# Defining and Preserving More C Behaviors: Verified Compilation Using a Concrete Memory Model

**Andrew Tolmach**
Portland State University, Portland, OR, USA

**Chris Chhak**
Portland State University, Portland, OR, USA

**Sean Anderson**
Portland State University, Portland, OR, USA

## Abstract

We propose a concrete ("pointer as integer") memory semantics for C that supports verified compilation to a target environment having simple "public vs. private" data protection based on tagging or sandboxing (such as the WebAssembly virtual machine). Our semantics gives definition to a range of legacy programming idioms that cause undefined behavior in standard C, and are not covered by existing verified compilers, but that often work in practice. Compiler correctness in this context implies that target programs are secure against all control-flow attacks (although not against data-only attacks). To avoid tying our semantics too closely to particular compiler implementation choices, it is parameterized by an novel form of oracle that non-deterministically chooses the addresses of stack and heap allocations. As a proof-of-concept, we formalize a small RTL-like language and verify two-way refinement for a compiler from this language to a low-level machine and runtime system with hardware tagging. Our Coq formalization and proofs are provided as supplementary material.

**2012 ACM Subject Classification** Security and privacy → Logic and verification; Software and its engineering → Compilers; Software and its engineering → Semantics

**Keywords and phrases** Compiler verification, C language semantics, Coq proof assistant

**Digital Object Identifier** 10.4230/LIPIcs.ITP.2024.32

**Supplementary Material** *Software*: `https://github.com/hope-pdx/MoreC.git`

**Funding** Work supported by the National Science Foundation under Grant No. 2048499.

## 1 Introduction

Undefined memory behaviors (UBs) in C programs, notably buffer overflows, are a major source of bugs and security exploits in real world systems. One approach to this problem is to detect or prevent memory UBs at runtime, turning silent violations into observable failures. A wide variety of hardware and software mechanisms have been proposed to achieve this [11, 29, 30, 12, 18, 42, 33, 13]. These mechanisms offer complex and non-obvious trade-offs between execution overhead and implementation complexity on one hand, and precision and reliability of memory safety enforcement on the other. For example, we might naively wish to trap all violations of spatial and temporal safety [37] at the granularity of individual C memory objects, but this may cause unacceptable execution overhead; moreover, it may break running systems, because lots of legacy C code relies on UB idioms that (usually) work in practice. Thus, we may be driven to employ more coarse-grained protection.

One simple idea is to classify all in-memory data as either public or private [41]. *Public data* consists of stack-allocated arrays and variables whose addresses are taken, heap objects allocated by `malloc`, and globals. *Private data* includes everything else the compiler puts in memory: control information such as saved return addresses or heap metadata, unaddressable

variables, function parameters and return values, spilled temporaries, etc. *Public vs. Private (PvP) protection* enforces that program loads and stores via C data pointers can touch only public data; private data is protected and accessed only by (trustworthy) compiler-generated or runtime system code. However, no protection boundaries are enforced between different pieces of public data; e.g., an out-of-bounds store to one array might overwrite the contents of another array. PvP can be implemented by a variety of techniques, including sandboxing or tagging (see §2). It is essentially the memory model provided by the WebAssembly virtual machine [14], and it has been proposed as the minimal "backstop" protection model for the Tagged C system of security policy enforcement [9, 2].

When properly used by a compiler, PvP (combined with a mechanism for preventing corruption or forging of function pointers) should suffice to prevent all *control-flow attacks* [37] i.e., it should be impossible for a compiled program to corrupt its own control flow. It would be natural to formalize this intuition as part of the specification and verification of compiler correctness. But conventional C compilers, including the CompCert verified compiler [20], make no guarantees at all about programs that exhibit UB. Indeed, compilers like `gcc` and Clang notoriously take advantage of the assumption that "UB cannot happen" to perform aggressive optimizations that often surprise programmers and can lead to serious security vulnerabilities [43, 36]. Hence, if we want to use compiler semantic preservation to characterize the security guarantees provided by PvP, we need to start from a source semantics that defines more memory behaviors than standard C.

To this end, we describe the design of a *concrete* memory semantics for C and a strategy for verified compilation from C with this semantics to a target machine and runtime system that enforce PvP protections. The concrete semantics treats (data) pointers as word-size machine integers. Load and store are well-defined at every address (i.e., every integer), possibly as an explicit *out-of-bounds (OOB)* failstop. Memory is finite, so *out-of-memory (OOM)* failstops are also possible. Since pointers are integers, the concrete semantics supports arbitrary arithmetic operations over them, including many useful low-level programming idioms that are UB in ISO standard C (see §3).

Using concrete pointers distinguishes us from most verified C compiler efforts, which follow the pioneering example of CompCert [22] by modeling pointers as a pair of abstract block identifier and offset. CompCert's use of a single, abstract, infinite memory model across all compiler phases considerably eases verification; making do without this is part of the challenge we take up in this work. More fundamentally, changing to a concrete pointer semantics inhibits some useful compiler optimizations (see §3), which may make compiled code run more slowly. The payoff is that we significantly widen the set of source programs for which a verified compiler guarantees to preserve correct control-flow behavior.

At the same time, we don't want our C semantics to be *too* concrete: it obviously should not depend on the choice of PvP enforcement mechanism or the details of the compiler or runtime system. This goal motivates the most novel aspect of our approach, which is the treatment of memory allocation. The concrete source semantics must assign a well-defined integer address to each stack and heap object. These integers must not change during compilation—otherwise, semantics preservation would break. But stack frame offsets are not determined until late in the compilation pipeline, and heap locations not until the `malloc` implementation executes at run time. So the source semantics must somehow predict where each object will actually live by using the details of exactly how the compiler and runtime work. Yet putting these details in the semantics itself would pollute it horribly, making the source language definition implementation-dependent. Our solution is to parameterize the source semantics with an *oracle*—essentially an external source of non-determinism—which it

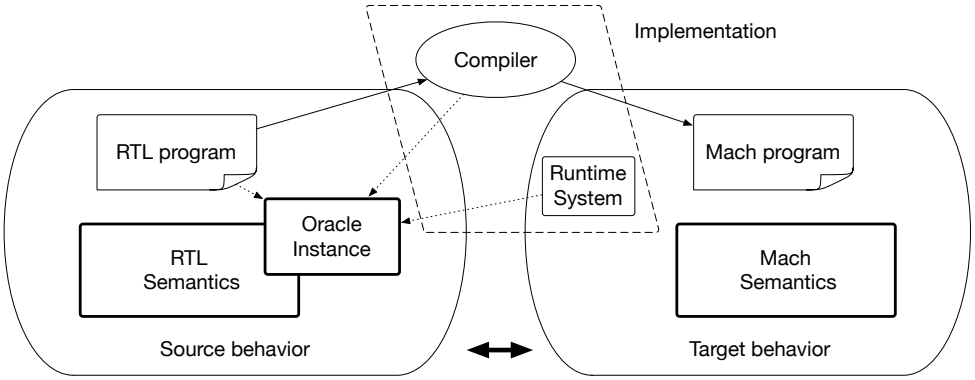


**Figure 1** Proof Architecture. Dotted lines represent information dependencies; the thick double-headed arrow is bi-directional refinement.

consults to obtain the addresses of allocated objects. The oracle is packaged into a memory model with an abstract interface equipped with a small set of axioms that characterize its behavior, which is carefully designed to be independent of the PvP enforcement mechanism. The C semantics can be understood (and used to reason about C program behavior) based just on the memory model interface and axioms; it is entirely independent of how the oracle is instantiated. For any given *implementation* (compiler and runtime system for a particular PvP target), we will be able to construct a corresponding memory model *instantiation* that validates the model's axioms. A proof of semantic preservation for a compiler connects the behavior of the source semantics equipped with a particular instantiation to the behavior of the corresponding implementation (see Figure 1). The **design and axiomatization of the oracular memory model** is our first technical contribution (§4).

Our second technical contribution is a **proof-of-concept verified compiler**, fully formalized in Coq [10], that prototypes key aspects of our approach (§5). We start by formalizing the oracular memory model's interface and axiomatic properties. We then define a small toy source language RTL with functions and concrete memory, and give it a semantics parameterized by the memory model. We define a simple RISC-like target machine Mach and runtime system with tag-based memory protection, and a compiler from source to target. Then we instantiate the memory model so that its allocation behavior matches the target implementation and verify that the instantiation obeys the axioms. Finally, we prove bi-directional refinement between the source semantics, operating under that instantiation, and the target implementation. Much of our formal framework is borrowed from CompCert, but our memories use concrete addresses rather than abstract blocks and offsets, and, unlike in CompCert, the source and target model memory quite differently.

Determinizing the memory behavior of RTL with an oracle introduces some technical complexity in our proofs, but having a deterministic source language lets us use a forward simulation to prove semantic equivalence, which is much easier to construct than a backward one. As usual in CompCert [21], we rely on determinism of the target language Mach to derive a backward simulation automatically from the forward one. In a full compiler, RTL might itself be the target language for an earlier pass to be verified by forward simulation, giving another reason for wanting it to be deterministic. Moreover, defining an explicit oracle lets us prove that the memory model axiomatization is consistent.

We view this work as just an initial step in a larger project of building a "boring" [4] but fully secure C compiler. In particular, this paper considers only data pointers; function

pointer protection is also essential to guarantee correct control flow (see §7). More broadly, to get formal guarantees for the full range of programs accepted by standard compilers, we need a source semantics from which *all* UB has been removed (including non-memory UB such as integer overflow). Ultimately, we would like to show a secure compilation property [32] for systems that link C code against arbitrary machine code restricted to access only public data.

## 2 Background: PvP Enforcement

Our memory oracle can be instantiated using a wide range of PvP mechanisms, both hardware and software. These fall into two main categories: tagging and sandboxing. Our formal proof-of-concept compiler development assumes a tagged target, but could readily be retargeted for sandboxing.

*Tagging* mechanisms adjoin a metadata tag to each byte (or chunk of adjacent bytes) in memory, and to each pointer, and compare them at load and store operations. A one-bit tag suffices to distinguish public from private addresses. Using tagging lets the compiler keep the ordinary unified single-stack layout, with public and private data interleaved freely within each stack frame. Similarly, the runtime system can use tags to protect private metadata (e.g. block lengths, free list pointers, etc.) appearing in heap block headers or within unused blocks, as is common in conventional heap allocator implementations. To be secure and reasonably efficient, tagging requires hardware support, which is becoming increasingly common, e.g. via ARM MTE [34, 3, 24] or the PIPE ISA extension [13, 1].

*Sandboxing* places all public data in a contiguous region of memory and then forces all public loads and stores to lie within that region. It has been widely used for browser protection [44] and has recently been popularized as part of the WebAssembly virtual machine [14]. To use sandboxing, the compiler must adopt a non-standard memory layout that places just the public parts of the stack and heap in the sandbox. While direct hardware support for sandboxing is possible (e.g. via x86 segment-based addressing [44]), it is often implemented using software instrumentation. The simplest approach is just to add explicit bounds checks around each public access; this is expensive, but gives a hard failstop in the event of an error. Software Fault Isolation (SFI) [40, 17] is a cheaper alternative based on the assumption that the sandbox size and base address alignment have the form $2^n$; then any arbitrary integer can be "warped" to an address within the sandbox by zeroing all but its low-order $n$ bits and or-ing in the base address. OOB accesses do not failstop, but memory outside the sandbox is never corrupted.

## 3 Concrete Memory Semantics for C

In this section, we sketch the form of concrete memory semantics we envisage for C, and examine some of its consequences. Public memory is structured into *regions*, which are used to store scalars, `struct`s, or `union`s whose address is (potentially) taken, arrays, and `malloc`ed heap objects. Region allocation and deallocation can occur either implicitly, e.g. for locals during function entry/exit and for globals at program start, or explicitly, for heap data managed by `malloc` and `free` calls in program code. The semantics assigns a concrete integer address to the base of each allocated region by consulting an *oracular memory model*, which is a parameter of the semantics.

The semantics makes no distinction between data pointers and word-sized machine integers. Thus pointers support the same arithmetic and bitwise operations as ordinary

```
void *memmove(void *s1,
              const void *s2,
              size_t n) {
  char *dest = (char *) s1;
  const char *src = (const char *) s2;
  if (dest <= src)  // UB in standard C
    while (n--)
      *dest++ = *src++;
  else {
    src += n;
    dest += n;
    while (n--)
      *--dest = *--src;
  }
  return s1;
}
```

```
extern char hash(void *p);
int main() {
  int *p = (int *) malloc(sizeof(int));
  *p = 0;
  int *q = // UB in standard C
     (int *) ((uintptr_t) p | (hash(p) & 0xF));
  int *r = (int *) (((uintptr_t) q >> 4) << 4);
  return *r;
}

void bad() {
  char p[4], q[4]; // may failstop with OOM
  q[2] = 0;
  p[6] = 1;        // may failstop with OOB
  print(q[2]);     // if reached, prints number
}
```

**Figure 2** Concrete pointer examples (adapted from [6, 19])

integers, and casts between pointers and integers are semantic no-ops. Loads and stores through *any* integer that points into a public region succeed and obey the usual "good variables" properties (i.e., a load from a given location returns the value most recently stored there). Loads or stores though an integer that points at an *unallocated* location might not obey the good variables properties, and might also cause execution to failstop, again based on a decision by the oracle. Accesses that would make the implementation halt due to a PvP violation or an unmapped page fault should correspond to source OOB failstops; those that the implementation deems harmless or "warps" into allocated locations can be allowed. At a minimum, for the semantics preservation theorems to hold, the oracle must refuse to overwrite any location that contains private compiler-generated data. But the source semantics doesn't know anything about private data, just that accesses outside allocated regions are unreliable.

Figure 2 illustrates some code that has memory-related UB in ISO standard C and in CompCert, but is well defined in our concrete semantics. Function `main` performs bit-level operations on the representation of `p`, "stealing" the low order 4 bits to hold a hash code; it relies on the result of `malloc` being 16-byte aligned, so the low order bits can be safely zeroed again before the pointer is used. Function `memmove` works even when source and target buffers overlap; it relies on making a comparison between `dest` and `src`. Function `bad` performs an out-of-bounds write; in our semantics, depending on where `p` and `q` live in the stack frame, the code will either failstop with OOB on the assignment to `p[6]`, or continue and print some number (0 or 1 in any reasonable instantiation of our memory model)—but it won't behave *arbitrarily*.

**Out of Memory.** Since actual machine memory is finite and pointers are represented as machine integers with limited range [28], any oracle instance will need to refuse some allocation requests due to stack overflow or heap exhaustion. Running out of memory causes execution to enter a *failstop* state. This is different from *getting stuck*, which corresponds to a UB for which the compiled code can do anything; failstopping is a well-defined behavior that must be preserved by compiled code. Thus a target will always run out of memory when the source does; this is essential for the preservation of safety properties (since otherwise the target could perform a bad behavior even though the source failstops).

To avoid polluting its axiomatization with implementation-specific details about memory

```
extern void f(int *p);
void g(int c) {
  int x[10], y[10], u, v, w;
  x[0] = 42;  y[0] = 99; u = x[0]; // optimize to u = 42 ?
  y[c] = 99;  v = x[0]; // optimize to v = 42 ?
  f(y); w = x[0]; // optimize to w = 42 ?
}
```

**Figure 3** Potential Redundant Load Optimizations

consumption, we allow the oracle to fail with OOM at any time. To gain confidence that our compiler correctness results are not vacuous, we have also proven reverse refinement from target to source—showing that the source runs out of memory *only* when the target does. We can confirm that the resulting allocation behavior is reasonable by inspecting the (completely concrete) target semantics and the generated target code.

**Optimization.** Using concrete semantics inevitably limits a compiler's optimization opportunities. Most significantly, any store to an unknown location and any function call potentially overwrites arbitrary public locations. (Private data, including ordinary scalars whose addresses are not taken, *are* preserved across unknown stores and function calls just as in ordinary C semantics.) This invalidates non-aliasing analyses, and so can prevent removal of some redundant public data loads and stores and also reduce the applicability of register promotion optimizations [25].

For example, in the code in Figure 3, all the suggested redundant load optimizations are valid in standard C (and performed by `gcc` and Clang): the first two because the stores to `y` can be assumed to be in bounds and the last because the address of `x` does not escape to `f`. But only the first is valid in our semantics, because in a concrete world, an out-of-bounds store to `y[c]` might indeed overwrite `x[0]`, and the unknown function `f` might overwrite *any* public location. The desire to maintain these optimizations while supporting more liberal pointer arithmetic has led to considerable work on hybrid models that allow some form of interoperation between block-based and concrete views [16, 15, 27]. Despite this, we are unaware of any empirical studies that assess how important alias-based optimizations actually are for real C workloads. We note that CompCert [23] doesn't perform the second or third optimizations in Figure 3 either, although they are valid under its memory model.

A second limit on optimization is a subtle consequence of working with finite memory while preserving refinement [19]. In our oracular approach, the source semantics locates each in-memory object at exactly the same address as in the target implementation. If the compiler were allowed to remove an allocation operation as part of dead code elimination, the oracle would have no idea where to put the object! Nor could it just announce OOM, since then the source would failstop whereas the target might continue, violating forward refinement. The upshot is that the compiler cannot optimize away dead allocations; this is unfortunate, but perhaps not too important in practice (e.g., CompCert doesn't currently perform such optimizations either).

Fortunately, many other optimizations involving allocations are still valid; in particular, CompCert-style function inlining and tail-call elimination should both be possible in our framework because they only adjust the locations of public stack allocations, not their total size. (Tail call elimination can also change the order of allocations and deallocations, but only for zero-sized blocks, which does not matter.)

## 4 Axiomatic Memory Model

The C memory model used by our concrete semantics gives an abstract characterization of *memories* that are modified and inspected using a set of *operator* functions. We now describe the model in detail, giving the signatures of the operators and a set of axioms specifying their behavior. The design goal for the axiomatization is to capture just those properties that are necessary for a C semantics and program logic to employ the memory model, while constraining possible instantiations as little as possible.

The model interface consists of an abstract type $\mathcal{M}$ of memories, with the following constants and operators:

| | | | | | |
|---|---|---|---|---|---|
| `initm` | :: | $\mathcal{M}$ | `hpFree` | :: | $\mathcal{M} \to \mathcal{A} \to \lfloor \mathcal{M} \rfloor$ |
| `stkAlloc` | :: | $\mathcal{M} \to \mathcal{L} \to \mathcal{S} \to \mathcal{G} \to \lfloor (\mathcal{R}, \mathcal{M}) \rfloor$ | `perturb` | :: | $\mathcal{M} \to \mathcal{L} \to \lfloor \mathcal{M} \rfloor$ |
| `stkFree` | :: | $\mathcal{M} \to \lfloor \mathcal{M} \rfloor$ | `load` | :: | $\mathcal{M} \to \mathcal{A} \to \lfloor \mathcal{V} \rfloor$ |
| `hpAlloc` | :: | $\mathcal{M} \to \mathcal{L} \to \mathcal{S} \to \mathcal{G} \to \lfloor (\mathcal{R}, \mathcal{M}) \rfloor$ | `store` | :: | $\mathcal{M} \to \mathcal{A} \to \mathcal{V} \to \lfloor \mathcal{M} \rfloor$ |

*Notation:* These functions are all partial, as indicated by the fact that they return option types, where we write $\lfloor T \rfloor$ for `(Option` $T$`)`, $\lfloor t \rfloor$ for `(Some` $t$`)` and $\emptyset$ for `None`. Memories $M \in \mathcal{M}$ are byte-indexed. Addresses $\mathcal{A}$ and sizes $\mathcal{S}$ are non-negative integers. Values $\mathcal{V}$ are machine bytes. Labels $\mathcal{L}$ are an arbitrary type, explained further below. Alignments $\mathcal{G}$ are positive integral powers of two. Regions $r \in \mathcal{R} = \mathcal{A} \times \mathcal{S}$ are half-open intervals with base $bs\ r$ and size $sz\ r$; we say $a$ is in the *footprint* of $r$, written $a \in r$, iff $bs\ r \leq a < bs\ r + sz\ r$.

The basic meaning of most of these operations should be clear from their names and type signatures. Memory is organized into *(allocated) regions*, which are intended to hold public data. Stack regions are intended for public data associated with function activations, allocated and freed as part of compiler-generated function call/entry and exit/return code. Heap regions are intended for explicitly allocated storage, allocated and freed by the C library `malloc` and `free` calls; they are also used for globals allocated at program start-up time (and never freed). Formally, the only difference between them is how a region to be freed is identified: for the stack, it is implicitly the most recently allocated region; for the heap, it is explicitly specified by a region base address.

Load and store are single-byte operations taking a concrete address. This byte-based interface can be used as a foundation on which a semantics can build a higher-level interface supporting reads and writes of multi-byte types, assuming suitable functions for encoding and decoding these in terms of bytes, and additional alignment checks where needed [22].

Figure 4 gives the full axiomatization of the memory operators. Although the precise layout of memory is deliberately kept abstract, the axioms use two observation functions that expose information about the allocated regions. $S_M$ is the stack of allocated stack regions in $M$, represented as a list in stack order (with most recently pushed region at the head). $H_M$ is the set of the allocated heap regions in $M$, represented as an (unordered) list with no duplicates. We write $A_M$ for $S_M$ ++ $H_M$, the full (multi)set of allocated regions in $M$.

*Further notation:* We write :: for list cons, [] for the empty list, $\in$ for list membership, ++ for list concatenation, and $L_1 \cong L_2$ if $L_1$ and $L_2$ are equal considered as multisets (i.e., are permutations of each other). $M'$ is *value-stable on* $M$, written $M \lessapprox M'$, iff $\forall a \in r \in A_M, \texttt{load}\ M\ a = \texttt{load}\ M'\ a$. We assume that machine addresses have $w$ bits.

We start with the basic invariants on memories that are maintained by all operations. RWF specifies basic well-formedness conditions on regions. It implies that:

**(i)** all addresses in a region (and the address immediately following) are representable in an unsigned machine word, allowing them to be computed using machine arithmetic;

$$\frac{r \in A_M}{0 < bs\ r \leq bs\ r + sz\ r < 2^w - 1}\ (\textsc{RWF}) \qquad \frac{A_M = rs_1 +\!\!+ r_1 :: rs_2 +\!\!+ r_2 :: rs_3}{\forall a.a \in r_1 \Rightarrow a \notin r_2}\ (\textsc{RDisj})$$

$$\frac{H_M = rs_1 +\!\!+ r_1 :: rs_2 +\!\!+ r_2 :: rs_3}{bs\ r_1 \neq bs\ r_2}\ (\textsc{HpRDist}) \qquad A_{\texttt{initm}} = []\ (\textsc{Init})$$

$$\frac{\texttt{stkAlloc}\ M\ lbl\ s\ g = \lfloor (r, M') \rfloor}{S_{M'} = r :: S_M \wedge H_{M'} \cong H_M}\ (\textsc{StkA}) \qquad \frac{\texttt{hpAlloc}\ M\ lbl\ s\ g = \lfloor (r, M') \rfloor}{H_{M'} \cong r :: H_M \wedge S_{M'} = S_M}\ (\textsc{HpA})$$

$$\frac{\texttt{stkAlloc}\ M\ lbl\ s\ g = \lfloor (r, M') \rfloor}{sz\ r \geq s \wedge (bs\ r)\ mod\ g = 0}\ (\textsc{StkAR}) \qquad \frac{\texttt{hpAlloc}\ M\ lbl\ s\ g = \lfloor (r, M') \rfloor}{sz\ r \geq s \wedge (bs\ r)\ mod\ g = 0}\ (\textsc{HpAR})$$

$$\frac{\texttt{stkAlloc}\ M\ lbl\ s\ g = \lfloor (r, M') \rfloor}{M \lessapprox M'}\ (\textsc{StkAV}) \qquad \frac{\texttt{hpAlloc}\ M\ lbl\ s\ g = \lfloor (r, M') \rfloor}{M \lessapprox M'}\ (\textsc{HpAV})$$

$$\frac{S_M \neq []}{\texttt{stkFree}\ M \neq \emptyset}\ (\textsc{StkFOK}) \qquad \frac{\texttt{hpFree}\ M\ a = \lfloor M' \rfloor}{\exists r \in H_M.bs\ r = a \wedge H_M \cong r :: H_{M'} \wedge S_{M'} = S_M}\ (\textsc{HpF})$$

$$\frac{\texttt{stkFree}\ M = \lfloor M' \rfloor}{\exists r.S_M = r :: S_{M'} \wedge H_{M'} \cong H_M}\ (\textsc{StkF}) \qquad \frac{r \in H_M}{\texttt{hpFree}\ M\ (bs\ r) \neq \emptyset}\ (\textsc{HpFOK})$$

$$\frac{\texttt{stkFree}\ M = \lfloor M' \rfloor}{M' \lessapprox M}\ (\textsc{StkFV}) \qquad \frac{\texttt{hpFree}\ M\ a = \lfloor M' \rfloor}{M' \lessapprox M}\ (\textsc{HpFV})$$

$$\frac{\texttt{perturb}\ M\ lbl = \lfloor M' \rfloor}{S_M = S'_M \wedge H_M \cong H'_M \wedge M \lessapprox M' \wedge M' \lessapprox M}\ (\textsc{Pert})$$

$$\frac{a \in r \in A_M}{\texttt{store}\ M\ a\ v \neq \emptyset}\ (\textsc{StOK}) \qquad \frac{a \in r \in A_M}{\texttt{load}\ M\ a \neq \emptyset}\ (\textsc{LdOK})$$

$$\frac{\texttt{store}\ M\ a\ v = \lfloor M' \rfloor}{\texttt{load}\ M'\ a = \lfloor v \rfloor}\ (\textsc{LdStEq}) \qquad \frac{\texttt{store}\ M\ a\ v = \lfloor M' \rfloor}{S_M = S'_M \wedge H_M \cong H'_M}\ (\textsc{StR})$$

$$\frac{a \in r \in A_M \qquad \texttt{store}\ M\ a\ v = \lfloor M' \rfloor \qquad a' \neq a}{\texttt{load}\ M'\ a' = \texttt{load}\ M\ a'}\ (\textsc{LdStNeq})$$

**Figure 4** Axiomatization of memory constants and operators

**(ii)** the C `NULL` pointer, which we assume to be 0 (as is natural on almost all current machines), does not point into any region;

**(iii)** regions may be empty (have size 0), which can help avoid special cases in a semantics.

RDisj states the basic property that no address can be within the footprint of two allocated regions, which is essential for reasoning about separation. Combined with RWF it further implies that the total amount of allocated memory is bounded. Since the address passed to `hpFree` should unambiguously identify a single region, we impose a further invariant HpRDist to guarantee this (and hence $H_M$ is indeed a set).

The heap and stack are initially empty (Init). A successful allocation pushes a new region onto the stack regions (StkA) or adds a new region to the heap (HpA), preserving the values in existing allocated regions (StkAV,HpAV). The new region is of at least the requested size and alignment (StkAR, HpAR). Standard `malloc` behavior can be obtained by using `hpAlloc` with the maximum required alignment for any primitive type (typically $w/4$, allowing the lower-order $\log_2 w - 2$ bits to be "stolen" for other uses).

Each allocation operation carries a *label* from some type $\mathcal{L}$, which is a parameter of the

entire model. Labels are contextual clues that an instantiation can use to place allocations at the same locations as an actual target implementation. For example, if public stack allocation occurs as part of function entry, the labels on `stkAlloc` might carry the name or definition of the function, giving an instantiation access to the size of the function's private data and hence enabling it to calculate the location of the public region. Crucially, labels have no impact at all on the axiomatized behavior of the interface. In this sense, they do not affect the source semantics; they just make it possible to prove compiler semantic preservation for a particular choice of labeling scheme, instantiation, and target implementation.

The axioms make no guarantees about when `stkAlloc` or `HpAlloc` will succeed; as discussed in §3, they may return $\emptyset$ at any time, to indicate OOM. On the other hand, `StkFree` is guaranteed to succeed iff the stack is non-empty (StkFOK, StkF) and `HpFree` succeeds iff the heap contains an allocated region at the specified address (HpFOK, HpOK). Again, these operations do not affect values in other allocated regions (StkFV,HpFV).

The `load` and `store` operations always succeed in allocated regions (StOK,LdOK). They *might* also succeed at unallocated addresses. It is important not to require failure on these accesses, because trapping them may be expensive or impossible in some implementations. The usual "good variables" properties hold for stores into an allocated region (LdStEq, LdStNeq). In fact, LdStEq holds even for addresses *outside* allocated regions, although this is not likely to be useful in practice. The restriction to allocated addresses for LdStNeq is essential to allow implementations like SFI masking that "warp" out-of-bounds stores into essentially unpredictable (but in-bounds) stores. Note that the axioms say nothing about initial values; that is, freshly allocated regions (and all unallocated addresses) start with defined but unpredictable contents.

Since all the operations (except `load`) return a potentially changed memory, they destroy any guarantees about the contents of *unallocated* locations, reflecting the fact that an implementation may change the layout or contents of private data at these points. But an implementation might also do this at other points that do not correspond to a source-level memory operation. For example, the stack pointer might change during function call/entry or exit/return sequences, even if no public data is allocated (or even accessed) at these points. To account for this, a source semantics can use the `perturb` pseudo-operation to signal places where such changes may occur in the target, without altering the status and contents of allocated data (Pert). A semantics should sprinkle `perturb` calls generously, as this allows a wider range of target implementations to support an instantiation of the model.

## 5 Verified Proof-of-Concept Compiler

We now describe the Coq implementation and verification of a proof-of-concept compiler and runtime based on the memory model. The implementation is coded in Coq's internal functional language, Gallina. Our source and target languages are highly simplified, while still containing enough features to exercise the memory model and to illustrate the interesting and challenging parts of the semantics preservation proof.

The formalisation of the memory model closely follows the design given in §4, except that memory is 64bit-word-indexed rather than byte-indexed, so values $\mathcal{V}$ are 64-bit words and pointers occupy a single location, and there are no alignments.

### 5.1 Source language

The source language (Figure 5, left) is a simple Register Transfer Language (RTL). A program is a list of named functions, each with an unbounded number of local pseudo-registers, jointly

| | | | |
|---|---|---|---|
| $i$ | := `mov` $r_s$ $r_d$ | move register | |
| | \| `movi` $n$ $r_d$ | move immediate | |
| | \| `mov&` $r_d$ | move array base | |
| | \| `op`$_\oplus$ $r_{s_1}$ $r_{s_2}$ $r_d$ | binary operation | |
| | \| `ld` $r_a$ $r_d$ | load | |
| | \| `st` $r_s$ $r_a$ | store | |
| | \| `call` $f$ $\vec{r}$ $r_d$ | call | |
| $\oplus$ | := `+` \| `-` | binary operator | |
| $f$ | := $f(\vec{r_a}, n) = \vec{i}$ `ret` $r_r$ | function | |
| $pr$ | := $\vec{f}$ | RTL program | |

| | | | |
|---|---|---|---|
| $r$ | := `SP` \| `RV` \| `RA` \| `GP1` \| `GP2` \| `GP3` | | |
| $i$ | := `mov` $r_s$ $r_d$ | move register | |
| | \| `movi` $n$ $r_d$ | move immediate | |
| | \| `op`$_\oplus$ $r_{s_1}$ $r_{s_2}$ $r_d$ | binary operation | |
| | \| `ld` $p$ $ofs(r_a)$ $r_d$ | load | |
| | \| `st` $p$ $t$ $r_s$ $ofs(r_a)$ | store | |
| | \| `jal` $id$ | jump-and-link | |
| | \| `builtin` $b$ | built-in | |
| $p$ | := `hi` \| `lo` | privilege | |
| $t$ | := `pro` \| `unpro` | protection tag | |
| $b$ | := `malloc` \| `free` | built-in | |
| $f$ | := $id : \vec{i}$ | function | |
| $pr$ | := $\vec{f}$ | Mach program | |

**Figure 5** Syntax for RTL (left) and Mach (right)

operating on a word-addressed memory. Registers and memory contain 64-bit machine words; there is absolutely no distinction between integers and pointers. Function call is an atomic operation passing all arguments at once, and the call stack is implicit. Since intrafunction control flow is unimportant for the compilation issues we wish to explore, we dispense with it altogether: function bodies are just straight-line instruction sequences.

There are two ways of allocating memory. For the **stack**, each function activation implicitly allocates a local array of statically fixed size. Just as with C local arrays or address-taken scalars, this storage can be used by the function itself and by callees who are given a pointer to it. For the **heap**, there are C-like built-in functions `malloc`:$words \to ptr$ and `free`:$ptr \to void$ that do explicit allocation and deallocation. The RTL semantics implements these features using calls to the corresponding memory model operations.

Function $f(\vec{r_a}, n) = \vec{i}$ `ret` $r_r$ takes its arguments in $\vec{r_a}$, allocates a local array of constant size $n$, executes the straight-line sequence of instructions $\vec{i}$, deallocates its local array, and returns the value in register $r_r$. Functions can use any number of additional local registers, which are implicitly initialized to 0. The `mov&` $r_d$ instruction moves the base address of the local array into $r_d$. Other instructions should be self-explanatory.

Programs can have one of four behaviors:

- Terminate with a result value. The first function in the program is taken to be the program's entry point and its final return value is the program's overall result.
- Failstop with a memory error. This can be either a failed allocation (`OOM`) or an unsuccessful access to an unallocated memory address (`OOB`).
- Diverge. Despite the lack of control-flow instructions, this can happen via an infinite recursion, provided the functions involved have zero-size local arrays (otherwise the program will eventually run out of memory).
- Get stuck. This can only happen if the program calls a function that does not exist or has the wrong number of arguments. It is easy to define a static typing judgement that rules out such programs.

The formal semantics of RTL is given by a straightforward instruction stepping relation over machine states. We include an explicit `Failstop` state. The state maintains an implicit call stack of pending activations, including return addresses and local registers. A single memory is threaded throughout, with memory model operations invoked at the following places in the semantics:

- Function entry: `stkAlloc` is invoked to allocate the local array, passing the current code location (function id and remaining instructions) as the context label. If the allocation succeeds, the array base is remembered in the state for future use by the `mov&` instruction. If it fails, the machine enters the `Failstop OOM` state.
- Function exit: `stkFree` is invoked to deallocate the local array. We can use the StkFOK axiom to prove that this can never fail in this semantics.
- Function call and return: `perturb` is invoked (again with the code location as label) to reflect the fact that the target implementation may change the stack to pass arguments or clean up after a call; if either operation fails, the machine enters `Failstop OOM`.
- Execution of `malloc` built-in: `hpAlloc` is called with the requested number of words. If this fails, the machine enters `Failstop OOM`.
- Execution of `free` built-in: `hpFree` is called with the specified address; if this fails, the address must be bogus, and the machine enters `Failstop OOB`.
- Load and store: these are done directly by the memory model's `load` and `store` operations; if they fail, the machine enters `Failstop OOB`. Recall that accesses to allocated locations are guaranteed to succeed, but out-of-bounds accesses will not necessarily fail.

Although the memory model abstracts over the behavior of the allocation oracle, for any given instantiation of the memory model, RTL itself is deterministic.

## 5.2 Target machine

The compiler's target language (Figure 5, right) is a simple RISC-like assembly code (Mach). The instruction set is very similar to RTL, but operates over a small fixed set of registers, including a stack pointer (`SP`), return address register (`RA`), return value register (`RV`), and general-purpose registers (`GP1-3`). Registers and memory can contain either 64-bit integers or abstract code pointers (actually just sequences of instructions). Machine code lives separately from data memory, and is divided into named functions, the first of which is the program's entry point `main`. Again, for simplicity, each function is a straight-line sequence of instructions, and there are no intrafunction branch instructions. Functions are invoked by `jal`, which stores the "return address" (the remaining instructions in the caller) in `RA`. After the last instruction in a function, the machine "returns" by continuing execution at the "address" in `RA`. `RV` is intended to hold the result value of a function when it returns.

The machine's memory is modeled as a single 64bit-word-addressed array. In addition to a containing a value (the *payload*), each address has an associated single-bit *protection* tag, either `pro`*tected* or `unpro`*tected*. Store operations set both the payload and the tag. Load and store operations are parameterized by a corresponding *privilege* flag: `hi` or `lo`. Low-privilege instructions can only access unprotected locations; high-privilege instructions can access all locations. Protection violations cause a failstop. This corresponds to a simplified version of ARM MTE [34, 3] or PIPE [13]. The initial values in memory are defined but arbitrary and are all tagged as `unpro`. The intention is that private data such as return addresses, arguments, and variables will be tagged `pro` and accessed with `hi` instructions, whereas public data such as the per-function arrays will be tagged `unpro` and accessed with `lo` instructions.

The overall structure of memory is specified by a small set of parameters defining the bases and limits of a *stack area* and a *heap area*, organized as shown in Figure 6. All loads and stores are constrained to lie within one of these areas, which correspond to the pages mapped by a process in a real machine. Accesses outside these areas cause failstops. The machine initializes `SP` to `StkBase`; thereafter, the stack is managed entirely by code generated by the compiler. The heap is managed by the `malloc` and `free` functions that, in a real system, would be part of the runtime system code. For simplicity (and because our target

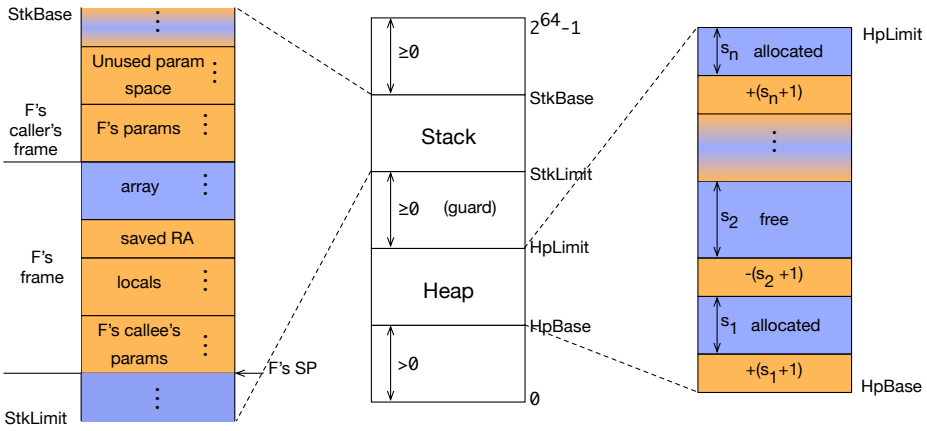


**Figure 6** Layout of memory (center), stack frames (left), and heap (right); orange regions are `pro` and blue regions are `unpro`.

language is so impoverished) here we write these functions in Gallina instead and invoke them via special built-in machine instructions; see §5.4.

Mach programs are deterministic, and can have the same four kinds of behaviors as RTL programs. Termination is signaled by returning from `main`; the overall program result value is in `RV`. Failstops can be due either to memory errors (protection violation or out-of-bounds access) or a failure by a built-in function (out of memory in `malloc` or bad argument to `free`). Although real machines don't get stuck, this one can, by trying to jump to a function that doesn't exist or return when `RA` does not contain a code pointer; these are artifacts of our abstract view of code that would disappear if we used concrete code addresses.

## 5.3 Compilation scheme and stack layout

The compiler from RTL to Mach has a very simple structure: each function is translated independently, and each source instruction is translated to a fixed sequence of target instructions. The target code maintains a conventional stack of function activation frames, growing towards lower addresses. A function's frame (Figure 6, left) holds its parameters and local variables, its return address, and its local array; thus private and public data are interleaved. There is no attempt to perform register allocation; instead, each RTL parameter and pseudo-register is mapped to a fixed frame offset and loaded/stored each time it is used. We adopt a calling convention where the entire responsibility for constructing and freeing stack frames is given to the callee. The size of the frame is computed statically and the `SP` is adjusted just once at function entry and again at function exit. We view parameters to any callees as part of the *caller*'s frame, which must include enough space to hold the maximum number of parameters needed by any call the function makes, computed by a simple pass over the function body. Since the sizes of all frame components are known statically, they can be accessed as offsets from the `SP`; there is no need for a separate frame pointer.

Initially, the entire memory is tagged `unpro`, but all components of stack frames except the local array should be tagged `pro`. The compiler generates function entry code to do this one word at a time, and matching exit code that retags everything `unpro`. Setting a tag also requires writing a payload value; both these operations write zeros. The dual approach of making `pro` be the default and unprotecting/reprotecting the array component would also work, but could be much less efficient, since arrays can be arbitrarily large. To protect

non-stack memory, target code must failstop if the stack overflows its predefined area, i.e. if `SP` goes below `StkLimit`. We achieve this via the usual trick of placing a "guard" area of invalid addresses between the stack and the heap.

## 5.4 Runtime allocator and heap layout

As described in §5.2, the target machine is equipped with a runtime heap memory allocator packaged as C-like `malloc` and `free` functions. For simplicity and convenience, we build this code into the machine's semantics, but in a real implementation it would be part of the runtime system; our Gallina code is "honest" in the sense that it uses only memory operations already provided by the machine's instruction set. We use a very simple heap organization (Figure 6, right). The heap region is divided into objects, each consisting of a one word header followed by zero or more words of data. The header is an integer whose absolute value is the size of the object (including the header itself), and whose sign indicates whether the object is currently allocated (sign +1) or free (sign -1). Headers are tagged `pro`; all other head addresses are tagged `unpro`. Initially, the heap consists of a single free object.

To allocate a new heap region of size $s \geq 0$, `malloc` loops over the objects in increasing address order, starting at `HpBase`, until it finds a free object of size $n \geq s$. It returns that object's address (the first word above the header), after using a privileged write to flip the sign bit in the header to mark the object as allocated. If $n > s$, the object is first split into two objects, which involves changing the size in the header of the existing object and transforming a data word into a header for the second object, again using privileged writes. If no sufficiently large object is found before the loop reaches `HpLimit`, `malloc` failstops.

To deallocate a heap region at a specified address, `free` must operate similarly: it loops over the objects in order starting at `HpBase` until it finds an allocated object at that address, and then uses a privileged write to flip the sign bit in that object's header, marking it as free. The newly freed object is then coalesced with its neighbors on each side if they were already free. If no matching address is found, the specified address was invalid, so `free` failstops.

Obviously this scheme (especially for `free`) has poor efficiency properties compared to real C memory manager implementations. But it does illustrate a realistic interleaving of (public) data and (private) allocator metadata in memory, and shows how a tag-based machine can protect the metadata, while keeping proofs fairly simple. Extending the algorithms to manage multiple free lists, etc. would be straightforward but tedious. Implementing a more realistic $O(1)$ time `free` safely could be done by storing and protecting metadata separately.

## 5.5 Memory model instantiation

We now describe how the memory model (§4) can be instantiated so that the addresses it assigns to stack and heap allocations requested by the source semantics (§5.1) match those generated by compiled code (§5.3) and the runtime heap allocator (§5.4). This is the most novel aspect of our approach. In essence, we work backwards from the compiler and runtime design to identify (just) those aspects of target state that affect public memory layout, and use these to define instantiations of the $\mathcal{M}$ type and the model's operations and observation functions that validate the model's axioms. The semantics preservation proof (§5.6) confirms that we have done so correctly for this source and target.

The construction of this instantiation is parameterized by an RTL source program, which is consulted to obtain the definitions of functions passed as labels to `StkAlloc`, from which the oracle can calculate private stack data sizes. The instantiation itself is then passed as a parameter to the definition of the RTL semantics. For the proof-of-concept system, we define

$\mathcal{M} = \{m : (vals : Vals) \times (st : St) \times (hp : Hp) \mid MInv\ m\}$, i.e. a Coq subset type containing a triple of value map *vals*, stack *st*, and heap *hp*, which obeys an invariant *MInv*. We discuss each component in turn.

**Values.** To support the model's `load` and `store` operations, the instantiation simply maintains a map *vals* of type $Vals = \mathcal{A} \to \mathcal{V}$ that contains the correct values for all public (`unpro`) locations, and is arbitrary at other locations.

**Stack.** The source semantics uses stack allocation operations only for the local array created as part of each function entry sequence; the model instantiation must predict the runtime location of this array. To do this, the instantiation maintains an abstract stack $st : St$ of function activations kept in one-to-one correspondence with the frames of the concrete implementation stack. Here *St* is lists of abstract *frames*; each invocation of `stkAlloc` with size $s$ and label $f$ pushes a new frame $(f, s)$ onto *st*. The instantiation uses the function definition for $f$ and knowledge of how the compiler lays out concrete frames (Figure 6) to compute the array address to return (or $\emptyset$ if this address is below `StkLimit`).

Each `stkFree` pops the top frame from *st*. Since the implementation retags private frame locations as `unpro` during function exit, resetting their payload values to 0, the instantiation must also zero the corresponding locations in *vals*.

In this particular implementation all the work associated with building and tearing down frames is done in the callee. Although the semantics invokes `perturb` operations on the caller side before and after each call, nothing happens to the concrete stack at these points, so `perturb` $M\ l$ always returns $\lfloor M \rfloor$.

The observation function $S_M$ is obtained simply by mapping over *st* and extracting the array base and size from each frame. The instantiation also defines a predicate identifying accessible (i.e. public) addresses in the stack area, which includes both addresses in allocated regions (i.e. local arrays) and those beyond the end of the stack pointer; `load` and `store` on inaccessible stack area addresses return $\emptyset$.

**Heap.** RTL's invocations of `hpAlloc` and `hpFree` are in one-to-one correspondence with calls to the "runtime system" primitives `malloc` and `free`. Hence, the model instantiation simply maintains a slightly abstracted version $hp : Hp$ of the runtime's heap data structure and executes abstracted versions of the runtime algorithms over it. *Hp* is lists of *objects* maintained in increasing address order, with the first object being understood to start at `HpBase`. Each object is described as a pair $(s, af)$ where $s$ is the object size and $af$ is a boolean is-allocated flag. The heap accessibility predicate holds for addresses within both allocated and unallocated objects, but not the private header words. When the implementation of `free` performs coalescing, it will retag some private headers as `unpro` and reset their payload values to 0, so the model instantiation must also zero the corresponding locations in *vals*.

**Invariant.** By design, the type of $\mathcal{M}$ allows very little "junk," but we do need to carry two simple technical invariants in *MInv*: for *st*, the calculated `SP` must always lie within the stack area; for *hp*, the total size of the objects (including headers) always equals `HpLimit - HpBase`. Among other things, these let us prove that all accessible addresses are representable in a machine word (axiom RWF).

## 5.6 Semantic preservation

We say a pair of RTL and Mach behaviors are *equivalent* (written $\approx$) if they both Terminate with the same value, both Diverge, both get Stuck, or both Failstop (for *any* reason). Our notion of semantic preservation is two-way refinement: if an RTL program does not get Stuck, then each of its behaviors is equivalent to a behavior of the corresponding Mach program, and vice-versa. Since both Mach and RTL (with a specific memory model instantiation) are deterministic, this simplifies to the following strong result. For any RTL program $P$, let $\mathcal{I}(P)$ be the memory model instantiation defined as in §5.5, applied to $P$. Then we have:

▶ **Theorem 1.** ***(Equivalent Behavior)***
*Let $S$ be an RTL source program and $T$ be the corresponding compiled Mach target program. Let $\mathcal{B}_{RTL}[I](S)$ be the behavior of $S$ under the RTL semantics with memory model instantiation $I$, and $\mathcal{B}_{Mach}(T)$ be the behavior of $T$ under the Mach semantics. Finally, let $B_{RTL} = \mathcal{B}_{RTL}[\mathcal{I}(S)](S)$ and $B_{Mach} = \mathcal{B}_{Mach}(T)$. Then, if $B_{RTL}$ is not Stuck, $B_{Mach} \approx B_{RTL}$.*

As noted in §5.1, we can easily define a static typing judgement on RTL programs that rules out Stuck behavior. Then we have a corollary stating that if $S$ is well-typed, $B_{Mach} \approx B_{RTL}$. The proof of this theorem is based on a standard stepwise forward simulation lemma showing that each RTL step corresponds to one or more Mach steps while preserving a matching relation between states. Iterating this lemma directly gives forward refinement; coupled with determinacy of the target semantics it also implies reverse refinement [21].

As usual, the main challenge of the proof lies in defining the matching relation by cases over the possible RTL states, which all include the memory model's internal state $M$ as one component. To describe Mach memory, we use a version of CompCert's separation logic library, modified to work for tagged concrete memory. A typical separation logic assertion is the following characterization of the Mach memory corresponding to a single stack frame, as the separated conjunction of three simpler assertions:

```
Definition frame_contents (f:RTL.function) (sp:Z) (retaddr:mval)
                          (sB:RTL.regbank) (vm:MemInst.valmem) : massert :=
      match_env f sB sp ** hasvalue (sp + ra_ofs f) retaddr t_pro
      ** match_pub vm (sp + arr_ofs f) (sp + arr_ofs f + sz_a f).
```

Roughly, this says that a frame based at Mach memory address `sp` contains separate slots for the private values of parameter and local registers of the RTL function (`match_env`), the private saved return address (`hasvalue`), and the public array contents stored in $M$'s *vals* component (`match_pub`). More complex assertions are used to characterize the memory of the stack as a whole. In addition to matching memory contents, we must maintain a *three-way* relation among the RTL call stack, the *st* component of $M$, and the Mach `SP`, in order to guarantee that source and target share the same notion of accessible stack memory.

Heap matching is easier, because RTL delegates all knowledge of heap structure to $M$. The following definition characterizes the heap segment `h` starting at Mach memory base address `b`; note the encoding of the abstract allocation flag `af` into the sign of the size in the header.

```
Fixpoint heap_contents (h:MemInst.heap) (b:Z) (vm:MemInst.valmem) : massert :=
  match h with
  | [] => pure True
  | (sz,af)::rest => hasvalue b (if af then sz+1 else -(sz+1)) t_pro
                  ** match_pub vm (b+1) (b+1+sz)
                  ** heap_contents rest (b+1+sz) vm
  end.
```

The overall matching relation combines stack matching, heap matching, and a rather large number of purely technical invariants. The simulation proof itself is quite lengthy, but fairly straightforward. We develop a series of lemmas to show that whenever source and target are in matching states, each publicly accessible address in $M$ points to an `unpro` location in Mach memory with the same value, and each inaccessible address points to a `pro` Mach address. These are then used to prove that private (resp. public) stores in RTL can be simulated by privileged (resp. unprivileged) stores in Mach, preserving state matching; furthermore, failing stores in RTL can be simulated by failing stores in Mach. We prove auxiliary lemmas about fixed sequences of code generated during compilation for function call, entry and exit. Similarly, we prove that the low-level Mach implementations of `malloc` and `free` correctly simulate those in the $M$ implementation.

The overall structure of our development is inspired by CompCert, and we make direct use of the following CompCert modules: `Separation` (modified as noted above), `Integers` (for machine integers), `Smallstep` and `Behaviors` (both modified to remove traces and add Failstop states), and various low-level libraries.

## 6 Related Work

**Memory models.** Norrish [31] gives a mechanized C semantics with a concrete byte-level memory, which was later refined by Tuch, et al. [39, 38] to incorporate type-based non-aliasing. The main focus of this line of work is to support verification of C programs, rather than of C compilers. Memarian's Cerberus system [26] includes an (unaxiomatized) C memory model interface that abstracts over various alternatives for pointer semantics, some fairly concrete.

Several variants of the original CompCert memory model [22] treat memory and pointer representations more concretely. CompCertTSO [35], which extends CompCert v1.5 to target a low-level machine with a TSO relaxed memory model, switches from abstract to concrete pointers early in the compilation pipeline. Since CompCert v1.5 lacked alias analysis, the impact of concrete addressing on optimization does not seem to have been considered. Mullen et al. [28] append a new peephole optimization pass to the CompCert pipeline that uses a low-level version of assembly code in which pointers have been mapped to concrete integers. Our concrete semantics should validate all of their transformations. CompCertMC [41] extends the CompCert proof chain to a machine language with a flat, concrete memory model, similar to our Mach language. None of these systems expose concrete pointers at source level, or attempt to give meaning to OOB behaviors.

**Oracles.** Carbonneaux et al. [8] introduce the idea of a memory oracle that collects information from the compiler about the size of each target stack frame, and reflects that back to the source. CompcertMC [41] and CompCertS [6, 5] employ such an oracle to derive a bound on total stack size from inspection of the source program. Our oracle is much more detailed, as it says *where* each allocation goes, not just how big it is; also, unlike these systems, we describe heap allocation in detail. On the other hand, we deliberately avoid exposing private data size information in the source semantics itself, so we cannot support bounds calculations there. Another difference is that these systems treat source level OOM as a stuck behavior for which the compiler makes no guarantees, whereas we treat OOM as a distinct behavior that is verifiably preserved by the compiler.

**Low-level pointer arithmetic.** Kang et al. [15] propose a hybrid *quasi-concrete* memory model in which abstract pointers acquire concrete addresses only when they are cast to

integers. The intent is to support both arithmetic operations on cast values and aggressive optimization when casting has not occurred. As in our semantics, OOM becomes an observable behavior of the source program, which is guaranteed to be preserved by the target; one unintuitive feature is that these OOMs occur at cast time rather than at allocation time. OOB behaviors remain UB, hence not preserved.

Besson et al. [6, 5] propose a memory model that keeps CompCert's abstract representation of pointers, but also supports certain bit-level arithmetic computations. Their CompCertS semantics builds a symbolic representation of each computed pointer value in terms of abstract block addresses; this has well-defined semantics iff its value is invariant under all possible legal block placements. For example, CompCertS gives the expected semantics to function `main` of Figure 2, since the value of `r` does not depend on where `p` lives, but not to `memmove`, since the result of comparing `dest` and `src` depends on the relative placements of the corresponding blocks. Our semantics gives defined semantics to both functions.

In a separate line of work, Besson et al. [7] show how to modify CompCert to turn the bit-level pointer operations needed for SFI memory address "warping" into well-defined behaviors that are preserved by compilation, by adding a compiler pass that "arithmetises" all pointers into integer offsets within a single sandbox block, referenced by a shadow stack pointer [17]; the behavior of globals and `malloc` is axiomatized to use the sandbox as well. They prove in Coq that transformed programs are well-defined and safe. They do not propose a concrete pointer semantics at source level; indeed, they do not attempt to prove that their new pass preserves source behavior (which would require adding a notion of OOM at source level). Their transformation inhibits later optimizations much like our semantics; their benchmarks suggest that this can have noticeable, though inconsistent, effects on performance.

## 7 Conclusions and Future Work

We have proposed a concrete memory semantics for C, with no memory UBs, suitable for compiling to a target having public vs. private memory enforcement, and equipped with a novel oracle that predicts the concrete placement of memory allocations based on the actual behavior of the compiler and runtime system. Our proof-of-concept verified compiler demonstrates the feasibility of this approach for a very simple, but characteristic, subset of C and a tag-based enforcement mechanism.

Our next work is to extend this subset by incorporating function pointers, which can be protected in the target environment using a range of hardware and software tagging or trampolining techniques [7] analogous to those used for data pointers. Once again, it is desirable to abstract away the details of enforcement when specifying the source semantics; we believe this can be done in a style similar to the oracular memory model we use here.

A significant question is whether our oracular approach supports vertical compositionality across a multi-phase compiler. It is particularly desirable that oracles can be constructed modularly, i.e. that an oracle instantiation for each stage can be constructed from an *arbitrary* instantiation of the successor stage's oracle. We have some preliminary work on an inlining phase suggesting that this is indeed possible, but much more experience is needed. Ultimately, we hope to use this approach to build a complete *Concrete C* compiler with a UB-free source semantics and the full scope and depth of CompCert.

## References


1 Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Catalin Hritcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *2015 IEEE Symposium on Security and Privacy*, pages 813–830, 2015. doi:10.1109/SP.2015.55.

2 Sean Anderson, Allison Naaktgeboren, and Andrew Tolmach. Flexible runtime security enforcement with tagged C. In Panagiotis Katsaros and Laura Nenzi, editors, *Runtime Verification - 23rd International Conference, RV 2023, Thessaloniki, Greece, October 3-6, 2023, Proceedings*, volume 14245 of *Lecture Notes in Computer Science*, pages 231–250. Springer, 2023. doi:10.1007/978-3-031-44267-4_12.

3 ARM. Armv8.5-a memory tagging extension white paper. URL: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

4 Daniel J. Bernstein. boringcc. Boring crypto google group post, 2015. URL: https://groups.google.com/g/boring-crypto/c/48qa1kWignU/m/o8GGp2K1DAAJ.

5 Frédéric Besson, Sandrine Blazy, and Pierre Wilke. CompCertS: A Memory-Aware Verified C Compiler using a Pointer as Integer Semantics. *Journal of Automated Reasoning*, 63(2):369–392, August 2019. URL: https://inria.hal.science/hal-02401182, doi:10.1007/s10817-018-9496-y.

6 Frédéric Besson, Sandrine Blazy, and Pierre Wilke. A verified compcert front-end for a memory model supporting pointer arithmetic and uninitialised data. *J. Autom. Reasoning*, 62(4):433–480, 2019. doi:10.1007/s10817-017-9439-z.

7 Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas P. Jensen, and Pierre Wilke. Compiling sandboxes: Formally verified software fault isolation. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 499–524. Springer, 2019. doi:10.1007/978-3-030-17184-1_18.

8 Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-end verification of stack-space bounds for C programs. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 270–281. ACM, 2014. doi:10.1145/2594291.2594301.

9 CHR Chhak, Andrew Tolmach, and Sean Anderson. Towards formally verified compilation of tag-based policy enforcement. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, pages 137–151, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3437992.3439929.

10 Coq Development Team. *The Coq Reference Manual, version 8.18.0*, September 2023. URL: https://coq.inria.fr/doc/V8.18.0/refman/.

11 Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, page 5, USA, 1998. USENIX Association. URL: https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention.

12 Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. HardBound: Architectural support for spatial safety of the C programming language. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 103–114, 2008. URL: http://acg.cis.upenn.edu/papers/asplos08_hardbound.pdf.

13 Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and Andre DeHon. Architectural support

for software-defined metadata processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 487–502, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2694344.2694383, doi:10.1145/2694344.2694383.

14 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*, pages 185–200, 2017. URL: https://www.cs.tufts.edu/~nr/cs257/archive/andreas-rossberg/webassembly.pdf, doi:10.1145/3062341.3062363.

15 Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A formal C memory model supporting integer-pointer casts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 326–335, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2737924.2738005.

16 Robbert Krebbers. *The C Standard Formalized in Coq*. PhD thesis, Radboud University Nijmegen, December 2015. URL: http://robbertkrebbers.nl/research/thesis.pdf.

17 Joshua Kroll, Gordon Stewart, and Andrew Appel. Portable software fault isolation. In *27th IEEE Computer Security Foundations Symposium*. IEEE, 2014. URL: http://www.cs.princeton.edu/~appel/papers/psfi.pdf.

18 Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 147–163. USENIX Association, 2014. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov.

19 Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. Reconciling high-level optimizations and low-level code in LLVM. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276495.

20 Xavier Leroy. A Formally Verified Compiler Back-End. *J. Autom. Reason.*, 43(4):363–446, December 2009. doi:10.1007/s10817-009-9155-4.

21 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. URL: https://xavierleroy.org/publi/compcert-CACM.pdf, doi:10.1145/1538788.1538814.

22 Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008. URL: http://xavierleroy.org/publi/memory-model-journal.pdf.

23 Xavier Leroy et al. Compcert 3.10. URL: https://github.com/AbsInt/CompCert/releases/tag/v3.10.

24 Hans Liljestrand, Carlos Chinea Perez, Rémi Denis-Courmont, Jan-Erik Ekberg, and N. Asokan. Color my world: Deterministic tagging for memory safety. *CoRR*, abs/2204.03781, 2022. doi:10.48550/arXiv.2204.03781.

25 John Lu and Keith D. Cooper. Register promotion in c programs. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, page 308–319, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/258915.258943.

26 Kayvan Memarian. *The Cerberus C Semantics*. PhD thesis, University of Cambridge, 2023. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-981.pdf.

27 Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C semantics and pointer provenance. *Proceedings of the ACM on Programming Languages*, 3, 2019. URL: https://dl.acm.org/citation.cfm?id=3290380.

28 Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. Verified peephole optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*, pages 448–461, 2016. URL: http://peek.uwplse.org/pldi16.pdf, doi:10.1145/2908080.2908109.

29 Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 245–258. ACM, 2009. URL: http://repository.upenn.edu/cgi/viewcontent.cgi?article=1941&context=cis_reports.

30 Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler enforced temporal safety for C. *SIGPLAN Not.*, 45(8):31–40, June 2010. URL: http://doi.acm.org/10.1145/1837855.1806657, doi:10.1145/1837855.1806657.

31 Michael Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-453.pdf.

32 Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51(6), February 2019. doi:10.1145/3280984.

33 Alexander L. Richardson. *Complete Spatial Safety for C and C++ using CHERI capabilities*. PhD thesis, University of Cambridge, October 2019. URL: https://www.repository.cam.ac.uk/bitstream/handle/1810/307454/thesis-hardbound.pdf.

34 Kostya Serebryany. ARM memory tagging extension and how it improves C/C++ memory safety. *login Usenix Mag.*, 44(2), 2019. URL: https://www.usenix.org/publications/login/summer2019/serebryany.

35 Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *Journal of the ACM*, 60(3):22, 2013. URL: http://doi.acm.org/10.1145/2487241.2487248, doi:10.1145/2487241.2487248.

36 Laurent Simon, David Chisnall, and Ross J. Anderson. What you get is what you C: Controlling side effects in mainstream C compilers. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–15. IEEE, 2018. URL: https://www.cl.cam.ac.uk/~rja14/Papers/whatyouc.pdf, doi:10.1109/EuroSP.2018.00009.

37 Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, pages 48–62. IEEE Computer Society, 2013. URL: http://lenx.100871.net/papers/War-oakland-CR.pdf, doi:10.1109/SP.2013.13.

38 Harvey Tuch. Formal verification of C systems code. *J. Autom. Reasoning*, 42:125–187, 04 2009. doi:10.1007/s10817-009-9120-2.

39 Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. *SIGPLAN Not.*, 42(1):97–108, January 2007. doi:10.1145/1190215.1190234.

40 Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 203–216, 1993. URL: http://www.eecs.harvard.edu/~greg/cs255sp2004/wahbe93efficient.pdf.

41 Yuting Wang, Pierre Wilke, and Zhong Shao. An abstract stack based approach to verified compositional compilation to machine code. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290375.

42 Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Proceedings - IEEE Symposium on Security and Privacy*, 2015. doi:10.1109/SP.2015.9.

43 Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. Silent bugs matter: A study of compiler-introduced security bugs. In *32nd*

*USENIX Security Symposium (USENIX Security 23)*, pages 3655–3672, Anaheim, CA, August 2023. USENIX Association. URL: `https://www.usenix.org/conference/usenixsecurity23/presentation/xu-jianhao`.

44 Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010. URL: `http://research.google.com/pubs/archive/34913.pdf`, `doi:10.1145/1629175.1629203`.