Mind the Heap: Preventing Capability Leakage Across CHERI Compartments

Seyedhamed Ghavamnia
Bloomberg
New York, New York, USA
sghavamnia@bloomberg.net

Maurice D. Hanisch*
Caltech
Pasadena, California, USA
mhanisch@caltech.edu

Julien Vanegue
Bloomberg
New York, New York, USA
Imperial College London
London, UK
jvanegue@bloomberg.net

Abstract

CHERI enforces spatial memory safety and referential pointer integrity by replacing integer pointers with unforgeable capabilities that carry bounds and permissions. This rules out many conventional pointer-fabrication attacks and shifts exploitation toward the reuse of existing privileged capabilities obtained through vulnerabilities or unintended disclosure. CheriBSD's library compartmentalization reduces such disclosure by isolating per-compartment execution stacks. However, heap memory remains shared across compartments, and may lead to capability leakage through both reuse and explicit sharing.

In this paper, we show that CHERI's compartmentalization guarantees can be bypassed through two distinct attack vectors, even in the absence of allocator bugs. First, cross-compartment reuse of freed heap regions can leak privileged capabilities through residual allocator state. Second, unnecessary sharing of compartment-owned heap objects can leak capabilities to less-trusted compartments through ordinary program data structures and interfaces. To mitigate unnecessary sharing of live capabilities, we introduce *type-directed capability sealing*, which uses static analysis of library interfaces and CHERI sealing to preserve opaque-handle semantics for library-owned objects. As a complementary defense against residual capability reuse, we adapt the established practice of per-compartment allocation pools to CheriBSD's library-compartmentalization runtime. Our design integrates with CheriBSD's existing library compartmentalization mechanisms without requiring application or library modifications. Experimental results on micro- and macrobenchmarks show that the combined system incurs minimal overhead (less than 5% on average) while preventing capability leakage through both heap reuse and unnecessary cross-compartment sharing.

CCS Concepts

• **Security and privacy → Systems security**; **Operating systems security**;

*Work was performed during internship at Bloomberg.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
CCS '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846803

Keywords

CHERI, Compartmentalization, Type-based Sealing, User-space Allocator

ACM Reference Format:
Seyedhamed Ghavamnia, Maurice D. Hanisch, and Julien Vanegue. 2026. Mind the Heap: Preventing Capability Leakage Across CHERI Compartments. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846803

1 Introduction

CHERI is a hardware architecture [61] that strengthens memory safety by replacing conventional integer pointers with *capabilities*: unforgeable, metadata-bearing references that carry bounds and permissions and are enforced by hardware. A central property of this design is *capability integrity*: capabilities are associated with hardware validity tags that cannot be forged by software, preventing attackers from synthesizing new authoritative pointers through ordinary memory corruption. These guarantees substantially change the attacker's strategy. Many classic exploits rely on forging or corrupting pointers to reach sensitive code or data. Under CHERI, direct pointer fabrication is largely eliminated, so attacks increasingly depend on reusing existing valid capabilities [55]; making capability leakage a central security concern.

CHERI compartmentalization provides a hardware- and software-assisted mechanism for decomposing a process into protection domains and enforcing least privilege between trusted and untrusted components. In CheriBSD's library-based compartmentalization model, accidental capability leakage is reduced by separating execution stacks across compartments and mediating cross-domain calls via dedicated entry and exit paths [27, 61]. However, even with stacks and control transfers isolated, heap memory remains shared across compartments. This creates two opportunities for unintended capability leakage: (1) residual capabilities may be exposed through cross-compartment reuse of freed heap regions, and (2) live heap objects may explicitly carry capabilities across library boundaries through ordinary data structures.

The allocator is central to the first problem. User-space allocators acquire coarse-grained memory from the kernel, subdivide it to satisfy application requests, and in CHERI must return correctly bounded capabilities for each allocation. Consequently, the allocator and its interaction with the runtime form part of the effective trusted computing base (TCB).

The second problem arises from ordinary sharing patterns in systems software. Heap objects often contain capability-bearing state, such as dispatch tables, callbacks, or references to cryptographic keys and metadata [1, 4]. Libraries routinely return pointers to such objects, which programs then store in higher-level struct types and later pass to other libraries. This is commonly referred to as *data oversharing* and has been studied in previous works [36, 37, 44], but remains challenging. Refactoring existing software to eliminate these flows would require invasive changes to interfaces and struct types. Instead, the more practical goal is to preserve such sharing while preventing references from remaining usable in compartments that do not need authority to dereference them.

In this paper, we first demonstrate how allocator-level issues can erode CHERI's security guarantees, using proof-of-concept vulnerabilities that enable capability exposure. We then show that, even in the absence of allocator bugs, CHERI's compartmentalization guarantees can be bypassed using two distinct heap-specific attack vectors. First, cross-compartment reuse of freed heap regions can leak capabilities through residual allocator state. Second, unnecessary sharing of library-owned heap objects can expose capabilities through ordinary program struct types and interfaces.

To mitigate these risks, we design and implement two complementary mechanisms. Our primary design contribution is *type-directed capability sealing*, which automatically identifies library-owned heap objects whose capabilities are shared across compartments and uses CHERI sealing to preserve opaque-handle semantics. We complement this mechanism with a CheriBSD-specific adaptation of established per-compartment allocation pools, preventing freed memory from being reused across library compartments. Together, these mechanisms prevent unintended capability leakage through both heap reuse and unnecessary cross-compartment sharing while preserving compatibility with existing code. Furthermore, our work extends the original formalization of temporal safety[64] and its `Rocq` mechanization[65] which only apply sealing and unsealing logic for `store` and `load` operations without concern for application or function-level heap compartmentalization.

In summary, this paper makes the following contributions:

- We conduct an in-depth security analysis of the default allocator in CheriBSD, identifying and reporting an implementation defect that can undermine CHERI's security guarantees.
- We show that, even in the absence of allocator bugs, capability leakage can occur through two heap-related mechanisms: cross-compartment reuse of freed heap regions and sharing of heap object capabilities across compartments.
- We design and implement *type-directed capability sealing*, a static-analysis mechanism that protects library-owned capabilities shared across compartments by sealing references that need not be dereferenced outside their owning library.
- We adapt and evaluate established per-compartment allocation for CheriBSD's library-compartmentalization runtime, preventing cross-compartment reuse while preserving existing allocation and free interfaces.
- We evaluate the performance overhead of our mechanisms using micro- and macrobenchmarks, showing they have minimal overhead.

2 Background

In this section we briefly introduce CHERI and its security guarantees, and then describe how library compartmentalization is currently implemented in CheriBSD 25.03.

2.1 CHERI Hardware Guarantees

The CHERI architecture enforces *spatial memory safety* and *referential pointer integrity* by replacing conventional integer pointers with *capabilities*—architectural pointers augmented with metadata such as bounds and permissions. Capability integrity complements spatial safety by restricting how capabilities can be created and manipulated. Each valid capability is associated with a hardware-maintained validity tag, and capabilities can be produced only by deriving them from existing valid capabilities via dedicated CHERI capability instructions and registers.

Sealed Capabilities. CHERI's protection model includes *sealing*, which allows software to transform a capability into a non dereferenceable token that can still be stored and passed between components. A sealed capability cannot be used as an ordinary pointer until it is unsealed by suitable authority, making sealing a natural mechanism for encapsulation and opaque-handle patterns. More generally, the CHERI ISA treats sealing as one of the core operations for building higher-level isolation mechanisms on top of capabilities [9, 60]. In Arm Morello, sealing is represented through the capability `ObjectType` field: non-zero values denote sealed capabilities and zero is used for unsealed capabilities. Morello allocates 15 bits to this field, giving software a finite object-type space that is shared with any architecture- or runtime-reserved uses [9, 10].

2.2 Software Supporting CHERI

CheriABI. CHERI's architectural guarantees rely on a CHERI-aware ABI and operating system stack. In CheriABI [24], the kernel begins with broad authority and *delegates* it by deriving progressively more restricted capabilities as memory is allocated and mapped. Userspace receives its initial capabilities via the runtime linker and obtains further authority through system calls that return appropriately bounded and permissioned capabilities. Consequently, both the kernel and the runtime linker (RTLD) are part of the trusted computing base (TCB).

Temporal Safety. CHERI does not provide *temporal safety* in hardware; instead, CheriBSD enforces it in software through Cornucopia-style capability revocation, which clears tags from stale capabilities [25, 26]. On deallocation, freed regions are quarantined and *painted* in a shadow map; a revocation sweep then traverses process state and clears tags from capabilities still pointing into painted regions. Only once no live tagged capabilities remain is the region *unpainted* and reusable. Our work is orthogonal: it prevents cross-compartment capability leakage through heap reuse while remaining compatible with revocation.

Userspace Allocators. Userspace allocators are security-critical in CHERI systems because they mediate most heap allocations and derive the capabilities returned to applications. When additional memory is needed, the allocator obtains backing memory from the kernel, e.g., via `mmap`; assuming a CHERI-aware kernel, the returned

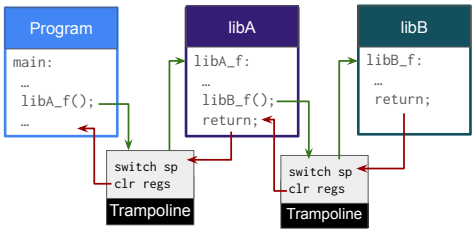


Figure 1: The runtime linker creates the library compartmentalization trampolines that change the execution stack and clear sensitive registers upon switching the compartment.

capability is bounded to the mapped region,[1] and the allocator subdivides it to satisfy individual requests. It must therefore return correctly bounded object capabilities and safely manage freed and reusable chunks. Defects can grant incorrect authority or expose residual capability-bearing memory.

2.3 CHERI Library Compartmentalization

CHERI provides architectural primitives (e.g., capability sealing) that can be used to construct protection domains, but practical compartmentalization requires system-level support beyond these low-level mechanisms [61]. CheriBSD 25.03 provides an experimental *library compartmentalization* facility (often referred to as `c18n`) implemented in the runtime linker (as shown in Figure 1). When enabled, each dynamically linked library executes as a separate compartment, and the runtime linker mediates cross-compartment calls by inserting trampolines and establishing a least-privilege capability environment based on ELF linkage (e.g., imported symbols) [27]. CheriBSD also includes ongoing work toward *sub-library* compartmentalization (finer-grained compartments such as object-file or feature-level partitions), but this support remains incomplete; therefore, we target the library-level model in this work.

3 Threat Model

We consider compartmentalized applications running on CheriBSD with library-based compartmentalization enabled. Our trusted computing base (TCB) consists of the CHERI hardware, the CheriBSD kernel, the run-time linker, and the trusted build and instrumentation pipeline used to derive and enforce our type-directed sealing policies. Accordingly, the analyzed program configuration (including its libraries and source- or bitcode-level inputs) is fixed before deployment, and the generated sealing policy and accessor functions are assumed correct for that configuration.

After deployment, we assume that a bug or vulnerability may allow an attacker to compromise a non-TCB compartment and execute with that compartment's authority. The attacker's goal is to obtain capabilities owned by other compartments, either by reclaiming freed heap memory that still contains residual capabilities or by misusing references to shared heap objects. A compromised compartment may invoke its permitted interfaces arbitrarily, retain and replay opaque handles, and exploit any authority it already holds. Our goal is to prevent such a compartment from gaining additional authority through cross-compartment heap reuse or by directly dereferencing shared objects whose types it does not own, except where the benign program requires field access.

[1] Subject to representational imprecision from capability compression [62].

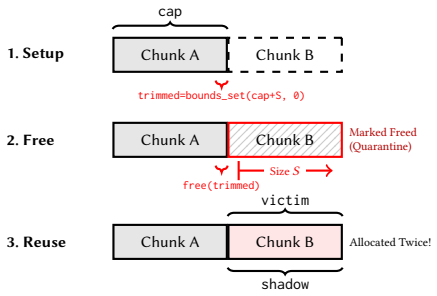


Figure 2: Shadow Allocation Exploit overview: Misaligned quarantine bookkeeping targets the adjacent Chunk B, allowing it to be subsequently allocated to both attacker and victim simultaneously.

4 Problem Overview and Motivation

In this section, we discover CHERI heap allocator vulnerabilities in CheriBSD's default allocator that enables capability exposure. We further generalize this to show how capability leakage can arise from heap reuse or capability sharing between compartments.

4.1 CHERI Allocator Vulnerabilities

Userspace allocators are critical to achieving the security guarantees that CHERI is designed to provide (as discussed in Section 2.2). To illustrate that vulnerabilities arise in practice, we investigated vulnerabilities discovered in the userspace allocators of CheriBSD. We responsibly disclosed these issues, and a fix has since been merged into the upstream repository for *jemalloc*.

Vulnerabilities in jemalloc. The default allocator, *jemalloc*, was found to accept *trimmed* capabilities (non-base pointers) in `free()`, desynchronizing its quarantine bookkeeping from architectural revocation. This can lead to two distinct classes of exploitation:

- **Shadow Allocation**: By passing a zero-length capability pointing to the end of an allocation (Chunk A), the allocator incorrectly places the *adjacent* Chunk B into quarantine. As shown in Figure 2, this compromises the allocator state and makes it return two capabilities to the same physical region.
- **Unpainted Capability Windows**: This vulnerability exploits the gap between shadow-map painting and architectural invalidation. By timing interleaved revocation cycles, an attacker can trick the allocator into unpainting the shadow bits of a chunk's header just before the revoker inspects it. This effectively hides a dangling pointer from invalidation, resulting in a persistent Use-After-Realloc condition.

Vulnerabilities in dlmalloc. Implementation defects are not limited to jemalloc. We discovered a similar class of vulnerabilities in an earlier CheriBSD allocator based on *dlmalloc*. Insufficient validation allowed an attacker to craft malformed chunk headers using `CHERI_PERM_SW_VMEM` permissions, triggering heap corruption or overlapping allocations..

4.2 Heap-based Capability Reuse Attack

Heap reuse can disclose capabilities even in the absence of allocator *implementation* bugs. Because allocations are typically returned uninitialized and freed chunks are placed back on allocator free lists without being zeroed, newly allocated memory may expose residual contents from a prior object, including capabilities. We refer to this class of exploits as *heap-based capability reuse attacks*, in which an adversary seeks to obtain previously allocated and freed heap chunks still containing useful capabilities.

While analogous reuse attacks exist in conventional systems, their impact is amplified under CHERI. Since capabilities are unforgeable, an adversary cannot fabricate high-privilege pointers and must instead obtain existing valid capabilities through disclosure or reuse. Residual heap contents therefore become a particularly valuable target. In CheriBSD's library compartmentalization model, call boundaries and stack state are constrained, but freed heap memory may still be reissued across compartments. If a privileged compartment frees an object containing sensitive capabilities and the same chunk is later reused by a less-trusted compartment, those capabilities may be disclosed.

To confirm that such capabilities persist in practice, we scanned Nginx running in pure-capability mode on Arm Morello under CheriBSD using a `ptrace`-based capability scanner. Under load, we identified 1,526 heap-resident executable sentry capabilities corresponding to 158 distinct Nginx functions. These were not isolated artifacts, but covered core Nginx functionality, including the I/O layer, the full HTTP filter pipeline, and the HTTP phase-dispatch functions. Several per-request handler slots also appeared repeatedly, yielding hundreds of instances at peak load. Consequently, an attacker who obtains read access to a freed `ngx_http_request_t` object may recover valid sentry capabilities into Nginx's active execution surface without forging any pointer. A straightforward mitigation is to clear memory on allocation or free. However, as shown in Section 7.3.1, pervasive clearing is often too costly.

4.3 Heap Data Oversharing

Programs and their associated libraries frequently share state and global conditions through heap objects. Often, an application acts as a conduit, storing pointers to objects allocated by one library and subsequently passing them to other, unrelated libraries.

Historically, sharing a single heap has been highly problematic, as vulnerabilities in one library can lead to the exposure or corruption of objects owned by completely different libraries [3, 5, 6]. For instance, CVE-2017-7529 [3] exploits an integer overflow in nginx's range filter module to read adjacent heap memory, exposing SSL session keys. Similarly, CVE-2022-41741 [6], a buffer overflow in the MP4 module, can corrupt adjacent `SSL_CTX` structures.

While CHERI capabilities effectively prevent these spatial overwrites and overreads, the need to pass complex structures between modules introduces a new attack surface: *capability leakage*. Consider how nginx handles TLS connections: it receives an `SSL*` pointer from OpenSSL's `SSL_new()` and stores it in an `ngx_ssl_connection_s` structure. Later, during gzip compression of HTTPS responses, nginx passes its `ngx_http_gzip_ctx_t` context, which holds a reference to that connection structure,to zlib.

As a result, zlib, a library with no legitimate need to access TLS state, receives a valid, dereferenceable capability to OpenSSL's `SSL` object. The impact is severe due to capability transitivity. The `SSL` struct type includes pointers to the `SSL_CTX`, `SSL_SESSION`, and the `EVP_CIPHER_CTX` struct types, the latter holding active encryption keys. A single unsealed `SSL*` capability thus transitively exposes the entire TLS stack to any library that receives it. This motivates a mechanism to allow sharing of these capabilities but preventing their use outside the trusted context.

5 Design

Our design addresses two distinct sources of heap-mediated capability propagation across compartments. First, freed heap memory may be reused by a different compartment, exposing residual capabilities left by the previous occupant of that region. Second, pointers to live heap objects may be intentionally shared across compartment boundaries even when the receiving compartment does not require authority to dereference them. We address these problems with two complementary mechanisms: *heap compartmentalization*, which confines heap reuse to the allocating compartment, and *type-directed capability sealing*, which preserves opaque-handle semantics for shared compartment-owned objects.

These mechanisms rely on two different notions of ownership. Allocation ownership is an established isolation policy that restricts heap reuse to the compartment that owns an allocation. Our primary design contribution is a type-ownership policy that is automatically derived from cross-compartment use and enforced through CHERI sealing and generated accessors. Thus, compartment-aware allocation controls storage reuse, whereas type-directed sealing controls dereference authority.

Both mechanisms are designed to integrate with CheriBSD 25.03's library-level compartmentalization model, in which dynamic libraries execute as distinct compartments and cross-compartment transfers are mediated by the runtime linker. Under this model, heap compartmentalization requires the allocator to identify the currently active compartment at each allocation operation, while type-directed sealing requires determining which shared struct types are owned by which compartments and where references to their objects cross compartment boundaries.

5.1 Heap Compartmentalization

We use heap compartmentalization as supporting infrastructure for preventing residual capability disclosure through cross-compartment heap reuse. The general strategy of restricting allocator reuse by compartment is established in prior compartmentalization systems [32, 63]; our contribution is its integration with CheriBSD's library-compartment transitions and its evaluation on CHERI applications.

Heap compartmentalization addresses capability leakage through allocator reuse. In a conventional process-wide heap, memory freed by one compartment may later be reallocated to another, potentially exposing residual capabilities left in the reused chunk. Our goal is to enforce *allocation ownership*: memory allocated for a compartment must be reused only within that same compartment's heap domain.

5.1.1 Allocator Model. Our design gives each compartment a logically distinct heap domain, even though all memory remains managed by a single allocator implementation. Allocation requests are attributed to the compartment whose code is currently executing, and freed memory is returned to that same compartment's reuse domain. This preserves compatibility with existing applications and libraries: no compartment identifier must be passed explicitly at allocation sites, unlike prior approaches based on compiler-assisted identifier propagation [2].

To realize this policy, the allocator maintains a per-thread *current compartment identifier* and uses it to select the target heap domain for allocation. Memory freed by any compartment is returned to the heap domain of the compartment that originally allocated it. This preserves cross-compartment allocation/free behavior while ensuring that recycled heap memory is never reused outside its allocation-owning compartment. Direct `mmap`-allocated memory lies outside this policy, but because it is provisioned by the kernel rather than drawn from allocator free lists, it cannot recover allocator-recycled chunks from other compartments.

5.1.2 Transition Consistency. Since allocation attribution depends on the currently active compartment, heap compartmentalization must remain synchronized with execution context. We therefore treat the current compartment identifier as trusted per-thread state and update it only on trusted compartment transitions. In CheriBSD's library compartmentalization model, such transitions are already mediated by runtime-linker-generated trampolines, making the runtime linker the natural place to maintain allocator state.

Accordingly, each cross-compartment transition updates the allocator's thread-local compartment identifier before control enters the callee. An allocation is therefore owned by the compartment active at the allocation request call site, not by the original top-level caller. This naturally handles well-nested cross-compartment call chains and preserves compatibility with existing code, while inheriting thread-safety and asynchronous-execution correctness from CheriBSD's existing compartmentalization machinery.

5.2 Type-Directed Protection for Shared Heap Objects

Heap compartmentalization prevents capability leakage through allocator reuse, but it does not address objects whose pointers are intentionally shared across compartments while still live. In compartmentalized applications, one compartment may return a pointer to an internal object, store it inside a larger struct type, or pass it through another interface to a different compartment; leaking authority to them.

This pattern is common in C software. A program may maintain a context struct type containing pointers to objects owned by multiple compartments, and that context may later be passed across additional compartment boundaries. Refactoring existing software to eliminate these flows would require pervasive changes to interfaces and struct types. Instead, our design preserves these programming patterns while ensuring that shared objects remain directly dereferenceable only within the compartment that owns their representation.

Our approach therefore has two steps. First, we derive type ownership and cross-compartment sharing behavior through static analysis. Second, we use this information to apply CHERI sealing to shared references whenever they leave their owning compartment.

5.2.1 Type Ownership and Sharing Classification. Our design relies on a notion of *type ownership*: a struct type is owned by the compartment that defines and manipulates its internal representation. For example, OpenSSL owns types such as `SSL_CTX` and `SSL`, while the nginx compartment owns types such as `ngx_connection_t`, even though other compartments may store and forward pointers to objects of those types. This notion of ownership determines which compartment is responsible for protecting a shared reference when it crosses a compartment boundary.

To apply this policy, we derive a per-compartment summary of struct type ownership, interface exposure, and cross-compartment use. This summary identifies which types are owned by each compartment, where capabilities to objects of those types cross compartment boundaries, and whether non-owning code uses them opaquely or through direct field access.

Concretely, the analysis records four kinds of information. First, it identifies the struct types defined within each compartment and attributes ownership of those types accordingly. Second, it records the struct-typed parameters and return types of externally-visible functions, thereby identifying the interface points through which heap-object references may cross compartment boundaries. Here, externally visible includes both exported functions and address-taken functions (to overapproximate indirect function calls). Third, it combines these summaries with cross-compartment call information to identify the actual sharing relationships that arise in the application. Finally, for each shared type, it determines whether non-owning code merely stores and forwards the pointer or directly accesses fields of the object (by analyzing function bodies).

This yields a three-way classification of shared types. A type is *sealable* if all non-owning compartments treat it as an opaque handle. A type *requires mediated access* if non-owning code accesses only a restricted subset of fields. A type is *not practically sealable* if non-owning code depends extensively on direct field access, making fine- grained mediation unattractive or ineffective. This classification drives selective enforcement: opaque-handle types can be sealed directly, while the small number of shared types with limited field access require explicit mediation.

5.2.2 Sealing Policy. Guided by the sharing analysis, we protect cross-compartment heap-object references using CHERI's sealing mechanism. The governing rule is simple: a shared object may be directly dereferenced only within the compartment that owns its type, unless the owner explicitly exposes restricted access to selected fields. Accordingly, when a capability to an owned object leaves its owning compartment, it is converted into a sealed capability. The resulting reference remains passable and storable, but it is no longer directly dereferenceable by non-owning code.

Each compartment therefore holds sealing authority for the types that it owns. The owning compartment seals references to its objects before those references cross a compartment boundary and unseals them when they return. Protection is applied at function entry, function return, call sites, and selected field accesses according to type ownership (Table 1 and Figures 3 and 4).

Table 1: Cross-compartment sealing scenarios. Each scenario describes a pointer-passing pattern at a compartment boundary and the corresponding instrumentation action. All seal and unseal operations are guarded: null pointers and already-sealed/unsealed pointers are skipped at runtime. Ext. func: an externally-visible function (exported functions and address-taken functions); Int. func: only internally visible; Ind. call: indirect function call

#	Scenario	Action	Condition
A. Function entry (callee side)			
A1	Ext. func	`unseal`	Arg type is owned
A2	Int. func	None	
B. Function return (callee side)			
B1	Ext. func	`seal`	Return type is owned
B2	Int. func	None	
C. Handling return value (caller side)			
C1	Ext. func	`unseal`	Caller owns return type
C2	Ind. call	`unseal`	Caller owns return type
C3	Int. func	None	
D. Calling a function (caller side)			
D1	Ext. func	`seal`	Arg type is owned
D2	Ind. call	`seal`	Func-ptr base ≠ current code base, owned arg-type
D3	Int. func	None	
E. Field access			
E	Struct field	`accessor`	Non-owner type

Table 1 summarizes the cases in which sealing, unsealing, or accessor mediation is required. These cases are most easily understood by separating callee-side handling, caller-side handling, and field accesses.

- **Callee-side function entry (A).** On entry to a function, the relevant question is whether the callee may receive sealed arguments. Internal functions require no action (A2), since they are only reached within the current compartment and therefore never receive cross-compartment sealed arguments. Externally-visible functions, by contrast, may be invoked from another compartment, so any argument whose type is owned by the callee compartment is unsealed on entry (A1). This is illustrated in Figure 3 by `SSL_new` and `SSL_set_ex_data`, which unseal their owned OpenSSL arguments on entry.
- **Callee-side function return (B).** On return, an externally-visible function seals its return value whenever the returned type is owned by the callee compartment (B1). Internal functions perform no such transformation (B2), since their return values do not cross a compartment boundary. Figure 3 shows this in `SSL_new`, where the returned `SSL*` is sealed before returning to nginx.
- **Caller-side direct calls and returns (C1, C3, D1, D3).** For direct calls, the caller can distinguish externally-visible from internal callees statically. When calling an externally-visible function, the caller seals any argument whose type it owns before the call (D1). When calling an internal function, no sealing is applied (D3). The same distinction governs returned values: after a call to an externally-visible function, the caller unseals the result if the return type is owned locally (C1), whereas returns from internal calls are left unchanged (C3). Return-side unsealing is based on the type as used by the caller, rather than the callee's declared return type, which correctly handles interfaces that return a generic `void*`. Figure 3 shows D1 when nginx seals `ngx_conn_t*` before passing it to `SSL_set_ex_data`, and C1 when nginx unseals the result of `SSL_get_ex_data`.
- **Caller-side indirect calls and returns (C2, D2).** Indirect calls are less straightforward because the callee is not known statically. For outgoing arguments, we avoid full pointer analysis and instead emit a runtime check that compares the base address of the function pointer capability against the base address of the current compartment (from the program counter capability). If the two bases differ, the call is treated as cross-compartment; any argument whose type is owned by the caller is then sealed before the call (D2). If the bases match, the call is treated like an internal call. Figure 4 shows this in `ssl_rd_bytes`, where the indirect call through `info_cb` is instrumented with a runtime base-address check; `SSL*` is sealed only if the target is in a different compartment.
  Return handling after an indirect call is decided separately. Because the caller does not know which function was invoked, it cannot infer from callee identity whether the result has been sealed. Instead, it uses the return type at the call site and performs a guarded unseal whenever that type is owned locally (C2). Figure 4 shows this in `ssl_get_by_char`, where the returned `SSL_CI*` is unsealed after an indirect call through a method pointer.
- **Field accesses (E).** When a non-owning compartment directly accesses a field of a protected type, sealing alone is insufficient because the object must remain opaque outside the owning compartment. In this case, the direct field access is replaced with an accessor function supplied by the owning compartment (E). Figure 4 shows this for nginx's accesses to `ASN1_INTEGER.length` and `ASN1_INTEGER.data` in `ngx_ssl_ocsp_key`.

A key invariant is that protection decisions remain local to the owner of the data. The callee decides whether a returned value must leave the compartment in sealed form based on the ownership of the returned type, while the caller decides whether a received value should be unsealed only when it owns the underlying type. This includes cases where the callee exposes a generic `void*` interface but the caller's debug information resolves the concrete type. A non-owning compartment therefore cannot influence the decision to expose another compartment's internal representation.

Finally, protection is applied to the shared pointer itself rather than recursively to the full reachable object graph. If a compartment shares a pointer to one of its own objects, that top-level pointer is sealed before it crosses the boundary, but the system does not recursively traverse the object and seal every capability reachable from

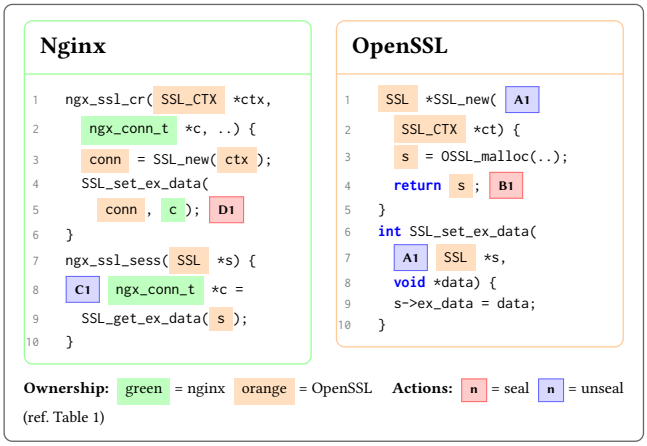


Figure 3: Direct-call sealing scenarios across nginx and OpenSSL. D1 nginx seals `ngx_conn_t*` before passing it to `SSL_set_ex_data`. C1 nginx later unseals the returned value from `SSL_get_ex_data` when the type is owned locally. A1 OpenSSL unseals owned arguments on entry, and B1 seals owned return values such as the `SSL*` returned by `SSL_new`.

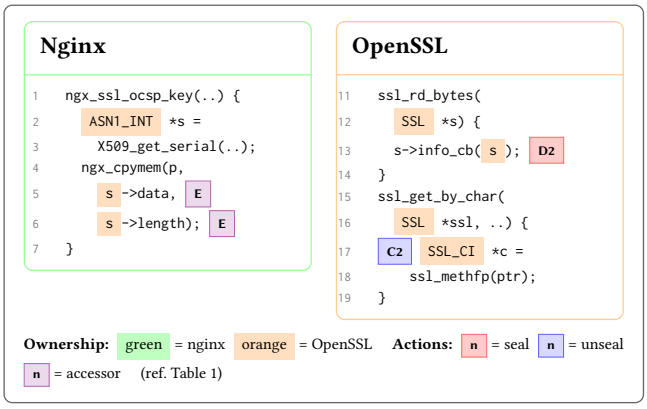


Figure 4: Accessor-mediated and indirect-call scenarios. E nginx reads `data` and `length` from OpenSSL's `ASN1_INTEGER` via generated accessor calls, since nginx does not own the type. D2 an indirect call site is instrumented to seal an owned argument when a runtime base-address check determines that the target lies outside the current compartment. C2 after an indirect call, the caller performs a guarded unseal when the returned type is owned locally.

its fields. This choice preserves compatibility with existing code while remaining consistent with the intended opaque-handle abstraction: non-owning code cannot follow internal pointers except through explicitly mediated access paths.

6 Implementation

We implement heap compartmentalization in CheriBSD 25.03 by modifying the default allocator, *jemalloc*, and the runtime-linker trampolines used for library compartmentalization. We implement type-directed capability sealing using an LLVM 15-based analysis and instrumentation pipeline. This section describes the concrete realization of the design.

6.1 Heap Compartmentalization

6.1.1 Allocator Compartmentalization. We implement heap compartmentalization in CheriBSD's default allocator, *jemalloc*. Jemalloc already partitions allocation state into multiple arenas, which serve as the isolation boundary in our design. We therefore map each compartment identifier and thread combination to a dedicated arena and confine reuse to that arena, without changing the external allocator API.

To track the last active compartment (at the time of calling an allocation function), we add an externally visible 64-bit thread-local variable to `libc`, which contains jemalloc. This variable stores the last active compartment identifier and is updated by the runtime during compartment transitions. We then use this identifier at the point where jemalloc selects the arena for each allocation request.

Specifically, jemalloc selects a thread cache and an arena through `tcache_get` and `arena_choose_impl`; we extend both to consult the thread-local compartment identifier. Each thread maintains a small fixed-size power-of-two hash table (open addressing, linear probing) mapping compartment identifiers to per-compartment thread caches, allocated lazily on first use. On a compartment change the allocator switches the active thread cache, and on first visit to a compartment it creates a fresh arena private to that (thread, compartment) pair. Because these arenas are not published globally, threads sharing a compartment identifier do not share an arena, avoiding the bin-lock contention a shared per-compartment arena would introduce.

6.1.2 Runtime Loader. CheriBSD's library compartmentalization mechanism uses runtime-linker-generated trampolines to mediate transitions between library compartments. We extend this path so that, on each domain transition, the trampoline also updates the thread-local compartment identifier used by jemalloc.

Initializing TLS metadata. Because the allocator's compartment identifier is thread-local, the runtime loader must determine how to locate that variable within each thread's TLS block. We therefore extend RTLD initialization so that, when loading `libc`, it identifies the TLS metadata corresponding to the allocator's compartment identifier and records the information needed to reconstruct its address for each thread during compartment transitions.

RTLD trampoline updates. When library compartmentalization is enabled, RTLD resolves cross-compartment calls to generated trampolines rather than directly to the callee entry point. These trampolines already perform the stack switch required for cross-compartment execution. We extend the per-compartment trampoline state so that, in addition to the target stack address, it also carries the TLS location of the allocator's compartment identifier for the current thread. Prior to entry to the callee compartment, the trampoline updates that TLS slot with the callee's compartment

identifier before transferring control. This design preserves the existing ABI and requires no source-level changes to applications or libraries. The RTLD modification is allocator-agnostic except for knowledge of the allocator's thread-local compartment identifier.

6.2 Type-Directed Capability Sealing

We implement type-directed capability sealing using LLVM 15. The tooling comprises three stages: a per-compartment type-usage analysis, a cross-compartment sharing analysis, and an instrumentation pass that inserts seal and unseal operations at the required boundaries. Each target compartment is compiled to LLVM bitcode with debug information (`-g`) and processed independently.

6.2.1 Type-Usage Analysis. The first stage is an LLVM module pass, which extracts a type summary from each bitcode module and emits a JSON manifest describing the compartment's interface with respect to its struct types. For each struct type, the pass records the type name, field count, and field layout. Since LLVM 15 uses opaque pointers, the pass relies on DWARF debug information to recover the concrete struct types associated with function parameters and return values. The resulting manifest is compartment-local: it describes only the code and types present in that module and can therefore be reused across different applications that link against the same component.

6.2.2 Cross-Compartment Sharing Analysis. The second stage combines the per-compartment manifests to determine which types cross compartment boundaries and how they should be protected. A struct type is considered *shared* if it appears as a parameter or return type on a call edge between two different compartments. For each shared type, the analysis determines the owning compartment and classifies how each non-owning compartment uses it (e.g., sealable). System-defined types such as `struct sockaddr` are excluded, since they are intentionally shared across all compartments. The analysis then emits an instrumentation configuration specifying the required actions based on the scenarios discussed in Table 1.

6.2.3 Sealing Enforcement. The third stage is an LLVM module pass, which reads the instrumentation configuration and inserts CHERI seal and unseal operations into each module.

Compartment initialization. For each module, the pass creates a file-scope global variable, to hold the compartment's sealing capability. It also generates an initialization function, registered as a `.init_array` constructor, which derives a per-compartment sealer with a unique object type.

Seal and unseal insertion. At each configured enforcement point, the pass emits the corresponding CHERI intrinsic. All operations are guarded by a null check and a sealed-state check so that null values and already correctly-stated capabilities bypass the transformation. This is needed since the same function may be called both internally and externally (affecting the sealing status of its arguments).

Internal call compensation. Since we always seal the return value of an externally-visible function, internal callers within the same compartment may receive a sealed return value that they must unseal. The pass handles this using the resolved return type at the call site, including indirect calls whose return types are recoverable from debug information.

Accessor generation. For types that require mediated access, the pass generates accessor functions in the owning compartment's bitcode. Each accessor unseals the pointer, performs the required field access, and returns the result. In the non-owning module, direct field accesses are replaced with calls to the corresponding accessor. This preserves the seal boundary while allowing controlled, field-granular cross-compartment data access.

7 Evaluation

We evaluate the practicality and security benefit of our two mechanisms on CheriBSD 25.03 running on Arm Morello, a CHERI-enabled platform that enforces capability bounds and permissions in hardware. Our evaluation addresses three questions. First, what coverage does type-directed capability sealing achieve across realistic compartmentalized applications? Second, what runtime and memory overheads are introduced by heap compartmentalization and sealing? Third, how effectively do the two mechanisms prevent cross-compartment capability propagation through heap reuse and explicit sharing? Unless otherwise stated, all experiments use CheriBSD's standard security configuration, including library compartmentalization and temporal-safety support (through Cornucopia [25, 26]), and compare an unmodified CheriBSD 25.03 baseline against our modified runtime and allocator.

7.1 Experimental Setup

We evaluate four configurations:

(1) **Baseline:** unmodified CheriBSD 25.03 with the default runtime linker and default `libc`/jemalloc (with library compartmentalization enabled).
(2) **HC-off:** modified runtime linker and modified `libc`/jemalloc, with heap compartmentalization disabled.
(3) **HC-on:** modified runtime linker and modified `libc`/jemalloc, with heap compartmentalization enabled.
(4) **Seal:** Type-directed capability sealing instrumentation.

This structure isolates the cost of each component. Comparing Baseline and HC-off captures framework overhead introduced by our changes independent of heap partitioning itself. Comparing HC-off and HC-on isolates the cost of heap compartmentalization. Comparing Baseline and Seal reports the overhead of type-directed sealing instrumentation.

7.2 Sealing Coverage and Compatibility

We first evaluate the applicability of type-directed capability sealing across a set of compartmentalized applications. Our analysis excludes system-defined libraries and types from sealing. In practice, few structured types are naturally owned by these libraries, and those that do exist (e.g., socket-related types) are typically accessed across many compartments, limiting the security benefit of sealing them. Table 2 summarizes the cross-compartment shared types, their classification, and the number of instrumentation points.

Across the six applications, we identify 161 shared struct types that cross compartment boundaries. Of these, 150 (93%) are fully sealable, 8 (5%) require mediated field access through accessors,

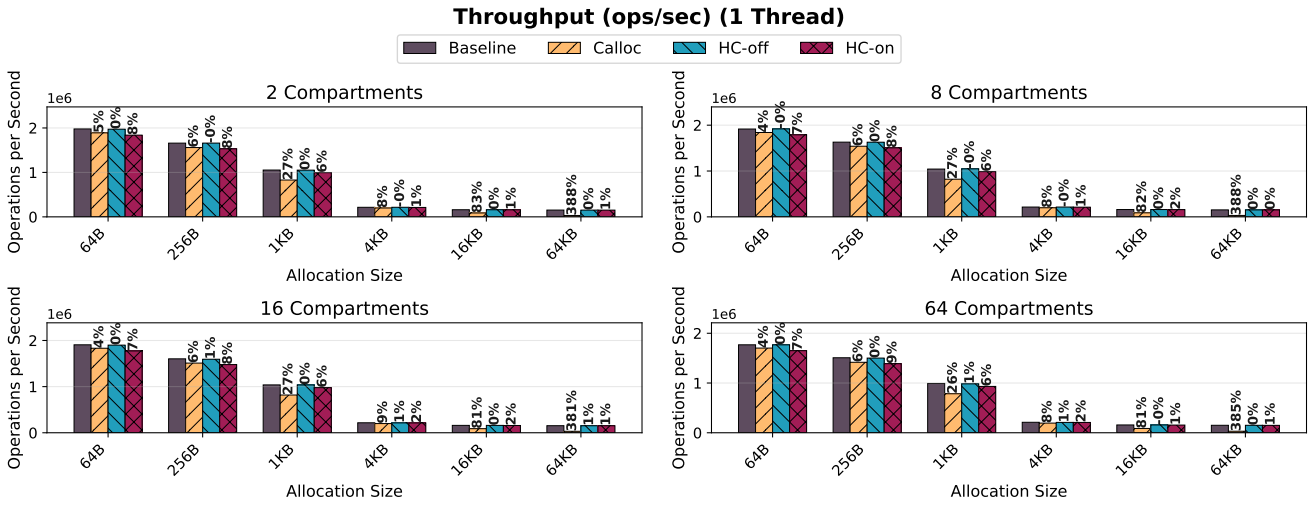


Figure 5: Microbenchmark results for heap compartmentalization. We report throughput for Baseline, `calloc`, HC-off, and HC-on while varying the number of compartments and allocation size.

Table 2: Cross-compartment sealing instrumentation summary for 6 applications.

App	#Compartments	Struct Types				Instrumentation Points		
		Shared	Seal	Acc.	NoSeal	Seal Ins.	Unseal Ins.	Acc. Repl.
nginx	4	39	37	2	0	1,300	6,754	21
redis	2	11	11	0	0	581	3,650	0
memcached	3	14	14	0	0	425	2,348	0
curl	3	41	37	3	1	1,772	8,350	68
wget	3	25	23	1	1	631	4,022	14
lighttpd	3	31	28	2	1	2,118	9,728	10
Total		**161**	**150**	**8**	**3**	**6,827**	**34,852**	**113**

and only 3 (2%) are not practically sealable. This indicates that most shared objects in these applications already follow an opaque-handle discipline and can therefore be protected with sealing.

We further validated functional compatibility on all six applications in Table 2 by running their existing test suites under the Seal configuration and comparing the results against their baseline. Across all six applications, type-directed sealing preserved baseline test-suite behavior: every test that passed on the CheriBSD baseline also passed under Seal. This indicates that the inserted seal/unseal operations and accessor replacements preserve intended application behavior across the evaluated software.

7.3 Performance Evaluation

7.3.1 Microbenchmark: Heap Compartmentalization Overhead. We use a custom microbenchmark to isolate the cost of heap compartmentalization. The benchmark forces allocator requests to originate from distinct DSOs so that runtime-linker-mediated compartment switches occur at the intended boundary. We evaluate Baseline, HC-off, and HC-on, and also compare against a `calloc`-based configuration that represents the conventional alternative of eager zeroing. We do not include Seal here, since the benchmark does not exercise realistic shared-object interfaces.

To exercise compartment-aware allocation, the benchmark routes allocation and free requests through a configurable set of lightweight shared libraries, each exporting wrappers around `malloc` and `free`. The executable links against up to 64 such DSOs so we can test up to 64 active compartments at runtime, ensuring that allocator calls originate from distinct DSOs and trigger compartment-ID updates. In our evaluation, we vary the number of active compartments (2, 8, 16, 64), the number of threads (1, 4, 8), and the allocation size (64B, 256B, 1KB, 4KB, 16KB, 64KB). The `calloc` variant serves as a reference point for mitigating residual-data reuse through eager clearing rather than compartmentalized reuse control.

We leverage throughput (operations per second) to evaluate overhead (shown in Figure 5. Throughput and memory usage are largely insensitive to the number of compartments and threads, making our approach scalable. The throughput overhead is as low as 1% and the average across all settings is 4%. The memory overhead is close to 1%. Compared with `calloc`, which can have up to 388% overhead, our approach avoids the high cost of eager zeroing while still preventing cross-compartment reuse of residual capability-bearing memory.

7.3.2 Macrobenchmark. We evaluate the runtime overhead on three server applications: Nginx (4 compartments), Lighttpd (3 compartments), and Redis (2 compartments). We focus on server workloads because they exercise realistic cross-compartment interfaces under sustained request processing. Memcached is excluded: under large loads, the CheriBSD baseline failed with a bus error due to a CheriBSD compatibility issue unrelated to our instrumentation. To isolate sealing overhead from client-side concurrency effects,

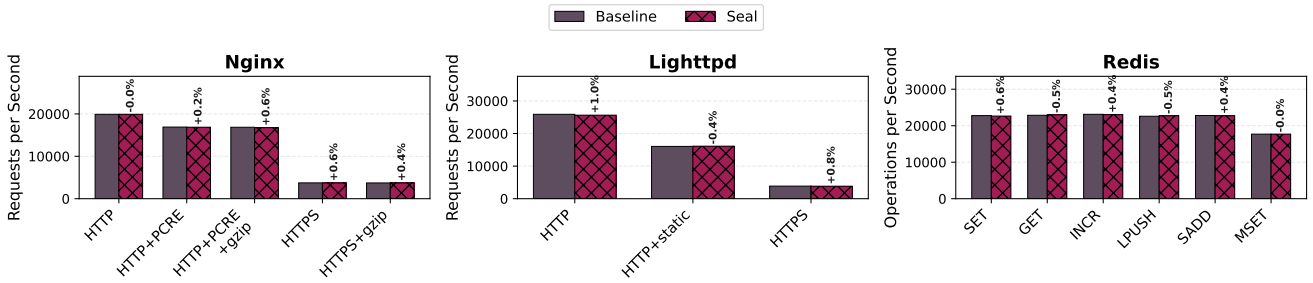


Figure 6: Macrobenchmark results for applying sealing protection for server applications in our dataset.

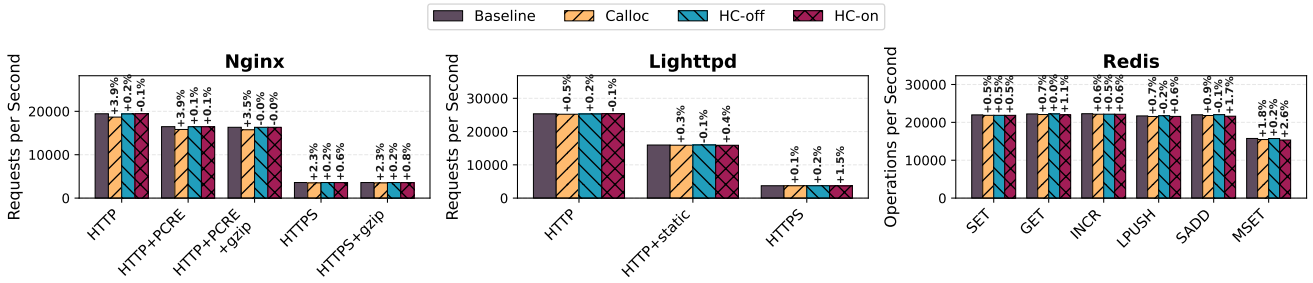


Figure 7: Macrobenchmark results for comparing calloc and heap compartmentalization performance.

each experiment uses a single client thread, and we report averages over ten runs. For Nginx and Lighttpd, we use ApacheBench (ab) to measure request throughput for HTTP and HTTPS workloads. For Redis, we use `redis-benchmark` with 100,000 operations per command type. Figure 6 shows that overhead is negligible for all workloads and applications and remains below 1% in all cases.

`calloc` *Macrobenchmark.* We evaluate the impact of replacing `malloc` with `calloc` in our target applications using macrobenchmarks (without applying type-directed sealing). Figure 7 shows how the overhead of calloc and heap compartmentalization differ between different applications. While calloc has a maximum of 3.9% overhead for Nginx, HC always remains below 1%. Whereas the SADD and MSET benchmarks for Redis incur 1.7% and 2.6% overhead of HC, and 0.9% and 1.8% overhead for calloc.

7.4 Security Evaluation

7.4.1 Preventing Cross-Compartment Heap Reuse. To validate that heap compartmentalization prevents cross-compartment reuse, we re-executed the microbenchmark from Section 7.3.1 while recording the address ranges returned by each allocation. Under the unmodified CheriBSD baseline, same-sized allocations frequently reuse recently freed chunks, and our harness observes overlap across compartments in all tested configurations. With heap compartmentalization enabled, we do not observe any cross-compartment overlap: allocations are consistently served from compartment-specific regions, and freed chunks are returned to their allocation-owning compartment rather than being reissued to other compartments.

7.4.2 Protecting Explicitly Shared Heap Objects. Most cross compartment struct types are directly amenable to sealing. Across the six applications, 93% of the shared types are used opaquely by non-owning compartments and therefore require no field access. To assess whether the remaining accessor-mediated cases weaken protection, we examined each accessed field and its transitive reachability.

Table 3 identifies the five struct types for which at least one application requires field-level access. Three of them, `z_stream_s`, `asn1_string_st`, and `ossl_param_st`, expose only scalar state or raw data buffers and do not permit traversal to other typed struct objects. `X509_info_st` contains struct pointers (e.g., `X509*` and `X509_CRL*`), but these do not recursively expose a larger internal object graph; reachability terminates at the first level. The remaining type, `GENERAL_NAME_st`, is a tagged union for which the non-owning compartment accesses all fields, and is therefore excluded from sealing because accessor mediation would provide little or no additional protection.

The most security-sensitive case is `X509_info_st`, which may expose certificate pointers. These accesses nevertheless reflect intended application behavior: Curl inspects certificate state to configure TLS connections and validate peers. Importantly, the capability to `X509_info_st` remains sealed, and any subsequent sharing of it across compartments preserves that sealed representation. A security risk arises only if the program explicitly extracts one of the struct's fields and passes that field to another compartment.

Runtime validation. To further validate that sealing eliminates traversable cross-compartment object paths, we performed a limited

Table 3: Struct types requiring accessor functions. #Fields reports the number of fields accessed by non-owning compartments (union across all consuming applications) over the total number of fields in the struct. †Cannot-seal: all fields accessed by non-owner; excluded from sealing.

Owner	Struct Type	Apps	#Fields	Struct Ptr
zlib	z_stream[1]	curl, lighttpd, nginx, wget	10/14	No
OpenSSL	asn1_str[2]	curl, lighttpd, nginx	3/4	No
	GEN_N[3,†]	curl, wget	2/2	Yes
	X509_inf[4]	curl	2/6	Yes
	ossl_par[5]	lighttpd	3/5	No

[1]`z_stream_s` [2]`asn1_string_st` [3]`X509_info_st` [4]`ossl_param_st` [5]`GENERAL_NAME_st`

runtime reachability analysis on nginx by embedding a capability scanner directly into the main program. The scanner performs a typed traversal rooted at the global `ngx_cycle_t`, following the application's internal struct types to pointers that cross from nginx into OpenSSL, including the `SSL_CTX*` stored in server configuration state. For each such boundary capability, the scanner checks whether the capability is sealed before attempting further traversal. In the baseline binary, these pointers are unsealed, and the scanner can continue into the OpenSSL compartment and reach internal objects, whereas sealing in the instrumented binary stops the traversal at the compartment boundary.

8 Formal Heap Compartmentalization Model

We proceed to defining the formal model of heap compartmentalization using CHERI *seal* and *unseal* primitives. Our formalization is fully mechanized in the Rocq [23] proof assistant and available as an artifact to this paper [28]. Our work is distinct from the existing CHERI formalization by Zaliva et al. [64] and its subsequent mechanization [65] where sealing and unsealing is only applied by store and load operations without concern for heap compartmentalization. In our extended logic, new rules for function call and return commands ensure that capabilities are sealed and unsealed when getting in and out of their originating compartment.

We reuse the framework of concurrent adversarial separation logic (CASL [57]), an extensible logic to reason about memory safety and other security properties from which $\textsc{Atom}_{\mathsf{che}}$ is instantiated. We leverage CASL for its ability to express both correct and incorrect computations. For space reasons, we focus the section on proving correct computations examples. We wish to establish the following capability invariant $\mathcal{INV}$:

Any capability shared outside its owner compartment must be sealed

We first define the syntax of the program logic $\textsc{Atom}_{\mathsf{che}}$ with new CHERI specific primitives:

$$\textsc{Atom}_{\mathsf{che}} \ni \mathbf{a} ::= \mathsf{compartenter}\ \tau \mid \mathsf{f}(\vec{x}) \mid (*\mathsf{f})(\vec{x}) \mid \mathsf{return_f}\ \vec{x} \mid$$

$$x := y \mid x\ = \mathsf{alloc}(sz) \mid \mathsf{free}(x) \mid \mathsf{y} = *\mathsf{x} \mid *\mathsf{x} = \mathsf{y} \mid \mathsf{assert}\ P$$

Figure 8 contains the $\textsc{Atom}_{\mathsf{che}}$ proof rules sufficient for the motivational example in figure 3 of section 5. We capture the capability allocation and sealing state of each pointer variable using a triple representation of the form (A, U, τ) where the first component is the allocation state A (allocated) or F (freed), the second component is the sealing state S (sealed) or U (unsealed), and the third component is the capability owner compartment identifier τ.

These CHERI rules ensure that allocation and sealing states are correctly computed based on the allowed transition system, which is enforced at the function boundary in the logic. Distinguishing between *caller* and *callee* based sealing and unsealing is not explicit in the logic, which could be covered at the cost of conciseness of our rules. Generic n-ary rules for `Call` and `Return` actions can easily be defined in a similar fashion, and we keep our current rules for simplicity since they are enough to express our examples.

9 Discussion

Heap compartmentalization generalizability. Our transition-time changes for applying heap compartmentalization are largely allocator-agnostic, since CheriBSD already mediates cross-compartment calls through the runtime loader. In contrast, making arbitrary allocators compartment-aware is non-trivial because it depends on allocator-specific metadata, caching, and reuse policies. Designing a reusable abstraction that preserves TCB-only control remains open; thus, our prototype targets CheriBSD's default allocator.

CheriBSD maturity. Our implementation targets CheriBSD 25.03, the latest stable release available at the time of this work. CheriBSD's compartmentalization support is still evolving; in particular, function-pointer wrapping for trampoline-based transitions is present in 25.03 but not enabled by default. Our allocator assumes the intended model in which all cross-compartment transfers, including indirect calls, are mediated by run-time-linker trampolines. Thus, the robustness of compartment-ID tracking partly depends on the maturity of this transition path. We consider this a limitation of the current software substrate, rather than of the design itself.

Allocation and type-ownership mismatch. Our analysis assumes allocation and type ownership coincide, but rare cases violate this assumption. In curl, a zlib-owned struct is embedded within a curl-owned type; curl derives a pointer to the embedded struct and passes it to zlib. We handle this by having the owner seal the derived capability for subsequent accesses. This does not strengthen isolation because the non-owner still controls the enclosing allocation. Stronger guarantees for such cases are left to future work.

Type confusion attacks. Our approach prevents non-owning compartments from directly accessing capabilities that belong to other compartments, but it does not validate that a sealed capability passed to an owning compartment has the expected type. Consequently, a malicious compartment may supply a validly sealed capability of an unintended type, causing the owner to interpret the referenced object under an incorrect type assumption. Such misuse can induce type confusion and potentially violate the integrity of the target compartment. Enforcing type-safe use of sealed capabilities is left to future work.

Heap Memory Fragmentation. Our heap compartmentalization does not reserve a fixed amount of memory for each compartment. Instead, each compartment's arena obtains memory on demand,

CHERI-Alloc

$$\left[x \mapsto (F, U, \tau) \wedge \tau_{cur} = \tau\right] x = \text{alloc}(sz) \left[x \mapsto (A, U, \tau) \wedge \tau_{cur} = \tau\right]$$

CHERI-Assert

$$\frac{P \Rightarrow Q}{\left[P\right] \text{ assert } Q \left[P\right]}$$

CASL-Frame

$$\frac{\left[P\right] c \left[Q\right] \qquad stable(R, \mathcal{R} \cup \mathcal{G})}{\left[P \star R\right] c \left[Q \star R\right]}$$

CHERI-Assign

$$\frac{\sigma' = U \text{ if } \texttt{holder(x)} = \tau \text{ else } S}{\left[x = - \star y \mapsto (A, \sigma, \tau)\right] \texttt{x := y} \left[x \mapsto (A, \sigma', \tau) \star y \mapsto (A, \sigma, \tau)\right]}$$

CHERI-Compart-Enter

$$\left[\tau_{cur} = \tau_1\right] \text{compartenter } \tau_2 \left[\tau_{cur} = \tau_2\right]$$

CHERI-Call-Unseal

$$\frac{f \in \mathcal{F}_{ext}(\tau_2) \qquad holder(x) = \tau_2}{\left[x \mapsto (A, S, \tau_2) \wedge \tau_{cur} = \tau_1\right] \texttt{f}(x) \left[x \mapsto (A, U, \tau_2) \wedge \tau_{cur} = \tau_2\right]}$$

CHERI-Ret-Seal

$$\frac{f \in \mathcal{F}_{ext}(\tau_1) \qquad caller(f) \in \mathcal{F}(\tau_2)}{\left[x \mapsto (A, U, \tau_1) \wedge \tau_{cur} = \tau_1\right] \text{return}_{\texttt{f}}\ (x) \left[x \mapsto (A, S, \tau_1) \wedge \tau_{cur} = \tau_2\right]}$$

CHERI-Call-Mixed

$$\frac{f \in \mathcal{F}_{ext}(\tau_2) \qquad holder(y) = \tau_2}{\left[x \mapsto (A, U, \tau_1) \star y \mapsto (A, S, \tau_2) \wedge \tau_{cur} = \tau_1\right] \texttt{f}(x, y) \left[x \mapsto (A, S, \tau_1) \star y \mapsto (A, U, \tau_2) \wedge \tau_{cur} = \tau_2\right]}$$

CHERI-Ret-Mixed

$$\frac{f \in \mathcal{F}_{ext}(\tau_2) \qquad caller(f) \in \mathcal{F}(\tau_1) \qquad holder(x) = \tau_1}{\left[x \mapsto (A, S, \tau_1) \star y \mapsto (A, U, \tau_2) \wedge \tau_{cur} = \tau_2\right] \text{return}_{\texttt{f}}\ (x, y) \left[x \mapsto (A, U, \tau_1) \star y \mapsto (A, S, \tau_2) \wedge \tau_{cur} = \tau_1\right]}$$

Logic Command	**Proof Steps**
	ngx_ssl_create_conn(ngx_conn_t *c, ... *ctx) {
Alloc × 2 + CompEnter	$\mathcal{P}_1 = \texttt{c} \mapsto (A, U, \tau_{\texttt{ngx}}) \star \texttt{ctx} \mapsto (A, S, \tau_{\texttt{ssl}}) \wedge \tau_{\texttt{cur}} = \tau_{\texttt{ngx}}$
	conn = SSL_new(ctx);
CallUnseal + Frame	$\mathcal{P}_2 = \texttt{ct} \mapsto (A, U, \tau_{ssl}) \wedge \tau_{cur} = \tau_{ssl}$
	s = OSSL_malloc();
Alloc	$\mathcal{P}_3 = \texttt{s} \mapsto (A, U, \tau_{\texttt{ssl}}) \star \texttt{ct} \mapsto (A, U, \tau_{\texttt{ssl}}) \wedge \tau_{\texttt{cur}} = \tau_{\texttt{ssl}}$
	return s;
RetSeal + Assign	$\mathcal{P}_4 = \texttt{c} \mapsto (A, U, \tau_{\texttt{ngx}}) \star \texttt{conn} \mapsto (A, S, \tau_{\texttt{ssl}}) \star \texttt{ctx} \mapsto (A, S, \tau_{\texttt{ssl}}) \wedge \tau_{\texttt{cur}} = \tau_{\texttt{ngx}}$
	SSL_set_ex_data(conn, c);
CallMix + Assign + Frame	$\mathcal{P}_5 = \texttt{s} \mapsto (A, U, \tau_{\texttt{ssl}}) \star \texttt{data} \mapsto (A, S, \tau_{\texttt{ngx}}) \star \texttt{s.ex_data} = - \wedge \tau_{\texttt{cur}} = \tau_{\texttt{ssl}}$
	s->ex_data := data;
Assign	$\mathcal{P}_6 = \texttt{s} \mapsto (A, U, \tau_{\texttt{ssl}}) \star \texttt{data} \mapsto (A, S, \tau_{\texttt{ngx}}) \star \texttt{s.ex_data} \mapsto (A, S, \tau_{\texttt{ngx}}) \wedge \tau_{\texttt{cur}} = \tau_{\texttt{ssl}}$
	return (...);
RetMix + Assign	$\mathcal{P}_7 = \texttt{conn} \mapsto (A, S, \tau_{\texttt{ssl}}) \star \texttt{s.ex_data} \mapsto (A, S, \tau_{\texttt{ngx}}) \star \texttt{c} \mapsto (A, U, \tau_{\texttt{ngx}}) \star \texttt{ctx} \mapsto (A, S, \tau_{ssl}) \wedge \tau_{\texttt{cur}} = \tau_{\texttt{ngx}}$
	assert $\mathcal{INV}$;
Assert	$\{\mathcal{P}_7\}$

Figure 8: Top: Minimal CHERI logic rules for safe capability sealing and unsealing with heap compartmentalization. Each capability is represented as a triple of its allocation state (where A stands for Allocated and F for Free), their sealing state (where U stands for Unsealed and S for Sealed), and their owning compartment identifier. Allocation reuse across compartments is prevented while (un)sealing allows safe sharing between compartments. Bottom: Proof of the `nginx` + `openssl` example from Figure 3. The capability invariant $\mathcal{INV}$ is defined as: $\forall p.p \mapsto (-, x, \tau_p) \wedge holder(p) \neq \tau_p \Rightarrow x = S$ and reads as: *Any capability p shared from its owner compartment τ_p to another compartment holder(p) must be sealed* when holder(p) is a different compartment than the owner of p. We note τ_{cur} as the current compartment identifier. For each compartment, we distinguish internal functions $\mathcal{F}_{int}$ and external functions $\mathcal{F}_{ext}$ with $\mathcal{F}(\tau) = \mathcal{F}_{int}(\tau) \cup \mathcal{F}_{ext}(\tau)$ and $\mathcal{F}_{int}(\tau) \cap \mathcal{F}_{ext}(\tau) = \emptyset$. The frame rule inherited from CASL [57] requires stability conditions and is solely given so the reader can follow the paper's examples more easily.

so the design imposes no static per-compartment heap limit. However, memory freed by one compartment can be reused only by its allocation-owning compartment. This restriction may increase fragmentation and cause process-wide memory exhaustion earlier than in the baseline, particularly under adversarial allocation patterns. Our design therefore does not provide availability isolation against memory-exhaustion attacks.

Accessor and Sealing Decisions. Sealing depends on how non-owners use a type. Opaque types are sealed directly; for partial

field access, they remain sealed and accesses are replaced with owner-side accessors. A type remains unsealed only when non-owning code accesses its complete representation, since generating accessors for every field would effectively re-expose the entire object while providing little additional security. This occurred for only 3 of the 161 shared types; stronger protection would require redesigning their interfaces to expose a narrower representation.

10 Related Works

10.1 Uninitialized Memory

Uninitialized memory reads are a well-known source of information disclosure in low-level systems, and prior work spans both *detection* and *mitigation*. Dynamic tools such as Valgrind Memcheck and Dr. Memory can track undefined bytes and report uninitialized reads at runtime [16, 58]. Compiler-based sanitization reduces overhead relative to full dynamic instrumentation and has been proposed in the past [42, 59]. Other techniques expose uninitialized reads by perturbing execution or memory state, including probabilistic/redundant allocation (DieHard) and Differential Replay [13, 19].

Another line of work aims to *reduce residual-data exposure* by initializing or scrubbing memory: secure deallocation shortens the lifetime of sensitive data by clearing on (or after) free [22]; UniSan proactively initializes kernel allocations to prevent leakage across privilege boundaries [47]; and SafeInit enforces practical initialization discipline via compiler support [49]. Most closely related to CHERI, MonCHÉRI proposes *conditional capabilities* to enforce write-before-read semantics without pervasive zeroing, but it relies on constraints such as sequential initialization to preserve correctness [31]. In contrast, our work focuses on *cross-compartment* disclosure of residual capabilities through heap reuse.

10.2 Heap Partitioning

Heap partitioning has long been used to improve locality, scalability, ownership tracking, and security [12–14, 38, 39, 52]. Per-compartment allocation pools are also an established compartmentalization practice: Native Client [63] and WebAssembly-based sandboxes can maintain separate allocator state within each sandbox or linear-memory instance. We therefore do not claim novelty for partitioning allocator reuse or maintaining trusted current-compartment state. Our contribution is a CheriBSD-specific integration: library compartments share a process-wide `libc`/jemalloc instance, so we bind arena selection to trusted RTLD transitions and return freed chunks to their allocation-owning arena without changing application or library allocator interfaces.

10.3 Data Sharing and Interface Hardening

The software-compartmentalization SoK [41] separates compartmentalization into policy definition (P1), software abstraction (P2), and runtime enforcement (P3). We assume CheriBSD's library partition, infer a type-ownership policy within it (P1), represent shared objects using sealed handles and accessors (P2), and enforce the policy with CHERI sealing and lightweight runtime checks (P3). Our contribution is therefore not a new pointer-analysis algorithm, but the combination of lightweight static analysis with CHERI authority enforcement.

Rather than compute a complete points-to or object graph, we recover type ownership, interface exposure, and field use. CHERI metadata resolves residual uncertainty at runtime: function capabilities distinguish cross-compartment indirect calls, while guarded seal/unseal operations tolerate capabilities arriving in either state. This lets us remain conservative without treating every uncertain interaction as cross-compartment [41].

Automatic partitioning and message passing. Privilege-separation systems such as PrivTrans [17], Wedge [15], Glamdring [43], Google SAPI [30], and CompARTist [35] isolate components using processes, enclaves, or generated RPC interfaces. PtrSplit [44] automates much of the resulting marshalling, while Program Mandering [45] additionally optimizes compartment boundaries. These approaches provide strong isolation, but pointer-rich APIs must be serialized, copied, or translated into new handle protocols. Our approach instead preserves the existing address space, object layout, pointer identity, and call interface: sealed capabilities can still be stored and passed back, but not dereferenced by non-owners.

Cali [11] similarly avoids deep-copy marshalling by placing allocations shared with an isolated library in shared memory. Its protection unit is an allocation or region; ours is the authority-bearing reference, allowing an object to remain reachable while restricting dereference authority to its owner.

Typed interface hardening. RLBox [51] exposes sandbox values through C++ tainted types and requires validation at relevant call sites, while LXFI [48] constrains kernel-extension interfaces through software fault isolation, type restrictions, and API-integrity rules. These systems provide broader protection against hostile outputs, callbacks, and control-flow misuse, but require sandbox-aware APIs, validation logic, specifications, or wrappers. Our mechanism addresses the narrower problem of unnecessary dereference authority and automatically inserts enforcement without changing existing application or library interfaces.

Kernel–driver shared state. KSplit [37] uses data flow and alias analysis to identify state shared between Linux drivers and the kernel, then generates specifications and synchronization for isolated copies. Since both sides may modify shared state, it must reason about aliases, marshalling direction, and synchronization.

Our target pattern is simpler: a library such as OpenSSL defines an object representation, while other compartments typically only store and forward its pointer. We therefore assign ownership by type and expose non-owners only to opaque sealed handles. Small required field subsets are mediated by owner-side accessors; types extensively used by multiple compartments remain unsealed. This occurred for only three of 161 shared types in our evaluation.

Burtsev et al. [18] likewise advocate isolated heaps, single ownership, and explicit interfaces for kernel–driver state, but assign ownership per object and allow it to change dynamically. Our stable type ownership is less expressive, but enables automatic enforcement at existing C library interfaces.

Selective protection of sensitive data. DataShield [20], SeCage [46], xMP [56], type-based data isolation [50], and compiler-based in-memory encryption [53] protect explicitly designated sensitive data using isolation, bounds checks, virtualization, or encryption. DynPTA [54], for example, combines conservative pointer analysis

with dynamic labels to avoid encrypting nonsensitive objects. These systems track sensitivity across potentially aliasing accesses; our runtime checks instead resolve only compartment-boundary and capability-state uncertainty.

Compartment-interface vulnerability analysis. DUI Detector [34], ConfFuzz [40], and CivScope [21] show that compartment interfaces can remain vulnerable to attacker-controlled dereferences, confused deputies, and hidden-state exposure. These systems primarily detect or characterize unsafe interfaces. Our mechanism instead enforces one specific property: non-owners cannot dereference unnecessarily delegated capabilities to owner-defined heap objects. It does not address scalar validation, API ordering, TOCTOU, or general interface vulnerabilities.

CHERI compartment systems. CHERIoT [7, 8] uses sealed capabilities as type-safe opaque handles throughout a clean-slate embedded RTOS and provides hardware/software mechanisms, such as local or no-capture capabilities, unavailable in Morello/CheriBSD. Our contribution is not the use of sealing itself, but automatically identifying C references that already follow opaque-handle semantics and retrofitting sealing into existing CheriBSD applications without changing their public interfaces. To our knowledge, prior work has not combined automatic type-ownership inference with CHERI sealing for existing CheriBSD library compartments.

11 Conclusion

In this paper, we showed how capability leakage is still a problem in compartmentalized CHERI systems. To mitigate this attack vector, we proposed type-directed capability sealing (whose logic is certified in Rocq) , and complemented it with a CheriBSD-specific adaptation of established per-compartment allocation pools to prevent capability leakage through cross-compartment heap reuse.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback, the CHERI teams at the University of Cambridge and Capabilities Limited for their support, and Alfredo Mazzinghi, Brooks Davis, Dapeng Gao, Jessica Clarke, and Robert N. M. Watson for helpful discussions and feedback.

References

[1] Nginx. https://www.nginx.com/.
[2] [rfc] a framework for allocator partitioning hints. https://discourse.llvm.org/t/rfc-a-framework-for-allocator-partitioning-hints/87434.
[3] CVE-2017-7529: Integer overflow in nginx range filter. https://nvd.nist.gov/vuln/detail/CVE-2017-7529, 2017. CVSS 7.5. Affects nginx 0.5.6–1.13.2.
[4] OpenSSL. https://www.openssl.org/, 2020.
[5] CVE-2021-23017: Off-by-one in nginx dns resolver. https://nvd.nist.gov/vuln/detail/CVE-2021-23017, 2021. CVSS 9.8. Affects nginx 0.6.18–1.20.0.
[6] CVE-2022-41741: nginx mp4 module out-of-bounds write. https://nvd.nist.gov/vuln/detail/CVE-2022-41741, 2022. CVSS 7.8. Affects nginx 1.1.3–1.23.1.
[7] Saar Amar, Tony Chen, David Chisnall, Nathaniel Wesley Filardo, Ben Laurie, Hugo Lefeuvre, Kunyan Liu, Simon W Moore, Robert Norton-Wright, Margo Seltzer, et al. Cheriot rtos: An os for fine-grained memory-safe compartments on low-cost embedded devices. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 67–84, 2025.
[8] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W Moore, Yucong Tao, Robert NM Watson, et al. Cheriot: Complete memory safety for embedded devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 641–653, 2023.
[9] Arm Ltd. Morello supplement to the arm c language extensions, 2022. Accessed 2026-04-27.
[10] Arm Ltd. Morello prototype architecture overview user guide, 2023. Accessed 2026-04-27.
[11] Markus Bauer and Christian Rossow. Cali: Compiler-assisted library isolation. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 550–564, 2021.
[12] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.
[13] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.
[14] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 114–124, 2001.
[15] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into Reduced-Privilege compartments. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*, San Francisco, CA, April 2008. USENIX Association.
[16] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 213–223, April 2011. Online: https://ieeexplore.ieee.org/document/5764689.
[17] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, 2004.
[18] Anton Burtsev, Vikram Narayanan, Yongzhe Huang, Kaiming Huang, Gang Tan, and Trent Jaeger. Evolving operating system kernels towards secure kernel-driver interfaces. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 166–173, 2023.
[19] Mengchen Cao, Xiantong Hou, Tao Wang, Hunter Qu, Yajin Zhou, Xiaolong Bai, and Fuwei Wang. Different is good: Detecting the use of uninitialized variables through differential replay. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*, pages 1883–1897, November 2019.
[20] Scott A. Carr and Mathias Payer. DataShield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 193–204, 2017.
[21] Yi Chien, Vlad-Andrei Bădoiu, Yudi Yang, Yuqian Huo, Kelly Kaoudis, Hugo Lefeuvre, Pierre Olivier, and Nathan Dautenhahn. Civscope: Analyzing potential memory corruption bugs in compartment interfaces. In *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification*, pages 33–40, 2023.
[22] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the 14th USENIX Security Symposium (USENIX Security '05)*, Baltimore, MD, USA, 2005. USENIX Association.
[23] Thierry Coquand, Gérard Huet, Christine Paulin, et al. The coq proof assistant, 1984.
[24] Brooks Davis, Robert NM Watson, Alexander Richardson, Peter G Neumann, Simon W Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, et al. Cheriabi: Enforcing valid pointer provenance and minimizing pointer privilege in the posix c run-time environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 379–393, 2019.
[25] Nathaniel Wesley Filardo, Brett F Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, et al. Cornucopia: Temporal safety for cheri heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 608–625. IEEE, 2020.
[26] Nathaniel Wesley Filardo, Brett F Gutstein, Jonathan Woodruff, Jessica Clarke, Peter Rugg, Brooks Davis, Mark Johnston, Robert Norton, David Chisnall, Simon W Moore, et al. Cornucopia reloaded: Load barriers for cheri heap temporal safety. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 251–268, 2024.
[27] Dapeng Gao and Robert NM Watson. Library-based compartmentalisation on cheri. In *Programming Languages for Architecture 2023*, 2023.
[28] Seyedhamed Ghavamnia, Maurice D. Hanisch, and Julien Vanegue. CAPWARD: Capability warding by automatically restricting dereference. https://github.com/bloomberg-science/capward, 2026. Source code and evaluation artifact.
[29] Seyedhamed Ghavamnia, Maurice D. Hanisch, and Julien Vanegue. CAPWARD: Capability warding by automatically restricting dereference, 2026.
[30] Google. Sandboxed api (SAPI). Google Code Sandboxing documentation. Accessed 2026-08-15.
[31] Merve Gülmez, Håkan Englund, Jan Tobias Mühlberg, and Thomas Nyman. Mon chéri: Mitigating uninitialized memory access with conditional capabilities. In

2025 IEEE Symposium on Security and Privacy (SP), pages 829–847. IEEE, 2025.
[32] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and Jean-François Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, pages 185–200, 2017.
[33] Maurice D. Hanisch. jemalloc: resolve sub-capabilities in je_malloc_underlying_allocation. https://github.com/CTSRD-CHERI/cheribsd/pull/2448.
[34] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. Identifying arbitrary memory access vulnerabilities in privilege-separated software. In *European Symposium on Research in Computer Security*, pages 312–331. Springer, 2015.
[35] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. The art of app compartmentalization: Compiler-based library privilege separation on stock android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1037–1049, 2017.
[36] Yongzhe Huang, Kaiming Huang, Matthew Ennis, Vikram Narayanan, Anton Burtsev, Trent Jaeger, and Gang Tan. Sok: Understanding the attack surface in device driver isolation frameworks. *arXiv preprint arXiv:2412.16754*, 2024.
[37] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. {KSplit}: Automating device driver isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 613–631, 2022.
[38] Nouraldin Jaber and Milind Kulkarni. Data structure–aware heap partitioning. In *Proceedings of the International Conference on Compiler Construction (CC)*, pages 109–119, 2017.
[39] Chris Lattner and Vikram Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.
[40] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Yi Chien, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. Assessing the impact of interface vulnerabilities in compartmentalized software. February 2023.
[41] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. Sok: Software compartmentalization. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3107–3126. IEEE, 2025.
[42] Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing. {CAMP}: Compiler and allocator-based heap memory protection. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4015–4032, 2024.
[43] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for intel {SGX}. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298, 2017.
[44] Shen Liu, Gang Tan, and Trent Jaeger. PtrSplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2359–2371, 2017.
[45] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. Program-mandering: Quantitative privilege separation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1023–1040, 2019.
[46] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pages 1607–1619, 2015.
[47] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, pages 920–932. Association for Computing Machinery, October 2016.
[48] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 115–128, 2011.
[49] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. Safeinit: Comprehensive and practical mitigation of uninitialized read vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2017. Internet Society.
[50] Alyssa Milburn, Erik Van Der Kouwe, and Cristiano Giuffrida. Mitigating information leakage vulnerabilities with type-based data isolation. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1049–1065. IEEE, 2022.
[51] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the Firefox renderer. In *29th USENIX Security Symposium (USENIX) Security 20)*, pages 699–716, 2020.
[52] Gene Novark and Emery D. Berger. DieHarder: Securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 573–584, 2010.
[53] Tapti Palit, Fabian Monrose, and Michalis Polychronakis. Mitigating data leakage by protecting memory-resident sensitive data. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 598–611. ACM, 2019.
[54] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. Dynpta: Combining static and dynamic analysis for practical selective data protection. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1919–1937. IEEE, 2021.
[55] Simon Pope. Security analysis of cheri isa, October 2020. Analyzes the theoretical impact of CHERI on memory-safety vulnerabilities reported to MSRC in 2019.
[56] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. xMP: Selective memory protection for kernel and user space. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 584–598, 2020.
[57] Azalea Raad, Julien Vanegue, Josh Berdine, and Peter O'Hearn. A general approach to under-approximate reasoning about concurrent programs. In *34th International Conference on Concurrency Theory (CONCUR 2023)*, pages 25–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2023.
[58] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association. Online: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/using-valgrind-detect-undefined-value-errors-bit.
[59] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: Fast detector of uninitialized memory use in c++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55, 2015.
[60] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Michael Roe, et al. Capability hardware enhanced RISC instructions: CHERI instruction-set architecture. Technical Report UCAM-CL-TR-987, University of Cambridge, Computer Laboratory, 2024.
[61] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.
[62] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W Filardo, A Theodore Markettos, et al. Cheri concentrate: Practical compressed capabilities. *IEEE Transactions on Computers*, 68(10):1455–1469, 2019.
[63] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Orm, Shiki Okasaka, Neha Narula, Nicholas Fullagar, and Google Inc. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2009.
[64] Vadim Zaliva, Kayvan Memarian, Ricardo Almeida, Jessica Clarke, Brooks Davis, Alexander Richardson, David Chisnall, Brian Campbell, Ian Stark, Robert NM Watson, et al. Formal mechanised semantics of cheri c: capabilities, undefined behaviour, and provenance. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 181–196, 2024.
[65] Vadim Zaliva, Kayvan Memarian, Brian Campbell, Ricardo Almeida, Nathaniel Filardo, Ian Stark, and Peter Sewell. A cheri c memory model for verified temporal safety. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 112–126, 2025.

A Open Science

We have reported and provided fixes for the allocator issues we identified that have been merged upstream [33]. Our tool, the data generated in the paper, and a more complete version of our paper are available on GitHub [28]. For long-term availability, the version of the artifact corresponding to the results reported in this paper has also been archived on Zenodo [29].

B Ethical Considerations

This work studies capability leakage and the affected stakeholders include CHERI users, maintainers, application developers, and the broader research community. We identified allocator defects that could weaken CHERI's security guarantees and responsibly disclosed them to the CHERI/CheriBSD developers. We also provided fixes, which have since been merged upstream. Our experiments used controlled test systems and involved no human subjects, private user data, or production services.