A CHERI C Memory Model for Verified Temporal Safety

Vadim Zaliva
University of Cambridge
Cambridge, United Kingdom
Vadim.Zaliva@cl.cam.ac.uk

Kayvan Memarian
University of Cambridge
Cambridge, United Kingdom
Kayvan.Memarian@cl.cam.ac.uk

Brian Campbell
University of Edinburgh
Edinburgh, United Kingdom
Brian.Campbell@ed.ac.uk

Ricardo Almeida
University of Edinburgh
Edinburgh, United Kingdom
Ricardo.Almeida@ed.ac.uk

Nathaniel Filardo
University of Cambridge
Cambridge, United Kingdom
nwf20@cam.ac.uk

Ian Stark
University of Edinburgh
Edinburgh, United Kingdom
Ian.Stark@ed.ac.uk

Peter Sewell
University of Cambridge
Cambridge, United Kingdom
Peter.Sewell@cl.cam.ac.uk

Abstract

Memory safety concerns continue to be a major source of security vulnerabilities. The CHERI architecture, as instantiated in prototype CHERI-RISC-V cores, the Arm Morello system, and Microsoft's CHERIoT embedded core, provides fine-grained memory access control through unforgeable hardware capabilities. The impact of CHERI on spatial memory safety is well understood. This paper systematically examines temporal memory safety within CHERI C – a dialect of the C programming language for CHERI – and proposes a formal approach to defining and ensuring it. In particular: 1) we examine the impact of five existing capability revocation mechanisms on CHERI C semantics and present a specialised object memory model tailored to CHERI C; 2) we introduce a new CHERI-specific pointer provenance tracking scheme; and 3) we formally define the security guarantees provided by this memory model, supported by a Coq proof of their correctness, expressed as invariants of the memory state.

CCS Concepts: • **Software and its engineering → Compilers**; **Formal language definitions**; *Software functional properties*; • **Theory of computation → Semantics and reasoning**; • **Security and privacy → Logic and verification**.

Keywords: CHERI, memory safety, temporal safety, formal verification, memory model, pointer provenance, C language, Coq, capability revocation

CPP '25, January 20–21, 2025, Denver, CO, USA

ACM ISBN 979-8-4007-1347-7/25/01
https://doi.org/10.1145/3703595.3705878

ACM Reference Format:
Vadim Zaliva, Kayvan Memarian, Brian Campbell, Ricardo Almeida, Nathaniel Filardo, Ian Stark, and Peter Sewell. 2025. A CHERI C Memory Model for Verified Temporal Safety. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '25), January 20–21, 2025, Denver, CO, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3703595.3705878

1 Introduction

Memory handling errors in C and C++ code continue to expose systems to severe security vulnerabilities, and form the core of many exploit chains used in the wild. For example, they are involved in around 70% of vulnerabilities tracked by Project Zero [17], and hold prominent positions in the Common Weakness Enumeration project's list of the most dangerous and most stubborn software weaknesses [33]. A considerable amount of effort has been made to mitigate these vulnerabilities, from static and dynamic code analysis, through adjusting the execution environment and compilation to reduce exposure, to rewriting code in memory-safe languages [32]. However, many of these have limitations, and the sheer volume of legacy code makes rewriting most code in a safe language a distant prospect.

The CHERI architecture [37, 39] provides a more robust form of fine-grained memory protection for legacy code by extending existing instruction set architectures with hardware capabilities. These combine an address with metadata giving the bounds of the object pointed to, permissions for accessing it, and an out-of-band validity tag that makes them unforgeable. This promises a substantial improvement in safety against memory errors, with the Microsoft Security Response Center estimating that between one third and two thirds of vulnerabilities reported to them in 2019 would be "deterministically mitigated" by CHERI, depending on the

level of protection used [20]. There is substantial and growing experience with CHERI, including the Arm Morello experimental application-class implementation [15], multiple RISC-V designs including the CHERIoT microcontroller-class processor [2], LLVM and GCC ports that can use capabilities for pointers, and a large body of ported code including a FreeBSD port with a desktop environment [35].

The current proposed semantics for CHERI C [42] deals with the consequences of using capabilities for pointers, which allows C programs to benefit from the hardware's built-in protection. This protection provides *spatial safety*, where accesses are limited to the addresses of object the pointer was derived from; for example, a buffer access cannot overrun into a neighbouring object. Conspicuously absent, however, is discussion of *temporal safety*. In this paper, we enrich the existing proposal with a model of *capability revocation*, define several increasingly stringent security invariants precluding larger classes of temporal safety violations, and show that our memory model preserves the strongest of these invariants.

In the context of the C programming language (and the CHERI C dialect), *temporal safety* ensures that memory objects are accessed only within their intended lifetime. In particular, two undesirable scenarios are considered: 1) *use-after-free* (UAF), where a memory object is accessed after it has been deallocated but before the underlying address space has been repurposed; and 2) *use-after-reallocation* (UAR), when a pointer to a previously allocated memory block is used after that memory has been freed and subsequently reallocated to a new object. Here, reallocation includes any reuse *by the allocator* for its own purposes. For some allocator implementations, there may be no or limited opportunity for UAF, only UAR; traditional designs relied heavily on in-band ("intrusive") data structures to track released space, reusing the first few words of objects passed to `free` within the allocator to hold, for example, pointers to other available memory. UAR in particular can lead to severe security vulnerabilities, as the *aliasing* of objects invites data corruption or data exposure. UAF does not involve aliasing, but it, too, is not without its risks: attempted UAF accesses can cause faults, as the allocator is free to release the address space and underlying storage of free objects (such as by unmapping virtual pages); and the possibility of a UAF likely implies the possibility of a UAR later, with address space having been repurposed by the allocator and/or operating system.

As a concrete example, consider Listing 1. This program performs an allocation, frees the resulting object, and then makes another allocation of the same size as the first. Historically, heap allocators likely placed the two objects in the same underlying memory: the freed memory would be at the front of the free list and an exact match for the size requested. That is, `pa` and `qa` would be equal after line 9, and, in most C implementations, `p` would alias the object reachable through `q`. This sets the stage for UAR of a heap ("dynamic storage duration") object. Even if `qa` is not equal to `pa` and the memory underlying `p` is not reallocated, UAF of `p` after line 7 could still fault, if the allocator releases said memory. It is also possible to have UAF or UAR of stack ("automatic storage duration") objects.

```
1 #include <stdint.h>
2 #include <stdlib.h>
3
4 int main() {
5   int *p = malloc(4096);
6   ptraddr_t pa = (ptraddr_t)p;
7   free(p);
8   int *q = malloc(4096);
9   ptraddr_t qa = (ptraddr_t)q;
10  /* ... */
11 }
```

Listing 1. A C example of potential temporal unsafety.

More generally, all instances of UAF or UAR are attempts to refer to an object outside of its lifetime, resulting in *undefined behaviour* (UB) according to ISO C [19, §6.2.4]. Thus, even though both temporal safety violation scenarios could not occur in *well-defined* (that is, without UBs) CHERI C programs, in the presence of UB, compilers may generate code attempting illegal accesses. These could be mitigated by CHERI hardware in the presence of revocation.

In this paper, we will explore CHERI C semantics for CHERI capability revocation mechanisms, which we describe in more detail in Section 2.

Another aspect of CHERI C semantics that we discuss in Section 3 is the subtle relationship between compile-time *pointer provenance* tracking (the relationship between pointers and corresponding memory allocations used in compiler alias analysis) and runtime CHERI memory protections (hardware-enforced pointer bounds), augmented by a revocation mechanism (as defined in Section 2).

We make the following contributions:

(1) We introduce a new CHERI-specific pointer provenance tracking scheme: *PNVI-CHERI*, which is conceptually simpler than schemes previously proposed for CHERI C (Section 3).
(2) We present a specialised object memory model tailored for CHERI C, which is based on *PNVI-CHERI* and includes support for a generalised version of five existing CHERI capability revocation mechanisms; it is defined in Coq (Section 4).
(3) We integrate our new CHERI memory model (extracted from the Coq definitions) into existing executable CHERI C semantics and test it against an existing CHERI C semantics test suite (Section 4).
(4) We formally define and prove in Coq the security guarantees provided by this memory model, which precludes both UAF and UAR scenarios. It is applicable to

CHERIoT, non-parallel CHERIvoke, both Cornucopia and Reloaded—right after a full revocation sweep,[1] and the fixed-point of the revocation sweep algorithm for CheriOS (Section 5).

(5) Additionally, we present a framework for monadic reasoning, developed during the course of this work, which may be useful for reasoning about a wide class of programs that utilise state and error monads (Section 6).

Our implementation of the memory model in Coq, along with proofs, is available in a public Git repository [2].

Several limitations apply to this work, briefly summarised below. All of them are subjects for future research and are discussed further in Section 9.

Concurrency. Our memory model does not currently support concurrency because it is based on a version of the Cerberus C semantics without support for the C/C++11 concurrency.

Stack vs Heap. While the semantics of the *abstract C machine* and, consequently, our memory model, do not differentiate between stack and heap memory, the existing revocation mechanisms do.

Pointer leakage via memory reuse. The C standard does not require allocated heap memory to be initialised, leaving it with *indeterminate* values. These often reflect the old program state from reused memory, but random values would still be valid. A major risk for CHERI C is that such values could be valid capabilities to live objects, leading to unintended and covert capability flow. Requiring capability tags to be reset in new allocations may be sufficient, but it could incur runtime costs.

2 Revocation

There are several existing attempts to address heap temporal safety for CHERI C/C++: CHERIvoke [40], Cornucopia [12], and Cornucopia Reloaded [11] provide replacement allocators (and novel kernel machinery) for CheriABI processes in CheriBSD [9], and both the CheriOS [10] and CHERIoT [2, 3] operating systems have provided spatially- and temporally-safe shared heap allocators. These efforts have relied on various *sweeping revocation* implementations to locate and invalidate capabilities pointing to freed allocations, in keeping with CHERI's design desideratum of no additional indirection (through, for example, tables of live objects).

These systems conceptually operate by refining the heap memory life-cycle, which historically consisted of just a single distinction between *live* and *free* (or *dead*). That is, a location in heap memory was either considered to be actively used as part of a live object or available for (re)use. These systems introduce a new state, *quarantined*, which describes memory from released objects to which references (pointers, capabilities) may yet exist, while *free* now describes heap memory to which no references exist (aside from those held by the heap implementation itself).

The safety guaranteed by these systems is the prevention of UAR by finding and clearing the tag of all capabilities with bounds within quarantined regions during a revocation sweep. (This is permitted by ISO C, as even the value of a pointer to an object past the end of its lifetime is indeterminate; referring back to Listing 1, `p` and copies thereof are indeterminate values after line 7.) After the sweep, it is safe to make available for reuse the address ranges of memory quarantined before the sweep began. In general, preventing *UAF* is seen as less urgent; UAF either accesses quarantined objects (unaliased and unmodified by the runtime) or faults, if backing memory is released or if revocation invalidates the pointer. Concretely, in terms of Listing 1, the promise is that, by line 10, either 1) `pa` and `qa` are guaranteed to be distinct, because the first allocation was in quarantine when the second took place; or 2) `p` is no longer valid, and so the equality of `pa` and `qa` does not imply aliasing of `p` and `q`.

An implementation of heap temporal safety via revocation consists of two primary components: per-allocator quarantine tracking logic and a system-provided sweeping service. Allocators, which are the runtime basis of the language's object model, are responsible for ensuring that address space undergoes revocation. The system (usually, the kernel) provides the revocation service to ensure that pointers in "hard-to-reach" places (especially, machine registers, save areas, and other kernel state held on behalf of a process), in addition to (process) memory, are properly swept. The interface between the two involves allowing the allocator to request that a sweep begin, a way to monitor the sweep's progress, and a concrete representation of quarantine. CheriOS's sweep revoked a single contiguous region of address space, while the CHERIvoke, Cornucopia (Reloaded), and CHERIoT lineage have used a bitmap at the same granularity as CHERI capability tags. Concurrent (and parallel) revocation implementations bring significant benefits, but slightly complicate the interface by requiring that the allocator track when, relative to a given sweep, an object entered quarantine.

The implementation of such a sweeping service often takes advantage of privileged architectural mechanisms to accelerate its operation, and it is largely these mechanisms that differentiate the implementations:

CheriOS relied on a custom architectural extension to clear the tag of loaded capabilities pointing into the region being reclaimed. Its revocation service was simply a loop around an atomic load and store to commit to memory this filtered view of memory.

CHERIvoke was more of a microbenchmark limit study than a practical implementation. It operated by stopping

[1] assuming no calls to *free()* were made during the sweep.
[2] https://github.com/rems-project/cerberus

all application threads for the duration of the revocation sweep. To avoid useless work, it proposed an architectural "card marking" scheme, tracking capability stores in the MMU, which would have allowed skipping pages without capabilities.

Cornucopia is a "mostly concurrent," two-phase implementation built on the above card-marking scheme. First, Cornucopia attempts to revoke *concurrently* with the application, visiting only pages possibly holding capabilities, and tracking which pages are newly- or *re*-marked during this work. Second, the application is paused and only this (hopefully smaller) set of pages are (re)visited. In practice, this showed significant reduction in tail latencies relative to a one-phase approximation of CHERIvoke.

Cornucopia Reloaded continues using Cornucopia's card-marking scheme, but only to avoid pages devoid of capabilities. Using a novel architectural mechanism of *capability load faults* to front-run the application, it provides a view of memory as if revocation had completed as soon as it began. A background scan ensures that all pages are revoked. Tail latencies showed significant improvement over Cornucopia.

CHERIoT takes advantage of its small scale to have its capability load *instruction* perform similar front-running of the application. Concretely, the load instruction snoops on quarantine state, now *architectural* and held in memory tightly coupled to the (singular) CPU pipeline. Its revocation service is, similar to that of CheriOS, simply a commitment of this filtered view to memory; software implementation is possible, but there is hardware acceleration in the CHERIoT-Ibex implementation [3, §3.3.3].

Beyond this basic framework, additional architectural extensions can enable *prompt invalidation* of pointers into quarantine. These facilities serve to close the distinction between UAF and UAR, preventing access even to quarantined address space not yet recycled, providing software the illusion that revocation is instantaneous (while still allowing the actual revocation and eventual address reuse to occur asynchronously, concurrently, and/or in parallel with the program). CHERIoT's front-running of the application in fact ensures that no valid capability can be obtained with bounds in the quarantined region, regardless of whether revocation is actively ongoing in the background (whereas Reloaded's front-running happens only during a revocation sweep and is best considered a performance optimization and not a security mechanism). The sketched hybridization of CHERI and memory coloring [11, §7.3] proposes colors as a replacement for quarantine bitmaps and renders mis-colored capabilities inert, even if tagged. In the other systems above, the validity of capabilities with bounds in quarantine will depend on the algorithm's details and the timing of the revocation sweep.

Below, we extend the CHERI C memory model with a revocation mechanism, a conceptual simplification used to approximate the effects of these revocation implementations.

Caveat: Addresses are Visible at Runtime in C. We note in passing that revocation is concerned solely with *pointers* and, importantly, does not consider *addresses* as needing special handling; after a pointer-to-integer cast (except to `(u)intptr_t`), the integer is handled as any other integer would be, including being ignored by the revoker. This means that software may observe that a particular address has been used to hold a series of objects (with disjoint lifetimes). While this cannot give rise to violations of memory safety, it can still be a source of bugs. Consider a hash-table associating metadata to objects, keyed on addresses; freeing an object without removing the associated metadata is possible, and stale metadata may be erroneously re-associated with a different object. We consider these to be *logic* bugs separable from *safety* concerns. One possible way of addressing this is by attaching provenance information to integers created from pointers, similar to the *PVI* ("provenance via integers") model [28], but this model was shown to add significant complexity to the semantics of integer operations, affecting their algebraic properties, and disallowing common optimisations.

3 Provenance, Capabilities, and Revocation

The concept of *pointer provenance* is not explicitly mentioned in the ISO C standard [19], but it was confirmed by WG14 committee discussions concerning DR260 [38]. In particular, it was concluded:

> *In addition the C Standard does not prohibit an implementation from tracking the provenance of the bit-pattern representing a value. An indeterminate value happening to have a bit pattern that is identical to a bit pattern representing a determinate value is not sufficient to allow access to the indeterminate value free from undefined behavior.*

This led to the development, in consultation with the ISO C standards committee, of the working draft Technical Specification [16], and there is ongoing effort to publish it as an ISO Technical Specification.

The concept of pointer provenance allows the C abstract machine to track the origin of pointer values, and this information can be used by the semantics to justify the soundness of existing compiler optimisations that are based on *alias analysis*. It must be noted that provenance is not normally explicit at runtime, but rather meta-information associated with pointers in the language semantics, and used to define when accesses give rise to undefined behaviour.

Consider the example shown in Listing 2. In the absence of pointer provenance tracking, most compiler optimisations based on alias analysis cannot be justified, as demonstrated

```c
#include <stdio.h>
#include <string.h>
#include <stdint.h>

int y=2, x=1;
int main() {
  int *p = &x + 1;
  int *q = &y ;
  printf("Addresses: p=%p q=%p\n", (void*)p, (void*)q);
  if((ptraddr_t)&p == (ptraddr_t)&q) {
    *p = 11; // does this have undefined behaviour ?
    printf ("x=%d y=%d *p=%d *q=%d\n", x, y, *p, *q);
  }
}
```

Listing 2. A program raising a question of pointer provenance, slightly modified from an example in [16, §1.2].

by the disagreement in how different compilers compile this program. Depending on their decision on whether `p` and `q` could alias, the optimisation could lead to the resulting programs printing different values, as was shown for *GCC*, *Clang*, and *ICC* in [16]. Several flavours of provenance tracking are discussed in that work, but we focus on the *PNVI* family – "Provenance Not Via Integers" – where provenance is associated only with pointer types. Applying the *PNVI-ae-udi* provenance model in particular makes it clear that the pointer `p` is associated with the allocation of `x`, and since it has been altered via pointer arithmetic to point one-past the original allocation, an attempt to write using it is indeed *undefined behaviour*.

The goal of revocation, as described in Section 2, is to remove from the system all pointers into spans of address space that once held allocations. Central to this concept of revocation is the requirement that any given pointer can be associated with its originating allocation. This association must be possible even if the pointer has been offset within the object, making it a so-called *interior* pointer, or has been taken one-past-the-end of the allocation, *out of bounds*. This gives rise to a *capability derivation* tree relationship between pointers: the allocator internally holds a pointer to its heap, from which it derives pointers to individual allocations, and from which the application derives, for example, interior pointers. The pointer derivation relationship is *monotonic*, which means derived pointers' bounds and permissions cannot exceed those of the pointers they are derived from. CHERI, unlike some other capability systems, does not explicitly track this relationship; nevertheless, its capability tag bits can be thought of as attesting to the existence of a derivation chain from a primordial, root capability.

The CHERI C semantics proposed in [42] recommends *PNVI-ae-udi* for pointer provenance tracking. We claim that a significant part of *PNVI-ae-udi* is not applicable to CHERI C while some other parts of it could expressed using meta information provided by CHERI. To this end we propose a conceptually simpler provenance tracking model for CHERI C, which we term *PNVI-CHERI*.

PNVI-CHERI is based on the *PNVI-plain* model, rather than on *PNVI-ae-udi*. The "udi" part of *PNVI-ae-udi* stands for *user-disambiguation* for provenance recovery for pointers that are one-past a storage instance. This is no longer applicable, as the *PNVI-CHERI* provenance lookup mechanism proposed below has no ambiguity even for one-past pointers. The "ae" part of *PNVI-ae-udi* stands for *exposed-address*, which is also not relevant in CHERI C, as the *exposure* mechanism features only in the context of *integer-to-pointer* casts. CHERI C permits such casts only from the capability-carrying `intptr_t` and `uintptr_t` types; round-trip casts via these types preserve capability metadata.

The key observation is that all PNVI mechanisms described in [16] associate additional metadata, such as an allocation ID, with pointer objects. However, in CHERI C, pointers already contain sufficient meta-information to recover their provenance. Thus, semantics can implement provenance tracking by utilising and estimating this runtime information instead.

The *PNVI-CHERI* lookup mechanism uses capability *bounds* instead of its address to locate the associated allocation. Each pointer to a newly allocated object will initially have its bounds within the footprint of the corresponding allocation. Due to the *capability monotonicity* property [36], the bounds can never escape the original allocation footprint. Thus, the allocation corresponding to a given pointer can be found by comparing its bounds[3] to a list of existing live and quarantined allocations, as shown in Algorithm 1:

Algorithm 1 Capability provenance lookup in PNVI-CHERI

Require: Capability *cap*, Allocation Map *allocmap*
Ensure: Returns `Some(alloc_id)` if found, or `None`.

```
1:  function LOOKUP_PROVENANCE(cap, allocmap)
2:    if ¬ GET_TAG(cap) then
3:      return None // Invalid capability
4:    end if
5:    base ⟵ GET_BASE(cap) // Get the lower bound
6:    for all (alloc_id, (quarantine, start, size)) ∈ allocmap
      do // Iterate over key-value pairs in the map
7:      if start ≤ base < start + size then
8:        return Some(alloc_id) // Return the allocation ID if
      base is within the range
9:      end if
10:   end for
11:   return None // No allocation found for the base address
12: end function
```

There is one special case where the lower bound of a capability could escape the original allocation. It arises when

[3] It is sufficient to check the lower bound only.

object bounds are set one-past the original allocation, but the bounds length is set to zero. This could only be done in CHERI C by using `cheri_bounds_set` intrinsic, as in the C language there is no legitimate way to create a 0-length object. Zero-length capabilities confer no authority and might be considered a nuisance encoding artefact rather than a meaningful semantic object. We choose not to allow them and propose to specify in CHERI C semantics that setting the capability bounds length to zero with `cheri_bounds_set` is `undefined behaviour`.

Since the revocation algorithms we are considering remove regions from quarantine only after all capabilities associated with them have been revoked, all valid capabilities are guaranteed to be resolved by the lookup mechanism described above to either live or quarantined allocations.

The ISO C [19] + *PNVI-ae-udi* [16], as well as CHERI C semantics rules, call for several checks dependent on pointer provenance to flag various undefined behaviours, such as out-of-bounds access or dangling pointers:

(1) Checking whether a pointer refers to a live allocation.
(2) Checking whether two pointers have the same provenance when subtracted or compared using relational operator such as <.
(3) Checking whether a pointer is inside the bounds of the corresponding memory allocation's footprint.

In *PNVI-CHERI*, Algorithm 1 allows finding the corresponding allocation record for any valid pointer. The returned allocation will either be live or quarantined, as specified by a boolean flag in the `allocation` record in the memory state described in Section 4. Checking whether two pointers have the same provenance can be trivially done by looking up and comparing their allocations. Finally, the capability bounds checks supersede the allocation bounds check, as they cannot exceed the footprint of the allocation.

To revisit our example in Listing 2, the bounds of pointer `&x` will remain unchanged after its address was incremented in line 7, and on line 11 it could be unambiguously resolved to belong to allocation `x`. The write attempt could not only then be flagged as UB, but additionally, if the compiler chose to generate code attempting the write anyway, it would be intercepted and prevented by the CHERI runtime.

4 CHERI C Memory Model

We have defined in Coq a memory object model for CHERI C that employs the *PNVI-CHERI* provenance tracking, as introduced in Section 3. This builds on the existing formalisation of CHERI C from [42]. It is defined as an instantiation of the memory model interface of the Cerberus C semantics [27], which gives us an executable oracle for a large fragment of the C11 language equipped with our CHERI C memory model in place of the PNVI models.

Cerberus defines the semantics of C11 as an elaboration of the syntax of C expressions and statements into expressions in a *Core* language, which in turn is given an operational semantics. Details regarding the memory state, pointer values, and memory operations (loads, stores, ...) are abstracted by a memory model interface from the elaboration and the operational semantics of Core. This allows the two to be integrated with minimal or no modifications to different memory models, while avoiding the need to rework aspects of the semantics of C11 unrelated to memory and the semantics of pointers.

As required by the interface, the memory model defines a type for the memory state, and operations which may use or modify this state (e.g. allocation of objects, `store`). These stateful operations use a `memM` monad, which combines *state* and *error* monads. The interface also defines the type of pointer values, and constructors and destructors (e.g. `array_shift_ptrval` corresponding to pointer arithmetic in the surface language).

The type `mem_state` describes the concrete content of bytes in memory, along with the abstract state of capabilities and the memory allocation subsystem:

$$
\begin{aligned}
\text{mem\_state} &\triangleq \text{allocmap} \times \text{bytemap} \times \text{capmap} \times \mathbb{Z} \times \mathbb{A} \\
\text{allocation} &\triangleq \mathbb{B} \times \mathbb{A} \times \mathbb{N} \\
\text{allocmap} &\triangleq \mathbb{Z} \rightharpoonup \text{allocation} \\
\text{bytemap} &\triangleq \mathbb{A} \rightharpoonup \text{byte} \\
\text{capmap} &\triangleq \mathbb{A} \rightharpoonup \mathbb{B} \times \mathbb{B} \times \mathbb{B} \\
\mathbb{A} &\triangleq \{x \in \mathbb{Z} \mid 0 \leq x < \text{ADDR\_LIMIT}\}
\end{aligned}
$$

The memory content consists of bytes addressable via bounded positive integers (type $\mathbb{A}$), which corresponds to the bytemap field in the memory state. The map is partial to account for uninitialised memory, which is treated as *unspecified* byte values. In Coq, it is implemented as a finite map $\mathbb{A} \rightarrow$ option byte.

Additionally, for each capability-size aligned address, we store some meta information for the capability that could potentially be stored at this location (the `capmap` field of the memory state). The first piece of this meta information is the "tag" representing the capability's validity. The additional two boolean flags respectively track whether the capability tag and bounds values are unspecified.[4] The capability at a given location is valid if and only if there exists a mapping in `capmap` for its address, for which the `tag` is set, and `tag_unspecified` is `false`.

All memory is allocated and released via the corresponding methods of the memory interface. The state of the allocator is maintained as a list of allocations identified by a unique allocation ID (the `allocmap` field of the memory state). Each `allocation` object bears a boolean flag indicating whether this allocation is in quarantine, along with the starting address and the allocation size in bytes. All new allocations are "live" and recorded with the `quarantine` flag set to `false`.

[4] See [42] for a discussion on why tags and bounds could be unspecified.

When memory is released, the corresponding allocation is put into quarantine by raising this flag in the allocation record. Finally, after quarantined allocations are processed by the revocation sweep, their records are expunged from `allocmap`.

The two remaining components of the memory state are the high watermark for allocation IDs, `next_alloc_id`, and the low watermark for already allocated memory, `last_address`.

The low-level memory operations attempt to faithfully simulate CHERI hardware. In particular, they ensure that all memory accesses require valid capabilities, which enforce access bounds and permissions. They also guard the integrity of the capabilities, with any attempts that could lead to forging capabilities (such as modifying individual bytes of a capability representation in memory) resulting in the tag being cleared. Additionally, valid capabilities can be stored and loaded only from capability-size aligned addresses.

The high-level interface of the memory model provides the memory operations and constructors/destructors for pointer values and integer values needed by the *Core* language used to elaborate C11.

The `memcpy` function from the standard C library is implemented as part of the memory model. Unlike in ISO C, in CHERI C, it cannot be implemented as a simple byte-copy loop because it needs to preserve valid capabilities at capability-size aligned locations that fit completely in the copied region.

There are several existing CHERI implementations that differ in address width and bounds and permission encoding schemes, and the definitions of capabilities' fields, such as *flags*, *object type*, and *seal type*. Many of these details are not pertinent to CHERI C semantics, and we have abstracted them in our memory model implementation using the Coq module system. Some of these types, like *object type*, are opaque and only require an equality predicate, while others, at an abstract level, expose a minimal required set of properties. For example, the permissions set may vary, but at the very least, we need to distinguish between *read* and *write* permissions in the memory model. For the abstract address type, the value is opaque, but the interface provides *bitwise complement*, equality, *less than* operations, and proof that the latter is *irreflexive*.

Additionally, following from the Cerberus API, the semantics is parametrised by a `Implementation` module type, which provides various implementation-defined parameters, such as storage sizes for scalar types (integers, floats, and pointers) and their alignment constraints.

Finally, some aspects of the CHERI C semantics and the memory model, in particular, are controlled by a `Switches` module type. This is a mechanism in the Cerberus model to select between semantic "flavours" (e.g. whether are not relational operators between pointers to disjoint objects is given defined semantics). In the executable semantics, some of these can be controlled via command-line options. One switch, in particular, relevant to this paper, enables instant revocation. The *revocation sweep* is implemented as a method of the memory model interface that can be synchronously called at any time. In the *instant revocation* mode (controlled by a switch), it is automatically invoked at the end of the `kill` method (which models the end of the lifetime of an object).

This fine-grained parametrisation allows the memory model to be potentially instantiated, for example, for 32-bit or 64-bit architectures, using Morello or RISC-V flavours of CHERI capabilities, with instant or delayed revocation.

The executable version of the semantics parametrises the memory model to emulate the 64-bit Morello architecture, including capabilities encoding and operations as implemented by [1].

Our *PNVI-CHERI* memory model was integrated into the Cerberus executable semantics. Apart from our memory model, Cerberus is implemented in OCaml, and we extracted OCaml code from Coq and interfaced it with the rest of the implementation. It successfully passed a semantics test suite of 94 tests, covering various aspects of CHERI C semantics, showing no discrepancies with the previous *PNVI-ae-udi*-based CHERI C semantics. These results give us added confidence that *PNVI-CHERI* is a suitable replacement for *PNVI-ae-udi* in CHERI C.

5 Memory Invariants

What guarantees of temporal safety does *PNVI-CHERI* provide? To answer this, we must consider the properties of the *PNVI-CHERI*-based memory model. The memory model serves as a stateful interface between CHERI C semantics and the memory state. We can impose a *memory invariant* that must hold for any sequence of memory model method invocations, provided that the initial state satisfies this invariant It may include unreachable states, as the actual state transition graph is constrained by the semantics.

In the Cerberus semantics, all objects, including local variables, reside in memory. Access to memory is granted only via capabilities; hence, the set of live capabilities in memory defines the set of memory regions accessible to a C program. By expressing an invariant that places constraints on these available capabilities, we can reason about potential risks of accessing freed or quarantined memory in the context of temporal memory safety.

In this section, we present several invariants on the state of our *PNVI-CHERI* memory model, corresponding to different revocation algorithms.

5.1 Base

The *base* invariant only enforces internal consistency of the memory model data structure and does not place any restrictions on memory content:

$$
\begin{aligned}
&\forall m = (\text{am}, \text{bm}, cm, \text{next\_alloc\_id}, \text{last\_address}),\\
&\quad \wedge\ \forall i, j \in \text{dom}(\text{am}),\ i \neq j \implies\\
&\qquad \text{am}(i) = (\_, \text{start}_i, \text{len}_i) \wedge \text{am}(j) = (\_, \text{start}_j, \text{len}_j) \implies\\
&\qquad \text{start}_i + \text{len}_i \leq \text{start}_j \vee \text{start}_j + \text{len}_j \leq \text{start}_i\\
&\qquad \vee\ \text{len}_i = 0 \vee \text{len}_j = 0 \qquad (1)\\
&\quad \wedge\ \forall x \in \text{dom}(\text{am}),\ \text{am}(x) = (\text{quarantine}, \text{start}, \text{len}) \implies\\
&\qquad \text{start} + \text{len} < \text{ADDR\_LIMIT} \qquad (2)\\
&\quad \wedge\ \forall k \in \text{dom}(cm),\ k \mod \text{PTR\_ALIGN} = 0 \qquad (3)\\
&\quad \wedge\ \forall k \in \text{dom}(\text{am}),\ k < \text{next\_alloc\_id} \qquad (4)\\
&\quad \wedge\ \forall x \in \text{dom}(\text{am}),\ \text{am}(x) = (\_, \text{start}, \_) \implies\\
&\qquad \text{start} \geq \text{last\_address} \qquad (5)
\end{aligned}
$$

Clause (1) ensures that allocations do not overlap. Clause (2) states that all allocations must fit in the address space. Clause (3) states that all capability metadata stored in the *capmap* map are for pointer-aligned addresses. Clause (4) states that *next_alloc_id* is indeed a high watermark for allocation IDs. Finally, Clause (5) states that *last_address* is indeed the low watermark for allocated memory blocks (the allocator starts from the top and allocates downwards in the address space).

The invariant does not impose any restrictions on capabilities in the memory, and any valid capability in memory could potentially have access to any memory location, regardless of the region it belongs to: live, quarantined, or free.

5.2 The "Dirty" Invariant (Between Sweeps)

This invariant applies to the memory state for CHERIvoke, CheriOS, Cornucopia and Cornucopia Reloaded in the "dirty" state (between sweeps). In this state, there may exist capabilities in memory pointing to regions which have been freed but not yet processed by the revocation sweep pass, and thus are living in quarantine. The *dirty* memory invariant in addition to the *base* invariant includes the following clause:

$$
\begin{aligned}
&\forall \text{addr}, \text{g},\ \text{cm}(\text{addr}) = (\text{true}, \_, \text{false}) \implies\\
&\quad \forall \text{bs},\ \text{fetch\_bytes}(\text{bm}, \text{addr}, \text{PTR\_SIZE}) = \text{bs} \implies\\
&\qquad \exists \text{c},\ \text{decode\_cap}(\text{bs}) = \text{c} \wedge\\
&\qquad\quad \exists \text{base}, \text{size}, \text{alloc\_id},\\
&\qquad\qquad \text{am}(\text{alloc\_id}) = (\_, \text{base}, \text{size}) \wedge\\
&\qquad\qquad\quad \text{base} \leq \text{cap\_get\_base}(\text{c}) \wedge\\
&\qquad\qquad\quad \text{cap\_get\_base}(\text{c}) < \text{base} + \text{size} \qquad (6)
\end{aligned}
$$

In Clause (6), `fetch_bytes` is a function which returns a list of `option byte` values from `bytemap` for a given address range, `decode_cap` attempts to construct a capability from these bytes, and `cap_get_base` returns the lower bound of the capability.

The invariant as whole states that for all addresses with a valid tag in `capmap`, there should be a corresponding byte region that decodes as a capability, and there must be an allocation (live or quarantineed) that includes the lower bound of this capability. Or in plain English:

Property 1. *Any tagged capability in memory allows access only within the bounds of live or quarantined allocations.*

5.3 The "Clean" Invariant (Post-Sweep)

The next invariant describes the state of the memory in the case of instant revocation: as if a revocation sweep was executed each time immediately after *free()*. This matches one of the deployment scenarios for CHERIvoke (without parallelism) described in [40]. Coincidentally, it also applies to CHERIoT. For Cornucopia and Reloaded, it describes the state of the memory right after a full revocation sweep, assuming no other calls to *free()* were made during the sweep. It is also applicable to the fixed-point of the revocation sweep algorithm for CheriOS.

The *clean* memory invariant includes *base* and *dirty* and the following additional clause:

$$\forall x \in \text{dom}(\text{am}), \text{am}(x) = (\text{false}, \_, \_) \qquad (7)$$

Clause (7) states that all allocations are live (the `quarantine` flag is `false`). Freed allocations are placed into quarantine, and the quarantine list is emptied by the revocation sweep. This improved security property could be stated in English as:

Property 2. *Any tagged capability in memory allows access only within the bounds of live allocations.*

It could be shown this invariant is stronger that *dirty* (*clean* implies *dirty*).

Memory regions and access to them, as allowed by invariants, are shown in Figure 1. The *base* invariant does not impose any restrictions. The *dirty* invariant gives access to both live and quarantined memory, while the *clean* invariant gives access only to live memory.

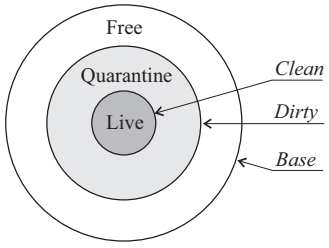

Figure 1. Invariants and accessible memory

6 Proving Invariants

We have proven in Coq that our memory model preserves the *clean* invariant, the strictest of the invariants we defined. To recall from Section 4, the memory model interface contains

stateless and stateful operations. Only the latter can change the memory state. We have proven that all stateful operations of the memory model preserve the memory invariant.

The high-level proof statements are expressed with the help of the logical typeclass: `Class PreservesInvariant {invr: mem_state_r → Prop} {T:Type} (s:mem_state_r)(M: memM T): Prop`. The class has a single constructor which states that if we start from a given state that satisfies the invariant predicate, the result of stateful operations will be either an error or will have a state satisfying the same invariant. The instances are universally quantified for all arguments of a given method. For example, Listing 3 shows the instantiation (with proof omitted) for the `kill` operation of the memory model:

```
Instance kill_PreservesInvariant
  (loc: location) (is_dyn: 𝔹) (ptr: pointer_value):
    ∀ s, PreservesInvariant mem_invariant s
               (kill loc is_dyn ptr).
```

Listing 3. `PreservesInvariant` instance for `kill`

We have proven all 36 instances of `PreservesInvariant` for high-level operations of the memory model as well as for many auxiliary functions. For functions that are known not to modify the state, we have proven instances of the `SameState` typeclass, which is similar to `PreservesInvariant` but for operations which are executed in `memM` monad but do not modify state. Then, we have proven that `SameState` implies `PreservesInvariant`.

Let's dive a little deeper into our proof techniques. Our `memM` monad is an instance of our "error with state" `errS` monad, parameterised by `memory state` and `memory error` types: `Definition memM := errS mem_state memMError.` An abridged definition of the `errS` monad, using typeclasses from ExtLib [26], is shown in Listing 4.

```
Variable (St : Type) (ErrT: Type).

Notation err := (sum ErrT).
Definition errS A := St → (St*(ErrT+A)).
Instance Monad_errS: Monad errS.
Instance Exception_errS : MonadExc ErrT errS.
Instance State_errS: MonadState St errS.
Definition evalErrS {A:Type} (c:errS A) (initial:St): err A.
Definition execErrS {A:Type} (c:errS A) (initial:St): err St.
```

Listing 4. `errS` monad

The `evalErrS` returns a value, and `execErrS` returns the state. Additional operations (not shown) provided by the state and exception monads are `get`, `put`, `update`, and `raise`.

Stateful memory model methods are written in a monadic style within the `memM` monad, where operations are sequenced using `bind`, values are injected with `ret`, and errors are flagged using `raise`. The state can be examined using `get` and implicitly modified using `put` or `update`.

The most burdensome aspect of reasoning deals with sequencing computations using `bind`. We defined several relevant lemmas (shown in Listing 5) that could be useful in different contexts for reasoning about sequenced computations.

```
Variable T T': Type.
Variable m: memM T'.
Variable c: T' → memM T.
Variable s: mem_state.

Instance bind_PreservesInvariant:
  PreservesInvariant s m →
  (∀ s' x, PreservesInvariant s' (c x)) →
PreservesInvariant s (bind m c).

Instance bind_PreservesInvariant_full:
  invr s →
  (∀ s' x, m s = (s', inr x) →
  (match execErrS (c x) s' with
    | inl _ ⇒ True
    | inr b ⇒ invr b
    end)) →
PreservesInvariant s (bind m c).

Instance bind_PreservesInvariant_value:
  (invr s → (∀ s' x, m s = (s', inr x) →
  (invr s' ∧ PreservesInvariant s' (c x)))) →
PreservesInvariant s (bind m c).
```

Listing 5. `PreservesInvariant` instances for `bind`

The simplest one is `bind_PreservesInvariant`, which does not expose the bound variable and assumes that the continuation is always invariant-preserving. The `bind_PreservesInvariant_full` lemma allows reasoning about the value of the bound variable m and the intermediate state s', but does not require the computation m to preserve the invariant. It is useful in cases where a sequence of computations preserves an invariant while the individual steps do not necessarily do so. The last one, `bind_PreservesInvariant_value`, allows reasoning about the value of the bound variable x and the intermediate state s', which must also satisfy the invariant.

Some additional lemmas for monadic reasoning are shown in Listing 6.

```
Variable s: mem_state.
Variable T: Type.

Instance bind_get_PreservesInvariant
  {c: mem_state → memM T}:
  PreservesInvariant s (c s) →
PreservesInvariant s (bind get c).

Instance get_PreservesInvariant: PreservesInvariant s get.

Instance ret_PreservesInvariant:
  ∀ (x:T), PreservesInvariant s (ret x).
```

```
12
13 Instance raise_PreservesInvariant
14   ∀ e, PreservesInvariant s (raise e).
15
16 Instance update_PreservesInvariant
17   {f: mem_state → mem_state}:
18   (invr s → invr (f s)) →
19   PreservesInvariant s (update f).
20
21 Instance put_PreservesInvariant:
22   ∀ s', invr s' → PreservesInvariant s (put s').
23
24 Instance put_PreservesInvariant_dep:
25   ∀ s', (invr s → invr s') → PreservesInvariant s (put s').
```

Listing 6. Additional `PreservesInvariant` instances

A special case of `bind` applied to `get`, found in `bind_get_PreservesInvariant`, exploits the fact that `get` does not modify the state (a fact also used in `get_PreservesInvariant`). The `ret` in `ret_PreservesInvariant` is trivially preserving as it does not modify the state. The `raise` operation raises an error, which satisfies the `PreservesInvariant` formulation, as proven by `raise_PreservesInvariant`. In `update_PreservesInvariant`, the `update` is state-preserving as long as the state transition function is also state-preserving. Finally, for the `put` operation, we have two lemmas. In the first one, `put_PreservesInvariant`, there is no relation between the original state s and the new state s' as long as s' satisfies the invariant. In the second one, `put_PreservesInvariant_dep`, the invariant compliance of s' depends on that of s.

Manual proving of thousands of lines of code turned out to be tedious, so we wrote some `Ltac` automation to aid with it. For example, an abridged version of the `preserve_step` tactic is shown in Listing 7.

```
1 Ltac preserves_step :=
2   match goal with
3   |[⊢ PreservesInvariant _ _ (bind get _)] ⇒
4           apply bind_get_PreservesInvariant
5   |[⊢ PreservesInvariant _ _ (bind _ _)] ⇒
6           apply bind_PreservesInvariant
7   |[⊢ PreservesInvariant _ _ (raise _)] ⇒
8           apply raise_PreservesInvariant
9   |[⊢ PreservesInvariant _ _ (ret _)] ⇒
10          apply ret_PreservesInvariant
11  |[⊢ PreservesInvariant _ _ get] ⇒
12          apply get_PreservesInvariant
13  |[⊢ PreservesInvariant _ _ (put _) ] ⇒
14          apply put_PreservesInvariant
15  |[⊢ PreservesInvariant _ _ (update _)] ⇒
16          apply update_PreservesInvariant
17  |[⊢ PreservesInvariant _ _] ⇒ typeclasses eauto
18 end.
```

Listing 7. `preserves_step` tactic

This tactic allows quick stepping through monadic expressions in the goal by introducing necessary intermediate variables, hypotheses, and more granular sub-goals. While the handling of monadic primitives is hardcoded into the tactic, it will attempt to resolve `PreservesInvariant` goals for user-defined functions using the Coq typeclass resolution mechanism. Available `PreservesInvariant` instances for additional functions will be applied, if possible, via the `typeclasses eauto` default clause. There are additional tactics that allow performing these steps repeatedly, destructing `match` and `let` expressions along the way.

Additionally, there is a set of lemmas for inverting monadic operations in the hypotheses, along with corresponding automation. A snippet of these (a lemma and an `Ltac` script fragment) is shown in Listing 8.

```
1 Lemma bind_memM_inv
2   {T T': Type}
3   {m: memM T'}
4   {c: T' → memM T}
5   {x: T}
6   {s s': mem_state}:
7 (bind m c) s = (s', inr x) →
8 ∃ s'' y, m s = (s'', inr y) ∧ c y s'' = (s', inr x).
9
10 Ltac state_inv_step := match goal with
11   (* memM bind: *)
12   |[ H: (bind _ (λ x ⇒ _)) ?s = (_ ,inr _) ⊢ _ ] ⇒
13     let H1 := fresh H in
14     let H2 := fresh H in
15     let x' := fresh x in
16     let s' := fresh s in
17     apply bind_memM_inv in H;
18     destruct H as [s' [x' [H1 H2]]];
19   (* ... *)
20 end.
```

Listing 8. Sample definitions for inverting monadic operations

The resulting automation has proven to be handy, unwinding all state and error monad-related mechanics and leaving the user to prove the actual state modifications and invariants for their results.

Although we defined our monadic reasoning lemmas and automation for the `memM` monad, `mem_state` state, and `PreservesInvariant` predicate, the underlying framework could be easily generalised for all instances of the `errS` monad and arbitrary state types and invariant predicates. We plan to eventually refactor and release this as a general-purpose library for reasoning about monadic programs with error handling and state.

Because the memory model interface was defined by the OCaml implementation of Cerberus, some data structures passed to the memory model could not be proven to be consistent, even though the underlying OCaml code ensures they are correct by construction. For example, when dealing

with a `struct` type, the list of field values needs to correspond exactly to the list of field types in the `struct` definition. In a pure Coq implementation, this would have been enforced by dependent types. To avoid adding additional assumptions or axioms to our proofs, we opted to add assertions to the memory model implementation, which allowed us to derive the necessary guarantees. These assertions could be later removed for performance, if required.

The Coq implementation of the memory model is relatively compact: the core logic consists of 3,481 lines of code (LoC) plus 4,018 LoC of auxiliary definitions (helper functions and data structures). The proofs of the invariant amount to 9,546 LoC.

7 Discussion of the Results

The relation between memory states described by *clean* and *dirty* invariants[5], which we have proven (as defined in Section 5), is shown in Figure 2.

Initially, the empty memory state is *clean*. Any subsequent calls to *free()* transition it to *dirty*, where it remains until the revocation pass is executed via *revoke()*.

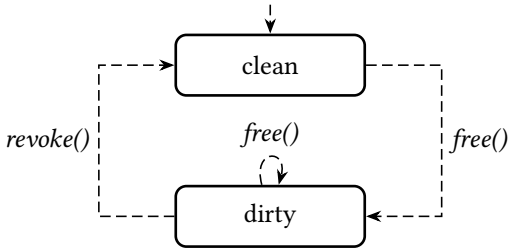

Figure 2. State transitions between *clean* and *dirty* states.

In the CHERI capability machine, memory access is governed by valid capabilities, which collectively define the footprint of memory accessible to a program. Initially, in an empty memory state, neither use-after-return (UAR) nor use-after-free (UAF) issues are possible, as no live capabilities exist.

Subsequent calls to *free* result in freed memory blocks being placed into a quarantine state. These blocks remain accessible under the *dirty* memory invariant (as defined in Section 5), potentially allowing UAF. However, new allocations are not permitted to overlap with either live or quarantined regions, as enforced by the invariants, thereby ensuring that UAR is impossible in this stage.

A revocation pass transitions memory back to a *clean* state, as specified by the *clean* memory invariant (Section 5). This process invalidates all capabilities pointing to quarantined regions while preserving those referencing live memory, ensuring that neither UAF nor UAR can occur in the *clean* state.

[5]The *base* invariant is part of both *clean* and *dirty* invariants and always holds.

In summary, the *dirty* and *clean* memory invariants guarantee the absence of UAR across all memory states and the prevention of UAF in the *clean* state.

To recap, our proposed memory model builds upon the previously proposed in [42] CHERI C memory model but replaces the use of *PNVI-ae-udi* with *PNVI-CHERI*. The principal innovation lies in achieving greater abstraction simplification while preserving compatibility with the CHERI C semantics. While not formally proven, we claim *soundness* and *completeness* properties of the proposed memory model with respect to the original *PNVI-ae-udi*-based CHERI C memory model from [42], defined as follows:

Soundness. The proposed memory model retains compatibility with the behaviours permitted by *PNVI-ae-udi*, ensuring no additional behaviours are introduced. The sole deviation is the introduction of new undefined behaviour concerning the handling of zero-length capability bounds, as elaborated in Section 2.

Completeness. Compiler optimisations predicated on pointer provenance remain valid under the new model. Although *PNVI-CHERI* and *PNVI-ae-udi* adopt distinct approaches to pointer provenance tracking, both ultimately address the same fundamental question: "What is the provenance of a given pointer?" As the answers provided by the two models align, the observable behaviours of CHERI C programs remain consistent across both semantics.

8 Related Work

Our work builds upon the CHERI C semantics and the capability-enabled memory model presented in [42], which itself builds on the foundational Cerberus ISO C mechanised semantics and the provenance-aware memory model [27].

Our contributions, summarised in Section 1, extend this existing work. Specifically, we modify the CHERI C memory model to introduce the *PNVI-CHERI* pointer provenance tracking scheme and a new pointer revocation mechanism. Additionally, the formulation of invariants and proofs of their correctness for our enhanced memory model represents completely new work, addressing the concept of temporal safety in CHERI C for the first time.

Our implementation if CHERI C memory model is based on [42], but it uses *PNVI-CHERI* instead of *PNVI-ae-udi*, which allowed for significant simplifications to the internal logic and data structures. Some examples of simplification are: pointer values and bytes in memory no longer carry provenance information; allocations do not carry an `exposure` flag; and memory bytes do not keep track of an optional integer offset. All provenance-related checks and related logic were re-implemented using capabilities' fields instead of provenance meta-information, and the redundancies between capability bounds and allocation footprint checks

were eliminated. This made our model simpler both conceptually and implementation-wise.

We are aware of only two other formal verification projects related to CHERI C: a bounded model checker [8], and a CompCert-style CHERI C memory model [31].

There are several other memory models proposed for ISO C. They are not easy to compare directly as their design is impacted by how the rest of C semantics is formalised, but they could be roughly classified by the approach they have taken to represent C objects in memory.

Memory models influenced by CompCert's memory model [25] use two-level block/offset addressing to index byte values. The CH_2O memory model [22] uses abstract trees, mirroring the structure of C data types, to store memory values, with paths through these trees used to address them. These models, sometimes referred to as *logical* memory models, are convenient for use from the perspective of C semantics, but they make it challenging to model pointer-to-integer casts, pointer bit manipulation, and memory exhaustion. These capabilities are necessary for reasoning about low-level compiler optimisations and linking them with underlying hardware formalisations to achieve the long-sought but elusive goal of "end-to-end" verification. By contrast, *concrete* memory models represent memory as an array of bytes, with each byte addressable by an integer. This simplifies the modelling of pointer-to-integer casts, pointer bit manipulation, and memory exhaustion. There is ongoing work [4–7, 21, 24] aiming to reconcile *logical* and *concrete* memory models. Some of the aforementioned memory models [24, 29] implement forms of pointer provenance and attempt to address temporal safety.

Our monadic reasoning framework was inspired by Interaction Trees (ITrees) [41], although it is significantly simpler, focusing exclusively on state and error monads. While ITrees provide a rich framework for reasoning about potentially non-terminating programs, non-determinism, and coinductive computations, our framework is designed for simplicity and ease of use, tailored specifically to deterministic programs with state and error handling.

9 Future Work

We proved the *clean* invariant but could also prove the *dirty* invariant, which should be relatively simple. Once both are proven, we could trivially demonstrate that the former implies the latter.

Using our model definition and reasoning framework, in the future we could prove additional model properties such as *pointer monotonicity*, which states that memory model operations, aside from new memory allocations, should not expand the *security perimeter*. The *security perimeter* is defined as the union of all regions of memory accessible via (transitively) available pointers and their permissions. Non-expansion of the security perimeter means that the accessible area of memory can only shrink, and the permissions for existing regions can only become more restrictive (e.g., changing from read-write to read-only, but not vice versa).

The CHERI C memory model, along with the base version of Cerberus C semantics on which it relies, does not model the C/C++11 concurrency memory model. Several existing efforts [23, 30] have sought to add concurrency to Cerberus, which could be leveraged in the future to implement a concurrent version of CHERI C semantics. It should be noted, however, that our *PNVI-CHERI* provenance tracking scheme is independent of concurrency semantics and should apply equally to concurrent implementations.

While the abstract semantics presented in this work do not differentiate between heap and stack temporal safety, to date all implementations of temporal safety mechanisms atop CHERI have done so for performance, as the stack is designed to reuse addresses promptly in ways that the heap is not. Indeed, several of the real-world implementations of CHERI revocation, and, in particular, the CHERIvoke family on which we model our semantics, offer *exclusively* heap temporal safety, leaving stack temporal safety out of scope. Instead, the semantics presented here assume that it is possible to act on all of memory *atomically*. As such, the prompt reuse of address space in stack allocations is not a concern, and the practical concerns around concurrency and performance that have driven implementation work can be ignored. There have been efforts to extend the architecture in various ways to better facilitate stack temporal safety [13, 14, 18, 34], CHERIoT has mechanisms more narrowly targeting *inter-compartment* stack temporal safety [3, §5.2], and CheriOS built a temporally-safe stack C *ABI* atop its heap temporal safety mechanism [10, §4.1]. However, these require more machinery and are not without their own practical caveats. A completely satisfactory way of capturing our semantics' indifference to heap- vs. stack-allocated objects remains elusive. That said, stack objects' lifetimes are often simpler than heap objects', with compiler optimization techniques such as *escape analysis* can often ensure that no references to a stack object outlive the object itself [10, § C]; one could imagine moving (possibly-)escaping objects into the heap.

Temporal safety as treated in this work is a very particular aspect of memory safety at large, ensuring properties of valid pointers within program memory. Notably, these properties are orthogonal to the *flow* of pointers within the program. Meanwhile, the C programming language allows for allocations to hold "indeterminate" values until initialization. In practice, these "indeterminate" values are often old state from the program, previously held in the object(s) whose memory has been reused, but in principle arbitrary values conjured "out of the air" would still be semantically correct. A risk to CHERI C safety is that these indeterminate values could be valid capabilities to live objects in the system, resulting in unintentional and covert capability flow between objects in the program. Such flows will not violate

the properties proven in this work. In practice, some implementations of allocators, for added security at the cost of performance, enforce zero-initialisation of all allocations before making them available to the program. In CHERI C, clearing the tags is sufficient, and would also be a side effect of zero-initialisation. The added benefit of zero-initialisation, compared to just tag clearing, is that it also prevents *information* leaks via non-capabilities, as opposed to *authority* leaks via capabilities. Both initialisation behaviours can be formally modelled in our memory model, allowing for a stronger memory invariant to be stated and proven.

10 Conclusions

We introduce a new CHERI-specific pointer provenance tracking scheme: *PNVI-CHERI*. It is defined in the spirit of existing *PNVI-plan* and *PNVI-ae-udi* schemes, re-using the provenance tracking terminology and framework of the working draft ISO Technical Specification [16], and should provide a logical extension of this specification for CHERI C. The resulting *PNVI-CHERI* provides compiler writers with a framework to reason about pointer aliasing and confirm the correctness of potential optimisations.

The resulting *PNVI-CHERI*-based CHERI C memory model is significantly simpler and easier to understand than the *PNVI-ae-udi*-based one from [42], which could help facilitate its wider adoption not only by the research community but also by practitioners developing tools for CHERI C. The memory model formalisation in Coq enables other researchers to formally prove properties of the CHERI memory model.

We analysed several existing low-level CHERI revocation mechanisms and modelled their impact on CHERI C semantics. Moreover, we have formally defined and proved security properties of the memory model related to temporal safety.

Acknowledgements

This work was supported by the UK Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme, to deliver a DSbDtech enabled digital platform (grant 105694). This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 789108, ERC AdG ELVER). This work was funded in part by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee for ERC-AdG-2022, EP/Y035976/1 SAFER.

References

[1] Ricardo Almeida et al. 2024. *coq-cheri-capabilities* library. GitHub repository. https://github.com/rems-project/coq-cheri-capabilities Accessed 2024-09-13.

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

[3] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2023. CHERIoT: Complete Memory Safety for Embedded Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 641–653. https://doi.org/10.1145/3613424.3614266

[4] Calvin Beck, Irene Yoon, Hanxi Chen, Yannick Zakowski, and Steve Zdancewic. 2024. A Two-Phase Infinite/Finite Low-Level Memory Model: Reconciling Integer–Pointer Casts, Finite Space, and undef at the LLVM IR Level of Abstraction. *Proc. ACM Program. Lang.* 8, ICFP, Article 263 (aug 2024), 29 pages. https://doi.org/10.1145/3674652

[5] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2014. A Precise and Abstract Memory Model for C Using Symbolic Values. In *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8858)*, Jacques Garrigue (Ed.). Springer, 449–468. https://doi.org/10.1007/978-3-319-12736-1_24

[6] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2015. A Concrete Memory Model for CompCert. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9236)*, Christian Urban and Xingyuan Zhang (Eds.). Springer, 67–83. https://doi.org/10.1007/978-3-319-22102-1_5

[7] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2017. CompCertS: A Memory-Aware Verified C Compiler Using Pointer as Integer Semantics. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10499)*, Mauricio Ayala-Rincón and César A. Muñoz (Eds.). Springer, 81–97. https://doi.org/10.1007/978-3-319-66107-0_6

[8] Franz Brauße, Fedor Shmarov, Rafael Menezes, Mikhail R. Gadelha, Konstantin Korovin, Giles Reger, and Lucas C. Cordeiro. 2022. ESBMC-CHERI: Towards Verification of C Programs for CHERI Platforms with ESBMC. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 773–776. https://doi.org/10.1145/3533767.3543289

[9] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 379–393. https://doi.org/10.1145/3297858.3304042

[10] Lawrence G. Esswood. 2021. *CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor.* Ph. D. Dissertation. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-961

[11] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Jessica Clarke, Peter Rugg, Brooks Davis, Mark Johnston, Robert Norton, David Chisnall, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2024. Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 251–268.

https://doi.org/10.1145/3620665.3640416

[12] Wesley Nathaniel Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*. 608–625. https://doi.org/10.1109/SP40000.2020.00098

[13] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.* 5, POPL, Article 6 (jan 2021), 30 pages. https://doi.org/10.1145/3434287

[14] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. 2022. Le temps des cerises: efficient temporal stack safety on capability machines using directed capabilities. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 74 (April 2022), 30 pages. https://doi.org/10.1145/3527318

[15] Richard Grisenthwaite, Graeme Barnes, Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Jonathan Woodruff. 2023. The Arm Morello Evaluation Platform - Validating CHERI-Based Security in a High-Performance System. *IEEE Micro* 43, 3 (2023), 50–57. https://doi.org/10.1109/MM.2023.3264676

[16] Jens Gustedt, Peter Sewell, Kayvan Memarian, Victor BF Gomes, and Martin Uecker. 2022. A Provenance-aware Memory Object Model for C. Working draft ISO Technical Specification TS6010.

[17] Ben Hawkes. 2019. 0day In the Wild. (2019). Project Zero team blog, Google. https://googleprojectzero.blogspot.com/p/0day.html. Accessed 2023-04-19.

[18] Sander Huyghebaert, Thomas Van Strydonck, Steven Keuchel, and Dominique Devriese. 2020. Uninitialized Capabilities. arXiv:2006.01608 [cs.PL] https://arxiv.org/abs/2006.01608

[19] International Organization for Standardization. 2018. *Programming Languages – C.* International Standard 9899:2018. ISO/IEC, Geneva, Switzerland.

[20] Nicolas Joly, Saif ElSherei, and Saar Amar. 2020. *Security Analysis of CHERI ISA.* Technical Report. Microsoft Security Response Center. https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf

[21] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A formal C memory model supporting integer-pointer casts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 326–335. https://doi.org/10.1145/2737924.2738005

[22] Robbert Jan Krebbers. 2015. *The C Standard Formalized in Coq.* Ph. D. Dissertation. Radboud University, Nijmegen, Netherlands.

[23] Stella Lau, Victor B. F. Gomes, Kayvan Memarian, Jean Pichon-Pharabod, and Peter Sewell. 2019. Cerberus-BMC: A Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 387–397. https://doi.org/10.1007/978-3-030-25540-4_22

[24] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling high-level optimizations and low-level code in LLVM. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 125:1–125:28. https://doi.org/10.1145/3276495

[25] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2.* Research report RR-7987. INRIA. http://hal.inria.fr/hal-00703441

[26] Gregory Malecha et al. 2012. ExtLib Coq library. GitHub repository. https://github.com/coq-ext-lib/coq-ext-lib Accessed: 2024-09-11.

[27] Kayvan Memarian. 2023. *The Cerberus C semantics.* Technical Report UCAM-CL-TR-981. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-981

[28] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C Semantics and Pointer Provenance. *Proc. ACM Program. Lang.* 3, POPL, Article 67 (January 2019), 32 pages. https://doi.org/10.1145/3290380

[29] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *Proceedings of the 2010 International Symposium on Memory Management* (Toronto, Ontario, Canada) *(ISMM '10)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/1806651.1806657

[30] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. 2016. An operational semantics for C/C++11 concurrency. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 111–128. https://doi.org/10.1145/2983990.2983997

[31] Seung Hoon Park, Rekha R. Pai, and Tom Melham. 2023. A Formal CHERI-C Semantics for Verification. In *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13993)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer, 549–568. https://doi.org/10.1007/978-3-031-30823-9_28

[32] Alex Rebert and Christoph Kern. 2024. *Secure by Design: Google's Perspective on Memory Safety.* Technical Report. Google Security Engineering. https://research.google/pubs/secure-by-design-googles-perspective-on-memory-safety/

[33] CWE Team. 2023. CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/index.html Access: 2024-09-14.

[34] Stelios Tsampas, Dominique Devriese, and Frank Piessens. 2019. Temporal Safety for Stack Allocated Memory on Capability Machines. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. https://doi.org/10.1109/CSF.2019.00024

[35] Robert N. M. Watson, Ben Laurie, and Alex Richardson. 2021. *Assessing the Viability of an Open-Source CHERI Desktop Software Ecosystem.* Technical Report. Capabilities Limited. https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/20210917-capltd-cheri-desktop-report-version1-FINAL.pdf

[36] Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. 2019. *An Introduction to CHERI.* Technical Report UCAM-CL-TR-941. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-941

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

[38] WG14. 2004. Defect Report 260. http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm.

[39] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 457–468. https://doi.org/10.1109/ISCA.2014.6853201

[40] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 545–557. https://doi.org/10.1145/3352460.3358288

[41] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (dec 2019), 32 pages. https://doi.org/10.1145/3371119

[42] Vadim Zaliva, Kayvan Memarian, Ricardo Almeida, Jessica Clarke, Brooks Davis, Alexander Richardson, David Chisnall, Brian Campbell, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2024. 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 181–196. https://doi.org/10.1145/3617232.3624859

Received 2024-09-17; accepted 2024-11-19