Exploring C Semantics and Pointer Provenance

KAYVAN MEMARIAN, University of Cambridge, UK
VICTOR B. F. GOMES, University of Cambridge, UK
BROOKS DAVIS, SRI International, USA
STEPHEN KELL, University of Cambridge, UK
ALEXANDER RICHARDSON, University of Cambridge, UK
ROBERT N. M. WATSON, University of Cambridge, UK
PETER SEWELL, University of Cambridge, UK

The semantics of pointers and memory objects in C has been a vexed question for many years. C values cannot be treated as either purely abstract or purely concrete entities: the language exposes their representations, but compiler optimisations rely on analyses that reason about provenance and initialisation status, not just runtime representations. The ISO WG14 standard leaves much of this unclear, and in some respects differs with de facto standard usage — which itself is difficult to investigate.

In this paper we explore the possible source-language semantics for memory objects and pointers, in ISO C and in C as it is used and implemented in practice, focussing especially on pointer provenance. We aim to, as far as possible, reconcile the ISO C standard, mainstream compiler behaviour, and the semantics relied on by the corpus of existing C code. We present two coherent proposals, tracking provenance via integers and not; both address many design questions. We highlight some pros and cons and open questions, and illustrate the discussion with a library of test cases. We make our semantics executable as a test oracle, integrating it with the Cerberus semantics for much of the rest of C, which we have made substantially more complete and robust, and equipped with a web-interface GUI. This allows us to experimentally assess our proposals on those test cases. To assess their viability with respect to larger bodies of C code, we analyse the changes required and the resulting behaviour for a port of FreeBSD to CHERI, a research architecture supporting hardware capabilities, which (roughly speaking) traps on the memory safety violations which our proposals deem undefined behaviour. We also develop a new runtime instrumentation tool to detect possible provenance violations in normal C code, and apply it to some of the SPEC benchmarks. We compare our proposal with a source-language variant of the twin-allocation LLVM semantics proposal of Lee et al. Finally, we describe ongoing interactions with WG14, exploring how our proposals could be incorporated into the ISO standard.

CCS Concepts: • **Theory of computation → Operational semantics**; • **Software and its engineering → General programming languages**;

Additional Key Words and Phrases: C

ACM Reference Format:
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

1 INTRODUCTION

The semantics of pointers and memory objects in C has been a vexed question for many years. A priori, one might imagine two language-design extremes: a concrete model that exposes the memory semantics of the underlying hardware, with memory being simply a finite partial map from

2475-1421/2019/1-ART67
https://doi.org/10.1145/3290380

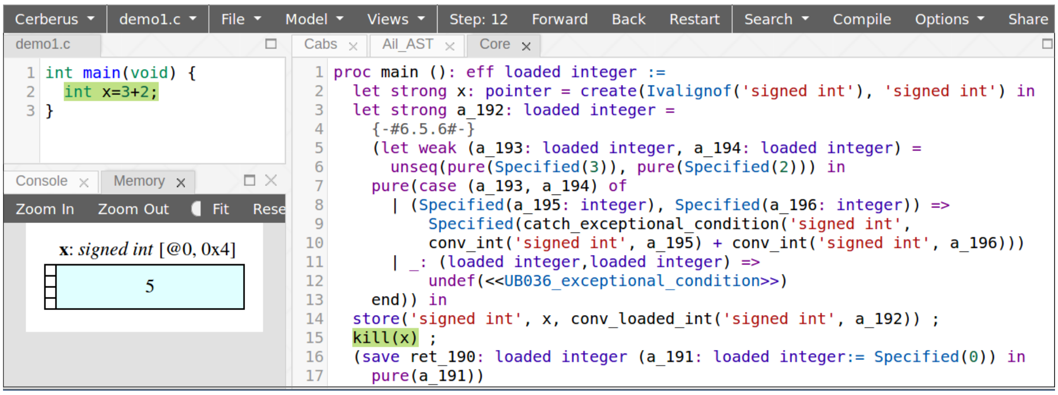
machine-word addresses to bytes, and an abstract model in which the language types enforce hard distinctions, e.g. between numeric types that support arithmetic and pointer types that support dereferencing. C is neither of these. Its values are not abstract: the language intentionally permits manipulation of their underlying representations, via casts between pointer and integer types, `char*` pointers to access representation bytes, and so on, to support low-level systems programming. But C values also cannot be considered to be simple concrete values: at runtime a C pointer will typically just be a machine word, but compiler analysis reasons about abstract notions of the provenance of pointers, and compiler optimisations rely on assumptions about these for soundness.

To understand exactly what is and is not allowed, as a C programmer, compiler or analysis tool writer, or semanticist, one might turn to the ISO language standard produced by WG14 [2011a]. However, that suffers from three problems, especially for these pointer and memory object issues. First, while in many respects the ISO standard is clear, in some it is not. Particularly relevant here, some compiler optimisations rely on alias analysis to deduce that two pointer values do not refer to the same object, which in turn relies on assumptions that the program only constructs pointer values in "reasonable" ways (with other programs regarded as having undefined behaviour, UB). The committee response to Defect Report DR260 [WG14 2004] states that implementations can track the origins (or "provenance") of pointer values, but exactly what this means is left undefined, and it has never been incorporated into the standard text. Even what a memory object is is not completely clear in the standard, especially for aggregate types and within heap regions.

Second, in some respects there are significant discrepancies between the ISO standard and the de facto standards, of C as it is implemented and used in practice. Major C codebases typically rely on particular compiler flags, e.g. `-fno-strict-aliasing` or `-fwrapv`, that substantially affect the semantics but which standard does not attempt to describe, and some idioms are UB in ISO C but widely relied on in practice. There is also not a unique de facto standard: in reality, one has to consider the expectations of expert C programmers and compiler writers, the behaviours of specific compilers, and the assumptions about the language implementations that the global C codebase relies upon to work correctly (in so far as it does). Our recent surveys [Memarian et al. 2016; Memarian and Sewell 2016b] of the first revealed many discrepancies, with widely conflicting responses to specific questions.

Third, the ISO standard is a prose document, as is typical for industry standards. The lack of mathematical precision, while also typical for industry standards, has surely contributed to the accumulated confusion about C, but, perhaps more importantly, the prose standard is not *executable as a test oracle*. One would like, given small test programs, to be able to automatically compute the sets of their allowed behaviours (including whether they have UB). Instead, one has to do painstaking argument with respect to the text and concepts of the standard, a time-consuming and error-prone task that requires great expertise, and which will sometimes run up against the areas where the standard is unclear or differs with practice. One also cannot use conventional implementations to find the sets of all allowed behaviours, as (a) the standard is a loose specification, while particular compilations will resolve many nondeterministic choices, and (b) conventional implementations cannot detect all sources of undefined behaviour (that is the main point of UB in the standard, to let implementations assume that source programs do not exhibit UB, together with supporting implementation variation beyond the UB boundary). Sanitisers and other tools can detect some UB cases, but not all, and each tool builds in its own more-or-less ad hoc C semantics.

This is not just an academic problem: disagreements over exactly what is or should be permitted in C have caused considerable tensions, e.g. between OS kernel and compiler developers, as increasingly aggressive optimisations can break code that worked on earlier compiler implementations.



```
int main(void) {
  int x=3+2;
}
```

```
proc main (): eff loaded integer :=
  let strong x: pointer = create(Ivalignof('signed int'), 'signed int') in
  let strong a_192: loaded integer =
    {-#6.5.6#-}
    (let weak (a_193: loaded integer, a_194: loaded integer) =
      unseq(pure(Specified(3)), pure(Specified(2))) in
    pure(case (a_193, a_194) of
      | (Specified(a_195: integer), Specified(a_196: integer)) =>
          Specified(catch_exceptional_condition('signed int',
          conv_int('signed int', a_195) + conv_int('signed int', a_196)))
      | _: (loaded integer,loaded integer) =>
          undef(<<UB036_exceptional_condition>>)
    end)) in
  store('signed int', x, conv_loaded_int('signed int', a_192)) ;
  kill(x) ;
  (save ret_190: loaded integer (a_191: loaded integer:= Specified(0)) in
    pure(a_191))
```

Fig. 1. An example C program, its elaboration into Core, and the memory graph during execution

Our first main contribution (§2–6) is an exploration of the design space and two candidate semantics for pointers and memory objects in C, taking both ISO and de facto C into account. We earlier [Chisnall et al. 2016; Memarian et al. 2016] identified many design questions. We focus here on the questions concerning pointer provenance, which we revise and extend. We develop two main coherent proposals that reconcile many design concerns; both are broadly consistent with the provenance intuitions of practitioners and ISO DR260, while still reasonably simple. We highlight their pros and cons and various outstanding open questions. These proposals cover many of the interactions between abstract and concrete views in C: casts between pointers and integers, access to the byte representations of values, etc. We compare our proposals with a source-language analogue of the LLVM twin-allocation semantics proposal of Lee et al. [2018] (§6).

Our second main contribution (§7–9) comprises substantial improvements to our Cerberus semantics for a large fragment of C [Memarian et al. 2016]. Cerberus defines semantics by elaborating C source into a purpose-built Core language; Core is abstracted on a clean memory object model interface that we instantiate with our memory object models. We thus make the combined semantics executable as test oracles, and confirm that they do the right thing for our library of semantic test cases (except for tests using IO, that Cerberus does not support). We make Cerberus more robust and usable with a new parser; extensive front-end improvements; a type system for Core; concrete, symbolic and interactive execution modes; and a web-interface GUI, linked to for each test and shown in Fig. 1, to permit easy experimentation with small examples. Cerberus now also identifies many of the clauses of the ISO C standard text captured by its definitions of type-checking and elaboration, displaying these in the GUI. The project page includes data for various compilers and other tools for these tests: GCC 8.1, Clang 6.0, ICC 19, UBSAN, ASAN, MSAN, CompCert [Leroy 2009; Leroy et al. 2018], RV-Match [Guth et al. 2016], CH2O [Krebbers 2015], and CHERI [Chisnall et al. 2015; Watson et al. 2018, 2015; Woodruff et al. 2014]. We include extensive validation of the combination of Cerberus and our provenance semantics on various existing test suites (§9): the GCC torture tests [FSF 2018a], the ITC Toyota benchmark [Shiraishi et al. 2015], the KCC example test suite [Hathhorn et al. 2015; KCC 2018], and a family of Csmith tests [Regehr et al. 2012].

Our third contribution is an empirical investigation of existing C code, examining the extent to which it is compatible with our proposed provenance semantics (§10). We do so in three ways. For the first two, we exploit data from the CHERI project [Chisnall et al. 2015; Watson et al. 2018, 2015; Woodruff et al. 2014], which has designed experimental architectures with hardware support for fine-grained memory protection and secure encapsulation, using capabilities, and ported software including the FreeBSD kernel and userspace to this. The CHERI-adapted FreeBSD userspace includes hundreds of libraries, daemons, and command-line tools, such as zlib, OpenSSL, and OpenSSH; other ported applications include Postgres, nginx, and WebKit, representing a broad range of code functionality, style, and vintage. CHERI C (when using capabilities for all pointers) is a stricter model than our provenance semantics, so the changes required for this tell us about the semantics relied on by the original code. We analyse the changes needed, and also run the resulting code in a CHERI emulator adapted to log all capability arithmetic that involves out-of-bounds pointer constructions. Third, we develop a novel dynamic instrumentation tool that can be applied to conventional (non-CHERI) code, tracking semantic provenance information in shadow memory, and present preliminary results from this.

Last but not least, our fourth contribution is an ongoing engagement with the ISO WG14 standards committee about these issues (§11).

Caveats and non-goals Our combined model covers many features of C, both syntactic and semantic, but not all. This paper focusses on provenance and pointer semantics. We do not address the memory object model issues relating to subobject provenance, uninitialised reads, and padding. We focus on the C commonly used for mainstream systems programming without effective types (with `-fno-strict-aliasing`). Our semantics is intended as a *source semantics* for C. For an intermediate language, e.g. LLVM, there are related but distinct goals (see §6).

Cerberus does not cover preprocessor features, C11 character-set features, general use of floating-point and complex types (beyond simple float constants and arithmetic from the underlying OCaml implementation), user-defined variadic functions (we do cover `printf`), bitfields, `volatile`, `restrict`, generic selection, `register`, flexible array members, some exotic initialisation forms, signals, `longjmp`, multiple translation units, most of the standard library, or concurrency.

We make Cerberus executable as a test oracle to explore all the behaviour of small test cases. It is not intended as a bug-finding tool for production C code.

2 BASIC POINTER PROVENANCE

C pointer values are typically represented at runtime as simple concrete numeric values, but mainstream compilers routinely exploit information about the *provenance* of pointers to reason that they cannot alias, and hence to justify optimisations. In this section we develop a provenance semantics for simple cases of the construction and use of pointers,

For example, consider the classic test on the right (note that this and many of the examples below are edge-cases, exploring the boundaries of what different semantic choices allow, and sometimes what behaviour existing compilers exhibit; they are not all intended as desirable code idioms).

```
// provenance_basic_global_yx.c (and an xy variant)
#include <stdio.h>
#include <string.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 (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11;  // does this have undefined behaviour?
    printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
  }
}
```

Depending on the implementation, `x` and `y` might in some executions happen to be allocated in adjacent memory, in which case `&x+1` and `&y` will have bitwise-identical representation values, the `memcmp` will succeed, and `p` (derived from a pointer to `x`) will have the same representation value as a pointer to a different object, `y`, at the point of the update `*p=11`. This can occur in practice, e.g. with GCC 8.1 -O2 on some platforms. Its output of `x=1 y=2 *p=11 *q=2` suggests that the compiler is reasoning that `*p` does not alias with `y` or `*q`, and hence that the initial value of `y=2` can be propagated to the final `printf`. ICC, e.g. ICC 19 -O2, also optimises here (for a variant with `x` and `y` swapped), producing `x=1 y=2 *p=11 *q=11`. In contrast, Clang 6.0 -O2 just outputs the `x=1 y=11 *p=11 *q=11` that one might expect from a concrete semantics. Note that this example does not involve type-based alias analysis, and the outcome is not affected by GCC or ICC's `-fno-strict-aliasing` flag; note also that the mere formation of the `&x+1` one-past pointer is explicitly permitted by the ISO standard.

These GCC and ICC outcomes would not be correct with respect to a concrete semantics, and so to make the existing compiler behaviour sound it is necessary for this program to be deemed to have undefined behaviour.

The current ISO standard text does not explicitly speak to this, but the 2004 ISO WG14 C standards committee response to Defect Report 260 (DR260 CR) [WG14 2004] hints at a notion of provenance associated to values that keeps track of their "origins":

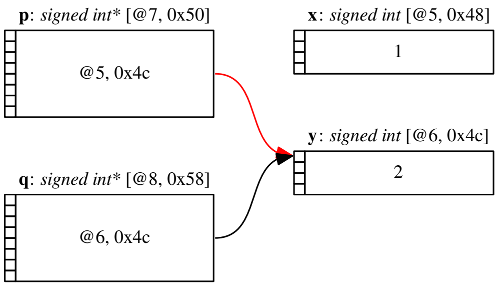
> *"Implementations are permitted to track the origins of a bit-pattern and [...]. They may also treat pointers based on different origins as distinct even though they are bitwise identical."*

However, DR260 CR has never been incorporated in the standard text, and it gives no more detail. This leaves many specific questions unclear: it is ambiguous whether some programming idioms are allowed or not, and exactly what compiler alias analysis and optimisation are allowed to do.

Basic provenance semantics for pointer values For simple cases of the construction and use of pointers, capturing the basic intuition suggested by DR260 CR in a precise semantics is straightforward: we associate a *provenance* with every pointer value, identifying the original allocation the pointer is derived from. In more detail:

- We take abstract-machine pointer values to be pairs (π, a), adding a *provenance* π, either $@i$ where i is an allocation ID, or the *empty* provenance `@empty`, to their concrete address a.
- On every allocation (of objects with static, thread, automatic, and allocated storage duration), the abstract machine chooses a fresh allocation ID i (unique across the entire execution), and the resulting pointer value carries that single allocation ID as its provenance $@i$.
- Provenance is preserved by pointer arithmetic that adds or subtracts an integer to a pointer.
- At any access via a pointer value, its numeric address must be consistent with its provenance, with undefined behaviour otherwise. In particular:
  - access via a pointer value which has provenance a single allocation ID $@i$ must be within the memory footprint of the corresponding original allocation, which must still be live.
  - all other accesses, including those via a pointer value with empty provenance, are undefined behaviour.

  This undefined behaviour is what justifies optimisation based on provenance alias analysis.



On the right is a provenance-semantics memory-state snapshot (from the Cerberus GUI) for `provenance_basic_global_xy.c`, just before the invalid access via `p`, showing how the provenance mismatch makes it UB.

All this is for the *C abstract machine* as defined in the standard: compilers might rely on provenance in their alias analysis and optimisation, but one would not expect normal implementations to record or manipulate provenance at runtime (though dynamic or static analysis tools might, as might non-standard implementations such as CHERI C). Provenances therefore do not have program-accessible runtime representations in the abstract machine.

Can one construct out-of-bounds (by more than one) pointer values by pointer arithmetic? Consider the example below, where q is transiently (more than one-past) out of bounds but brought back into bounds before being used for access. In ISO C, constructing such a pointer value is clearly stated to be undefined behaviour [WG14 2011a, 6.5.6p8]. This can be captured using the provenance of the pointer value to determine the relevant bounds. There are cases where such pointer arithmetic would go wrong on some platforms (some now exotic), e.g. where pointer arithmetic subtraction overflows, or if the transient value is not aligned and only aligned values are representable at the particular pointer type, or for hardware that does bounds checking, or where pointer arithmetic might wrap at values less than the obvious word size (e.g. "near" or "huge" 8086 pointers). However, transiently out-of-bounds pointer construction seems to be common in practice, as we see in §10, and in Chisnall et al. [2015]. It may be desirable to make it implementation-defined whether such pointer construction is allowed. That would continue to permit implementations in which it would go wrong to forbid it, but give a clear way for other implementations to document that they do not exploit this UB in compiler optimisations that may be surprising to programmers. Cerberus supports both semantics, with a switch.

```
// cheri_03_ii.c
int x[2];
int *p = &x[0];
int *q = p + 11; // defined behaviour?
q = q - 10;
*q = 1;
```

Inter-object pointer arithmetic The first example in this section relied on guessing (and then checking) the offset between two allocations. What if one instead calculates the offset, with pointer subtraction; should that let one move between objects, as below? In ISO C11, the q-p is UB (as a pointer subtraction between pointers to different objects, which in some abstract-machine executions are not one-past-related). In a variant semantics that allows construction of more-than-one-past pointers, one would have to to choose whether the *r=11 access is UB or not. The basic provenance semantics will forbid it, because r will retain the provenance of the x allocation, but its address is not in bounds for that. This is probably the most desirable semantics: we have found very few example idioms that intentionally use inter-object pointer arithmetic, and the freedom that forbidding it gives to alias analysis and optimisation seems significant.

```
// pointer_offset_from_ptr_subtraction_global_xy.c
#include <stdio.h>
#include <string.h>
#include <stddef.h>
int x=1, y=2;
int main() {
  int *p = &x;
  int *q = &y;
  ptrdiff_t offset = q - p;
  int *r = p + offset;
  if (memcmp(&r, &q, sizeof(r)) == 0) {
    *r = 11; // is this free of UB?
    printf("y=%d *q=%d *r=%d\n",y,*q,*r);
  }
}
```

Pointer equality comparison and provenance A priori, pointer equality comparison (with == or !=) might be expected to just compare their numeric addresses, but we observe GCC 8.1 -O2 sometimes regarding two pointers with the same address but different provenance as nonequal (provenance_equality_global_xy.c). Unsurprisingly, this happens in some circumstances but not others, e.g. if the test is pulled into a simple separate function, but not if in a separate compilation unit. To be conservative w.r.t. current compiler behaviour, pointer equality in the semantics should give false if the addresses are not equal, but nondeterministically (at each runtime occurrence) either take provenance into account or not if the addresses are equal – this specification looseness accommodating implementation variation. Alternatively, one could require numeric comparisons, which would be a simpler semantics for programmers but force that GCC behaviour to be regarded as a bug. Cerberus supports both options. One might also imagine making it UB to compare pointers that are not strictly within their original allocation, but that would break loops that test against a one-past pointer, or requiring equality to *always* take provenance into account, but that would require implementations to track provenance at runtime.

The current ISO C11 standard text is too strong here unless numeric comparison is required: 6.5.9p6 says *"Two pointers compare equal **if and only if** both are [...] or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space"*, which requires such pointers to compare equal – reasonable pre-DR260CR, but debatable after it.

3 POINTER CONSTRUCTION VIA CASTS, REPRESENTATION ACCESSES, ETC.

To support low-level systems programming, C provides many other ways to construct and manipulate pointer values:

- casts of pointers to integer types and back, possibly with integer arithmetic, e.g. to force alignment, or to store information in unused bits of pointers;
- copying pointer values with `memcpy`;
- manipulation of the representation bytes of pointers, e.g. via user code that copies them via `char*` or `unsigned char*` accesses;
- type punning between pointer and integer values;
- I/O, using either `fprintf`/`fscanf` and the `%p` format, `fwrite`/`fread` on the pointer representation bytes, or pointer/integer casts and integer I/O;
- copying pointer values with `realloc`;
- constructing pointer values that embody knowledge established from linking, and from constants that represent the addresses of memory-mapped devices.

A satisfactory semantics has to address all these, together with the implications on optimisation. We define and explore two main alternatives:

- **PVI**: a semantics that tracks provenance via integer computation, associating a provenance with all integer values (not just pointer values), preserving provenance through integer/pointer casts, and making some particular choices for the provenance results of integer and pointer `+/-` integer operations; or
- **PNVI**: a semantics that does not track provenance via integers, but instead, at integer-to-pointer cast points, checks whether the given address points within a live object and, if so, recreates the corresponding provenance. We explain in the next section why this is not as damaging to optimisation as it may sound.

For the latter, we also mention three variants:

- **PNVI-address-taken**: a variant that restricts the above to objects whose address has been taken;
- **PNVI-escaped**: potential variants that additionally restrict to objects whose address has been taken and (in some sense) escaped; and
- **PNVI-wildcard**: a variant that gives a "wildcard" provenance to the results of integer-to-pointer casts, delaying checks to access time.

The PVI semantics, which we developed informally in ISO WG14 working papers [Memarian et al. 2018; Memarian and Sewell 2016a], was motivated in part by the GCC documentation [FSF 2018b]:

> *"When casting from pointer to integer and back again, the resulting pointer must reference the same object as the original pointer, otherwise the behavior is undefined. That is, one may not use integer arithmetic to avoid the undefined behavior of pointer arithmetic as proscribed in C99 and C11 6.5.6/8."*

which presumes there is *an* "original" pointer, and by experimental data for `uintptr_t` analogues of the first test of §2, which suggested that GCC and ICC sometimes track provenance via integers (see `xy` and `yx` variants). However, discussions at the 2018 GNU Tools Cauldron suggest instead that at least some key developers regard the result of casts from integer types as potentially broadly aliasing, at least in their GIMPLE IR, and such test results as long-standing bugs in the RTL backend.

Shifting to a provenance semantics that does not track provenance via integers would be a substantial simplification, in the definition of the semantics, in how easy it is for people to understand, and in the consequences for existing code (which might otherwise need additional annotations for exotic idioms). That leads us to articulate and explore the various options above, to see which could be broadly acceptable.

Pointer/integer casts The ISO standard (6.3.2.3) leaves conversions between pointer and integer types almost entirely implementation-defined, except for conversion of integer constant `0` and null pointers, and for the optional `intptr_t` and `uintptr_t` types, for which it guarantees that any *"valid pointer to* **`void`***"* can be converted and back, and that *"the result will compare equal to the original pointer"*. As we have seen, in a post-DR260CR provenance-aware semantics, *"compare equal"* is not enough to guarantee the two are interchangeable, which was clearly the intent of that phrasing. Both PVI and PNVI support this, by preserving or reconstructing the original provenance respectively (`provenance_roundtrip_via_intptr_t.c`).

Inter-object integer arithmetic Below is a `uintptr_t` analogue of the last test of §2, attempting to move between objects with `uintptr_t` arithmetic. In PVI, this remains UB. First, the integer values of `ux` and `uy` have the provenances of the allocations of `x` and `y` respectively. Then `offset` is a subtraction of two integer values with non-equal single provenances; we define the result of such to have the empty provenance. Adding that empty-provenance result to `ux` preserves the original `x`-allocation provenance of the latter, as does the cast to **`int`**`*`. Then the final `*p=11` access is via a pointer value whose address is not consistent with its provenance.

```
// pointer_offset_from_int_subtraction_global_xy.c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int x=1, y=2;
int main() {
  uintptr_t ux = (uintptr_t)&x;
  uintptr_t uy = (uintptr_t)&y;
  uintptr_t offset = uy - ux;
  printf("Addresses: &x=%"PRIuPTR" &y=%"PRIuPTR\
       " offset=%"PRIuPTR" \n",ux,uy,offset);
  int *p = (int *)(ux + offset);
  int *q = &y;
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11; // is this free of UB?
    printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
  }
}
```

In PNVI, on the other hand, this has defined behaviour. The integer values are pure integers, and at the **`int`**`*` cast the value of `ux+offset` matches the address of `y` (live and of the right type), so the resulting pointer value takes on the provenance of the `y` allocation. Similarly, PVI forbids while PNVI allows (here contrary to current GCC/ICC O2) `uintptr_t` analogues of the first test of §2 (`provenance_basic_using_uintptr_t_global_xy.c`).

Both choices are defensible here: PVI will permit more aggressive alias analysis for pointers computed via integers (though those may be relatively uncommon), while PNVI will allow not just this test, which as written is probably not idiomatic desirable C, but also the essentially identical XOR doubly linked list idiom, using only one pointer per node by storing the XOR of two (`pointer_offset_xor_global.c`). Opinions differ as to whether that idiom matters for modern code.

There are other real-world but rare cases of inter-object arithmetic, e.g. in the implementations of Linux and FreeBSD per-CPU variables, in fixing up pointers after a `realloc`, and in dynamic linking (though arguably some of these are not between C abstract-machine objects). These are rare enough that it seems reasonable to require additional source annotation, or some other mechanism, to prevent compilers implicitly assuming that uses of such pointers as undefined.

Pointer provenance for pointer bit manipulations It is a standard idiom in systems code to use otherwise unused bits of pointers: low-order bits for pointers known to be aligned, and/or high-order bits beyond the addressable range. The example on the right (which assumes `_Alignof(int) >= 4`) does this: casting a pointer to `uintptr_t` and back, using bitwise logical operations on the integer value to store some tag bits.

To allow this, we suggest that the set of unused bits for pointer types of each alignment should be made implementation-defined. In PVI we make the binary operations used here, combining an integer value that has some provenance ID with a pure integer, preserve that provenance. In PNVI the intermediate value of `q` will have empty provenance, but the value of `r` used for the access will re-acquire the correct provenance at cast time.

```
// provenance_tag_bits_via_uintptr_t_1.c
#include <stdio.h>
#include <stdint.h>
int x=1;
int main() {
  int *p = &x;
  // cast &x to an integer
  uintptr_t i = (uintptr_t) p;
  // set low-order bit
  i = i | 1u;
  // cast back to a pointer
  int *q = (int *) i; // does this have UB?
  // cast to integer and mask out low-order bits
  uintptr_t j = ((uintptr_t)q) & ~((uintptr_t)3u);
  // cast back to a pointer
  int *r = (int *) j;
  // are r and p now equivalent?
  *r = 11;           //  does this have UB?
  _Bool b = (r==p);  //  is this true?
  printf("x=%i *r=%i (r==p)=%s\n",x,*r,b?"t":"f");
}
```

Algebraic properties of integer operations The PVI definitions of the provenance results of integer operations, chosen to make the previous two examples respectively forbidden and allowed, has an unfortunate consequence: it makes those operations no longer associative (compare this and this; the latter is UB in PVI). It is unclear whether this would be acceptable in practice, either for C programmers or for compiler optimisation. One could conceivably switch to a PVI-multiple variant, allowing provenances to be finite sets of allocation IDs. That would allow the `pointer_offset_from_int_subtraction_global_xy.c` example above, but perhaps too much else besides. PNVI does not suffer from this problem.

Copying pointer values with `memcpy()` This clearly has to be allowed (`pointer_copy_memcpy.c`), and so, to make the results usable for accessing memory without UB, `memcpy()` and similar functions have to preserve the original provenance. The ISO C11 text does not explicitly address this (in a pre-provenance semantics, before DR260, it did not need to). One could do so by special-casing `memcpy()` and similar functions to preserve provenance, but the following questions suggest less ad hoc approaches.

Copying pointer values bytewise, with user-`memcpy` One of the key aspects of C is that it supports manipulation of object representations, e.g. as in the following naive user implementation of a `memcpy`-like function, which constructs a pointer value from copied bytes. This too should be allowed. PVI makes it legal by regarding each byte (as an integer value) as having the provenance of the original pointer, and the result pointer, being composed of representation bytes of which at least one has that provenance and none have a conflicting provenance, as having the same. PNVI makes it legal in a different way: there, the representation bytes have no provenance, but when reading a pointer value from the copied memory, the read will be from multiple representation-byte writes. We use essentially the same semantics for such reads as for integer-to-pointer casts: checking at read-time that the address is within a live object, and giving the result the corresponding provenance. One could instead require, more restrictively, that the result has all the original bytes of some legitimate pointer. There may not be much reasonable code that would be sensitive to the distinctions between these, but there is some, e.g. manipulations of pointers where one knows the high-order bytes are the same. This is important in some language runtimes, using 32- or 48-bit values for pointers in a 64-bit architecture.

```
// pointer_copy_user_dataflow_direct_bytewise.c
#include <stdio.h>
#include <string.h>
int  x=1;
void user_memcpy(unsigned char* dest,
                 unsigned char *src, size_t n) {
  while (n > 0)  {
    *dest = *src;
    src += 1; dest += 1; n -= 1;
  }
}
int main() {
  int *p = &x;
  int *q;
  user_memcpy((unsigned char*)&q,
              (unsigned char*)&p, sizeof(int *));
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

As Lee observes [private communication], to make it legal for compilers to replace user-memcpy by the library version, one might want the two to have exactly the same semantics.

Real `memcpy()` implementations are more complex. The glibc `memcpy()`[glibc 2018] involves copying byte-by-byte, as above, and also word-by-word and, using virtual memory manipulation, page-by-page. Word-by-word copying is not permitted by the ISO standard, as it violates the effective type rules, but we believe C2x should support it for suitably annotated code. Virtual memory manipulation is outside our scope at present.

Copying pointer values via encryption To more clearly delimit what idioms our proposals do and do not allow, consider copying pointers via code that encrypts or compresses a block of multiple pointers together, decrypting or uncompressing later. In PVI, this involves exactly the same combination of distinct-provenance values that (to prohibit inter-object arithmetic, and thereby enable alias analysis) we above regard as having empty-provenance results. As copying pointers in this way is a very rare idiom, we believe it reasonable to require such code to have additional annotations. In PNVI, it would just work, in the same way as `user_memcpy()`.

It has been argued that pointer construction via `intptr_t` and back via any value-dependent identity function should be required to work. That would admit the above, but defining that notion of "value-dependent" is exactly what is hard in the concurrency thin-air problem [Batty et al. 2015], and we do not believe that it is practical to make compilers respect dependencies in general.

Copying pointer values via control flow We also have to ask whether a usable pointer can be constructed via non-dataflow control-flow paths, e.g. if testing equality of an unprovenanced integer value against a valid pointer permits the integer to be used as if it had the same provenance as the pointer. We do not believe that this is relied on in practice, and our proposed PVI semantics does not permit it; it tracks provenance only through dataflow. For example, consider exotic versions of `memcpy` that make a control-flow choice on the value of each bit or each byte, reconstructing each with constants in each control-flow branch (`pointer_copy_user_ctrlflow_bytewise.c` and `pointer_copy_user_ctrlflow_bitwise.c`). These are value-dependent identity functions, and in PNVI would work, while PVI they would give empty-provenance pointer values and hence UB.

Integer comparison and provenance If integer values have associated provenance, as in PVI, one has to ask whether the result of an integer comparison should also be allowed to be provenance dependent (`provenance_equality_uintptr_t_global_xy.c`). GCC did do so at one point, but it was regarded as a bug and fixed (from 4.7.1 to 4.8). We propose that the numeric results of all operations on integers should be unaffected by the provenances of their arguments. .

Pointer provenance and union type punning Pointer values can also be constructed in C by type punning, e.g. writing a `uintptr_t` union member and then reading it as a pointer-type member (`provenance_union_punning_2_global_xy.c`). The ISO standard says *"the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type"*, but says little about that reinterpretation. We propose that these reinterpretations be required to be implementation-defined, (in PVI) that it be implementation-defined whether the result preserves the original provenance (e.g. where they are the identity), and (in PNVI) that the same integer-to-pointer cast semantics be used at such reads (the latter does not match current ICC O2 behaviour).

Pointer provenance via IO Consider now pointer provenance flowing via IO, e.g. writing the address of an object to a pipe or file and reading it back in. We have three versions: one using `fprintf`/`fscanf` and the `%p` format, one using `fwrite`/`fread` on the pointer representation bytes, and one converting the pointer to and from `uintptr_t` and using `fprintf`/`fscanf` on that value with the `PRIuPTR`/`SCNuPTR` formats (`provenance_via_io_percentp_global.c`, `provenance_via_io_bytewise_global.c`, and `provenance_via_io_uintptr_t_global.c`) The first gives a syntactic indication of a potentially escaping pointer value, while the others (after preprocessing) do not. Somewhat exotic though they are, these idioms are used in practice: in graphics code for serialisation/deserialisation (using `%p`), in xlib (using `SCNuPTR`), and in debuggers.

In the ISO standard, the text for `fprintf` and `scanf` for `%p` says that this should work: *"If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the* `%p` *conversion is undefined"* (again construing the pre-DR260 "compare equal" as implying the result should be usable for access), and the text for `uintptr_t` and the presence of `SCNuPTR` in `inttypes.h` weakly implies the same there.

But then what can compiler alias analyses assume about such a pointer read? In PNVI, this is simple: at `scanf`-time, for the `%p` version, or when a pointer is read from memory written by the other two, we can do a runtime check and potential acquisition of provenance exactly like an integer-to-pointer cast. For PVI, however, there are several options, none of which seem ideal: we could use a PNVI-like semantics, but that would be stylistically inconsistent with the rest of PVI; or (only for the first) we could restrict that to provenances that have been output via `%p`), or we could require new programmer annotation, at output and/or input points, to constrain alias analysis.

Pointers from device memory and linking In practice, concrete memory addresses or relationships between them sometimes are determined and relied on by programmers, in implementation-specific ways. Sometimes these are simply concrete absolute addresses which will never alias C stack, heap, or program memory, e.g. those of particular memory-mapped devices in an embedded system. Others are absolute addresses and relative layout of program code and data, usually involving one or more *linking* steps. For example, platforms may lay out certain regions of memory so as to obey particular relationships, e.g. in a commodity operating system where high addresses are used for kernel mappings, initial stack lives immediately below the arguments passed from the operating system, and so on. The details of linking and of platform memory maps are outside the scope of ISO C, but real C code may embody knowledge of them. Such code might be as simple as casting a platform-specified address, represented as an integer literal, to a pointer. It might be more subtle, such as assuming that one object directly follows another in memory—the programmer having established this property at link time (perhaps by a custom linker script). It is necessary to preserve the legitimacy of such C code, so that compilers may not view such memory accesses as undefined behaviour, even with increasing link-time optimisation.

We leave the design of exactly what escape-hatch mechanisms are needed here as an open problem. For memory-mapped devices, one could simply posit implementation-defined ranges of such memory which are guaranteed not to alias C objects. The more general linkage case is more interesting, but well outside current ISO C. The tracking of provenance through embedded assembly is similar.

4 IMPLICATIONS OF PROVENANCE SEMANTICS FOR OPTIMISATIONS

In an ideal world, a memory object semantics for C would be consistent with all existing mainstream code usage and compiler behaviour. In practice, we suspect that (absent a precise standard) these have diverged too much for that, making some compromise required. As we have already seen, the PNVI semantics would make some currently observed GCC and ICC behaviour unsound, though at least some key GCC developers already regard that behaviour as a longstanding unfixed bug, due to the lack of integer/pointer type distinctions in RTL. We now consider some other important cases, by example.

Optimisation based on equality tests Both provenance semantics let `p==q` hold in some cases where `p` and `q` are not interchangeable. As Lee et al. [2018] observe in the LLVM IR context, that may limit optimisations such as GVN (global value numbering) based on pointer equality tests. PVI suffers from the same problem also for integer comparisons, wherever the integers might have been cast from pointers and eventually be cast back. This may be more serious.

Can a function argument alias local variables of the function? In general one would like this to be forbidden, to let optimisation assume its absence. Consider first the example below, where `main()` guesses the address of `f()`'s local variable, passing it in as a pointer, and `f()` checks it before using it for an access. Here we see, for example, GCC -O0 optimising away the `if` and the write `*p=7`, even in executions where the `ADDRESS_PFI_1PG` constant is the same as the `printf`'d address of `j`. We believe that compiler behaviour should be permitted, and hence that this program should be deemed to have UB — or, in other words, that code should not normally be allowed to rely on implementation facts about the allocation addresses of C variables. The PVI semantics flags UB for the simple reason that `j` is created with the empty provenance, and hence `p` inherits that. The PNVI semantics also deems this to be UB, because at the point of the `(int*)i` cast the `j` allocation does not yet exist, so the cast gives a pointer with empty provenance; any execution that goes into the `if` would thus flag UB.

```
// pointer_from_integer_1pg.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(int *p) {
  int j=5;
  if (p==&j)
    *p=7;
  printf("j=%d &j=%p\n",j,(void*)&j);
}
int main() {
  uintptr_t i = ADDRESS_PFI_1PG;
  int *p = (int*)i;
  f(p);
}
```

Varying to do the cast to `int*` in `f()` instead of `main()`, passing in an integer `i` instead of a pointer (`pointer_from_integer_1ig.c`), this remains UB in PVI, but in PNVI becomes defined, as `j` exists at the point when the abstract machine does the `(int*)i` cast. At present we do not see any strong reason why making this defined would not be acceptable — it amounts to requiring compilers to be conservative for the results of integer-to-pointer casts where they cannot see the source of the integer, which we imagine to be a rare case — but this does not match current O2 or O3 compilation for GCC, Clang, or ICC.

Allocation-address nondeterminism Note that both of the previous examples take the address of `j` to guard their `*p=7` accesses. Removing the conditional guards gives tests that one would surely like to forbid (tests `pointer_from_integer_1p.c` and `pointer_from_integer_1i.c`). Both are forbidden in PVI for the same reason as before, and the first is forbidden in PNVI again because `j` does not exist at the cast point.

But the second forces us to think about how much allocation-address nondeterminism should be quantified over in the basic definition of undefined behaviour. For evaluation-order and concurrency nondeterminism, one would normally say that if there exists any execution that flags UB, then the program as a whole has UB (for the moment ignoring UB that occurs only on some paths following I/O input, which is another important question that the current ISO text does not address).

This view of UB seems to be unfortunate but inescapable. If one looks just at a single execution, then (at least between input points) we cannot temporally bound the effects of an UB, because compilers can and do re-order code w.r.t. the C abstract machine's sequencing of computation. In other words, UB may be flagged at some specific point in an abstract-machine trace, but its consequences on the observed implementation behaviour might happen much earlier (in practice, perhaps not very much earlier, but we do not have any good way of bounding how much). But then if one execution might have UB, and hence exhibit (in an implementation) arbitrary observable behaviour, then anything the standard might say about any other execution is irrelevant, because it can always be masked by that arbitrary observable behaviour.

Accordingly, our semantics nondeterministically chooses an arbitrary address for each allocation, subject only to alignment and no-overlap constraints (ultimately one would also need to build in constraints from programmer linking commands). Then in PNVI, the `..._1i.c` example is UB because, even though there is one execution in which the guess is correct, there is another (in fact many others) in which it is not. In those, the cast to `int*` gives a pointer with empty provenance, so the access flags UB — hence the whole program is UB, as desired.

Can a function access local variables of its parent? This too should be forbidden in general. The example on the right is forbidden by PVI, again for the simple reason that `p` has the empty provenance, and by PNVI by allocation-address nondeterminism, as there exist abstract-machine executions in which the guessed address is wrong. One cannot guard the access within `f()`, as the address of `j` is not available there. Guarding the call to `f()` with `if ((uintptr_t)&j == ADDRESS_PFI_2)` (`pointer_from_integer_2g.c`) again makes the example well-defined in PNVI, as the address is correct and `j` exists at the `int*` cast point, but notice again that the guard necessarily involves `&j`. This does not match current Clang at O2 or O3, which print `j=5`.

```
// pointer_from_integer_2.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f() {
  uintptr_t i=ADDRESS_PFI_2;
  int *p = (int*)i;
  *p=7;
}
int main() {
  int j=5;
  f();
  printf("j=%d\n",j);
}
```

The PNVI-address-taken, PNVI-escaped, and PNVI-wildcard alternatives An obvious refinement to PNVI is to restrict integer-to-pointer casts to recover the provenance only of objects that have had their address taken, recording that in the memory state. Perhaps surprisingly, that seems not to make much difference to the allowed tests, because the tests one might write tend to already be UB due to allocation-address nondeterminism, or to already take the address of an object to use it in a guard. PNVI-address-taken has the conceptual advantage of identifying these UBs without requiring examination of multiple executions, but the disadvantage that whether an address has been taken is a fragile syntactic property, e.g. not preserved by dead code elimination. One can also restrict further, to addresses that have in some sense escaped, but precisely defining a particular such sense is complex and somewhat arbitrary. A rather different model is to make the results of integer-to-pointer casts have a "wildcard" provenance, deferring the check that the address matches a live object from cast-time to access-time. This would make `pointer_from_integer_1pg.c` defined, which is surely not desirable.

The problem with lost address-takens and escapes Our PVI proposal allows computations that erase the numeric value (and hence a concrete view of the "semantic dependencies") of a pointer, but retain provenance. This makes examples like that below [Richard Smith, personal communication], in which the code correctly guesses an allocation address (which has the empty provenance) and adds that to a zero-valued quantity (with the correct provenance), allowed in PVI. We emphasise that we do not think it especially desirable to allow such examples; this is just a consequence of choosing a straightforward provenance-via-integer semantics that allows the bytewise copying and the bitwise manipulation of pointers above. In other words, it is not clear how it could be forbidden simply in PVI.

```
// provenance_lost_escape_1.c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include "charon_address_guesses.h"
int x=1; // assume allocation ID @1, at ADDR_PLE_1
int main() {
  int *p = &x;
  uintptr_t i1 = (intptr_t)p;            // (@1,ADDR_PLE_1)
  uintptr_t i2 = i1 & 0x00000000FFFFFFFF;//
  uintptr_t i3 = i2 & 0xFFFFFFFF00000000;// (@1,0x0)
  uintptr_t i4 = i3 + ADDR_PLE_1;        // (@1,ADDR_PLE_1)
  int *q = (int *)i4;
  printf("Addresses: p=%p\n",(void*)p);
  if (memcmp(&i1, &i4, sizeof(i1)) == 0) {
    *q = 11;  // does this have defined behaviour?
    printf("x=%d *p=%d *q=%d\n",x,*p,*q);
  }
}
```

However, in implementations some algebraic optimisations may be done before alias analysis, and those optimisations might erase the `&x`, replacing it and all the calculation of `i3` by `0x0` (a similar example would have `i3 = i1-i1`). But then alias analysis would be unable to see that `*q` could access `x`, and so report that it could not, and hence enable subsequent optimisations that are unsound w.r.t. PVI for this case. The basic point is that whether a variable has its address taken or escaped in the source language is not preserved by optimisation. A possible solution, which would need some implementation work for implementations that do track provenance through integers, but perhaps acceptably so, would be to require those initial optimisation passes to record the address-takens involved in computations they erase, so that that could be passed in explicitly to alias analysis. In contrast to the difficulties of preserving dependencies to avoid thin-air concurrency, this does not forbid optimisations that remove dependencies; it merely requires them to describe what they do.

In PNVI, the example is also allowed, but for a simpler reason that is not affected by such integer optimisation: the object exists at the **int**∗ cast. Implementations that take a conservative view of all pointers formed from integers would automatically be sound w.r.t. this. At present ICC is not, at O2 or O3.

Should PNVI allow one-past integer-to-pointer casts? For PNVI, one has to choose whether an integer that is one-past a live object (and not strictly within another) can be cast to a pointer with valid provenance, or whether this should give an empty-provenance pointer value. Lee observes that the latter may be necessary to make some optimisation sound [personal communication], and we imagine that this is not a normal idiom in practice, so we follow the stricter semantics here.

Summary of pros and cons Both semantics handle many cases as one would desire. Both suffer from the non-interchangeability of pointers that have compared equal (with ==) and PVI also from the same problem for integers. PVI suffers from the loss of algebraic properties of integer arithmetic operations, the difficulty of accommodating pointer I/O, and the problem with lost address-takens and escapes. PNVI makes some currently observable compiler behaviours unsound, though at least for GCC some of these are viewed by some developers as buggy in any case. PNVI is considerably simpler to define and explain than PVI, and on balance we think PNVI is preferable.

5 PROPOSED PVI AND PNVI SEMANTICS FOR PROVENANCE, IN DETAIL

We now give our proposed PVI and PNVI memory object semantics in more detail, as manually typeset mathematics simplified from the Cerberus mechanised Lem [Mulligan et al. 2014] source. We have removed most subobject details, function pointers, and some options. Neither the typeset models or the Lem source consider linking, or pointers constructed via I/O.

In Cerberus, the memory object model is factored out from Core with a clean interface, roughly as in [Memarian et al. 2016, Fig. 2]. This provides functions allocate_object (for objects with automatic or static storage duration, i.e. global and local variables), allocate_region (for the results of `malloc`, `calloc`, and `realloc`, i.e. heap-allocated regions), `kill` (for allocation lifetime end), `load`, and `store`, and the various operations on pointer and integer values, including casts, comparisons, shifting pointers by struct-member offsets, etc. The interface involves types `pointer_value` (p), `integer_value` (x), `floating_value`, and `mem_value` (v), which are abstract as far as Core is concerned. Distinguishing pointer and integer values gives more precise internal types. A pointer value can either be `null` or a pair (π, a) of a provenance π and address a. In PNVI, an integer value is simply a mathematical integer (within the appropriate range for the relevant C type), while in PVI, an integer value is a pair (π, n) of a provenance π and a mathematical integer n. Memory values are the storable entities, either a pointer, integer, floating-point, array, struct, or union value, or `unspec` for unspecified values, each together with their C type.

In both PVI and PNVI, a memory state is a pair (A, M), where A is a partial map from allocation IDs to either `killed` or allocation metadata (size n, optional C type τ (or `none` for allocated regions), base address a, permission flag $f \in$ {`readWrite`, `readOnly`}, and kind $k \in$ {`object`, `region`}), and M is a partial map from addresses to abstract bytes, which are triples of a provenance π, either a byte b or `unspec`, and an optional integer pointer-byte index j (or `none`). The last is used in PNVI to distinguish between loads of pointer values that were written as whole pointer writes vs those that were written byte-wise or in some other way.

Figures 2–5 give the main memory object semantics, as labelled transition relations from a memory state to either a memory state, UB, or OOM (out of memory), with labels for each operation (and their return value) in the Cerberus memory model interface.

For simplicity, we present the model without the ISO semantics that makes all pointers to an object or region indeterminate at the end of its lifetime, assume two's complement representations, assume NULL pointers have address 0, and allow NULL pointers to be constructed from any empty-provenance integer zero, not just integer constant expressions.

The M models the memory state in terms of low-level abstract bytes, but `store` and `load` take and return the higher-level memory values. We relate the two with functions $\mathrm{repr}(v)$, mapping a memory value to a list of abstract bytes, and $\mathrm{abst}(\tau, bs)$, mapping a list of abstract bytes to its interpretation as a memory value with C type τ.

The $\mathrm{repr}(v)$ function is defined by induction over the structure of its memory value parameter and returns a list of $\mathrm{sizeof}(\tau)$ abstract bytes, where τ is the C type of the parameter. The base cases are values with scalar types (integer, floating and pointers) and unspecified values. For an unspecified value of type τ, it returns a list with abstract bytes of the form (@`empty`, `unspec`, `none`). Non-null pointer values are represented with lists of abstract bytes that each have the provenance of the pointer value, the appropriate part of the two's complement encoding of the address, and the $0..\,\mathrm{sizeof}(\tau) - 1$ index of each byte. Null pointers are represented with lists of abstract bytes of the form (@`empty`, 0, `none`). In PVI, integer values are represented similarly to pointer values except that the third component of each abstract byte is `none`. In PNVI, integer values are represented by lists of abstract bytes whose first component is always the empty provenance and the third is again `none`. Floating-point values are similar, in both models, except that the provenance of the abstract bytes is always empty. For array and struct/union values the function is inductively applied to each subvalue and the resulting byte-lists concatenated. The layout of structs and unions follow an implementation-defined ABI, with padding bytes like those of unspecified values.

The $\mathrm{abst}(\tau, bs)$ function is defined by induction over τ. The base cases are again the scalar types. For these, $\mathrm{sizeof}(\tau)$ abstract bytes are consumed from bs and a scalar memory value is constructed from their second components: if any abstract byte has an `unspec` value, an unspecified value is constructed; otherwise, depending on τ, a pointer, integer or floating-point value is constructed using the two's complement or floating-point encoding. For pointers with address 0, the provenance is empty. For non-0 pointer values and integer values, in PVI the provenance is constructed as follows: if at least one abstract byte has non-empty provenance and all others have either the same or empty provenance, that provenance is taken, otherwise the empty provenance is taken. In PNVI, when constructing a pointer value, if the third components of the bytes all carry the appropriate index, the provenance of the result is the provenance of the bytes (which will all have the same provenance). Otherwise, the A part of the memory state is examined to find whether a live allocation exists with a footprint containing the pointer value that is being constructed. If so, its allocation ID is used for the provenance of the pointer value, otherwise the empty provenance is used. For array/struct types, $abst()$ recurses on the progressively shrinking list of abstract bytes.

In addition to the figures, some operations are desugared/elaborated to simpler expressions by the Cerberus pipeline (§7). Their PVI results have provenance as follows; their PNVI results are the same except that there integers have no provenance:

- the result of address-of (&) has the provenance of the object associated with the lvalue.
- prefix increment and decrement operators follow the corresponding pointer or integer arithmetic rules.
- the conditional operator has the provenance of the second or third operand; simple assignment has the provenance of the expression; compound assignment follows the pointer or integer arithmetic rules; the comma operator has the provenance of the second operand.
- integer unary `+`, unary `-`, and `~` operators preserve the original provenance; logical negation `!` has a value with empty provenance.
- **`sizeof`** and `_Alignof` operators give values with empty provenance.
- bitwise shifts has the provenance of their first operand.

$$[\textsc{label}\text{: allocate\_region}(al, n) = p]$$
$$\frac{i \notin \mathrm{dom}(A) \qquad a \in \mathrm{newAlloc}(A, al, n) \\ p = (@i, a)}{A, M \to A[i \mapsto (n, \mathrm{none}, a, \texttt{readWrite}, \texttt{region})], M}$$

$$[\textsc{label}\text{: load}(\tau, p) = v]$$
$$\frac{p = (@i, a) \qquad A(i) = (n, \_, a', f, k) \\ [a..a + \mathrm{sizeof}(\tau) - 1] \subseteq [a'..a' + n - 1] \\ v = \mathrm{abst}(\tau, M[a..a + \mathrm{sizeof}(\tau) - 1])}{A, M \to A, M}$$

$$[\textsc{label}\text{: allocate\_object}(al, \tau, \texttt{readWrite}) = p]$$
$$\frac{i \notin \mathrm{dom}(A) \qquad a \in \mathrm{newAlloc}(A, al, n) \\ n = \mathrm{sizeof}(\tau) \qquad p = (@i, a)}{A, M \to A(i \mapsto (n, \tau, a, \texttt{readWrite}, \texttt{object})), M}$$

$$[\textsc{label}\text{: store}(\tau, p, v)]$$
$$\frac{p = (@i, a) \qquad A(i) = (n, \_, a', \texttt{readWrite}, k) \\ [a..a + \mathrm{sizeof}(\tau) - 1] \subseteq [a'..a' + n - 1]}{A, M \to A, M([a..a + \mathrm{sizeof}(\tau) - 1] \mapsto \mathrm{repr}(v))}$$

$$[\textsc{label}\text{: allocate\_object}(al, \tau, \texttt{readOnly}(v)) = p]$$
$$\frac{i \notin \mathrm{dom}(A) \qquad a \in \mathrm{newAlloc}(A, al, n) \\ n = \mathrm{sizeof}(\tau) \qquad p = (@i, a)}{A, M \to A(i \mapsto (n, \tau, a, \texttt{readOnly}, \texttt{object})), M([a..a + n - 1] \mapsto \mathrm{repr}(v))}$$

$$[\textsc{label}\text{: kill}(p, k)]$$
$$\frac{p = (@i, a) \qquad k = k' \\ A(i) = (n, \_, a, f, k')}{A, M \to A(i \mapsto \texttt{killed}), M}$$

$$[\textsc{label}\text{: diff\_ptrval}(\tau, p, p') = x]$$
$$\frac{p = (@i, a) \qquad p' = (@i', a') \qquad i = i' \qquad A(i) = (n, \_, \hat{a}, f, k) \\ x = (@\texttt{empty}, (a - a')/\,\mathrm{sizeof}(\mathrm{dearray}(\tau))) \qquad a \in [\hat{a}..\hat{a} + n] \qquad a' \in [\hat{a}..\hat{a} + n]}{A, M \to A, M}$$

$$[\textsc{label}\text{: rel\_op\_ptrval}(p, p', op) = b]$$
$$\frac{p = (@i, a) \qquad p' = (@i', a') \qquad i = i' \\ b = op(a, a') \qquad op \in \{\le, <, >, \ge\}}{A, M \to A, M}$$

$$[\textsc{label}\text{: eq\_op\_ptrval}(p, p') = b]$$
$$\frac{\begin{cases} b = \texttt{true} & \text{if } p = p' \\ b \in \{(a = a'), \texttt{false}\} & \text{if } p = (\pi, a), p' = (\pi', a'), \text{ and } \pi \neq \pi' \\ b = \texttt{false} & \text{otherwise} \end{cases}}{A, M \to A, M}$$

$$\mathrm{iso\_array\_shift\_ptrval}(A, p, \tau, x) = \begin{cases} (@i, a') & \begin{array}{l} \text{if } p = (@i, a) \text{ and } x = (\pi', n) \text{ and} \\ a' = a + n * \mathrm{sizeof}(\tau) \text{ and} \\ A(i) = (n'', \_, a'', \_, \_) \text{ and} \\ a' \in [a''..a'' + n''] \end{array} \\ \text{UB: out of bounds} & \text{if all except the last conjunct above hold} \\ \text{UB: empty prov} & \text{if } p = (@\texttt{empty}, a) \\ \text{UB: killed prov} & \text{if } p = (@i, a) \text{ and } A(i) = \texttt{killed} \\ \text{UB: null pointer} & \text{if } p = \texttt{null} \end{cases}$$

$$\mathrm{member\_shift\_ptrval}(p, t, m) = \begin{cases} (\pi, a + \mathrm{offsetof\_ival}(\tau, m)), & \text{if } p = (\pi, a); \\ \mathrm{offsetof\_ival}(\tau, m), & \text{if } p = \texttt{null}. \end{cases}$$

Fig. 2. Provenance semantics: common rules. These rules are the same for PVI and PNVI, except that in the latter diff_ptrval constructs a pure integer v, instead of an integer value with @empty provenance. $\mathrm{dearray}(\tau)$ returns τ if it is not an array type, and otherwise returns its element type. $\mathrm{newAlloc}(A, al, n)$ returns the set of addresses of blocks of size n aligned by al that do not overlap with 0 or any other allocation in A.

allocate_region(al, n) / allocate_object$(al, \tau, \texttt{readwrite})$ / allocate_object$(al, \tau, \texttt{readOnly}(v))$:	
OOM out of memory	if newAlloc$(A, al, n) = \{\}$ or newAlloc$(A, al, \text{sizeof}(\tau)) = \{\}$
load(τ, p) / store(τ, p, v) / kill(p):	
UB null pointer	if $p = \texttt{null}$
UB empty provenance	if $p = (@\texttt{empty}, a)$
UB killed provenance	if $p = (@i, a)$ and $A(i) = \texttt{killed}$
load(τ, p) / store(τ, p, v):	
UB out of bounds	if $p = (@i, a)$, $A(i) = (n, \_, a', f, k)$, and $[a..a + \text{sizeof}(\tau) - 1] \not\subseteq [a'..a' + n - 1]$
store(τ, p, v):	
UB read-only	if $p = (@i, a)$ and $A(i) = (n, \_, a', \texttt{readOnly}, k)$
kill(p):	
UB non-alloc-address	if $p = (@i, a)$, $A(i) = (n, \_, a', f, k)$, and $a \neq a'$

(rules for diff_ptrval, rel_op_ptrval, and eq_op_ptrval omitted)

Fig. 3. Provenance semantics: error rules, for memory operations in state A, M, for both PVI and PNVI

$$\text{cast\_ival\_to\_ptrval}(\tau, x) = \begin{cases} \texttt{null}, & \text{if } x = (@\texttt{empty}, 0) \\ (\pi, n), & \text{otherwise, where } x = (\pi, n) \end{cases}$$

$$\text{cast\_ptrval\_to\_ival}(\tau, p) = \begin{cases} (@\texttt{empty}, 0), & \text{if } p = \texttt{null}; \\ (\pi, a), & \text{if } p = (\pi, a) \text{ and } a \in \texttt{value\_range}(\tau) \\ \text{UB}, & \text{otherwise} \end{cases}$$

$$\pi \oplus \pi' = \begin{cases} \pi, & \text{if } \pi = \pi' \text{ or } \pi' = @\texttt{empty}; \\ \pi', & \text{if } \pi = @\texttt{empty}; \\ @\texttt{empty}, & \text{otherwise.} \end{cases}$$

$$\text{op\_ival}(op, (\pi, n), (\pi', m)) = (\pi \oplus \pi', op(n, m)), \text{ where } op \in \{+, *, /, \%, \&, |, \wedge\}$$

$$\text{op\_ival}(-, (\pi, n), (\pi', m)) = \begin{cases} (@\texttt{empty}, n - m), & \text{if } \pi = @i \text{ and } \pi' = @i', \text{ whether } i = i' \text{ or not;} \\ (@i, n - m), & \text{if } \pi = @i \text{ and } \pi' = @\texttt{empty}; \\ (@\texttt{empty}, n - m), & \text{if } \pi = @\texttt{empty}. \end{cases}$$

$$\begin{aligned} \text{eq\_ival}((\pi, n), (\pi', m)) &= (n = m) \\ \text{lt\_ival}((\pi, n), (\pi', m)) &= (n < m) \\ \text{le\_ival}((\pi, n), (\pi', m)) &= (n \leq m) \end{aligned}$$

Fig. 4. PVI semantics

$$\text{cast\_ival\_to\_ptrval}(\tau, x) = \begin{cases} \texttt{null}, & \text{if } x = 0 \\ (@i, x), & \text{if } A(i) = (n, \_, a, f, k) \text{ and } [x..x + sizeof(\tau) - 1] \subseteq [a..a + n - 1] \\ (@\texttt{empty}, x), & \text{if there is no such } i \end{cases}$$

$$\text{cast\_ptrval\_to\_ival}(\tau, p) = \begin{cases} 0, & \text{if } p = \texttt{null}; \\ a, & \text{if } p = (\pi, a) \text{ and } a \in \texttt{value\_range}(\tau) \\ \text{UB}, & \text{otherwise} \end{cases}$$

Fig. 5. PNVI semantics

6 COMPARISON WITH THE TWIN-ALLOCATION SEMANTICS

Lee et al. [2018] address a closely related but subtly different problem, of the memory object model design for the LLVM IR. As an intermediate language, that is subject to different constraints to the C source-language semantics that is our focus here. It has to make standard compiler optimisations sound as IR-to-IR transformations: optimisations should not add observable behaviour w.r.t. the IR semantics. In contrast, a C source-language semantics need only make the end-to-end behaviour of standard compiler implementations sound, e.g. by being refinable to such IR semantics. An intermediate semantics can also be more liberal than a C source semantics, allowing more programs to be well-defined, and an LLVM IR semantics in particular can take advantage of facts about the analyses and optimisations done by that specific compiler. Then a source-language semantics for C should involve as little change to the ISO text and concepts as possible, and should be widely accessible, consistent with the intuitions of the standards committee, compiler writers, and C programmers.

Their *logical pointers*, obtained from allocations (or pointer arithmetic on logical pointers), are similar to our pointers with a single @i provenance. Their *physical pointers*, obtained from pointer-to-integer casts of logical pointers, are roughly analogous to pointers with a wildcard provenance. By itself, that would be much too liberal, so they add two additional mechanisms:

(1) to make it impossible to use such a pointer (when supplied as a function parameter) to access local variables, each physical pointer includes a *timestamp*, a call ID that maps (in their memory state) to the time stamp when the pointer was passed as argument to a function, or None if the pointer did not originate from an argument (or if that function has returned). This extra data lets them do an access-time check that rules out examples such as our `pointer_from_integer_1pg.c` of §4. In contrast, our PNVI's cast-time check can simply check whether an appropriate object exists when the integer-to-pointer cast is done.
(2) they defer pointer-construction bounds checking to access time: each physical pointer includes an *inbounds* set of physical addresses, of the previous pointer values that it has been constructed from, and at access-time it is checked that all of these are in-bounds. This deferred checking makes code-motion optimisations sound as IR-to-IR transformations, but it would not be appropriate for an ISO C source-language semantics, where such construction is UB irrespective of whether the pointer value is used for access.

They also rely heavily on allocation-address nondeterminism, as PNVI does, but take it further. They observe that for programs that almost or completely exhaust the allocatable address space, it is possible for code to indirectly learn facts about allocation addresses without explicitly casting them to integers, and that this can make some desirable optimisations unsound in general. To rule out such programs, they introduce *twin allocation*: their semantics makes one (or more) shadow allocations for each actual allocation, making it easy to reason that, for any example that guesses a concrete address, that there is another similar execution in which the guess is wrong. We conjecture that much the same could be achieved by restricting attention to programs that never "almost" run out of address space, i.e. that, in each execution, always leave at least space (suitably aligned) for one additional copy of the largest allocation they make in that execution.

Subject to that restriction, it seems plausible that one could establish a refinement from the PNVI C source-language semantics we describe here to their LLVM IR semantics. They demonstrate that the latter can be implemented (by adapting LLVM) with reasonable performance, so this would give a coherent and implementable end-to-end story.

7 RECALLING THE CERBERUS PIPELINE (BACKGROUND)

$$\begin{tikzcd}
\text{C source} \arrow[d, "\text{parsing}"] \\
\text{Cabs (C abstract syntax)} \arrow[d, "\text{desugaring}"] \\
\text{Ail (C abstract syntax, desugared)} \arrow[d, "\text{C type inference/checking}"] \\
\text{Typed Ail (C-type-annotated C abstract syntax)} \arrow[d, "\textbf{elaboration}"] \\
\text{Core} \arrow[d, "\text{Core-to-Core transformations}"] \\
\text{Core} \arrow[d, "\text{Core operational semantics and } \textbf{memory object model}"] \\
\text{executions}
\end{tikzcd}$$

Cerberus defines a semantics by elaboration: after front-end parsing and typechecking, it applies a compositional translation from C into a Core language. This translation makes many subtle aspects of C explicit: the loose specification of evaluation order, arithmetic conversions, undefined behaviour arising from overflow, and suchlike. Core is essentially a typed first-order call-by-value lambda calculus, extended with features to model the behaviour of those things. It has an executable small-step operational semantics, invoking the memory semantics for operations on memory and pointers. Cerberus is expressed largely as a pure functional definition in Lem [Mulligan et al. 2014] (approx. 20 000 lines), together with some OCaml (11 000), and TypeScript (3 000) code. The main pipeline is above. Fig. 1 shows a tiny example C program and its elaboration into Core, making explicit:

- the lifetime of `x`, with the Core `create` and `kill` (lines 2 and 15);
- its alignment constraint, computed from its C type with the Core `Ivalignof` (line 2);
- the fact that the arguments of the C `+` are unsequenced, with the Core `unseq` (line 6);
- other aspects of C's loose evaluation order with Core `let strong` and `let weak`;
- the fact that (in one variant of the semantics) a C signed `+` gives undefined behaviour if either argument is an unspecified value, with the Core `case` pattern match and `undef` (7–13);
- C arithmetic conversions, with the Core `conv_int` and `conv_loaded_int` library functions (lines 10 and 14), which are passed Core representations of the C result types; and
- the default `0` return of `main` (line 16).

Core arithmetic (e.g. the `+` at line 10) is all simply over the mathematical `integers` (paired with provenances in PVI), with conversions, wrapping, overflow checks, etc. all made explicit by the elaboration. Core distinguishes between values known to be `Specified` and a distinguished unspecified value at each type (a `loaded integer` can be either), and between effectful and `pure` Core expressions.

The semantics of the C store to `x`, on the other hand, is just a primitive `store` (line 14) in Core, with its behaviour left to the memory object model.

8 RE-ENGINEERING CERBERUS

From a programming-language semantics perspective, real C is a fantastically complex language, far more so than one might imagine from working with idealised C-like languages or from occasional C programming. Many aspects of this are well-defined by ISO, with sufficiently careful and informed reading thereof, but they are subtle and intricate. The previous Cerberus [Memarian et al. 2016] handled some of this in a principled way, but it was far from complete enough to handle the memory model tests we discuss here (it was validated chiefly with small Csmith-generated tests). To make it executable as a test oracle for memory-model tests, and to bring it closer to a generally usable tool, we have done substantial extension and re-engineering of the semantics. These are things that are handled by any full-fledged C front-end implementation, and by CIL [Necula et al. 2002], but Cerberus aims to have a clear relationship to the standard, to capture exactly what it says (where that is well-defined), and to report all undefined behaviours, and so we do not want to rely on (say) the Clang front-end. Moreover, the front-end also involves evaluation and a specification of C implementation-defined choices, for both of which we want to use the Cerberus definitions rather than some potentially incompatible semantics from another front-end.

Parser. We have implemented a new C parser, closely following the ISO standard grammar, incorporating the elegant lexer hack of Jourdan and Pottier [2017].

Core typechecker. We have implemented a type-checker for Core, in a bidirectional style [Pierce and Turner 2000]. This detected many elaboration bugs.

Desugaring of struct/union/array initialisers. ISO C defines an intricate syntax for initialisers, allowing an aggregate type to be initialised either with a flat list or with a more structured syntax, and with arbitrary switching between the two styles. There is implicit completion (with `0` or `NULL`) if insufficient values are given, and extra values are ignored. One can also use explicit designators: array indices, or struct/union member identifiers. Designators need not be contiguous or even in-order (though Cerberus currently requires the latter). Arrays of character type can be initialised with the bytes of string literals. Scalar initialisers can also have optional braces. Outer arrays can have unknown (`[]`) size, leaving this to be determined by the initialiser. Various violations give UB, but examples such as that on the right are well-defined.

```
// init.c
struct T {
  int x;
  struct T2 {
    int y[2];
    char z;
  } st;
  char c[3];
};

struct T arr1[3] =
  { 1, {{2,3,40,50}, 6,
    "foo"}, [2].st.z= 7};
```

Constant expressions. C constant expressions can involve implementation-defined choices and undefined behaviour, initialisation of pointers can include l-value expressions to statically allocated objects, and so on. This means the Cerberus desugaring phase sometimes needs to use the entire rest of the pipeline, including typechecking and evaluation (with an initially empty memory).

Qualifiers. C includes various qualifiers, **`const`**, **`volatile`**, **`restrict`**, and **`_Atomic`**. Cerberus supports **`const`**; the others are covered in the front-end but not in our dynamic semantics.

Additional features. We have also defined support for single-compilation-unit aspects of linkage, e.g. to detect the UB arising if something is declared with both internal and external linkage; proper treatment of enum desugaring; decay of array-typed expressions to pointers to their first elements; decay of function-typed expressions to function pointers; l-value conversion; **`switch`**; limited support for floats (but using OCaml floats rather than specifying their behaviour formally); and the C library functions that have arisen in tests (a few dozen).

Handling specification looseness ISO C is intentionally a very loose specification, to admit a wide variety of implementations. Mathematically, this can be handled straightforwardly with a nondeterministic semantics, but making Cerberus executable as a test oracle, able to enumerate the sets of all allowed behaviours of small test programs, needs careful design and some compromises. In some ways we are intentionally more specific than ISO, to capture facts about the de facto standard that much modern code depends on. We assume two's complement arithmetic, that integer types contain no padding bits, and that the character set is fixed to be a minor extension of ASCII. The numeric ranges, memory sizes, and alignment restrictions of integer types are implementation-defined in C. Cerberus is abstracted on a Core module defining these for C integer and floating types. For testing we use those of LP64, as commonly used on Macs and Linux. Cerberus supports the full evaluation-order looseness of C [Memarian et al. 2016, §5.6], but (without additional optimisation) this quickly leads to a combinatorial explosion of possible executions and is orthogonal to pointer provenance issues, so for the testing in this paper (for which evaluation order looseness is largely irrelevant) we use a “sequentialise” option to disable it.

Most seriously, the concrete addresses of allocations in C are visible to the program, via pointer comparisons, casts to integer types, access to representation bytes, etc. Our proposed mathematical semantics makes nondeterministic choices of allocation addresses, subject to alignment and disjointness constraints. In the tool (for both PVI and PNVI), we have two options:

A *concrete-allocation-address* mode, which uses a specific deterministic naive allocation strategy (resolving the newAlloc nondeterminism in the mathematical model). This performs well, and is what we use for most of the testing in this paper, but it cannot exhibit all allowed executions for code that observably depends on allocation addresses.

A *symbolic-allocation-address* mode, in which each allocation generates a fresh symbolic variable for its concrete address, together with constraints expressing that that is suitably aligned and disjoint. We use Z3 [de Moura and Bjørner 2008] to resolve such constraints as needed. Cerberus still has to make some nondeterministic choices, e.g. for conditional tests involving pointers:

```
int x,y; if (&x < &y ) { if (&y < &x) { return 10; }}
```

where for each `if` Cerberus explores two branches, with a new constraint in each for the conditional expression being true or false. Here in the true/true cases Z3 will detect unsatisfiability and Cerberus will throw away those executions. There is much more going on in even a simple C program than one might expect. For example, `printf("hello\n");` involves the allocation and initialisation of an object for the string constant, then allocation and initialisation of a temporary object for the argument to `printf`, then (as the argument of `printf` is a `char *`) seven loads of the individual bytes. This makes it hard to use the symbolic mode for larger examples, and so here we focus largely on the concrete-allocation-address mode; the other is less well-developed at present.

Web interface: Cerberus C explorer We have equipped Cerberus with a web interface for interactive exploration of the static and dynamic semantics of C programs (at http://cerberus.cl.cam.ac.uk/cerberus). It allows the user to navigate throughout the Cerberus pipeline shown in §7. Figure 1 is in fact a screen dump from this, with an example C program in the (editable) left-hand pane and the elaborated Core. The user can identify the elaboration of every subexpression in the code, by mousing over or selecting either the source or the Core.

Understanding what parts of the standard are relevant for particular snippets of C code is often hard, requiring great familiarity with the standard. Our semantics inserts ISO C standard annotations in the Ail and Core ASTs during type-checking and elaboration, indicating in many cases (but not all) which paragraphs of the standard are being captured by each clause. Web interface tooltips allow the user to immediately read the relevant paragraphs of N1570 [WG14 2011b], a publicly available committee draft essentially identical to the C11 standard. This annotation will also enable future automated coverage checking with respect to the standard, e.g. for test suites.

The interface allows the user to either interactively or pseudorandomly explore single execution paths, or (for small tests) perform an exhaustive search for all allowed executions, in either PVI or PNVI, and in either concrete-allocation-address or symbolic-allocation-address modes. It also provides a library of interesting test cases, including those of this paper, and integrates the Godbolt [2017] Compiler Explorer API, to show the assembly output for various compilers and platforms.

9 EXPERIMENTAL VALIDATION

Ideally, an executable-as-test-oracle C semantics would be accurate, robust, and clear enough to provide a definitive reference, e.g. so that it reports an UB if and only if there is one in a shared consensus interpretation of ISO. To the best of our knowledge, no semantics is yet at that point, and Cerberus is certainly not bug-free, but it is already a useful tool.

We validate our combined semantics (our memory object model linked with the rest of Cerberus) for our provenance tests and various other test suites; we detail this on the project web page and summarise here. We also report basic assessments of its performance. This testing uses Cerberus in concrete-allocation-address mode, with evaluation order sequentialisation.

Pointer Semantics First, for pointer provenance, we use our basic provenance semantics test suite, comprising the examples in the paper and variants thereof — currently 54 tests in 27 families. Cerberus produces the desired PVI and PNVI output for all except the 3 tests which involve file IO (Cerberus does not currently support a filesystem or `scanf`). Our testing found unintentional undefined behaviour in a few of the tests of [Memarian et al. 2016], e.g. involving signed integer overflow from unspecified values; we rewrote those tests accordingly.

We have also tried various other tools on these tests, though one needs care in interpreting the results, as some tests are only meaningful in executions with an address coincidence, and none are designed to aggressively exercise compiler optimisations. We exclude three tests that rely on address coincidences with non-address-taken variables. Leaving those aside, GCC 8.1, Clang 6.0, and ICC 19 all give output compatible with both PVI and PNVI, when compiled with or without optimisations and with or without strict aliasing, except that GCC and ICC are not currently sound at O2/O3 w.r.t. PNVI for `provenance_basic_using_uintptr_t_global_yx.c` and `provenance_union_punning_2_global_yx.c` (or variants thereof); none of GCC, Clang, and ICC are sound for PNVI at O2/O3 for `pointer_from_integer_1ig.c`; Clang is not sound for PNVI at O2/O3 for `pointer_from_integer_2g.c`; and ICC is not sound for PNVI at O2/O3 for `provenance_lost_escape_1.c`.

The CompCert 3.4 compiler gives results consistent with both PVI and PNVI for all tests, except at -O w.r.t. PNVI for `pointer_from_integer_1ig` and `pointer_from_integer_2g`. Many of our tests are outside of the scope of the CompCert interpreter and CH2O [Krebbers 2015], since they use standard library functions that these do not support.

RV-Match reports similar results to Cerberus, except that for four tests that Cerberus regards as UB it warns about an implementation-defined behaviour and then segfaults (having fallen back on native execution), without reporting any UB. It also appears to regard the subtraction of two integers derived from pointers to different objects as UB.

Other Test Suites The GCC torture test suite [FSF 2018a] has 1429 tests. Many of these tests were written pre-C89 in K&R style, which is not supported by Cerberus. We have converted all the files from K&R to ANSI syntax with `cproto` [Huang and Dickey 2018], and included a header file `cerberus.h`, which removes GNU attributes and replaces built-in types and functions (`__builtin_memset()` etc.) with standard library analogues. We additionally modified 103 files to make functions `void` or add a `return`. GCC is permissive w.r.t. that, but ISO and Cerberus forbid inconsistencies. We identified 237 tests that rely on GNU extensions and compiler flags (special GCC builtins, zero-length arrays, nested functions, `-fwrapv`, etc.), and 210 more use ISO features not supported by Cerberus (bitfields, `_Complex`, etc.). 8 tests simply call `abort()`, which fails the test. Of the remaining 974, Cerberus currently passes 948 (97.3%). Many of the failed tests are due to a bug in fancy struct/union initialisation (5 tests), or take too long to execute (11 tests).

The ITC Toyota benchmark [Shiraishi et al. 2015] aims to support quantitative comparison of static analysis tools. It consists of 1276 tests, half with defects and the other half without any defects (meant to detect false positives), in 51 files. We exclude 13 files which use features Cerberus does not support (mainly `pthreads` and floats), for a total of 526 non-defect tests, of which Cerberus runs 100% of them without errors, and 526 defect tests. Of the latter, we identify 160 that actually have defined behaviour w.r.t. ISO C. These “defects” include unused variables, integer sign lost because of unsigned cast, redundant conjunctions in if and while statements, and suchlike, and so

Defect Type	Cerberus			RV-Match	
	UB-defect	non-UB defect	non-defect	defect	non-defect
Static memory	66/67 (99%)	0	67/67 (100%)	100%	100%
Dynamic memory	81/82 (99%)	5	87/87 (100%)	94%	100%
Numerical	55/56 (98%)	46	102/102 (100%)	96%	100%
Resource management	54/61 (89%)	34	95/95 (100%)	93%	100%
Pointer-related	72/72 (100%)	10	82/82 (100%)	98%	100%
Inappropriate code	0/0 –	64	64/64 (100%)	0%	100%
Misc	16/28 (57%)	1	29/29 (100%)	63%	100%

Fig. 6. ITC Toyota benchmark results for Cerberus and RV-Match. For Cerberus, tests that use unsupported features were removed. The UB-defect column shows the number of "defect" tests that have UB, and the fraction that Cerberus detects. The non-UB-defect column shows the number of "defect" tests that are non-UB defects, for which Cerberus should not (and does not) report an error. The non-defect column are tests for which a tool should not report false positives. Technical difficulties prevented us running RV-Match on these tests, and so we just report the overall data from [Guth et al. 2016] rather than giving a similar breakdown.

are not things that Cerberus should detect. Cerberus currently detects 344 out of the remaining 363 (95%). Fig. 6 summarises these results. As measured with these tests, Cerberus and RV-Match have broadly comparable detection rates for undefined behaviour.

The KCC Example test suite [Hathhorn et al. 2015; KCC 2018] is, similarly to ITC Toyota, split between good tests and bad tests (UB, constraint violation and implementation defined), 409 in total, of which we support 293. Cerberus currently gets 94% of the latter correct. Of the 18 failures, 13 are various minor front-end bugs.

We have also tried 1191 Csmith [Regehr et al. 2012] tests, including those from Memarian et al. [2016], and excluding tests that time out on Clang after 5s. They range from `-max-expr-complexity` 1 to 4 and are 40-800 lines long. Cerberus currently gives the same results as Clang for 1190 tests; the remaining one times out after ten minutes.

Performance To give a very crude measure of performance, for tests that do not involve the memory layout model in interesting ways, we ran a small loop with varying numbers of iterations (on an i5-4670 CPU at 3.40GHz). This takes 0.15s for 1000 iterations, 1.27s for 10 000, and 12.7s for 100 000, apparently scaling linearly. Guth et al. [2016] report a compile-and-run time of 13s (on an 2.4 GHz Intel Xeon) for 10 000 iterations.

```
int ret = 0; for (int i = 0; i <= [...]; i++) { ret++; }
```

More importantly, the time needed to run most of the above test suites is quite reasonable: 22.5s to run our pointer provenance tests and many others (189 in total); 3 minutes to run the GCC Torture tests; and 25 minutes for those Csmith tests. The Toyota tests are combined into two large files (for the defect and non-defect tests); running all the non-defect tests takes 20s.

Compilation to OCaml The Cerberus architecture, factoring its C semantics via a relatively straightforward Core language and with a clean memory object model interface, can be exploited to implement other execution or analysis engines. As an experiment, we have also implemented a transcompiler from C to OCaml. It uses the Cerberus frontend, elaboration, and type-checking of Core, and its memory model, but compiles Core to OCaml rather than executing the Cerberus Core operational semantics as an interpreter. Initial testing suggests that this approach is around two orders of magnitude faster (it takes 10s to run 1 million iterations of the above program). However, this approach does not support all Cerberus's features: it cannot run in exhaustive mode and it detects fewer undefined behaviours.

10 EXPLORING EXISTING C CODE

The corpus of existing C code is the most important aspect of the de facto standard, and the hardest to change. It is also the hardest to investigate. Our previous surveys [Memarian et al. 2016; Memarian and Sewell 2016b] have explored what expert programmers believe about it, but ideally we would have solid data about the code itself, either via some static analysis (necessarily approximate) or a dynamic exploration (necessarily only covering the paths executed). Focussing on the latter, we cannot use existing mainstream C implementations, as they build in the assumption that the source does not exhibit UB. In principle we could use something like Cerberus, but, being aimed at a clear semantics rather than performance, that is not currently practical for large bodies of C; it also lacks features. Accordingly, we gather data from running C code in settings that approximate our provenance semantics, in several ways as below.

There is also useful data to be gained from the developers of bug-finding tools such as the address, memory, and undefined-behaviour sanitisers. These do not track pointer provenance or detect all UBs, but they do detect many cases that conventional implementations do not. To be usable, they have to avoid reporting what their clients would regard as false positives, irrespective of ISO. For example, they report that in real code they see many legitimate uses of pointers (for printing or comparison, not for access) after their object lifetimes, and that they believe it would not be viable to prohibit that [Kostya Serebryany, personal communication].

There is a delicate balance to be struck here: much C code contains bugs and poor practices, and a semantics and the standard should not generally be trying to legitimise those, but on the other hand there are non-ISO or ISO-unclear idioms that are used in essential and widespread ways in systems code, and these should not be regarded as unqualified undefined behaviour.

FreeBSD CHERI port annotation data CHERI [Chisnall et al. 2015; Watson et al. 2018, 2015; Woodruff et al. 2014] is an experimental architecture providing hardware support for fine-grained pointer-based memory protection and secure encapsulation. It has been developed as an extension of 64-bit MIPS, but similar features could be added elsewhere. CHERI-MIPS provides hardware *capabilities*, which consist of compressed 128-bit values including a base virtual address, an offset, a bound, and permissions. Additional tag memory, cleared by any non-capability writes, records whether each such unit of memory holds a valid capability. These and other features makes capabilities unforgeable by software: each capability must be derived from one of equal or larger rights. One can use capabilities instead of integers either to implement all pointers ("pure-capability" code) or selectively ("hybrid"). Pointer provenance is an essential element of CHERI's protections: pointers must be validly derived, and cannot be (for example) injected over the network, or improperly leaked between processes. Bounds set by the memory allocator prevent an improperly manipulated pointer from being used to access the wrong object. This is a broadly similar model to ISO C extended with our provenance semantics, except that where the latter has undefined behaviour for accesses without the right (ghost-state) provenance, CHERI guarantees to trap for accesses that are not via a valid (concrete-state) capability. CHERI C is stricter than our proposals in some ways: capabilities have to be suitably aligned, and cannot be copied bytewise.

In other work, we and others have ported a large body of C code to CHERI-MIPS, including the FreeBSD kernel (hybrid) and userspace (pure-capability), annotating per-file the kinds of changes needed. This allows us to understand the potential impact of adopting a provenance model (motivated here by pointer-aliasing concerns, but enforced in CHERI for security reasons) on a large and diverse C corpus. We have analysed the original plain-C idioms identified by this with respect to our provenance semantics. The main conclusion is that the port has been possible with significant effort but modest code change, with relatively few occurrences of idioms that are challenging w.r.t. provenance. The userspace part of the FreeBSD tree includes 824 UNIX programs and 198 libraries and 23000 .c and .h files (not all of which are compiled), of which 295 have been changed and annotated. We now consider some classes of changes:

In CHERI, pointers cannot be derived from integer values. 39 source files (tagged `pointer_as_integer`) fabricated pointers from integers, but almost all are just storing distinguished values to compare against. The only cases in which the result was actually used as a pointer are `mmap()` calls with `MAP_FIXED` for things like thread stacks.

Preserving pointers across stores and reloads from memory require use of tag-preserving types – pointers, or `intptr_t` – without which pointers will become non-dereferenceable. Memory copies may arise through explicit requests by the programmer – e.g. `memcpy` – but may also be implied in other operations, such as `qsort`, which is expected to preserve the dereferenceability of pointers stored in memory whose contents are being sorted. One `memcpy` implementation and three sort implementations were altered. Additionally, 9 files were altered to use an alternative hash implementation in place of Berkeley DB, which does not preserve alignment.

In a CHERI-aware OS, tags may be intentionally stripped during certain memory copies – for example, in the implementation of Inter-Process Communication (IPC) and loopback network communication, or when storing pointers into memory-mapped files. There seem to be no substantive uses of `%p` pointer IO, but pointers are sent over a socket between a parent and its `fork`'d child, a case beyond the scope of ISO C. 6 files were altered to handle two cases of this.

Similarly, pointer casts through integer types – often for pointer arithmetic – will discard capability metadata if not made through pointers or `intptr_t`. Using otherwise-unused bits of pointers (`pointer_bit_flags`) is indeed standard and common, with 11 occurrences. In 25 places, code uses integer arithmetic to check or adjust pointer alignment. And there is much code that expects `intptr_t` arithmetic to mirror pointer arithmetic.

Another example of provenance behavior in CHERI C is in pointer equality testing, which compares not just a capability's virtual address, but also bounds, permissions, and tag bit. This is a pragmatic choice arising from the desire that `realloc` can return a pointer with modified bounds yet an identical virtual address, and have the caller code use the newer rather than older pointer value. Using the incorrectly derived pointer will prevent access to newly available space beyond the bounds of the original pointer, as it will lead to a trap. However, this pointer equality semantics has a practical impact on other programmer-visible aspects of pointer comparison, in which the derivation of a pointer affects that comparison: a pointer one location past the end of a memory allocation will not compare as equal to a pointer to the next memory allocation, even though their virtual addresses are equal, due to differing bounds. There are many occurrences of `realloc()` (2104) and `reallocf()` (86), but most seem to be simple extensions of strings. Only 6 required adaption for CHERI, such as not deriving updated pointers to new allocations from old ones. Intraobject pointer arithmetic w.r.t. pointers to no-longer-live objects has to be supported (contrary to ISO) to handle some idioms for updating pointers post-`realloc()`. Apart from this, there seems to be basically no inter-object pointer arithmetic.

FreeBSD CHERI execution data Looking specifically at the construction of transiently out-of-bounds pointers, we instrumented the CHERI QEMU emulator to log every creation and modification of such a pointer at a capability type. This is possible without false positives on CHERI since capability manipulation must be performed using special instructions instead of integer instructions. Substantially out-of-bounds pointers are a particular concern in CHERI due to the use of bounds compressed relative to a pointer's virtual address: a pointer that goes too far out of bounds will no longer be representable, causing the tag bit to be cleared making the pointer un-dereferenceable.

We collected statistics for out-of-bounds pointer creation during a full run of the FreeBSD testsuite. We encountered 155214 pointers that were out-of-bounds by more that one byte. Of these, only 1205 pointed past the end. This indicates that despite construction of one-past-the-end pointers being legal, having pointers to before the object is more common in real-world code. We also found that 81% of these pointers are at most `sizeof(void*)` outsize of the bounds. While this is a surprising number of out-of-bounds pointers, there were at most 6 unique program counters generating these per process. One typical use of an out-of-bounds pointer is an idiom in which an array pointer is incremented prior to use within a loop, leading programmers to decrement the pointer below the lower bound prior to the loop. This idiom can be found in the zlib compression library, affecting dependent software such as gzip, OpenSSH, and libpng. The small number of locations creating out-of-bounds pointers suggests that the ISO C restrictions could probably be adhered to with only modest changes to the actual programs.

Shadow memory tool To test larger codebases against the semantics, we created a dynamic analysis tool using shadow memory to track the provenance of all pointers and integers. It is broadly similar to other dynamic analyses such as SoftBound [Nagarakatte et al. 2009] or Memcheck [Nethercote and Seward 2007], but it is implemented by source-to-source translation using CIL [Necula et al. 2002] (giving access to C source features). Pointers and integers carry metadata, as a one-byte provenance token analogous to the model's allocation ID. This allows integer types down to `char` to carry provenance information, while keeping shadow memory 1:1 with program memory size. The runtime is an extension of `liballocs` [Kell 2015], which supports querying for static, stack or heap objects at a given address. Provenance is propagated as in PVI, with small simplifications. The C library is not instrumented, but calls are wrapped to simulate propagation.

At every pointer dereference, the identity of the pointed-to allocation (queried from `liballocs`) is checked against the pointer's provenance (carried with it), which must match. False negatives, though unlikely, are possible because provenance tokens sometimes match coincidentally: because only 254 distinct tokens exist, and because each allocation's token is a deterministic function of its base address rather than being chosen at random (owing to limitations of metadata in `liballocs`). False positives arise from limitations dealing with C library functions (wrappers exist only for commonly used functions), and assembly code (for which the right semantics remain unclear). The tool's output has been manually validated against the relevant de-facto tests.

We tried the tool on ten C-language SPEC CPU2006 benchmarks, i.e. all those in pure C except `libquantum` (as CIL does not support C99 complex numbers). As expected, the majority (6) execute without provenance errors. In `mcf` the tool correctly aborts when offsetting is used to fix up pointers after a `realloc()`; these pointers retain the previous allocation's provenance. In `gcc` and `perlbench` custom allocators are used, which are not currently instrumented. In `h264ref` currently a callback from the C library (in `qsort()`) defeats current C library wrapping.

These benchmarks are a friendly case, since they are selected for portability and do little I/O. Most larger and more I/O-heavy workloads are beyond the tool's current robustness (especially its C library wrappers, and its slowdown factors). We have successfully run the Tiny C compiler `tcc` under the tool, and are currently improving it via testing on testing several other well-known codebases.

11 INTERACTING WITH ISO WG14 AND OTHER BODIES

The ISO C standard is maintained by its JTC1/SC22 WG14 committee [WG14 2017], with C++ maintained by WG21. It has produced several major versions: C90 (essentially identical to ANSI

C/C89), C99, C11, and C18, with various Technical Corrigenda in between. C18 is a mild revision of C11, and a future more substantial C2x revision is envisaged. WG14 meets twice per year, with attendees that typically include several members of major compiler groups, analysis tool vendors, and conformance suite vendors, around 15–25 people. C is relatively stable compared with many languages, with WG14 taking a largely conservative approach to changes. The standard is a prose document, and there seems little near-term prospect of replacing it as a whole with a normative mathematically rigorous definition: even if a complete such definition existed, it would not be accessible to many of the target readership. Even a prose rendering of such would likely be too radical a change. (Boehm and Adve [2008], Batty et al. [2011], and the C++ concurrency subgroup did introduce what is essentially a mathematical definition for ISO C++/C11 concurrency, transcribed into prose, but that affected only a small part of the standard.) If one wants to improve the standard's memory object model, therefore, one has to convince WG14 that there is a real problem, work with them to develop a solution that will be broadly acceptable, and show that it can be expressed in the style of the current standard. Towards this end, we have attended several WG14 meetings, and intend to continue doing so. This is a lengthy ongoing process and the outcome is hard to predict, but at recent meetings it appears that many members of the committee agree that there are serious questions about the memory object model and that a provenance-based semantics of some kind could capture the right intent. WG14 has instituted a C Memory Object Model Study Group to address the question.

We have also had informal discussions with members of the LLVM and GCC communities and the ISO WG21 C++ committee. There too there seem to be grounds for cautious optimism: the provenance semantics, in one or other form, appears to be broadly consistent with people's intuition.

12 RELATED WORK

There is a long history of work to formalise aspects of the ISO C standards and to build memory object models for particular C-like languages. We discuss much of this in detail in [Chisnall et al. 2016, §10, p66–83], and [Krebbers 2015, Ch.10] gives a useful survey. Work to formalise aspects of the standards includes Batty et al. [2011]; Cook and Subramanian [1994]; Ellison and Roşu [2012]; Gurevich and Huggins [1993]; Guth et al. [2016]; Hathhorn et al. [2015]; Krebbers and Wiedijk [2012]; Krebbers [2013, 2014, 2015]; Krebbers and Wiedijk [2013, 2015]; Norrish [1998, 1999]; Papaspyrou [1998]. Memory object models include those for CompCert by Leroy et al. [2012]; Leroy and Blazy [2008] and Besson et al. [2014, 2015], for CompCertTSO by Ševčík et al. [2013], the model used for seL4 verification by Tuch et al. [2007], and the model used for VCC by Cohen et al. [2009]. Most of this work adopts either a completely concrete model (e.g. those of Tuch et al. and Ellison et al.) or a very abstract model (e.g. the initial CompCert model and the fine-grained effective types model of Krebbers and Wiedijk [2013]. Later work for CompCert adds support for some low-level idioms, but not the full gamut thereof in de facto C [Besson et al. 2014, 2015, 2017; Krebbers et al. 2014; Leroy et al. 2012]. Analysis tools such as tis-interpreter [Cuoq et al. 2017; TrustInSoft 2017] and CBMC [Kroening and Tautschnig 2014] also have to deal with much of the semantics of C, although with implicit rather than explicit semantic models, as did CIL [Necula et al. 2002]. Jones [1992, 2009] describes a Model Implementation C Checker for static and dynamic conformance checking w.r.t. C90. This was a commercial product and exactly what it checked w.r.t. the issues described in this paper is not completely clear. It consisted of "over 100 000 lines of C", rather than a precise mathematical semantics.

The most closely related work to Cerberus as a whole is the KCC work of Ellison and Hathhorn, et al. This has been developed into a commercial product, RV-Match [Guth et al. 2016; Runtime Verification Inc. 2017], which claims to be a "complete formal ISO C11 semantics". RV-Match aims to be *"an automatic debugger for subtle bugs other tools can't find, with no false positives"*. It covers some C features that Cerberus does not, but puts less emphasis on a clear semantics that (where possible) is tightly coupled to ISO, and the choices it makes for memory object semantics are not clear to us. KCC had a very different structure to Cerberus: a rewrite system over a very large state, rather than Cerberus's compositional elaboration to Core. §9 includes more detailed comparison with RV-Match on various test suites. The CH2O work of Krebbers and Wiedijk, cited above, includes a Coq semantics for a moderately large subset of C, together with impressive development of metatheory for reasoning within the semantics. This semantics takes a stricter view of some aspects than ISO, to ease reasoning. The CH2O code extracted from Coq can run some tests, but that semantics is not feature-complete enough to run the pointer semantics tests we develop here.

The most closely related work on the memory object model side is that of Kang et al. [2015]; Lee et al. [2018, 2017]. These study the complementary problem of the semantics for the LLVM intermediate language, focussed on the semantics implicitly assumed by compiler optimisations. We discuss the relationship to the most recent of these in §6. Preliminary investigation suggests that the C source semantics outlined in this paper and the intermediate-language semantics they propose could be made compatible (up to refinement), which would give a coherent top-to-bottom story for the C language and its implementation.

13 CONCLUSION

We have provided an in-depth discussion of the design space and two candidate semantics for pointers and memory objects in C, taking both ISO C and de facto usage into account; re-engineered and extended the Cerberus semantics to cover many of the other aspects of C; and integrated the two to provide a semantics that is executable as a test oracle in various ways. We intend to make Cerberus available in an open-source form in due course. The conflicting requirements imposed by different implementations and usages of C may mean that there cannot be a single universally acceptable semantics, but this provides a solid basis for discussion, clarifies what C is, and has a realistic prospect of incorporation into future versions of the ISO standard. It should help clarify exactly what envelope of behaviour it is desirable to permit compilers (and alias analysis) to have.

This also creates many possibilities for future work, e.g. using Cerberus as a basis for precisely defining other dialects of C (such as CHERI C), as a test oracle for compiler alias analysis, to support test coverage measurement w.r.t. the standard, and as a basis for static and dynamic analysis tools grounded on an explicit and well-validated semantics of C.

ACKNOWLEDGMENTS

We thank all those with which we have discussed C semantics, especially Frédéric Besson, Richard Biener, Chandler Carruth, David Chisnall, Pascal Cuoq, Hal Finkel, Jens Gustedt, Chung-Kil Hur, Ralf Jung, Robbert Krebbers, Chris Lattner, Juneyoung Lee, Xavier Leroy, Nuno Lopes, Justus Matthiesen, Paul McKenney, Santosh Nagarakatte, John Regehr, Martin Sebor, Kostya Serebryany, Richard Smith, Hubert Tong, Martin Uecker, Freek Wiedijk, Steve Zdancewic, the other members of the WG14 C standard committee, and other attendees at EuroLLVM and the GNU Tools Cauldron. Any errors remain our own, of course.

This work was partially supported by EPSRC grant EP/K008528/1 (REMS), an ARM iCASE award, and ERC Advanced Grant ELVER 789108. This work is part of the CTSRD and ECATS projects sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 and HR0011-18-C-0016. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

REFERENCES

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Programming Languages and Systems - 24th European Symposium on Programming, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015.* 283–307. https://doi.org/10.1007/978-3-662-46669-8_12

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proc. POPL.*

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2014. A Precise and Abstract Memory Model for C Using Symbolic Values. In *Proc. Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings.* 449–468. https://doi.org/10.1007/978-3-319-12736-1_24

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.* 67–83. https://doi.org/10.1007/978-3-319-22102-1_5

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.* 81–97. https://doi.org/10.1007/978-3-319-66107-0_6

Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *Proc. PLDI.* ACM, New York, NY, USA, 68–78. https://doi.org/10.1145/1375581.1375591

David Chisnall, Justus Matthiesen, Kayvan Memarian, Kyndylan Nienhuis, Peter Sewell, and Robert N. M. Watson. 2016. C memory object and value semantics: the space of de facto and ISO standards. http://www.cl.cam.ac.uk/~pes20/cerberus/notes30.pdf (a revison of ISO SC22 WG14 N2013).

David Chisnall, Colin Rothwell, Robert N. M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. 2015. Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. In *Proc. ASPLOS: the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, New York, NY, USA, 117–130. https://doi.org/10.1145/2694344.2694367

Ernie Cohen, Michał Moskal, Stephan Tobies, and Wolfram Schulte. 2009. A Precise Yet Efficient Memory Model For C. *Electron. Notes Theor. Comput. Sci. (SSV 2009)* 254 (Oct. 2009), 85–103. https://doi.org/10.1016/j.entcs.2009.09.061

Jeffrey V. Cook and Sakthi Subramanian. 1994. *A Formal Semantics for C in Nqthm.* Technical Report 517D. Trusted Information Systems, Oct. http://web.archive.org/web/19970801000000*/http://www.tis.com/docs/research/assurance/ps/nqsem.ps.

Pascal Cuoq, Loïc Runarvot, and Alexander Cherepanov. 2017. Detecting Strict Aliasing Violations in the Wild. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings.* 14–33. https://doi.org/10.1007/978-3-319-52234-0_2

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.* 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Chucky Ellison and Grigore Roşu. 2012. An executable formal semantics of C with applications. In *Proc. POPL.*

FSF. 2018a. GNU Compiler Collection, Torture Test Suite. https://github.com/gcc-mirror/gcc/tree/master/gcc/testsuite/gcc.c-torture/execute.

FSF. 2018b. Using the GNU Compiler Collection (GCC) / 4.7 Arrays and pointers. https://gcc.gnu.org/onlinedocs/gcc/Arrays-and-pointers-implementation.html. Accessed 2018-10-22.

Matt Godbolt. 2017. Compiler Explorer. https://godbolt.org/.

Yuri Gurevich and James K. Huggins. 1993. The Semantics of the C Programming Language. In *Selected Papers from the Workshop on Computer Science Logic (CSL '92).* Springer-Verlag, London, UK, UK, 274–308. http://dl.acm.org/citation.cfm?id=647842.736414

Dwight Guth, Chris Hathhorn, Manasvi Saxena, and Grigore Rosu. 2016. RV-Match: Practical Semantics-Based Program Analysis. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I.* 447–453. https://doi.org/10.1007/978-3-319-41528-4_24

Chris Hathhorn, Chucky Ellison, and Grigore Rosu. 2015. Defining the undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015.* 336–345. https://doi.org/10.1145/2737924.2737979

Chin Huang and Thomas E. Dickey. 2018. cproto. https://invisible-island.net/cproto/cproto.html.

Derek Jones. 1992. Applications POSIX.1 conformance testing. http://www.knosof.co.uk/poschk.html. Presented at the EurOpen & USENIX Spring 1992 Workshop/Conference.

Derek M. Jones. 2009. The New C Standard: An Economic and Cultural Commentary. http://www.knosof.co.uk/cbook/.

Jacques-Henri Jourdan and François Pottier. 2017. A Simple, Possibly Correct LR Parser for C11. *ACM Trans. Program. Lang. Syst.* 39, 4 (2017), 14:1–14:36. https://doi.org/10.1145/3064848

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*. 326–335. https://doi.org/10.1145/2737924.2738005

KCC. 2018. Example Test Suite. https://github.com/kframework/c-semantics/tree/master/examples/c.

Stephen Kell. 2015. Towards a Dynamic Object Model Within Unix Processes. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2015)*. ACM, New York, NY, USA, 224–239. https://doi.org/10.1145/2814228.2814238

Krebbers and Wiedijk. 2012. N1637: Subtleties of the ANSI/ISO C standard. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1637.pdf.

Robbert Krebbers. 2013. Aliasing Restrictions of C11 Formalized in Coq. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings, LNCS 8307*. 50–65. https://doi.org/10.1007/978-3-319-03545-1_4

Robbert Krebbers. 2014. An operational and axiomatic semantics for non-determinism and sequence points in C. In *Proc. POPL*.

Robbert Krebbers. 2015. *The C standard formalized in Coq*. Ph.D. Dissertation. Radboud University Nijmegen.

Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. 2014. Formal C Semantics: CompCert and the C Standard. In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. 543–548. https://doi.org/10.1007/978-3-319-08970-6_36

Robbert Krebbers and Freek Wiedijk. 2013. Separation Logic for Non-local Control Flow and Block Scope Variables. In *Proc. FoSSaCS*.

Robbert Krebbers and Freek Wiedijk. 2015. A Typed C11 Semantics for Interactive Theorem Proving. In *Proceedings of the 2015 Conference on Certified Programs and Proofs (CPP '15)*. ACM, New York, NY, USA, 15–27. https://doi.org/10.1145/2676724.2693571

Daniel Kroening and Michael Tautschnig. 2014. CBMC - C Bounded Model Checker - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. 389–391. https://doi.org/10.1007/978-3-642-54862-8_26

Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling High-level Optimizations and Low-level Code with Twin Memory Allocation. In *Proceedings of the 2018 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2018, part of SPLASH 2018, Boston, MA, USA, November 4-9, 2018*. ACM.

Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming undefined behavior in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 633–647. https://doi.org/10.1145/3062341.3062343

Xavier Leroy. 2009. A formally verified compiler back-end. *J. Automated Reasoning* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

Xavier Leroy et al. 2018. CompCert 3.4. http://compcert.inria.fr/.

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

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

Kayvan Memarian, Victor Gomes, and Peter Sewell. 2018. n2263: Clarifying Pointer Provenance v4. ISO WG14 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2263.htm.

Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N.M. Watson, and Peter Sewell. 2016. Into the depths of C: elaborating the de facto standards. In *PLDI 2016: 37th annual ACM SIGPLAN conference on Programming Language Design and Implementation (Santa Barbara)*. http://www.cl.cam.ac.uk/users/pes20/cerberus/pldi16.pdf PLDI 2016 Distinguished Paper award.

Kayvan Memarian and Peter Sewell. 2016a. N2090: Clarifying Pointer Provenance (Draft Defect Report or Proposal for C2x). ISO WG14 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2090.htm.

Kayvan Memarian and Peter Sewell. 2016b. What is C in practice? (Cerberus survey v2): Analysis of Responses – with Comments. ISO SC22 WG14 N2015, http://www.cl.cam.ac.uk/~pes20/cerberus/analysis-2016-02-05-anon.txt.

Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: reusable engineering of real-world semantics. In *Proceedings of ICFP 2014: the 19th ACM SIGPLAN International Conference on Functional Programming*. 175–188. https://doi.org/10.1145/2628136.2628143

Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for C. In *Proc. PLDI.*

George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proc. CC.*

Nicholas Nethercote and Julian Seward. 2007. How to Shadow Every Byte of Memory Used by a Program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE '07).* ACM, New York, NY, USA, 65–74. https://doi.org/10.1145/1254810.1254820

Michael Norrish. 1998. *C formalised in HOL.* Technical Report UCAM-CL-TR-453. University of Cambridge, Computer Laboratory. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-453.pdf

Michael Norrish. 1999. Deterministic expressions in C. In *Proc. ESOP, 8th European Symposium on Programming, LNCS 1576.* Springer-Verlag, 147–161.

Nikolaos S. Papaspyrou. 1998. *A formal semantics for the C programming language.* Ph.D. Dissertation. National Technical University of Athens.

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44. https://doi.org/10.1145/345099.345100

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case Reduction for C Compiler Bugs. In *Proc. PLDI.*

Runtime Verification Inc. 2017. RV-Match. https://runtimeverification.com/match/.

Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. 2015. Test suites for benchmarks of static analysis tools. In *2015 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Gaithersburg, MD, USA, November 2-5, 2015.* IEEE Computer Society, 12–15. https://doi.org/10.1109/ISSREW.2015.7392027

`glibc`. 2018. `memcpy`. https://sourceware.org/git/?p=glibc.git;a=blob;f=string/memcpy.c;hb=HEAD.

TrustInSoft. 2017. `tis-interpreter`. http://trust-in-soft.com/tis-interpreter/. Accessed 2017-11-11.

Harvey Tuch, Gerwin Klein, and Michael Norrish. 2007. Types, bytes, and separation logic. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007.* 97–108. https://doi.org/10.1145/1190216.1190234

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (June 2013).

Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Sewell, Stacey Son, and Hongyan Xia. 2018. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7).* Technical Report UCAM-CL-TR-927. University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.html

Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015.* 20–37. https://doi.org/10.1109/SP.2015.9

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

WG14. 2011a. ISO/IEC 9899:2011.

WG14. 2011b. *Programming Languages — C.* ISO/IEC 9899:201x. Committee draft, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

WG14. 2017. JTC1/SC22/WG14 – C. http://www.open-std.org/jtc1/sc22/wg14/.

Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: revisiting RISC in an age of risk. In *ISCA '14: Proceeding of the 41st International Symposium on Computer Architecture.* IEEE Press, Piscataway, NJ, USA, 457–468. https://doi.org/10.1145/2678373.2665740