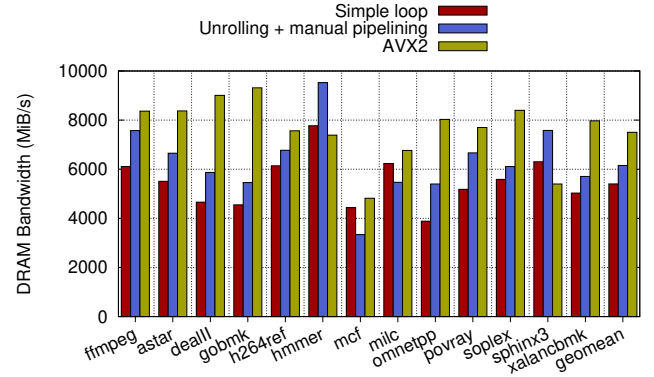

Figure 7: Memory bandwidth achieved for the sweep loop with different optimisations. The system's full read bandwidth is 19,405MiB/s.

This analytical model, along with the data in table 2, allows us to understand the sweeping overheads measured in figure 6. Xalancbmk and omnetpp have significant free rates and pointer densities over 85%, followed by dealII and soplex, whose pointer densities are 70% and 23% respectively. These four are indeed the only benchmarks with over 5% execution time overhead, as suggested by the model. Ffmpeg has a very high free rate, but a low pointer density, such that sweeping overhead does not break 5%.

6.2 Sweeping-Loop Optimisation

The speed of the memory sweep is critical to the performance of CHERIvoke. In figure 7, we evaluate the performance of several implementations of our sweeping-procedure kernel on each benchmark that features significant deallocation. A CHERIvoke sweep might approach the 19,405MiB/s read bandwidth of the system if the procedure is not compute bound, and if the indirect shadow lookup is entirely cached. We find that a naïve sweeping loop (presented in red) utilises only 28% of read bandwidth on average, and unrolling and manually pipelining the loop for better scheduling achieves 32%. We were able to fully vectorise the loop using AVX2 to sweep an entire cache line in 28 instructions, achieving 39% of the read bandwidth on average, but required an unconditional store to possibly clear dangling pointers, limiting us to memory copy performance. The performance of the AVX2 loop is roughly constant at almost 8GiB/s. AVX2 is not always the fastest; in hmmer and sphinx3 our vectorised implementation cannot compete with the unrolled loop. Mcf and milc see lower bandwidth utilisation, as their small, infrequent sweeping loops do not reach full throughput. Since none of these cases are allocation intensive, these outliers do not have a significant performance impact in figure 6.

6.3 Hardware Optimisations

Because of the new hardware optimisations introduced in section 3.4, we need not sweep all of memory. Two mechanisms avoid reading segments of memory without pointers: `PTE CapDirty` bits (section 3.4.2), which remove the need to scan through pages without capabilities, and the more fine-grained `CLoadTags` instruction (section 3.4.1), which allows us to skip cache lines with no tag-bits set. The results of our evaluations are shown in figure 8.
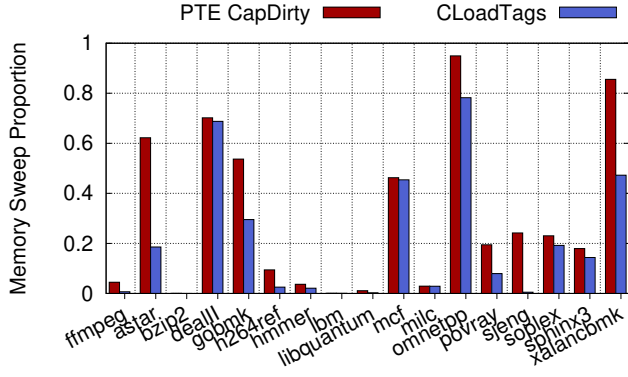
(a) Proportion of memory that needs to be swept for specific benchmarks, with work reduction both on a page-table granularity (`PTE CapDirty`) and on a cache-line granularity (`CLoadTags`).
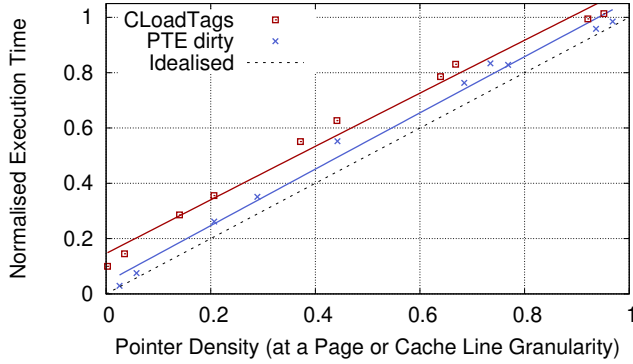
(b) Normalised execution time for sweeping through memory with the addition of PTE dirty bits to exclude capability-free pages, and CLoadTags instructions to exclude capability-free cache lines. Each is plotted versus their target granularities: PTE dirty is plotted against page density, and CLoadTags against cache-line density. The dotted line shows the ideal improvement from each technique.

Figure 8: Impact of the hardware optimisations from section 3.4 on both amount of memory that needs to be swept, and on resultant execution time as measured on CHERI.

Figure 8(a) shows the proportion of memory that must be swept under each optimisation, derived from the densities of capabilities both at the cache line and the page granularities. In most cases, the `PTE CapDirty` bits in the page table are sufficient to reach the achievable reduction in work, though there are several workloads where `CLoadTags` instructions allow a significant further reduction. Figure 8(b) shows how these mechanisms map to performance improvements on our CHERI FPGA hardware. We see that `PTE CapDirty` bits get close to an ideal performance improvement, in that the blue line is very close to the dotted $x = y$ line, and so the effect of not having to walk through pointer-free pages corresponds directly to a performance improvement. Performance with `CLoadTags` (orange) is more complex: though it can capture more fine-grained density data, and therefore theoretically reduce the amount of work more, its performance in practice is less close-to-ideal and can even lower performance. This reflects the larger
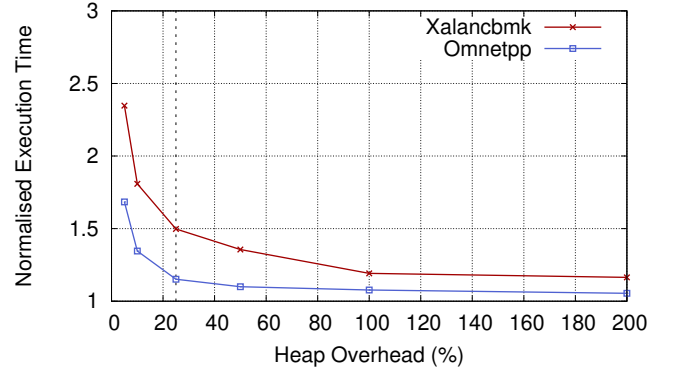
Figure 9: Normalised execution time for the two workloads with highest overheads, at varying heap overhead. Default setup shown by dotted line.

amount of work necessary to exploit this more fine-grained information. To determine if a cache line of 8 pointers can be skipped, `CLoadTags` must query the L1 and L2 caches and reach the tag cache of the system (around 10 cycles round trip in our FPGA implementation), and perform an unpredictable branch. In contrast, the `PTE CapDirty` implementation can skip a page of 256 pointers by inspecting page metadata. In practice, both coarse-grained (`PTE CapDirty`) and fine-grained (`CLoadTags`) optimisations are necessary for optimal work reduction.

6.4 Sweep-Frequency Trade-Offs

Time and space overheads can be traded off for one another in CHERIvoke. To see the extent of this, we re-evaluated xalancbmk and omnetpp, our workloads with the highest overheads at default settings, with different target heap-space overheads. The results of this are shown in figure 9. We see that the higher the heap overhead we are willing to tolerate, the less of a performance impact we will observe, even on highly allocation-intensive workloads.

There are two reasons for this. The first is that if we are willing to tolerate a higher heap overhead, deallocations can be left in quarantine for longer, and so we sweep proportionately less often as a result. This accounts for the majority of the performance increase we see with larger quarantine buffers, as most of the overhead of CHERIvoke is brought about via the sweeping procedure. The second is more subtle: for xalancbmk, by the time we reach 100% heap overhead, the normalised execution time is actually lower than the non-sweeping costs alone in figure 6. We found a consistent reduction in non-sweeping overheads corresponding to an increase in observed cache hit rate for the program as we moved to larger quarantine buffers. This counterintuitive result is caused by better allocation-fragmentation properties as we increase the heap size: under severe temporal fragmentation, it is better to quarantine memory for longer to allow cache lines to fall entirely out of use rather than frequently releasing small fragments in a severely fragmented heap.

6.5 Sweeping-Traffic Overhead

The results in figure 10 show the extra traffic generated from sweeping. We use Intel performance counters [43] to report the "off-core"
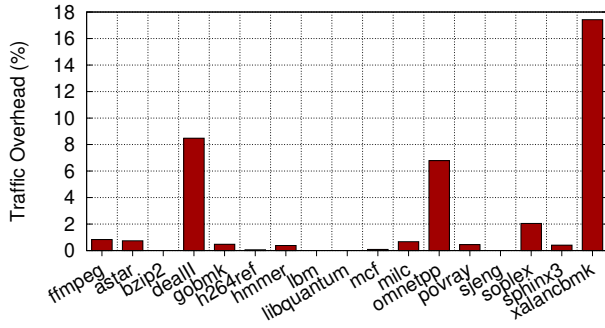
Figure 10: Off-core-traffic overhead.

traffic, which on our benchmark machine indicates the traffic to the shared L3 cache and above. We find that extra traffic utilisation is either comparable to (dealII) or significantly lower than (omnetpp, soplex, xalancbmk) the performance overhead. This is unsurprising: CHERIvoke only pays overhead on workloads that are allocation intensive, and workloads that are allocation intensive tend to be memory-bandwidth intensive, rendering CHERIvoke sweeping overheads less significant by comparison.

We can use this information to make judgements about the impact CHERIvoke has both on energy consumption and performance on multicores. In effect, energy-consumption overhead should scale comparably to performance overhead, as the additional factor, off-core traffic, and thus DRAM traffic, is comparable or smaller. Similarly, accesses to the shared L3-cache resource outside the core, which will affect performance of other applications running on a multicore, are typically minimal, and in allocation-intensive environments comparable to, though lower than, performance overhead.

6.6 Summary

CHERIvoke significantly outperforms any other system designed to provide strong performance guarantees [6, 12, 27, 41], both in average (4.7% runtime and 12.5% overall memory overheads) and worst case (51% performance and 35% memory). These overheads typically come from the sweeping procedure, which is a small code kernel that can be heavily optimised using vector instructions, and the cost of which can be analytically understood in simple terms of volume of freed data and density of pointers in memory. Our hardware extensions, `CLoadTags` and `PTE CapDirty`, both serve to significantly reduce the amount of work performed by CHERIvoke. Where performance overhead is high, memory can be traded to meet the target performance.

7 RELATED WORK

7.1 Revocation Techniques

Revocation techniques that do not make use of hardware capabilities have been explored. These include DangSan [41], DangNull [26], FreeSentry [48] and PSweeper [27]. These use the compiler to disambiguate pointers from data, add code for each pointer creation that inserts the pointer to a per-allocation list, and nullify all entries when data is freed. However, this per-allocation list is highly performance- and storage-intensive, which makes these techniques

infeasible for allocation-heavy workloads. Additionally, pointers can be hidden, so such techniques cannot guarantee temporal safety.

With CHERI, we can disambiguate pointers at run-time without any additional metadata, by using 1-bit tag metadata [22]. This means that we can instead sweep through memory to nullify any dangling pointers, avoiding the large memory and performance overheads associated with this complex metadata. It also means that the compiler need not be involved: the only change required is for the free method to add the quarantine list. CHERI also innately prevents hidden pointers, so can guarantee temporal safety.

BOGO [49], like CHERIvoke, builds temporal safety on top of spatial safety, in this case, Intel MPX. Due to a lack of a quarantine buffer for batching and due to the complex MPX table structure, BOGO's overheads are significantly higher than CHERIvoke: on SPEC CPU2006, CHERIvoke pays 4.7% average overhead and 50% worst case, whereas BOGO pays 60% average and 1,616% worst case.

7.2 Page-Table Techniques

Dangling pointers can be prevented from being used via protection at the granularity of the page table, by poisoning regions of memory upon a free. This is the technique used by Electric Fence [1]. Dhurjati and Adve [15] extend the technique to allow reuse of the underlying physical address to reduce overheads by aliasing virtual pages, and Dang et. al [12] present Oscar, which better supports concurrency and looks at more common workloads.

Page granularity can achieve low overheads when allocations are large. However, frequent small allocations can cause performance and memory overheads to increase enormously, as each allocation must be given its own virtual page, as well as increasing TLB pressure, causing significant slowdown.

7.3 Garbage Collection

Garbage collection solves the problem of use-after-frees by the inverse of pointer nullification: it prevents data from being freed until all references are removed. Examples of garbage collection used for this approach include FailSafe-C [35] and CCured [33].

As pointers can be hidden in low-level languages such as C and C++, this makes safe garbage collection a challenge [4, 5, 17]: we cannot trade the security issue of temporal safety for a program-safety issue of premature deletion of still-needed data. However, in the CHERI architecture, pointers cannot be hidden, as all memory accesses occur by unforgeable capabilities that can be distinguished from other data by tags. This means that CHERI avoids both the safety issue and any pointer aliasing from conservative garbage-collection techniques.

In addition, garbage collection suffers from two weaknesses that CHERIvoke does not. Because references to pointers may exist until long after the data is no longer being used, garbage collectors can suffer significant memory overhead even with frequent mark-sweep procedures. To counteract this, techniques such as the Boehm-Demers-Weiser garbage collector [6] also allow manual deallocation of objects. This means that use-after-free and use-after-reallocate violations can still occur in high-performance garbage collectors.

The second issue is related to performance. A garbage collector's marking procedure is significantly slower than CHERIvoke's

sweeping procedure, as marking involves a complex and memory-irregular graph search through each allocation, whereas sweeping can be performed at close to the rate of memory bandwidth via a simple, easy-to-optimise loop. Further, with CHERIvoke we know precisely how much memory can be reclaimed by each sweeping procedure, as this is supplied by the programmer with their manual deallocations. This means we can optimise by calling sweeping procedures only when there is sufficient useful work to be done (in our case, when the quarantine buffer is 25% of the rest of the heap), vastly reducing overheads without increasing memory usage.

Moreover, for many programmers, the malloc/free model is simply familiar. The semantics of explicitly managing memory is well understood and acceptable to a large class of programmers, as exemplified by the C and C++ communities. Existing codebases, especially legacy C and C++, need extra care to be ported to a GC model to function well under reasonable memory and performance overhead. On the contrary, malloc/free with sweeping revocation provides temporal safety without perturbing memory-allocation semantics, as well as having much more predictable memory and performance overhead — as this paper demonstrates.

7.4 Partial Temporal Safety

Techniques to reduce (but not eliminate) temporal-safety bugs have seen use both in academia and in practice. Cling [2] reduces the classes of use-after-free bugs that can be exploited by promoting type-safe reuse of pointers, based on size and call site, to reduce the provenance of reused memory, along with a more general delay-of-use to prevent memory exhaustion at the expense of security. Other techniques that use a delay-of-reuse technique, to make it harder for an attacker to reallocate data that is falsely freed but still in use, include DieHard [3], DieHarder [34], and FreeGuard [39].

7.5 Detection Methods

Runtime protection can fully guarantee temporal safety. However, some protection can also be brought about by detection methods designed to debug applications. An example of this strategy is AddressSanitizer [37], which poisons deallocated regions to flag up any accesses to them. The performance loss as a result is substantial, and so software AddressSanitizer can only be used in a debug setting. However, hardware acceleration of memory debug is implemented in the Sparc M7's ADI technique [24, 36] and in Arm MTE [18]. These use a small number of shadow bits to tag pointers, such that accesses to a region de- or re-allocated and tagged with a different bit value will fail. However, the small number of bits in these tags means that a motivated attacker can exhaust the space, to reallocate data with the correct tag. These techniques are therefore only suitable for runtime fault reporting rather than security. Another detection method is Undangle [7], which finds dangling pointers within a program at run-time, at the cost of false positives, since dangling pointers themselves may not result in future use.

7.6 Tagged Memory

CHERI [45] is just one way of using tagged memory to improve security or debug properties of a system. Other uses include annotating address validity, version numbers, object types and ownership [20]. While CHERI uses one bit per capability-aligned region to prevent

arbitrary changes of capabilities, other techniques use multiple bits to provide memory versioning. These include SPARC ADI [36] and Arm MTE [18]. Another tagged-memory debug technique, AArch64 HWASAN, combines memory tagging with a modified compiler toolchain for a hardware-assisted AddressSanitizer-like scheme [37, 38], by utilising unused top bits in pointers as memory tags to detect stale references.

CETS [32] uses word-length unique tags for memory accesses, such that a memory access will fail if the tag does not match the allocated region. This means that pointers are as large as CHERI's, but at the same time a large false-positive rate is suffered due to pointer hiding, which is valid in non-CHERI C. Unlike in CHERI, spatial safety cannot be guaranteed, and as there is no hardware support for CETS, it results in a significant performance loss. Watchdog [31] uses unique pointer and allocation identifiers to provide temporal safety in hardware: for the benchmarks in common between Watchdog and CHERIvoke, Watchdog pays 17% average overhead, whereas CHERIvoke pays less than 1%.

8 CONCLUSION

We have shown that it is possible to enforce temporal safety on modern systems with hardware capability support at low overhead. CHERIvoke, a technique that sweeps through memory to find architecturally visible capability pointers, and uses an efficient revocation shadow map to identify those that need to be revoked; it can achieve performance overheads of under 5% for a 25% heap size increase, and these can be traded off to match system requirements.

Our presentation of CHERIvoke considers only the fundamental mechanisms necessary for high-performance temporal safety; full implementations could be optimised further. Techniques such as reuse of physical addresses for page-size deallocations [12], type-based reuse of allocation data [2], and delaying of revocation by reusing locations over multiple MTE-style history bits [18] all have the potential to combine with CHERIvoke to make strong memory-safety properties cheap enough in all cases to become ubiquitous in all future systems.

REFERENCES

[1] 2015. Electric Fence. https://elinux.org/index.php?title=Electric_Fence
[2] Periklis Akritidis. 2010. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *USENIX Security*.
[3] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *PLDI*.
[4] Hans-J. Boehm. 1996. Simple Garbage-Collector-Safety. In *PLDI*.
[5] Hans-J. Boehm and David Chase. 1992. A Proposal for Garbage-Collector-Safe C Compilation. *Journal of C Language Translation* 4, 2 (1992).
[6] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exper.* 18, 9 (1988).
[7] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: Early Detection of Dangling Pointers in Use-after-free and Double-free Vulnerabilities. In *ISSTA*.
[8] Oliver Chang. 2016. Racing MIDI messages in Chrome. https://googleprojectzero.blogspot.com/2016/02/racing-midi-messages-in-chrome.html
[9] Oliver Chang. 2016. Racing MIDI messages in Chrome. https://googleprojectzero.blogspot.com/2016/02/racing-midi-messages-in-chrome.html.
[10] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-control-data Attacks Are Realistic Threats. In *SSYM*.
[11] The MITRE Corporation. 2018. CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html
[12] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2017. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *USENIX Security*.
[13] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment. In *ASPLOS*.
[14] Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multi-programmed computations. *Commun. ACM* 9, 3 (1966).
[15] Dinakar Dhurjati and Vikram Adve. 2006. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *DSN*.
[16] R. Kent Dybvig, David Eby, and Carl Bruggeman. 1994. *Don't stop the BIBOP: Flexible and Efficient Storage Management for Dynamically-Typed Languages*. Technical Report 400. Indiana University School of Informatics, Computing, and Engineering.
[17] John R. Ellis and David L. Detlefs. 1994. Safe, Efficient Garbage Collection for C++. In *CTEC*.
[18] Matthew Gretton-Dann. 2018. Arm A-Profile Architecture Developments 2018: Armv8.5-A. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv85a
[19] Richard Grisenthwaite. 2019. Supporting the UK in becoming a leading global player in cybersecurity. https://community.arm.com/blog/company/b/blog/posts/supporting-the-uk-in-becoming-a-leading-global-player-in-cybersecurity
[20] Richard H. Gumpertz. 1981. *Error Detection with Memory Tags*. Ph.D. Dissertation. Carnegie Mellon University.
[21] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (2006).
[22] 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. 2017. Efficient Tagged Memory. In *ICCD*.
[23] Piyus Kedia, Manuel Costa, Matthew Parkinson, Kapil Vaswani, Dimitrios Vytiniotis, and Aaron Blankstein. 2017. Simple, Fast, and Safe Manual Memory Management. In *PLDI*.
[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. 2016. SPARC M7: A 20 nm 32-Core 64 MB L3 Cache Processor. *IEEE J. of Solid-State Circuits* 51, 1 (2016).
[25] Doug Lea. 2000. A Memory Allocator. (2000). http://g.oswego.edu/dl/html/malloc.html
[26] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification.. In *NDSS*.
[27] Daiping Liu, Mingwei Zhang, and Haining Wang. 2018. A Robust and Efficient Defense Against Use-after-Free Exploits via Concurrent Pointer Sweeping. In *CCS*.
[28] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nuernberger, Wenke Lee, and Michael Backes. 2017. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In *NDSS*.
[29] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. 2017. SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities. In *NDSS*.
[30] S. S. Nagaraju, C. Craioveanu, E. Florio, and M. Miller. 2013. *Software vulnerability exploitation trends*. Technical Report. Microsoft.
[31] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *ISCA*.
[32] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *ISMM*.
[33] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.* 27, 3 (2005).
[34] Gene Novark and Emery D. Berger. 2010. DieHarder: Securing the Heap. In *CCS*.
[35] Yutaka Oiwa. 2009. Implementation of the Memory-safe Full ANSI-C Compiler. In *PLDI*.
[36] Oracle 2016. *Oracle's SPARC T7 and SPARC M7 Server Architecture*. Oracle.
[37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC*.
[38] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. *CoRR* abs/1802.09517 (2018).
[39] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. 2017. FreeGuard: A Faster Secure Heap Allocator. In *CCS*.
[40] Jr. Steele, Guy Lewis. 1977. *Data representations in PDP-10 MACLISP*. Technical Report AIM-420. MIT.
[41] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-free Detection. In *EuroSys*.
[42] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brook s Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE S&P*.
[43] Thomas Willhalm, Roman Dementiev, and Patrick Fay. 2012. *Intel Performance Counter Monitor - A Better Way to Measure CPU Utilization*. Intel.
[44] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Brooks Davis, Peter G Neumann, Robert Nicholas Maxwell Watson, Simon Moore, Anthony Fox, Robert Norton, and David Chisnall. 2019. Cheri concentrate: Practical compressed capabilities. *IEEE Trans. Comput.* (2019).
[45] 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. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *ISCA*.
[46] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel. In *CCS*.
[47] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel. In *CCS*.
[48] Yves Younan. 2015. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*.
[49] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2019. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *ASPLOS*.