VIP: Verifying Real-World C Idioms with Integer-Pointer Casts

RODOLPHE LEPIGRE, MPI-SWS, Germany
MICHAEL SAMMLER, MPI-SWS, Germany
KAYVAN MEMARIAN, University of Cambridge, UK
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands
DEREK DREYER, MPI-SWS, Germany
PETER SEWELL, University of Cambridge, UK

Systems code often requires fine-grained control over memory layout and pointers, expressed using low-level (*e.g.*, bitwise) operations on pointer values. Since these operations go beyond what basic pointer arithmetic in C allows, they are performed with the help of *integer-pointer casts*. Prior work has explored increasingly realistic memory object models for C that account for the desired semantics of integer-pointer casts while also being sound w.r.t. compiler optimisations, culminating in PNVI-ae-udi, the preferred memory object model in ongoing discussions within the ISO WG14 C standards committee. However, its complexity makes it an unappealing target for verification, and no tools currently exist to verify C programs under PNVI-ae-udi.

In this paper, we introduce VIP, a new memory object model aimed at supporting C verification. VIP sidesteps the complexities of PNVI-ae-udi with a simple but effective idea: a new construct that lets programmers express the intended provenances of integer-pointer casts explicitly. At the same time, we prove VIP compatible with PNVI-ae-udi, thus enabling verification on top of VIP to benefit from PNVI-ae-udi's validation with respect to practice. In particular, we build a verification tool, RefinedC-VIP, for verifying programs under VIP semantics. As the name suggests, RefinedC-VIP extends the recently developed RefinedC tool, which is automated yet also produces foundational proofs in Coq. We evaluate RefinedC-VIP on a range of systems-code idioms, and validate VIP's expressiveness via an implementation in the Cerberus C semantics.

CCS Concepts: • **Theory of computation** → **Operational semantics**; **Program verification**; **Separation logic**; • **Software and its engineering** → **General programming languages**.

Additional Key Words and Phrases: C programming language, memory model, pointer provenance, separation logic, proof automation, Iris, Coq.

ACM Reference Format:
Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. 2022. VIP: Verifying Real-World C Idioms with Integer-Pointer Casts. *Proc. ACM Program. Lang.* 6, POPL, Article 20 (January 2022), 32 pages. https://doi.org/10.1145/3498681

1 INTRODUCTION

Many forms of systems software—operating systems, hypervisors, device drivers, programming language runtimes, *etc.*—crucially rely on fine-grained control over memory and data layout. Such

Authors' addresses: Rodolphe Lepigre, MPI-SWS, Saarland Informatics Campus, Germany, lepigre@mpi-sws.org; Michael Sammler, MPI-SWS, Saarland Informatics Campus, Germany, msammler@mpi-sws.org; Kayvan Memarian, University of Cambridge, UK, kayvan.memarian@cl.cam.ac.uk; Robbert Krebbers, Radboud University Nijmegen, The Netherlands, mail@robbertkrebbers.nl; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org; Peter Sewell, University of Cambridge, UK, peter.sewell@cl.cam.ac.uk.

2475-1421/2022/1-ART20
https://doi.org/10.1145/3498681

control in turn sometimes demands the use of idioms that involve low-level arithmetic on pointer values, going beyond simple array-indexing calculations. For example:

(1) Pointer values often need to be rounded to a specific alignment, *e.g.*, in memory allocators.
(2) Language runtimes often employ pointer tagging, exploiting otherwise-unused bits in the representation of pointers to store metadata, *e.g.*, garbage collection flags.
(3) Language runtimes also often rely on otherwise-unused bits to encode both pointer values and bounded integers within single machine words, without indirection.
(4) Memory allocators often perform bitwise arithmetic on addresses to traverse an in-memory data structure, *e.g.*, an xor is used to find the "buddy" of a memory block in a buddy allocator.

In this paper, we explore the problem of how to verify the correctness of C programs that use such idioms. In C, these operations cannot be performed directly on pointer values. Instead, they are expressed by casting pointers to integers, and performing arbitrary operations on those raw integers, before casting them back to pointers. Hence, before we can talk about verifying programs that use these idioms, we must first ask a more basic question: what is the "right" *semantics* of memory and pointers in the presence of *integer-pointer casts*?

1.1 Existing Semantics for Pointers and Integer-Pointer Casts

Designing a memory object model for C supporting integer-pointer casts is notoriously difficult because it requires one to reconcile two key design goals:

(1) Supporting common C programming idioms involving pointers, including non-trivial arithmetic via integers as in the examples above; manipulating the representation bytes of structures involving pointers; and applying the address-of operator (&) to local variables.
(2) Being sound with respect to common compiler optimizations based on alias analysis.

At one extreme are *abstract models* like that of CompCert [Leroy and Blazy 2008; Leroy et al. 2012], in which pointers are represented as a pair of an abstract allocation identifier and an offset in the corresponding allocation block, or still more abstract models that add subobject structure [Krebbers 2015]. On the plus side, abstract models are well behaved with respect to compiler optimizations and correctness proofs thereof. For example, the CompCert model validates the constant propagation optimization across functions, and it also supports taking addresses of local variables. However, abstract models can only support restricted forms of conversions between pointers and integers—typically, round-trip casts—which are not sufficient to handle the idioms we target in general. It is possible to work around this limitation by restricting to certain coding patterns. For example, CertiKOS [Gu et al. 2015, 2018, 2019] and SeKVM [Li et al. 2021] model all allocatable data structures as a large array (sharing a single allocation identifier), and can thus replace some operations that usually require integer-pointer casts by simple pointer arithmetic, and Wang et al. [2019a] use hardwired support for tag bits. However, in this paper, we target real-world code that does not generally adhere to such coding patterns.

At the other extreme are *concrete models*, in which pointer values are simple machine words or integers, *e.g.*, as used in the C semantics of Norrish [1998] in HOL, seL4 [Tuch et al. 2007; Klein et al. 2009, 2014], and the early work on separation logic [O'Hearn et al. 2001; Reynolds 2002]. Concrete models trivially support integer-pointer casts with arbitrary arithmetic, but they are not sound in general with respect to some common compiler optimisations, *e.g.*, those that assume that pointers with different provenances cannot alias. A concrete model with addressable local variables and deterministic allocation cannot even validate optimisations that move local variable accesses around function calls. Concrete models with nondeterministic allocation can, but at the cost of having to reason about undefined behaviour in families of executions. The work on seL4 sidesteps these issues by using translation validation to guarantee that the particular program in

question is compiled soundly using GCC [Sewell et al. 2013] (GCC's optimisations are not sound with respect to the concrete model in general, but such unsoundness does not arise on seL4's code), and by restricting to a dialect of C in which one cannot take addresses of local variables.

To overcome this apparent dichotomy, much prior work has focused on designing "Goldilocks" memory models that are neither too concrete nor too abstract but just right [Gustedt et al. 2020; Memarian et al. 2019; Lee et al. 2018; Kang et al. 2015; Besson et al. 2015, 2014; Krebbers et al. 2014; Sevcík et al. 2013] (see §8 for a detailed discussion).

Among them, PNVI-ae-udi [Memarian et al. 2019] stands out as the most realistic to date. It has been validated against examples capturing a number of *de facto* coding idioms and compared against compiler behaviour, and it is the currently preferred memory object model in ongoing discussions within the ISO WG14 C standard committee, which is working towards an ISO Technical Specification [Gustedt et al. 2020]. This may later be incorporated into the C standard, which historically has not included a well-specified memory object model.

Unfortunately, PNVI-ae-udi's complexity makes it an unappealing target for verification, and there is currently no verification tool for C supporting it. The model is designed to "just work" for a wide variety of existing C idioms, without requiring any code changes. That is essential for uptake within the C standard, but it requires intricate semantic machinery. In PNVI, pointers are represented as a pair of an abstract provenance and a concrete address. The provenance is used on memory accesses to check that the involved memory is in bounds of the corresponding allocation, and that the allocation is still live. Pointer-to-integer casts simply drop the provenance information. The complexity arises in the handling of integer-to-pointer casts, to ensure that these result in a pointer with an appropriate provenance in the right cases. When an integer is cast into a pointer, PNVI-ae-udi attempts to reconstruct a valid provenance by looking for a suitable allocation. Only allocations whose provenance has previously been *exposed* are considered for this, where a provenance is exposed when a pointer to the corresponding allocation is cast to an integer.

So far, this might be manageable, but further subtleties arise from the need to support accesses to the representation bytes of pointers; the potential ambiguity of an integer address that is both one-past one allocation and at the start of another; and the need to correctly identify reads of pointer values that are (partially) from some representation-byte writes (not from an entire pointer-value write), which should be treated similarly to integer-to-pointer casts. Together, these add considerable complexity.

1.2 VIP: A Practical Approach to Verifying C Programs Under PNVI Semantics

In this paper, we build a verification tool for C programs that supports low-level integer-pointer cast idioms, as needed by systems software, and that ensures correctness under the PNVI-ae-udi semantics, thus inheriting its validation with respect to programming practice.

The key insight behind our approach is that, while PNVI-ae-udi had to support code in the wild unchanged, for a verification tool one expects in any case that the user may have to annotate the program with additional information when using tricky language features. Moreover, in the idioms we aim to support, there is always a pointer in scope at the integer-to-pointer cast with the right provenance (there are other cases where that does not hold, *e.g.*, xor linked lists or pointer hardening schemes used by certain allocators, but they appear to be much less common in practice). We can therefore rely on the user making their intent with respect to pointer provenance explicit, instead of the subtle and complex PNVI-ae-udi machinery that aims to just "get it right", without programmer guidance, in sufficiently many cases.

Specifically, we build our tool atop a new and simpler memory model—called VIP, for Verification-oriented model for Integer-Pointer casts—which we have designed to be more amenable to verification than PNVI-ae-udi, while still being compatible with it. The main idea of VIP is to provide

the user with a new construct, `copy_alloc_id`, by means of which they can explicitly annotate integer-to-pointer casts with a pointer value from which to copy the provenance.

This simple idea sidesteps the PNVI-ae-udi complexity: VIP enables us to factor the problem of verifying C programs that will be compiled by conventional compilers, assumed sound w.r.t. PNVI-ae-udi, into two, more tractable, problems.

(1) We first build a program logic that is sound under VIP and deploy it in RefinedC [Sammler et al. 2021], a recently-developed verification framework for C that is both *automated* (via a refined ownership type system) and *foundational* (producing machine-checked proofs in Coq). RefinedC originally relied on an abstract, CompCert-like memory model, which did not support integer-pointer casts. We lift this limitation by adapting RefinedC to be sound w.r.t. VIP, and evaluate it on various real-world idioms. This demonstrates that VIP is a suitable basis for verification tools supporting integer-pointer casts.
(2) We prove the VIP semantics sound w.r.t. an existing formalisation of PNVI-ae-udi, meaning that any behaviour observable under PNVI-ae-udi is also observable under VIP, and hence that programs verified w.r.t. VIP execute correctly on PNVI-ae-udi-compliant implementations. For this we map `copy_alloc_id` into a combination of casts mirroring its provenance manipulation (though one can also imagine built-in implementations in future compilers).

Put together, we obtain a new verification tool, RefinedC-VIP, which enables automated and foundational verification of C programs involving integer-pointer casts, and which is sound under the PNVI-ae-udi memory model.

Contributions.

- VIP (§3): a new verification-oriented C memory model supporting integer-to-pointer casts by combining concrete addresses and abstract provenances, and allowing pointer provenance to be explicitly copied from any pointer value on an integer-to-pointer cast.
- A proof (§5) that VIP is a sound abstraction of PNVI-ae-udi (§4) [Gustedt et al. 2020; Memarian et al. 2019]: a realistic memory model that has been extensively validated as a potential update to the ISO C standard and against de facto practice [Memarian et al. 2016].
- An integration of our memory model into the RefinedC verification framework in Coq [Sammler et al. 2021] to obtain the first (foundational and automated) program logic for C equipped to handle arbitrary pointer arithmetic via casts to integers and back (§6).
- An evaluation of the verification capabilities of VIP through a variety of examples inspired or directly taken from real-world C code (§7.1), including a simple allocator from pKVM, a hypervisor being developed by Google [Deacon 2020; Edge 2020].
- An empirical verification of the compatibility of VIP with PNVI-ae-udi and an evaluation of the annotations required by VIP (§7.2) by running an implementation of VIP in Cerberus [Memarian et al. 2019] on the test suite of Gustedt et al. [2020, Annex A.6].

Supplementary material. The supplementary material includes the detailed definitions and proofs for §3, §4, and §5 [Lepigre et al. 2021a], as well as the source code of RefinedC-VIP, our extension of Cerberus, and examples discussed in §7 [Lepigre et al. 2021b].

Limitations. (1) The `copy_alloc_id` instruction of VIP can only be used when a pointer with the expected provenance is available at the point of the (non-round-trip) integer-to-pointer cast. However, that seems to be the case for most practically important idioms. (2) RefinedC-VIP inherits all limitations of Cerberus and RefinedC. For instance, it does not support floating point numbers or bit fields. (3) While RefinedC-VIP allows the verification of concurrent programs, we ignore concurrency in our soundness proof as PNVI-ae-udi does not explicitly support it. However, we expect concurrency to be orthogonal to integer-pointer casts, which are our focus here. (4) The

```
1  #include <stdint.h>
2  //#include <refinedc.h> // uncomment for VIP
3  #define TAG_SIZE 3ULL
4  #define TAG_MOD  (1ULL << TAG_SIZE)
5  #define TAG_MASK (TAG_MOD - 1)
6
7  unsigned char tag_of(void* p){
8    uintptr_t i = (uintptr_t) p;
9    unsigned char t = i & TAG_MASK;
10   return t;
11 }
12
13 void* tag(void* p, unsigned char t){
14   uintptr_t i = (uintptr_t) p;
15   uintptr_t new_i = (i & ~TAG_MASK) | t;
16   void *q = (void*) new_i; // original code; comment for VIP
17   //void *q = copy_alloc_id(new_i, p); // uncomment for VIP
18   return q;
19 }
20
21 void* untag(void* p){
22   return tag(p, 0); }
23
24 // What follows is client code.
25 // It is not part of the library.
26
27 #include <stddef.h>
28
29 void client() {
30   size_t x = 0;
31
32   void *tp = tag(&x, 1);
33   assert(tag_of(tp) == 1);
34
35   size_t *px = (size_t*) untag(tp);
36   assert(*px == 0);
37 }
```

Fig. 1. Implementation of a pointer tagging library together with a simple client function. The version that we verify using RefinedC-VIP uses the `copy_alloc_id` construct. It is obtained by uncommenting Line 2, and by replacing Line 16 with an uncommented version of Line 17.

`copy_alloc_id` instruction would plausibly be useful for general systems programming (not just for C written for verification), to let compilers deduce stronger facts about the lack of aliasing on the resulting pointer value. However, we do not evaluate this here. (5) VIP and PNVI-ae-udi do not consider sub-object bounds or out-of-memory conditions. We ignore both throughout.

2 KEY IDEAS ILLUSTRATED WITH POINTER TAGGING

In this section we use a pointer tagging example (§2.1) to highlight the differences between VIP and the concrete, abstract, and PNVI-ae-udi models (§2.2). We demonstrate how to verify this example in RefinedC-VIP (§2.3), and then discuss the special case of round-trip casts (§2.4). We conclude by giving high-level intuitions for how we prove VIP to be a sound abstraction of PNVI-ae-udi (§2.5).

2.1 Implementation of a Pointer Tagging Library

Pointer tagging—storing metadata in unused bits of pointers—is a common use of integer-pointer casts. Figure 1 shows a pointer tagging library that assumes 8-byte alignment, so tags can be stored in the low-order three bits of pointer values, as per the `TAG_SIZE` macro on Line 3.

The library provides three functions: `tag_of(p)`, to read a tag from a pointer; `tag(p,t)`, to tag or re-tag a pointer; and `untag(p)`, to clear the tag of a pointer. The `client` function on the bottom right gives an example usage of the library. The variable `x` declared at Line 30 has type `size_t`, which (on the architecture we consider) guarantees that it is allocated at an address that is a multiple of `8`. As a consequence, a tagged pointer `tp` can be constructed from the address of `x` on Line 32 by calling `tag` with `&x` and a tag value smaller than `8`. On Line 33, the `client` function asserts that the tag of `tp` has the expected value `1`. Then, on Line 35, the tagged pointer `tp` is turned back into a usable untagged pointer `px` by calling `untag` to clear the tag. Finally, the assertion on Line 36 checks that `px` can indeed be used to access the value stored in `x`, which is still `0`.

Looking at the implementation of the library functions, `untag(p)` (Line 21) simply clears the tag bits of pointer `p`, using function `tag` with tag value `0`. The interesting functions are `tag_of` and `tag`,

since they directly involve integer-pointer casts. The function `tag_of(p)`, defined on Line 7, first casts its pointer argument `p` into an integer `i` on Line 8, and then uses a bit-masking operation to compute the tag value `t` from `i` on Line 9. The tag `t` is then returned on the next line. The function `tag(p)` (Line 13), converts the (possibly tagged) pointer `p` it receives as argument into an integer `i` (Line 14), then performs bitwise operations on `i` to obtain a tagged address `new_i` (Line 15), and finally casts `new_i` into a pointer `q` (Line 16), which is then returned.

2.2 The Example Under Various Memory Models

We now consider the execution of our `client` function in the main memory object models mentioned in the introduction. The role of a memory object model is to give a precise semantics to the various operations manipulating pointers (*e.g.*, pointer arithmetic and integer-pointer casts) and accessing the memory (*e.g.*, allocations, loads, and stores).

Concrete model. Under a concrete model, pointers are represented as concrete addresses (here 64-bit integers), and the memory itself is modeled as a finite partial map from addresses to bytes (8-bit integers). Going back to the `client` function of Figure 1, consider what happens when variable `x` (declared at Line 30) is allocated and initialised. First, a suitable 8-byte-aligned address, say `0b1001001000`, is picked for `x`, avoiding other live allocations (we ignore memory exhaustion). Variable `x` is then initialised by mapping the corresponding eight bytes of the memory to `0`, which amounts to storing the representation bytes of the `size_t` integer `0`. Other allocations, stores, loads, and deallocations (performed when the function returns) have similar semantics, manipulating the bytewise representation state. These operations match the low-level intuitions that many programmers have about memory—though the concrete model also allows operations that will not be compiled soundly w.r.t. this model by conventional aliasing-aware compilers.

Going back to the `client` function of Figure 1, let us see what happens on integer-pointer casts by considering the call to `tag(&x, 1)` on Line 32. First, note that the last three bits of the address `0b1001001000` (*i.e.*, the value of `&x`) are all `0`, since it is a suitably aligned (yet untagged) pointer. In the concrete model, all integer-pointer casts—including those in the implementation of `tag` on Line 14 and Line 16—are no-ops. As a consequence, the bitwise operations on Line 15 directly work on the address of `x`, and produces the address `0b1001001001` returned by the function, whose tag bits are `001`. The rest of the evaluation goes on similarly, and both assertions in `client` succeed.

Abstract model. Let us now consider the other side of the spectrum and see how this example is handled by abstract memory models like that of CompCert [Leroy and Blazy 2008; Leroy et al. 2012] or the original version of RefinedC [Sammler et al. 2021]. In such a model, pointers are represented as a pair of a block identifier (typically an integer), and an offset in the corresponding block. Memory is modeled as a finite partial map, associating block-offset pairs to either bytes (8-bit integers) or to pointer fragments (*e.g.*, byte n of some pointer value). In other words, abstract models essentially view the memory as a collection of disjoint memory blocks without concrete addresses. Moreover, since the representation bytes of pointers are abstract pointer fragments, interpreting them as integers leads to undefined behaviour.

In detail, declaration of the variable `x` on Line 30 leads to the allocation of a new abstract memory block. Depending on the model, this abstract representation leads to undefined behaviour in different ways: The original RefinedC memory model does not support any form of integer-pointer casts and thus the pointer-to-integer cast on Line 14 raises undefined behaviour. The CompCert model supports round-trip casts of pointers through integers by allowing integer variables to contain pointer values (*i.e.*, integer-pointer casts are defined as no-ops). However, all arithmetic operations on integer variables containing pointers lead to undefined behaviour. Thus, in this example, such a model would raise undefined behaviour on Line 15.

PNVI-ae-udi. PNVI-ae-udi combines concreteness and abstraction by representing pointers as pairs (π, a), where π is an abstract *provenance* (either an *allocation ID*, similar to the block identifiers of an abstract model, or a *symbolic provenance*), and a is a concrete address (similar to those of concrete models). Memory is modeled using three maps: (1) a map from addresses to bytes and some metadata, (2) a map from allocation IDs to allocation metadata recording the base address, the size, and whether the provenance has been exposed, and (3) a map associating symbolic provenances to either one or two allocation IDs (see §4 for more details).

Returning to the `client` function, the variable `x` declared at Line 30 is allocated at a location (π, a), where $\pi = @i$ for a fresh allocation identifier i, and a is a concrete 8-byte-aligned address. For simplicity, let us assume that a is `0b1001001000`, as we did for the concrete model. The interesting parts of the execution are the calls to the `tag` function on Line 32 and (indirectly) on Line 35. When evaluating `tag`, its argument pointer `p` is first cast into an integer on Line 14. In PNVI-ae-udi, this operation is simple: the concrete address a is used as the integer value, and the provenance π is marked as exposed (*i.e.*, its address has been observed). Thus, `i` is assigned the integer value `0b1001001000`. Then, the statement on Line 15 evaluates as under a concrete model: during the call to `tag(&x, 1)`, the value assigned to variable `new_i` is `0b1001001001` (with tag bits `001`).

Now, the tricky part under PNVI-ae-udi is to find a suitable provenance for the pointer with address `new_i`, created using `(void*) new_i` on Line 16. Given the intent of the library, we would expect this pointer to be assigned the same provenance as argument `p`. However, nothing in the code makes this intent of the programmer explicit, and thus PNVI-ae-udi uses complex machinery to find the right provenance. In particular, when casting an integer into a pointer, PNVI-ae-udi considers the provenances of any alive and exposed allocations containing the address. In our example, π is one such provenance since it is alive, it was exposed by the cast `(uintptr_t) p` on Line 14, and `new_i` is still in-bounds of the original allocation. If π is the only option, then PNVI-ae-udi uses it for the created pointer. However, if: (1) `new_i` is the address of the first byte of the allocation with provenance π, (2) `new_i` is *also* the address one past the end of another live allocation with provenance π' (implying that π' immediately precedes π in memory), and (3) π' is exposed, then both π and π' are valid choices for the pointer created on Line 16. In this case, PNVI-ae-udi relies on a *symbolic* provenance to account for both options at once (recorded in map (3) of the memory representation). This symbolic provenance is collapsed to a single option if and when some pointer operation is done that would otherwise be undefined behaviour. In our example, the call to `untag` on Line 35 can result in such a symbolic provenance (if conditions (2) and (3) above hold). The ambiguity is resolved when `px` is dereferenced on Line 36. For the access to be valid, the provenance of `px` must be resolved to that of `x` since dereferencing a one-past-the-end pointer is invalid.

Symbolic provenances allow PNVI-ae-udi to accept many programs without modification, but come with serious downsides making it unappealing for program verification. First, all operations on pointers need to take symbolic provenances into account, which can leads to significant complexity and subtle corner cases (*e.g.*, offset operations collapse a symbolic provenance if and only if the offset is not zero). Second, symbolic provenances can appear unexpectedly since integer-to-pointer casts are non-modular (*i.e.*, they can be influenced by exposed allocations of unrelated code, as we have seen in the example). Third, for verified code, symbolic provenances needlessly complicate reasoning about integer-to-pointer casts since the programmer usually knows the provenance the resulting pointer should have.

VIP. The key idea making our VIP memory model more suitable than PNVI-ae-udi for verification is that it lets the programmer explicitly say which provenance should be used for the results of integer-to-pointer casts, thus avoiding the complexities of symbolic provenances. To this end, VIP introduces an instruction `copy_alloc_id(i, p)` behaving essentially like `(void*)i`, but assigning

```
1  [[rc::parameters("r: {loc*Z}", "t: Z", "ty: type")]]
2  [[rc::args("r @ &tagged<TAG_MOD, ty>", "t @ int<u8>")]]
3  [[rc::requires("{0 ≤ t < TAG_MOD}")]]
4  [[rc::requires("[type_alive_own ty]")]]
5  [[rc::returns("{(r.1, t)} @ &tagged<TAG_MOD, ty>")]]
6  void* tag(void* p, unsigned char t);
7
8  [[rc::parameters("r: {loc*Z}", "ty: type", "v: val")]]
9  [[rc::args("at_value<v, r @ &tagged<TAG_MOD, ty>>")]]
10 [[rc::requires("[type_alive_own ty]")]]
11 [[rc::returns("{r.2} @ int<u8>")]]
12 [[rc::ensures("v: r @ &tagged<TAG_MOD, ty>")]]
13 [[rc::ensures("{0 ≤ r.2 < TAG_MOD}")]]
14 unsigned char tag_of(void* p);
15
16 [[rc::parameters("r: {loc*Z}", "ty: type")]]
17 [[rc::args("r @ &tagged<TAG_MOD, ty>")]]
18 [[rc::requires("[type_alive_own ty]")]]
19 [[rc::returns("{r.1} @ &own<ty>")]]
20 void* untag(void* p);
```

Fig. 2. RefinedC specifications for the functions of the pointer tagging library.

the provenance of pointer `p` to the resulting pointer. It allows programmers to make their intent clear via a small code modification. For example, the `tag` function is adapted by replacing the integer-to-pointer cast `(void*)new_i` on Line 16 with `copy_alloc_id(new_i, p)`.

With this modification, let us see how VIP handles the code of the `client` function. VIP is similar to PNVI-ae-udi in several ways (see §3 for details). Memory locations are represented in mostly the same way, as pairs of an abstract provenance and a concrete address. The memory is also modeled as in PNVI-ae-udi, with two modifications: (1) the metadata associated to allocations does not track their exposure, and (2) there is no symbolic provenance nor the associated map. Consequently, the evaluation proceeds roughly as under the PNVI-ae-udi semantics until Line 17, where `new_i` is cast into a pointer using `copy_alloc_id(new_i, p)`. As we already saw, this construct lets the user provide a pointer as second argument, and the provenance of this pointer is used for the newly created pointer. In other words—unlike under the PNVI-ae-udi semantics—pointers created by an integer-to-pointer cast have an explicit—and thus unambiguous—provenance. The most important consequence of this is that the VIP model more easily permits modular verification. Thanks to `copy_alloc_id`, `tag` can be verified without having to consider potentially ambiguous provenances.

2.3 Verification of a Pointer Tagging Library

We now explain how we can verify the pointer tagging library from Figure 1 using RefinedC-VIP—our new version of the RefinedC verification framework [Sammler et al. 2021] built atop VIP. RefinedC-VIP—like the original RefinedC—uses a type system featuring both ownership and refinement types to enable automated verification of C code. RefinedC-VIP extends the original RefinedC with support for integer-pointer casts and extends its standard library with types targeted at the verification of idioms relying on integer-pointer casts. In particular, RefinedC-VIP provides a `&tagged` type that allows one to generically account for the pointer tagging idiom for suitably aligned pointers. We used this type in the specification of our example library shown in Figure 2.

Annotations and types. Each function of the pointer tagging library is given a specification—or in other words, a RefinedC-VIP type—using annotations in the form of attributes. There are five different kinds of annotations here, and they are the same as in the original RefinedC except for the new type formers. They respectively specify a set of universally quantified parameters on which the specification depends (`rc::parameters`), the RefinedC types for the function arguments (`rc::args`) and for the return value (`rc::returns`), and possible pre- and post-conditions (`rc::requires` and `rc::ensures`). Given these annotations, together with the `copy_alloc_id` instruction in the implementation of the `tag` function, RefinedC-VIP is able to verify the bulk of the specification automatically, including all of the ownership reasoning. The remaining pure side-conditions (all

involving bitwise operators) are solved using two extra hints in the forms of lemmas to rewrite with. The verification process is modular: each function is type-checked independently by RefinedC.

We need to first introduce the RefinedC types used in the library. The most prominent type here is the new tagged pointer type `&tagged<n, ty>`, which is parameterised by an integer `n` and a RefinedC type `ty`. The parameter `n` tells us that the raw, untagged version of the considered pointer is a multiple of `n`, and hence that the tag is stored in its $\log_2(n)$ least significant bits. The parameter `ty` gives the type of the memory value pointed to by the raw pointer, together with the associated ownership. In our specifications, the value of `n` is fixed to be `8` through `TAG_MOD`, and the type `ty` is universally quantified using a parameter, which means that the library can be used at any type, although it need be verified only once. Now, additionally, it is important to note that `&tagged<n, ty>` is not just a type: it is a *refinement type*. A refinement is similar to a parameter, but it allows us to constrain the logical value attached to the elements of the type. Different refinement types have different types of refinements. For example, an integer type (*e.g.*, `int<u8>` in the type of the second argument of `tag`) is refined by a mathematical integer `t` (written `t @ int<u8>`), and an (untagged) pointer type (*e.g.*, `&own<ty>` in the return type of `untag`) is refined by a memory location. The refined tagged pointer type (written `r @ &tagged<n, ty>`) is refined by a pair `r` containing the untagged pointer value and the integer value of the tag.

Zooming in on the specifications. Let us now have a closer look at the specifications in Figure 2, starting with that of the `tag` function. It is parameterised by three universally quantified variables bound on Line 1: (1) a pair (`r`) of a location and an integer used as refinement for the type of the first argument of the function, (2) an integer `t` used similarly for the second argument, and (3) the type `ty` of the memory owned by the pointer. The first argument of the `tag` function given on the first part of Line 2—that is, the pointer to be tagged or re-tagged—is assigned the type `r @ &tagged<TAG_MOD, ty>`. Although the type of the argument is that of a tagged pointer, the type system also accepts any suitably-aligned owned pointer `&own<ty>` by subtyping. (And dually, subtyping can be used to turn a tagged pointer with tag value `0` into a standard owned pointer, which is useful to verify `untag` from the specification of `tag`.) The second argument of the function given on the second part of Line 2—the tag value to write—is assigned the type `t @ int<u8>`, that is, the type of an unsigned 8-bit integer (corresponding to C type `unsigned char`). On Lines 3-4 two preconditions are given for the function: `0 ≤ t < TAG_MOD` states that the tag integer `t` should be in the range of valid tag values, and `type_alive_own ty` proves that having a pointer to memory of type `ty` implies that the corresponding allocation is alive. The latter precondition is required since VIP—like PNVI-ae-udi—only allows pointer-to-integer casts for pointers that are alive. Note that this condition always trivially holds for common types like integers, owned pointers or structures containing only the former, since they directly own the necessary resources to prove the predicate. Finally, the last annotation given on Line 5 specifies that the return type of the function is a tagged pointer with refinement `(r.1, t)`. This means that the result has the same underlying raw pointer as the argument, but the tag is updated to `t`.

The specifications of `untag` and `tag_of` are similar, so we just point out a few key details. The return type of `untag` is a standard RefinedC owned pointer `{r.1} @ &own<ty>`, which is refined by the memory location taken from the refinement `r` of the tagged pointer argument. The type of the argument of `tag_of` is wrapped into the `at_value<v, ...>` type that gives the value of the argument the name `v`. This name is used in the postcondition on Line 12 to return the ownership of the tagged pointer that was passed in this argument.

Proof outline for the `tag` function. Having seen the specification of the library functions, let us now consider the proof outline of Figure 3, which sketches the process followed by the RefinedC-VIP automation to verify the `tag` function. At the beginning of the function, we can see

```
1  [[rc::parameters("r: {loc * Z}", "t: Z", "ty: type")]]
2  [[rc::args("r @ &tagged<TAG_MOD, ty>", "t @ int<u8>")]]
3  [[rc::requires("{0 ≤ t < TAG_MOD}", "[type_alive_own ty]")]]
4  [[rc::returns("{(r.1, t)} @ &tagged<TAG_MOD, ty>")]]
5  void* tag(void* p, unsigned char t){
6    // {p: r @ &tagged<TAG_MOD, ty>, t: t @ int<u8>, 0 ≤ t < TAG_MOD, type_alive_own ty}
7    uintptr_t i = (uintptr_t) p;
8    // {p: r @ &tagged<TAG_MOD, ty>, ..., i: (add_offset(r.1, r.2)) @ intptr<uintptr_t>}
9    // {p: r @ &tagged<TAG_MOD, ty>, ..., i: (r.1.1 + r.2) @ int<uintptr_t>}
10   uintptr_t new_i = (i & ~TAG_MASK) | t;
11   // {p: ..., ..., new_i: (((r.1.2 + r.2) & ~TAG_MASK) | t) @ int<uintptr_t>}
12   // {p: ..., ..., new_i: (r.1.2 + t) @ int<uintptr_t>}
13   void *q = copy_alloc_id(new_i, p);
14   // {p: r @ &tagged<TAG_MOD, ty>, ..., new_i: ..., q = val_of_loc((r.1.1, r.1.2 + t))}
15   // {p = val_of_loc(r.1), own r.1: ty, ..., new_i: ..., q = val_of_loc((r.1.1, r.1.2 + t))}
16   // {..., new_i: ..., q: ((r.1.1, r.1.2), t) @ &tagged<TAG_MOD, ty>}
17   // {..., new_i: ..., q: (r.1, t) @ &tagged<TAG_MOD, ty>}
18   return q;
19 }
```

Fig. 3. Informal Hoare-style proof outline for the verification of the `tag` function.

on Line 6 that the proof state holds the type assignments for the two arguments as well as the two preconditions. Then, at the point of the pointer-to-integer cast on Line 7, a typing rule for casts on type `&tagged` is applied. It consumes ownership of the type assignment for `p`, and gives it back together with a type assignment for the result of the cast. On Line 8, we can see that variable `i` is assigned type `(add_offset(r.1, r.2)) @ intptr<uintptr_t>`. It corresponds to an integer type whose underlying representation is a pointer value, and we will see how it is useful in §2.4. Note however, that it is refined by the actual pointer value stored in `p`, which is computed as an offset from the raw (untagged) pointer value with the current tag value. On Line 9, a subtyping rule is used to change the type of `i` into a standard integer, and the refinement is accordingly expressed with addition on integers (instead of pointer operation `add_offset`). Several typing rules for arithmetic operations on integers are then applied on Line 10, resulting in a type assignment of variable `new_i` on Line 11 with a corresponding sequence of mathematical operations. After several arithmetic simplifications, using the fact (derived from the type assignment of `p`) that `r.1.2` is a multiple of `8`, we finally obtain `(r.1.2 + t) @ int<uintptr_t>` as a type of `new_i` on Line 12.

At this point, all the arithmetic is done, and it remains to perform the integer-to-pointer cast of Line 13. To this end, the RefinedC-VIP typing rule for `copy_alloc_id` is applied, consuming the ownership associated to `new_i`, and giving it back when done. As we can see on Line 14, the rule provides us with a predicate `q = val_of_loc((r.1.1, r.1.2 + t))`, telling us that the value `q` is a pointer value whose provenance is taken from `r`, and hence that it is the same as that of the pointer the function received as argument. As expected, the address of `q` is also computed from that same pointer together with the new tag value `t`. Now, the last steps of the proof are all about transferring ownership from the type assignment of the argument `p`, to obtain a type assignment for `q`. On Line 15, the type assignment of `p` is decomposed into a definition for the value `p` and the actual ownership associated to the raw pointer. This ownership is then recomposed with the definition of the value `q` to obtain a type assignment for `q` on Line 16. And finally, the refinement is simplified on Line 17 to match the specified type for the return value of the function.

2.4 Round-Trip Casts

As we have seen in §2.2 and §2.3, the VIP model requires integer-to-pointer casts like `(void*)i` to be turned into `copy_alloc_id(i, p)`, where `p` is a pointer value holding a provenance that is suitable for the resulting pointer. However, there are situations where such a pointer `p` is not directly available. In particular, much systems C code uses pointers and integers interchangeably. Hence, it is common that, instead of a suitable pointer `p`, one only has access to an integer `ip` that was obtained from `p` using a pointer-to-integer cast. Two real-world examples of this are the early allocator of the pKVM hypervisor [pKVM developers 2020] (discussed in §7.1), which stores the address of its memory pool in an integer variable, and the OCaml language runtime, which encodes both (aligned) pointers and 63-bit integers in a 64-bit integer [Leroy and Doligez 1996].

In both examples, an integer `ip` obtained from a pointer-to-integer cast is available, and casting it back into a pointer would amount to completing a round-trip cast. To allow using such an integer `ip` as a source of provenance for `copy_alloc_id`, the VIP memory model—unlike PNVI-ae-udi—continues to track the provenance of a pointer in integers obtained from a pointer-to-integer cast until arithmetic is performed on it. As a consequence, a normal integer-to-pointer cast like `(void*) ip` can be used to obtain a valid pointer—with the initial provenance—which can in turn be used as second argument of `copy_alloc_id`.

2.5 Relating VIP and PNVI-ae-udi

As we have seen in the previous section, VIP and PNVI-ae-udi differ in four main ways. On the one hand, PNVI-ae-udi tracks (1) *ambiguities in provenance* and (2) *provenance exposure* whereas VIP does not. On the other hand, (3) VIP tracks provenance in integers (in a limited way, for round-trip casts) whereas PNVI-ae-udi does not, and (4) VIP relies on the `copy_alloc_id` primitive, which is not available in PNVI-ae-udi.

Despite these differences, our main theorem (Theorem 3 in §5) establishes that VIP can be soundly implemented atop PNVI-ae-udi. The proof of that theorem crucially relies on the fact that VIP can have *more* undefined behaviour than PNVI-ae-udi. For example, when an integer-to-pointer cast leads to a pointer with an ambiguous provenance in PNVI-ae-udi, it is perfectly fine in VIP to choose either one of the two possible provenances (difference 1 above). Intuitively, this divergence between the semantics is tolerable because there are two possible outcomes for such a choice. One possibility is that PNVI-ae-udi eventually resolves the ambiguity consistently with the choice made by VIP (or never needs to make a choice), in which case the two semantics stay in sync. The other possibility is that PNVI-ae-udi eventually resolves the ambiguity differently from how VIP did; however, in this case the very fact that PNVI-ae-udi resolved the ambiguity means that the choice PNVI-ae-udi didn't make (*i.e.*, the one VIP did) leads to undefined behaviour, so the soundness proof of VIP becomes trivial.

To be sound w.r.t. PNVI-ae-udi, VIP must also ensure that each provenance that it reconstructs for a pointer has previously been exposed under PNVI-ae-udi, despite the fact that VIP does not explicitly track exposure (difference 2 above). There are two different situations in which such a provenance is constructed. First, after a round-trip cast, the provenance of the original pointer (stored in the integer) is preserved, which is justified by the fact that only exposed provenances can appear in integers. Second, when an integer is cast into a pointer using `copy_alloc_id`, then the exposure is forced on the PNVI-ae-udi side by using the following implementation of the primitive.

```
void *copy_alloc_id(uintptr_t i, void *p){ (uintptr_t) p; return (void*) i; }
```

First, the pointer `p` is cast into an integer (that is discarded) to force the exposure of its provenance. Then, the expected integer-to-pointer cast is performed. At this point the provenance of `p` is known to be exposed and thus it can be used as the provenance of the resulting pointer.

$$
\begin{array}{rrcl}
\text{(Location)} & \ell & ::= & (\pi, a) \\
\text{(Pointer val.)} & p & ::= & \texttt{NULL} \mid \ell \mid \texttt{funptr(ident)} \\
\text{(C type)} & \tau & ::= & \tau[n] \mid \texttt{int}(\alpha) \mid \ldots \\
\text{(Value)} & v & ::= & \texttt{pointer}(\tau, p) \mid \texttt{integer}(\alpha, x) \mid \texttt{unspec}(\tau) \mid \texttt{array}(v_1, \ldots, v_n) \\
 & & & \mid \texttt{struct}(\texttt{T}, [f_1 = v_1; \ldots; f_n = v_n]) \mid \ldots \\
\text{(Rel. op.)} & \odot & ::= & \leq \mid < \mid \geq \mid > \mid = \mid \neq \\
\text{(Arith. op.)} & \oplus & ::= & + \mid - \mid \times \mid \ldots \\
\text{(Mem. event)} & e & ::= & \mathbf{allocate}(al, \tau) = p \mid \mathbf{kill}(p) = () \mid \mathbf{load}(\tau, p) = v \mid \mathbf{store}(\tau, p, v) = () \\
 & & & \mid \mathbf{diff\_pval}(\tau, p_1, p_2) = x \mid \mathbf{rel\_op\_pval}(\odot, p_1, p_2) = b \\
 & & & \mid \mathbf{eq\_pval}(p_1, p_2) = b \mid \mathbf{rel\_op\_ival}(\odot, x_1, x_2) = b \\
 & & & \mid \mathbf{arith\_bin\_op}(\oplus, x_1, x_2) = x \mid \mathbf{array\_offset}(p_1, \tau, n) = p_2 \\
 & & & \mid \mathbf{member\_offset}(p_1, \texttt{T}, f) = p_2 \mid \mathbf{cast\_ival\_to\_pval}(\tau, x) = p \\
 & & & \mid \mathbf{cast\_pval\_to\_ival}(\tau, p) = x \mid \mathbf{copy\_alloc\_id}(x, p_1) = p_2
\end{array}
$$

Fig. 4. Definition of memory events and other objects that are part of the common memory interface. They rely on an abstract notion of provenance (meta-variable π) and integer value (meta-variable x).

3 THE VIP MEMORY MODEL

In this section we formally present our VIP memory model. We first define a common memory interface (§3.1), shared by VIP and PNVI-ae-udi, to make the two precisely comparable. It is based on the interface used by the Cerberus C semantics [Memarian et al. 2019, 2016], giving confidence that it can handle a wide range of C code. The memory models are expressed as transition systems labeled with a parameterised form of *memory event*, instantiated for VIP (§3.2) and for PNVI-ae-udi (§4). The VIP memory state is defined in §3.3, and examples of reduction rules shown in §3.4.

3.1 Common Memory Model Interface

We first define a notion of *memory event*, shown in Figure 4, that is common to VIP and PNVI-ae-udi. Memory events and the associated objects are parameterised over a notion of "pointer provenance" (meta-variable π) and a notion of "integer value" (meta-variable x). Additionally, the definitions of Figure 4 use metavariables: $a \in \mathit{address}$ (with $\mathit{address} \triangleq \mathbb{Z}$) for a concrete address, `ident` for a function symbol, α for an integer type description, $al \in \mathbb{N}$ for an alignment constraint, $n \in \mathbb{N}$ for a natural number, `T` for a C struct or union, and f for a field name.

A memory location ℓ is defined to be a pair of an abstract provenance (defined in §3.2 for VIP) and a concrete address. A pointer value p is either the constant `NULL`, a location, or a function pointer identified by the corresponding (unique) function symbol. We assume for convenience that every function symbol `ident` is associated to a concrete address a = address_of_function(`ident`).

Our memory interface uses C types to describe the layout of memory, and (inductively defined) structured values. We leave out some details of these definitions here, as they are orthogonal to the presentation of the memory model. We however assume that there is a function sizeof(τ) on C types, giving the number of bytes occupied by a value of type τ.

After the definition of the relational and arithmetic operators, we get to memory events. These are formed of a name (in bold) describing the operation, a set of input parameters (in parentheses), and a result value. For example, $\mathbf{allocate}(al, \tau) = p$ corresponds to the allocation of an object at the address

p. The alignment parameter *al* indicates that the resulting pointer p should have an address which is a multiple of *al*, and the parameter τ indicates the type (and hence the size) of the allocation. The other forms of memory events are similar, so we will not describe all of them in detail. Note however that, since our memory model needs to precisely track the provenance of pointers, memory events are not restricted to the usual allocation, deallocation, load and store operations (*i.e.*, the first line of memory events in the figure). Instead, arithmetic operations (on pointers and on integers) as well as conversions between pointers and integers are also considered to be memory events. For example, $\mathbf{diff\_pval}(\tau, p_1, p_2) = x$ corresponds to the pointer difference operation, $\mathbf{array\_offset}(p_1, \tau, n) = p_2$ to the pointer offset operation, and $\mathbf{cast\_ival\_to\_pval}(\tau, x) = p$ to the integer-to-pointer cast operation. The last memory event in the figure, $\mathbf{copy\_alloc\_id}(x, p_1) = p_2$, corresponds to uses of the `copy_alloc_id` instruction introduced in §2, and we will go in more detail about it in §3.4.

3.2 Instantiation of the Memory Interface for VIP

To instantiate the common memory interface defined in the previous section for VIP, we need to give concrete definitions of pointer provenance and integer value.

$$\text{(provenance)} \quad \pi ::= \texttt{@empty} \mid @i \qquad \text{(Integer value)} \quad x ::= Loc(\ell) \mid Int(z \in \mathbb{Z})$$

A pointer either has the empty provenance or an allocation identifier $i \in alloc\_id$ (with $alloc\_id \triangleq \mathbb{Z}$). Integer values can be either locations ℓ or integers z, and we define the function $to\_int(x)$ converting integer values into integers as $to\_int(Loc((\pi, z))) = to\_int(Int(z)) = z$. Note that this representation of integers values lets VIP track pointer provenance via integers. As discussed in §2.4, this allows supporting round-trip casts in cases where no pointer with a suitable provenance is available to use as the second argument of the `copy_alloc_id` instruction.

3.3 Memory State of VIP

VIP is modeled as a labeled transition system over the state of the memory. We now precisely define what this state is. As one would expect, it needs to account for bytes of data stored in memory. They are here referred to as *memory bytes*, and defined as follows.

$$\text{(Byte)} \quad b ::= \texttt{unspec} \mid z \in [0 \,..\, 255] \qquad \text{(Mem. byte)} \quad mb ::= (\pi,\ b,\ \texttt{none} \mid n \in \mathbb{N})$$

A memory byte contains the description of a concrete byte b (which may be unspecified), together with two pieces of metadata. The first is the pointer provenance π associated to the byte (which may be `@empty` if there is no such provenance), and the second is an optional pointer fragment index n (or `none`) used in the representation of pointer values. Pointer values are formed of a sequence of memory bytes with consecutive fragment indices, starting at 0; we return to that later.

A state Σ of the memory in VIP is a pair (A, M) of an allocation map and a heap.

$$A : alloc\_id \rightharpoonup \{base : address;\ length : \mathbb{N};\ killed : \mathbb{B}\} \qquad M : address \rightharpoonup mbyte$$

The map A associates allocation identifiers (as found in the provenance of pointers) to corresponding allocation meta-data containing: the base address of the allocation, its length as a number of bytes, and a boolean indicating whether the allocation has been killed (*i.e.*, deallocated). In the following, we will use `killed` and `alive` as synonyms of *true* and *false* for more clarity.

The heap M maps concrete addresses to memory bytes, and we have $dom(M) \subseteq$ valid_addresses. The set valid_addresses is defined to only contain addresses that are representable on the pointers of the considered architecture, excluding the value 0 which is reserved for `NULL`, and excluding the last address of the address space to ensure that a pointer one past the end of an allocation never wraps around to 0. Moreover, it is enforced that the valid addresses do not intersect with code segments. As a consequence, the address of function pointers is not in valid_addresses.

To ensure that one only works with valid addresses, the set of addresses that are considered upon allocation is suitably restricted. This set is defined by function new_alloc(A, al, a) given below.

$$\text{live\_addresses}(A) \triangleq \cup_{(i \mapsto (a,n,\texttt{alive})) \in A} [a \,..\, a+n-1]$$

$$\text{new\_alloc}(A, al, n) \triangleq \left\{ a \;\middle|\; \begin{array}{l} \text{divides}(al, a) \;\wedge\; n > 0 \;\wedge \\ [a \,..\, a+n-1] \subseteq \text{valid\_addresses} \;\wedge \\ [a \,..\, a+n-1] \cap \text{live\_addresses}(A) = \varnothing \end{array} \right\}$$

An address a is considered for an allocation if four conditions are satisfied. First, the address should be aligned according to the given alignment constraint al, and the requested allocation size n should be strictly greater than zero. Then, the address range $[a \,..\, a+n-1]$ corresponding to the footprint of the potential allocation should only contain valid addresses, and it should not intersect with addresses in the range of live allocations (described by live_addresses(A)).

Relating structured values and memory bytes. As we will see in §3.4, the rules for the memory events $\mathbf{load}(\tau, p) = v$ and $\mathbf{store}(\tau, p, v) = ()$ need to convert between two value representations: the structured values appearing in the events, and the memory bytes actually stored on the heap. These two representations are related using two functions: a total function $\text{repr}(v)$ mapping a memory value to its representation as a sequence of memory bytes; and a partial function $\text{abst}(A, \tau, bs)$ interpreting the list of memory bytes bs as a structured value of C type τ. The abst function also takes the allocation map A as argument, as it must checks allocation bounds in some cases. The definitions of the two functions, which is quite subtle due to the fact that we support manipulating representation bytes of pointers, can be found in Lepigre et al. [2021a, §A].

3.4 Step Relation and Rules

Let us now consider the VIP step relation ($\rightarrow_{\text{VIP}}$). We write $\Sigma \xrightarrow{e}_{\text{VIP}} \Sigma'$ if the memory event e can be used to transition from the starting state $\Sigma = (A, M)$ to the end state $\Sigma' = (A', M')$. We say that a memory event e has undefined behaviour (UB) in state Σ if there is no end state Σ' to which a transition can be made. The step relation ($\rightarrow_{\text{VIP}}$) is defined by a set of rules, a selection of which is displayed in Figure 5. A transition appears as the conclusion of each rule, but the memory event labeling it is shown in square brackets next to the rule's name, above the premises.

Operations on memory. Starting at the top of Figure 5, VIP-ALLOC handles allocation with memory event $\mathbf{allocate}(al, \tau) = p$. Its premises require that the returned pointer value should be a location $(@i, a)$ (last premise), where i is a fresh allocation identifier (second premise), and a is a suitable address for an allocation of size n (third premise), where $n = \text{sizeof}(\tau)$ (first premise). When the rule is applied, the state (A, M) is modified in two ways. First, a new allocation with identifier i is recorded in A: its base address is a, its size is n bytes, and it is initially alive (*i.e.*, not deallocated). Second, the footprint of the allocation is updated in the heap M: all bytes are mapped to uninitialised memory bytes (whose second component is `unspec`).

Rule VIP-KILL handles deallocation using event $\mathbf{kill}(p) = ()$. It requires argument p to be a location $(@i, a)$ (first premise), and corresponding allocation i to be alive and to have base address a (second premise). When the rule applies, the resulting state records the fact that i was killed.

Rules VIP-LOAD and VIP-STORE handle events $\mathbf{load}(\tau, p) = v$ and $\mathbf{store}(\tau, p, v) = ()$ respectively. Both rules require p to be a location $(@i, a)$ (first premise), with the corresponding allocation being alive (second premise), and the footprint of the operation should be in bounds of the allocation (third premise). Rule VIP-LOAD then interprets the memory bytes stored at that footprint using the abst function, and it leaves the state unchanged. Rule VIP-STORE updates the heap using the appropriate memory bytes obtained with the repr function.

$$\text{VIP-ALLOC}\ [\mathbf{allocate}(al,\tau)=p]\quad \frac{n=\text{sizeof}(\tau)\qquad i\notin\text{dom}(A)\qquad a\in\text{new\_alloc}(A,al,n)\qquad p=(@i,a)}{(A,M)\rightarrow_{\text{VIP}}(A[i\mapsto(a,n,\texttt{alive})],M[a\,..\,a+n-1\mapsto(@\texttt{empty},\texttt{unspec},\texttt{none})])}$$

$$\text{VIP-KILL}\ [\mathbf{kill}(p)=()]\quad \frac{p=(@i,a)\qquad A(i)=(a,n,\texttt{alive})}{(A,M)\rightarrow_{\text{VIP}}(A[i\mapsto(a,n,\texttt{killed})],M)}$$

$$\text{VIP-ARITH-OP}\ [\mathbf{arith\_bin\_op}(\oplus,x_1,x_2)=x]\quad \frac{x=Int(to\_int(x_1)\oplus to\_int(x_2))}{(A,M)\rightarrow_{\text{VIP}}(A,M)}$$

$$\text{VIP-STORE}\ [\mathbf{store}(\tau,p,v)=()]\quad \frac{p=(@i,a)\qquad A(i)=(a_i,n_i,\texttt{alive})\qquad n=\text{sizeof}(\tau)\qquad [a\,..\,a+n-1]\subseteq[a_i\,..\,a_i+n_i-1]}{(A,M)\rightarrow_{\text{VIP}}(A,M[a\,..\,a+n-1\mapsto\text{repr}(v)])}$$

$$\text{VIP-LOAD}\ [\mathbf{load}(\tau,p)=v]\quad \frac{p=(@i,a)\qquad A(i)=(a_i,n_i,\texttt{alive})\qquad [a\,..\,a+\text{sizeof}(\tau)-1]\subseteq[a_i\,..\,a_i+n_i-1]\qquad v=\text{abst}(A,\tau,M[a\,..\,a+\text{sizeof}(\tau)-1])}{(A,M)\rightarrow_{\text{VIP}}(A,M)}$$

$$\text{VIP-P-I-CAST}\ [\mathbf{cast\_pval\_to\_ival}(\tau,p)=x]\quad \frac{p=(@i,a)\qquad x=Loc((@i,a))\qquad A(i)=(\_,\_,\texttt{alive})\qquad a\in\texttt{value\_range}(\tau)}{(A,M)\rightarrow_{\text{VIP}}(A,M)}$$

$$\text{VIP-COPY-A-ID}\ [\mathbf{copy\_alloc\_id}(x,p_1)=p_2]\quad \frac{p_1=(@i,\_)\qquad A(i)=(a,n,\texttt{alive})\qquad to\_int(x)\in[a\,..\,a+n]\qquad p_2=(@i,to\_int(x))}{(A,M)\rightarrow_{\text{VIP}}(A,M)}$$

$$\text{VIP-I-P-CAST-1}\ [\mathbf{cast\_ival\_to\_pval}(\tau,x)=p]\quad \frac{x=Loc((@i,a))\qquad A(i)=(a_i,n_i,\texttt{alive})\qquad a\in[a_i\,..\,a_i+n_i]\qquad p=(@i,a)}{(A,M)\rightarrow_{\text{VIP}}(A,M)}$$

$$\text{VIP-I-P-CAST-2}\ [\mathbf{cast\_ival\_to\_pval}(\tau,x)=p]\quad \frac{x=Int(z)\qquad z\in\text{valid\_addresses}\qquad p=(@\texttt{empty},z)}{(A,M)\rightarrow_{\text{VIP}}(A,M)}$$

Fig. 5. A selection of VIP reduction rules.

Pointer-to-integer casts and arithmetic. Consider VIP-P-I-CAST: one of the three VIP rules for integer-to-pointer cast events **cast_pval_to_ival**$(\tau,p)=x$. Unlike its sibling rules (found in Lepigre et al. [2021a, §B]), which are used when p is `NULL` or a function pointers, VIP-P-I-CAST applies when p is a location $(@i,a)$ (first premise), whose corresponding allocation is alive (third premise), and whose address is representable in the specified integer type (last premise). The result value of the operation is an integer value representing location $(@i,a)$ (second premise), and the memory state is unchanged. Importantly, note that the whole location is preserved, and not simply its address. Hence, provenance $@i$ is effectively stored in the integer value. This is key to the handling of round-trip casts as discussed in §2.4. Indeed, rule VIP-I-P-CAST-1 can then be used on such an integer value to get a pointer with the original provenance, without relying on `copy_alloc_id`.

Nonetheless, provenance can only flow via integers in a limited way. Any arithmetic operation leads to the provenance being lost, as can be seen in rule VIP-ARITH-OP, handling event **arith_bin_op**$(\oplus,x_1,x_2)=x$. By using the *to_int* function on the operands x_1 and x_2 in the first premise, all provenance information is lost. The result value is hence a simple integer—with no provenance—on which rule VIP-I-P-CAST-1 does not apply.

Copying the provenance of a pointer. To construct a pointer with a non-empty provenance—that can potentially be used for memory accesses—from an integer on which arithmetic has been performed, the special `copy_alloc_id` instruction needs to be used. Its semantics is reflected by rule VIP-COPY-A-ID, which handles memory events of the form **copy_alloc_id**$(x,p_1)=p_2$. The premises of the rule indicate that p_1 is expected to be a location with provenance $@i$ (first premise), where i identifies a live allocation (second premise), and the integer value of x should be in bounds or one

past the end of that allocation (third premise). The result p_2 of the operation is then defined (in the last premise) to be a location whose provenance is that of p_2 and whose address is x.

The second and third premises of VIP-COPY-A-ID are needed for the VIP semantics of `copy_alloc_id` to match the PNVI-ae-udi semantics of its implementation. In particular, note that integer-to-pointer casts under PNVI-ae-udi (including that in the implementation of `copy_alloc_id`) only consider provenances that are alive and for which the integer is in bounds (or one past the end). Hence, a matching requirement is necessary in VIP-COPY-A-ID to enforce soundness w.r.t. PNVI-ae-udi.

Concretely, consider the program shown on the right: If out-of-bounds, non-one-past-the-end pointers were allowed to be created with `copy_alloc_id`, then this program would be valid under VIP and yet have UB under PNVI-ae-udi, precisely the situation we don't want to arise. To see this, note that the pointer `px` created on Line 4 is clearly out of bounds and not one past the end of the allocation of `x`. It is however brought back in bounds using an offset operation before being dereferenced on Line 5, at an address that is exactly that of `&x`. The reason why this program has UB under PNVI-ae-udi is that the provenance of `&x` is not a valid option for `px` on Line 4, at the time of the integer-to-pointer cast found in the expansion of `copy_alloc_id`. As a consequence, the load operation `*(px - 2)` on Line 5 has UB since the provenance of the pointer in `px` is not that of `x`. On the other hand, without the second and third premises of VIP-COPY-A-ID, neither Line 4 nor Line 5 would raise UB under VIP, and the program would have well-defined behaviour.

```
1 int x = 42;
2 uintptr_t i = (uintptr_t) &x;
3 i = i + 2 * sizeof(int);
4 int *px = copy_alloc_id(i, &x);
5 assert(*(px - 2) == 42);
```

Integer-pointer casts. The last two rules of Figure 5 handle event $\mathbf{cast\_ival\_to\_pval}(\tau, x) = p$, corresponding to an integer-to-pointer cast. Rule VIP-I-P-CAST-1 is used, as mentioned previously, when the integer value x stores a location. Rule VIP-I-P-CAST-2 is used when x is just an integer, with no provenance, in which case the resulting location has provenance `@empty`. Note however that the integer z is required to be in the set of valid addresses, which rules out 0 and function pointers. They are handled using a separate rule.

4 DEFINITION OF PNVI-AE-UDI

In this section, we highlight key aspects of the PNVI-ae-udi memory model [Gustedt et al. 2020; Memarian et al. 2019], and its main differences with VIP. First, in §4.1, we instantiate the memory interface of §3.1 for PNVI-ae-udi. We then define the PNVI-ae-udi memory state in §4.2, before considering examples of reduction rules that we compare with those of VIP in §4.3. Finally, in §4.4, we discuss differences with PNVI-ae-udi as formalised by Gustedt et al. [2020].

4.1 Instantiation of the Memory Interface for PNVI-ae-udi

Let us now instantiate the memory interface of §3.1 for PNVI-ae-udi, as we did for VIP in §3.2. To this end, we define pointer provenance and integer values as follows.

$$\text{(provenance)} \quad \pi ::= \texttt{@empty} \mid @i \mid \iota \qquad\qquad \text{(Integer value)} \quad x ::= z \in \mathbb{Z}$$

Like in VIP, pointer provenance can be `@empty`, and it can also refer to an allocation by its identifier i as in $@i$. However, a key feature of PNVI-ae-udi is that *symbolic provenances* ι are used to deal with ambiguities arising at integer-to-pointer cast. Indeed, an integer may in some cases represent an address that is both one past the end of an allocation, and at the start of an adjacent one.

PNVI-ae-udi, unlike its PVI sibling [Memarian et al. 2019] or VIP, does not track pointer provenance via integers. Hence, integer values are just simple integers, and all provenance information is thus lost on pointer-to-integer casts. Nonetheless, PNVI-ae-udi supports round-trip casts of pointers, even for one-past-the-end addresses, thanks to exposure and symbolic provenances.

4.2 PNVI-ae-udi Memory State

The state of memory in PNVI-ae-udi is defined to be a tuple (A, S, M), where A and M are an allocation map and a heap similar to those of VIP, and S is a map used to track symbolic provenances.

$$A : \mathit{alloc\_id} \rightharpoonup \{\mathit{base} : \mathit{address};\ \mathit{length} : \mathbb{N};\ \mathit{killed} : \mathbb{B};\ \mathit{exposed} : \mathbb{B};\ \}$$

$$S : \mathit{provenance\_symbol} \rightharpoonup \{s \in \mathcal{P}(\mathit{alloc\_id}) \mid \mathit{card}(s) \in \{1, 2\}\} \qquad M : \mathit{address} \rightharpoonup \mathit{mbyte}$$

Note that the allocations recorded in A contain one more information compared to those of VIP, namely a boolean indicating whether the provenance of the allocation has been exposed. Indeed, as was mentioned in §2, PNVI-ae-udi restricts the reconstruction of a provenance on integer-to-pointer cast to those provenances that have been exposed. As we will shortly see, the *exposed* flag of an allocation is set when a pointer to that allocation is cast into an integer. In the following, we use `exposed` and `unexposed` as synonyms of *true* and *false* for clarity.

The symbolic provenance map S associates either one or two allocation identifiers to all provenance symbols in existence. Whenever the provenance of a pointer is ambiguous, and could thus be either i_1 or i_2, a new symbol ι is introduced and a mapping $\iota \mapsto \{i_1, i_2\}$ is inserted into S. If on a further operation it become apparent that one of the options, say i_2, is invalid, then it is removed and the mapping is updated to be $\iota \mapsto \{i_1\}$. In this latter case, the symbol ι is intuitively defined to be i_1, but an indirection is kept as part of S.

Note that the heap M uses the same representation of memory bytes as VIP. In fact, like VIP, PNVI-ae-udi relies on repr and abst functions to convert between structured values appearing in the $\mathbf{load}(\tau, p) = v$ and $\mathbf{store}(\tau, p, v) = ()$ events, and the memory bytes stored in M. As for VIP, the definition of these functions is intricate to handle manipulations of representation bytes of pointers. Note however that different definitions are used for the two memory models, since they have a different notion provenance, and the PNVI-ae-udi version also take exposure into account. The precise definition of repr and abst for PNVI-ae-udi is given in Lepigre et al. [2021a, §D].

4.3 PNVI-ae-udi Step Relation and Rules

PNVI-ae-udi has a relation ($\rightarrow_{\mathsf{PNVI}}$) similar to ($\rightarrow_{\mathsf{VIP}}$): $(A, S, M) \xrightarrow{e}_{\mathsf{PNVI}} (A', S', M')$ indicates that the memory event e can be used to transition from state (A, S, M) to state $\Sigma' = (A', S', M')$, and e has UB in a given state if no transition can be made. A selection of PNVI-ae-udi reduction rules is shown in Figure 6, focusing on integer-pointer cast operations.

Allocation, deallocation, load and store. The VIP rules VIP-ALLOC, VIP-KILL, VIP-LOAD and VIP-STORE from Figure 5 all have essentially identical counterparts in PNVI-ae-udi (not shown here, see Lepigre et al. [2021a, §E]), the only difference being that the *exposed* flag is set to `exposed` in the allocation rule, and ignored by the other operations. However, PNVI-ae-udi also requires another set of rules to handle the case where the pointer value p in the events $\mathbf{kill}(p) = ()$, $\mathbf{load}(\tau, p) = v$ and $\mathbf{store}(\tau, p, v) = ()$ has a symbolic provenance, and is thus of the form (ι, a). We give one example of such rule in Figure 6: PNVI-KILL-ι. Its second premise requires ι to be matched to an allocation identifier i whose corresponding allocation has base address a and has not yet been killed. Note that there can be at most one such i, since live allocations are always pairwise disjoint. In the case where $S(\iota)$ has two elements, the application of the rule thus eliminates one of the possibilities. We can see that the end state of the rule not only marks the allocation i as killed, but it also enforces that ι only has one associated allocation identifier, namely i.

Pointer-to-integer cast. We now consider two main PNVI-ae-udi rules for pointer-to-integer casts on a pointer value of the form (π, a). The former, PNVI-P-I-CAST is essentially similar to the VIP rule VIP-P-I-CAST—since it handles that case where $\pi = @i$—with two main differences. First,

$$\text{PNVI-KILL-}\iota\ [\textbf{KILL}(p) = ()]\quad \frac{p = (\iota, a) \qquad \exists i \in S(\iota),\ A(i) = (a, n, \texttt{alive}, t)}{(A, S, M) \rightarrow_{\text{PNVI}} (A[i \mapsto (a, n, \texttt{killed}, t)], S[\iota \mapsto \{i\}], M)}$$

$$\text{PNVI-P-I-CAST}\ [\textbf{CAST\_PVAL\_TO\_IVAL}(\tau, p) = x]\quad \frac{p = (@i, a) \qquad A(i) = (\_, \_, \texttt{alive}, \_) \qquad a \in \texttt{value\_range}(\tau) \qquad x = a}{(A, S, M) \rightarrow_{\text{PNVI}} (A[i \mapsto (a_i, n_i, \texttt{alive}, \texttt{exposed})], S, M)}$$

$$\text{PNVI-P-I-CAST-}\iota\ [\textbf{CAST\_PVAL\_TO\_IVAL}(\tau, p) = x]\quad \frac{p = (\iota, a) \qquad \exists i \in S(\iota),\ A(i) = (\_, \_, \texttt{alive}, \_) \qquad a \in \texttt{value\_range}(\tau) \qquad x = a}{(A, S, M) \rightarrow_{\text{PNVI}} (A, S, M)}$$

$$\text{PNVI-I-P-CAST}\ [\textbf{CAST\_IVAL\_TO\_PVAL}(\tau, x) = p]\quad \frac{\exists! i, A(i) = (a, n, \texttt{alive}, \texttt{exposed}) \qquad x \in [a\ ..\ a + n] \qquad p = (@i, x)}{(A, S, M) \rightarrow_{\text{PNVI}} (A, S, M)}$$

$$\text{PNVI-I-P-CAST-}\iota\ [\textbf{CAST\_IVAL\_TO\_PVAL}(\tau, x) = p]\quad \frac{\exists i_1,\ A(i_1) = (a_1, n_1, \texttt{alive}, \texttt{exposed}) \qquad x \in [a_1\ ..\ a_1 + n_1] \qquad \exists i_2,\ A(i_2) = (a_2, n_2, \texttt{alive}, \texttt{exposed}) \qquad x \in [a_1\ ..\ a_2 + n_2] \qquad i_1 \neq i_2 \qquad \iota \notin \text{dom}(S) \qquad p = (\iota, x)}{(A, S, M) \rightarrow_{\text{PNVI}} (A, S[\iota \mapsto \{i_1, i_2\}], M)}$$

Fig. 6. A selection of PNVI-ae-udi reduction rules.

the obtained integer value does not carry a provenance, since integer values are only integers in PNVI-ae-udi. And second, the PNVI-ae-udi rule performs the extra duty of marking the allocation as being exposed in the end state.

The second PNVI-ae-udi rule, PNVI-P-I-CAST-ι, handles the case where $\pi = \iota$, for which the second premise only requires ι to be mapped to some i such that the corresponding allocation is alive. Note that in the case where ι is mapped to two allocations identifiers, then we do not know how to resolve the ambiguity, and hence the map S is not modified in the end state. Also, symbolic provenances can only be mapped to allocation identifiers that have been exposed, so there is no need to update A as in the conclusion of PNVI-P-I-CAST.

Integer-to-pointer cast. We now consider the integer-to-pointer cast operation, which is responsible for the introduction of new symbolic provenances in the case of an ambiguity. Let us first consider the rule PNVI-I-P-CAST, which applies when the provenance to be reconstructed is unambiguous, as hinted by its first two premises. Indeed, they require that there is a unique allocation identifier i such that the corresponding allocation is alive, and the integer x is in bounds or one past the end of that allocation. In this case, since there is no other possibility, the provenance $@i$ is picked for the result pointer p in the last premise.

If on the other hand there are two different possible candidate allocation identifiers i_1 and i_2, then the rule PNVI-I-P-CAST-ι applies. In this case, a fresh provenance symbol ι is picked, and mapped to the set $\{i_1, i_2\}$ in the end state. The ambiguity is then tracked until it can be resolved by a subsequent operation that would only be valid with one of the two choices. Note that there can never be more than two candidate allocation identifiers, since an ambiguity only exists in the case where x is the address of the first byte of a live allocation which is immediately preceded by another live allocation (to which it points one past the end).

Implementation of copy_alloc_id. The `copy_alloc_id` instruction is a VIP-specific primitive: it is not directly supported by PNVI-ae-udi. However, to execute programs verified against VIP on PNVI-ae-udi-compliant implementations, we use the following implementation of `copy_alloc_id`:

```
void *copy_alloc_id(uintptr_t i, void *p){ (uintptr_t) p; return (void*) i; }
```

In terms of the PNVI-ae-udi memory model, this implementation produces two distinct events: (1) a **cast_pval_to_ival** event on the pointer-to-integer cast `(uintptr_t) p`, which has the side-effect of forcing the exposure of the provenance of `p`, and (2) a **cast_ival_to_pval** event on the actual integer-to-pointer cast `(void*) i`, upon which the resulting pointer may receive the provenance of `p` that was exposed by the former event. We will see in §5 how these two PNVI-ae-udi events are related to a VIP **copy_alloc_id**$(x, p_1) = p_2$ event to establish the soundness of VIP w.r.t. PNVI-ae-udi.

One remaining issue needs to be addressed with the above implementation of `copy_alloc_id`: since it is a function, additional events—not produced by VIP—are generated when allocating and deallocating its arguments. However, we can easily make sure that corresponding events also arise in VIP by wrapping the `copy_alloc_id` primitive into a similar function.

4.4 Differences from Previous Presentations of PNVI-ae-udi

To simplify the presentation and ease the statement of our soundness result in §5, our version of PNVI-ae-udi slightly differs from the formalisation of Gustedt et al. [2020]. Unlike in the original presentation, we only allow the allocation of objects (*i.e.*, variables) and not of memory regions (*i.e.*, heap allocations). We also do not support read-only allocations, nor the representation of floating point numbers. Nonetheless, we do not expect that it would be difficult to add these features to VIP and to our presentation of PNVI-ae-udi, since they are orthogonal to integer-pointer casts.

The original presentation of PNVI-ae-udi is parameterised by a number of switches influencing the behaviour of the model. In this paper, we have chosen to use both STRICT-POINTER-EQUALITY and ZOMBIE-POINTERS-ALLOW-ALL-IN-BOUNDS-ARITHMETIC. Moreover, we make the assumption that all pointer types have the same size, which is true in all current mainstream architectures.

Finally, we made a few cosmetic changes that do not influence the behaviour of PNVI-ae-udi. For example, the allocation map A uses a specific variant for the case of killed allocations in the original presentation, while we instead rely on a boolean flag to match VIP better.

5 SOUNDNESS OF VIP WITH RESPECT TO PNVI-AE-UDI

In this section, we explain how any verification tool based on VIP can be used to soundly reason about the execution of C programs under PNVI-ae-udi-compliant implementations. Let us assume that p is a program that has been verified with such a verification tool, say RefinedC-VIP, and that p is thus safe to execute under the VIP semantics. Using RefinedC-VIP, this first step is achieved using the following adequacy theorem (formally proved in the Coq proof assistant):

Theorem 1. *If* RefinedC-VIP *verifies a program p, it is safe to execute p under the* VIP *memory model, meaning that execution of p will not result in undefined behaviour.*

After having proved that p does not have UB under VIP, before compiling it with a compiler presumed sound under PNVI-ae-udi, we first have to implement the `copy_alloc_id` instruction as shown in §4.3. Formally, we need to translate the program p into a program $\downarrow p$ that can be run under PNVI-ae-udi. We then rely on the following theorem to transfer the guarantees offered by Theorem 1 to the translated program.

Theorem 2. *For all C programs p, if p is safe to execute under the* VIP *memory model, then $\downarrow p$ is safe to execute under the* PNVI-ae-udi *memory model.*

To prove Theorem 2, we establish a simulation between PNVI-ae-udi and VIP. More precisely, we show that every PNVI-ae-udi step can be simulated by a VIP step, and that every step that raises undefined behaviour in PNVI-ae-udi also raises undefined behaviour in VIP. Note however that it is perfectly fine to have more undefined behaviour under VIP than under PNVI-ae-udi. Indeed, Theorem 2 only considers programs that do not have undefined behaviour in VIP.

State relation. Making this proof formal is non-trivial since VIP and PNVI-ae-udi use different notions of provenance, integer value, and state. Hence, we cannot just prove that these objects are the *same* on both sides, only that they are *related* by a carefully-chosen relation ($\approx$). From the point of view of a program p, the fact that such objects are only related but not equal cannot be observed. Indeed, the memory interface treats them as abstract objects that p only manipulates via memory events. For corresponding events, what we prove is that if their inputs are related (in related states), then their output is also related (in possibly changed but still related states).

Finding a suitable definition for the relation ($\approx$) is challenging. In particular, it is not obvious how to relate a symbolic provenance ι from PNVI-ae-udi to a VIP provenance $@i$. Indeed, a particular allocation identifier i should only be related to ι if it is a valid candidate for the definition of the symbolic provenance. However, this information is not available by looking at ι only, since it is tracked in the S part of the PNVI-ae-udi state. To address this, we use the standard technique of parameterising our relation ($\approx$) by a *world* w, that can evolve along a pre-order ($\sqsubseteq$) during execution. We then use this world to track which allocation identifier i is related to each provenance symbol ι. The evolution of the world is invisible to the program p as ($\sqsubseteq$) preserves relatedness.

$$v_{\mathsf{PNVI}} \approx_w v_{\mathsf{VIP}} \ \wedge \ w \sqsubseteq w' \implies v_{\mathsf{PNVI}} \approx_{w'} v_{\mathsf{VIP}} \tag{1}$$

Event relation. Another difficulty for making the proof of Theorem 2 formal is the mismatch between the events of VIP and PNVI-ae-udi. Indeed, a **copy_alloc_id** event produced by a program p is replaced by a **cast_pval_to_ival** and a **cast_ival_to_pval** event in the translated program $\downarrow p$. To handle this, we define a relation ($\sim_w$) relating a list of PNVI-ae-udi events $\overline{e}_{\mathsf{PNVI}}$ to a single VIP event e_{VIP}. For events $f \neq$ **copy_alloc_id**, ($\sim_w$) simply relates a singleton PNVI-ae-udi event f to a VIP event f, provided the input parameters on either sides are related by ($\approx_w$). The **copy_alloc_id** event of VIP is related to the list of two events generated by its implementation given in §4.3, also requiring the inputs of the events to be related (see Lepigre et al. [2021a, §F] for details).

We can now finally state our main theorem establishing a simulation between PNVI-ae-udi and VIP events, and then use it to prove Theorem 2.

Theorem 3. *For all* PNVI-ae-udi *and* VIP *states such that* $\Sigma_{\mathsf{PNVI}} \approx_w \Sigma_{\mathsf{VIP}}$*, and for all* PNVI-ae-udi *event lists and* VIP *events such that* $\overline{e}_{\mathsf{PNVI}} \sim_w e_{\mathsf{VIP}}$ *and the last result value in* $\overline{e}_{\mathsf{PNVI}}$ *is* y*, we have:*

(1) If executing the events of the list $\overline{e}_{\mathsf{PNVI}}$ *in* Σ_{PNVI} *leads to UB, then* e_{VIP} *has UB in* Σ_{VIP}*.*

(2) Either e_{VIP} *has UB in* Σ_{VIP}*, or for all states* Σ'_{PNVI} *such that executing the events of* $\overline{e}_{\mathsf{PNVI}}$ *in* Σ_{PNVI} *leads to* Σ'_{PNVI}*, there exist* w'*,* y' *and* Σ'_{VIP} *with* $w \sqsubseteq w'$*,* $y \approx_{w'} y'$*,* $\Sigma'_{\mathsf{PNVI}} \approx_{w'} \Sigma'_{\mathsf{VIP}}$ *and executing the event* e'_{VIP} *(obtained from* e_{VIP} *by changing its result value to* y'*) in* Σ_{VIP} *leads to* Σ'_{VIP}*.*

Furthermore there exists w_0 *such that the empty states are related:* $(\varnothing, \varnothing, \varnothing) \approx_{w_0} (\varnothing, \varnothing)$*.*

Using Theorem 3, the proof of Theorem 2 follows by induction over the execution of the program p under PNVI-ae-udi and VIP. At each step, we have a world w such that $\Sigma_{\mathsf{PNVI}} \approx_w \Sigma_{\mathsf{VIP}}$, and all abstract objects ever given to p through the result of memory events are related by ($\approx_w$). We can establish this for the initial state using w_0 from Theorem 3, and the fact that the initial program has not yet received any object from the memory interface. For each step, if the next VIP event e_{VIP} is not **copy_alloc_id** then we have a single PNVI-ae-udi event e_{PNVI}, and we can satisfy the assumptions of Theorem 3 since the states and all objects (including input values) are related. If

e_{PNVI} has UB then, by Theorem 3, e_{VIP} has UB as well and we are done. If e_{PNVI} steps to Σ'_{PNVI} then Theorem 3 gives us w' with $w \sqsubseteq w'$, a result value y' that is $(\approx_{w'})$-related to the result value of e_{PNVI} and a resulting state Σ'_{VIP} with $\Sigma'_{\mathsf{PNVI}} \approx_{w'} \Sigma'_{\mathsf{VIP}}$. Now we only need to use property (1) to show that all objects known to p are related by $(\approx_{w'})$, and we are done. Now, if the next VIP event is **copy_alloc_id**, then we know that the next two events in PNVI-ae-udi are **cast_pval_to_ival** and a **cast_ival_to_pval** by definition of $\downarrow p$, and we can hence prove $(\sim_w)$ for these events. The rest of the proof follows similarly to the other case. □

Proof of the main theorem. We conclude this section by giving high-level intuitions for the proof of Theorem 3, fully detailed in Lepigre et al. [2021a, §F]. The main difficulty for the proof is picking the right definitions for worlds w and for $(\approx_w)$. The relation on states $\Sigma_{\mathsf{PNVI}} \approx_w \Sigma_{\mathsf{VIP}}$ is straightforward since the PNVI-ae-udi state tracks strictly more information than that of VIP (the exposure information and the map S of symbolic provenances) and thus $\Sigma_{\mathsf{PNVI}} \approx_w \Sigma_{\mathsf{VIP}}$ simply lifts $(\approx_w)$ to the parts that exist in both models. However, the situation is more complicated for provenances since we need to relate symbolic provenances ι in PNVI-ae-udi with concrete provenances $@i$ in VIP. This happens, for example, when executing a `copy_alloc_id`. Indeed, the final integer to pointer cast in the PNVI-ae-udi version can result in a symbolic provenance ι, while VIP always results in a concrete provenance $@i$ (taken from the second argument of `copy_alloc_id`). However, the premises of VIP-COPY-A-ID and the first pointer to integer cast in the expansion of `copy_alloc_id` ensure that i is always a possible option for ι, and that $i \in S(\iota)$. One way to look at this, is that VIP *eagerly* resolves the symbolic provenance ι to i. This is tracked during the proof via a map R (part of the world w) that maps symbolic provenances ι to concrete provenances:

$$(\iota, z) \approx_{(R,\_)} (@i, z) \text{ iff } R(\iota) = i \qquad\qquad R(\iota) \in S_{\mathsf{PNVI}}(\iota)$$

Interestingly, on integer-to-pointer casts, VIP may produce a pointer with `@empty` provenance, and PNVI-ae-udi one with a non-`@empty` provenance. Consequently, $(\approx_w)$ must relate `@empty` in VIP to arbitrary PNVI-ae-udi provenances: $(\pi, z) \approx_w (\texttt{@empty}, z)$. However, this is not at all a problem since `@empty` leads to strictly more UB than other provenances.

Another important part of the definition of $(\approx_w)$ is how it relates integers. In particular, VIP integer values may hold a provenance, while those of PNVI-ae-udi cannot. Here the key observation is that having a provenance $@i$ in a VIP integer is a witness that $@i$ was previously exposed in PNVI-ae-udi. This justifies VIP-I-P-CAST-1, as we can be sure that the provenance has been exposed and is thus a candidate for PNVI-I-P-CAST (resp. PNVI-I-P-CAST-ι). Formally, we track the fact that all provenances in integers have been exposed by a set of provenances E that is part of the world w:

$$z \approx_{(\_,E)} Loc((@i, z)) \text{ iff } i \in E \qquad\qquad i \in E \to A_{\mathsf{PNVI}}(i) = (\_, \_, \_, \texttt{exposed})$$

Finally, we define the pre-order $(\sqsubseteq)$ as $(R, E) \sqsubseteq (R', E') \triangleq R \subseteq R' \wedge E \subseteq E'$, to allow adding new elements to R and E during execution. It is easy to check that property (1) holds, and with these definitions, the proof of Theorem 3 given in Lepigre et al. [2021a, §F] is relatively straightforward.

6 REFINEDC-VIP

In this section, we show how VIP is used as the basis of our verification tool RefinedC-VIP, which extends RefinedC's deep embedding of C in Coq (called Caesium) with the VIP memory model, and adds new types and typing rules to support specifications and proofs about programs involving integer-pointer casts. In §6.1, we give the necessary background on the semantic typing approach used by RefinedC(-VIP). Then, in §6.2, we show how RefinedC-VIP enables verification of programs involving integer-pointer casts via new types and typing rules. Finally, in §6.3, we explain how RefinedC-VIP introduces a generalisation to a core algorithm of the RefinedC type checker that is required to support integer-pointer casts.

6.1 Semantic Typing in RefinedC

In RefinedC-VIP, as in RefinedC, verification works in two steps: First, the user gives a specification via a refinement and ownership type system. Then, an automatic type checker verifies the program against this specification by building a typing derivation. The RefinedC type system, similarly to that of RustBelt [Jung et al. 2018a], is built using the approach of *semantic typing*, in which types and typing judgements are given a meaning by interpreting them into Iris, a higher-order concurrent separation logic embedded into Coq [Jung et al. 2015, 2016; Krebbers et al. 2017; Jung et al. 2018b]. One advantage of this approach is that it is modular: new types can be defined by giving their interpretation into the model of types, and new typing rules can be added by proving them sound against the model of types and typing judgements.

Type assignments and model of types. Concretely, defining a new type τ means giving the logical interpretation of the type assignment $v \triangleleft_{\mathsf{v}} \tau$ (*i.e.*, assigning type τ to value v). The exact meaning of this type assignment can be chosen freely. For example, the type assignment for integers $v \triangleleft_{\mathsf{v}} n \mathbin{@} \mathsf{int}(\alpha)$ asserts that v is an integer with value n and size α (*e.g.*, `int` or `size_t` or similar). The type assignment can also assert ownership over memory (and other resources) thanks to the underlying separation logic. An example of this is the "owned pointer" type $\&_{\mathsf{own}}(\tau)$, where $v \triangleleft_{\mathsf{v}} \&_{\mathsf{own}}(\tau)$ asserts that v is a pointer that points to owned memory containing a value of type τ.

Typing judgements. RefinedC uses several different forms of typing judgements (roughly, one per syntactic construct of the language). Here we focus on the two that are used in §6.2:

$$\vdash_{\textsc{expr}} e \ \text{ when } \ P\,\{v, \tau.\ G(v, \tau)\} \qquad\qquad \vdash_{\textsc{subsume}} P_1 <: P_2\,\{G\}$$

The former is an expression typing judgement asserting that the expression e is well-behaved given the assertion P, which is usually a collection of type assignment for values occurring in e.[1] The latter is a subsumption judgement asserting that P_1 implies P_2, which amounts to a subtyping judgement in the usual case where P_1 and P_2 are type assignments for the same value. Both judgements have a continuation G, as is often encountered in weakest-precondition calculi like Iris's. In the expression judgement, this continuation receives the inferred type τ and resulting value v as arguments.

Typing rules. The typing rules of RefinedC are of the form $\frac{G}{F}$ where the conclusion F is a typing judgement and the premise G is a separation logic assertion that tells the type-checker how to proceed after applying this typing rule. Concretely, G is formulated in Lithium [Sammler et al. 2021]), a fragment of Iris which constitutes a separation logic programming language for implementing the RefinedC type-checker. However, the details of Lithium are beyond the scope of this paper.

6.2 New Types and Typing Rules for VIP

To support new idioms like integer-pointer casts in RefinedC, we introduce new types and typing rules for the involved operations. In the following, we go over these operations in the order they typically appear in programs: pointer-to-integer casts, then integer arithmetic, and finally pointer-to-integer casts.

Pointer-to-integer casts. Let us assume that we have a pointer ℓ with the owned pointer type $\&_{\mathsf{own}}(\tau)$, and that we want to cast it into an integer. To design a suitable typing rule, we need to consider the corresponding reduction rule in the operational semantics. For this reduction rule

[1] The expression judgement presented here differs from the presentation of Sammler et al. [2021] and the implementation of RefinedC, where a generic expression judgement (without any attached P) is used in combination with specialised judgements for each form of expression constructor (including an attached assertion for each value argument). We combine these judgements here into a single one for conciseness and simplicity.

VIP-P-I-CAST to apply, two conditions must be fulfilled: (1) the provenance of ℓ must be alive, and (2) the address of ℓ must fit in the resulting integer type α. As a consequence, proving the premise of our typing rule must require proving the corresponding RefinedC-VIP assertions $\mathsf{alive}(\ell)$ and $\ulcorner \ell.2 \in \alpha \urcorner$. Now, it remains to find a suitable type for the resulting integer $Loc(\ell)$. Since integers with provenance are a new feature of VIP, RefinedC-VIP defines a corresponding new type $\ell \ @ \ \mathsf{intptr}(\alpha)$, which is inhabited by $Loc(\ell)$. The type asserts that ℓ fits into an integer of type α, but does not assert any ownership of the memory pointed to by ℓ. Moreover, it also asserts that ℓ is in bounds (or one past the end) of its allocation—expressed with $\mathsf{in\_bounds}(\ell)$—which is needed on integer-to-pointer casts. Piecing everything together, we obtain the following rule:

$$\frac{\ell \lhd_\mathsf{v} \&_\mathsf{own}(\tau) \mathrel{-\!\!*} \left(\mathsf{in\_bounds}(\ell) \land \ulcorner \ell.2 \in \alpha \urcorner \land \mathsf{alive}(\ell) \land \forall v.\, G(v, \ell \ @ \ \mathsf{intptr}(\alpha))\right)}{\vdash_{\text{EXPR}} \mathsf{Cast}_{\mathsf{void}^* \hookrightarrow \alpha} \ \ell \ \text{ when } \ \ell \lhd_\mathsf{v} \&_\mathsf{own}(\tau) \ \{v_{res}, \tau_{res}.\ G(v_{res}, \tau_{res})\}}$$

Although it has been slightly simplified for presentation purposes, this rule is quite a mouthful. Let us go over it step by step, starting with the conclusion which is an expression typing judgement parameterised by the continuation G. The expression that is being type checked is $\mathsf{Cast}_{\mathsf{void}^* \hookrightarrow \alpha} \ \ell$: a cast of location ℓ (or more precisely, ℓ encoded as a value) into a C integer of type α. Note that the rule only applies if a type assignment $\ell \lhd_\mathsf{v} \&_\mathsf{own}(\tau)$ is provided.

Let us now go over the premise of our rule, which is expressed using (a fragment of) separation logic. First, the magic wand $\mathrel{-\!\!*}$ ensures that its left-hand side, *i.e.*, the type assignment $\ell \lhd_\mathsf{v} \&_\mathsf{own}(\tau)$, can be used to prove its right-hand side, *i.e.*, the conjunction of four predicates. This right-hand side contains the side-conditions that we mentioned above ($\mathsf{in\_bounds}(l)$: ℓ is in bounds of its allocation, $\ell.2 \in \alpha$: the address of ℓ fits in the integer type α, $\mathsf{alive}(\ell)$: ℓ is still alive). Moreover, the last conjunct requires proving the continuation G for the resulting type $\ell \ @ \ \mathsf{intptr}(\alpha)$.

Interestingly, the premise of our rule uses the standard conjunction $\land$ instead of the separating conjunction $*$ for the side-conditions. This allows reusing ownership assertions like $\ell \lhd_\mathsf{v} \&_\mathsf{own}(\tau)$ between the proofs of the side-conditions (in particular, $\mathsf{alive}(\ell)$) and the continuation G.

To be more concrete, let us now go back to the proof outline of Figure 3, which sketches the verification of the `tag` function. The standard owned pointer type $\ell \ @ \ \&_\mathsf{own}(\tau)$ of RefinedC is not expressive enough to account for pointer tagging. As a consequence, `tag` relies on a specific tagged pointer type $\mathsf{tagged}(m, \tau)$ that we added to the RefinedC standard library, and that is refined by a pair of a base pointer and a tag value. Intuitively, the type $(\ell, t) \ @ \ \mathsf{tagged}(m, \tau)$ contains the same ownership as $\ell \ @ \ \&_\mathsf{own}(\tau)$, but a value at that type corresponds to $\mathsf{add\_offset}(\ell, t)$ (assuming suitable alignment of ℓ) instead of just ℓ. As a consequence, the above typing rule for casts does not apply, and we must instead use a similar typing rule for tagged pointers.

$$\frac{v \lhd_\mathsf{v} (\ell, t) \mathrel{@} \mathsf{tagged}(m, \tau) \mathrel{-\!\!*} \left(\cdots \land \ulcorner \ell.2 + t \in \alpha \urcorner \land \mathsf{alive}(\ell) \land \forall v.\, G(v, \mathsf{add\_offset}(\ell, t) \mathrel{@} \mathsf{intptr}(\alpha))\right)}{\vdash_{\text{EXPR}} \mathsf{Cast}_{\mathsf{void}^* \hookrightarrow \alpha} \ v \ \text{ when } \ v \lhd_\mathsf{v} \ (\ell, t) \ @ \ \mathsf{tagged}(m, \tau) \ \{v_{res}, \tau_{res}.\ G(v_{res}, \tau_{res})\}}$$

For brevity, we elide details related to alignment and bounds condition with an ellipsis. In Figure 3, the above rule is applied when verifying the cast on Line 7. Note that the context prior to applying the rule, shown on Line 6, contains a type assignment of the right shape (for variable `p`): it is consumed when the rule is applied, but immediately placed back in the context (Line 8) thanks to the magic wand in the rule's premise. Aside from the continuation, all conjuncts in the rule's premise can be proved using the assertions from the context. For example, $\mathsf{alive}(\ell)$ can be proved thanks to a combination of `type_alive_own ty` and the type assignment for `p`. So, after applying the rule, we are left with proving the continuation for some value v corresponding here to variable `i` (since the proof sketch also does the store in the same step, which involves another rule not shown here), whose

type corresponds to the inferred type passed to the continuation (*i.e.*, add_offset(ℓ, t) @ intptr(α) where (ℓ, t) corresponds to parameter `r`, and α to `uintptr_t`).

Arithmetic. Let us now see how a value of type ℓ @ intptr(α)—resulting from a pointer-to-integer cast—can be used. First, one can perform standard integer operations on it, as seen in the example of §2.3. As formalised by the VIP-ARITH-OP reduction rule, these operations are allowed on integers with provenance, but any provenance is lost in the resulting value. In the type system, this is reflected by the fact that ℓ @ intptr(α) is a subtype of $\ell.2$ @ int(α).

$$\frac{\ulcorner n = \ell.2 \urcorner * G}{\vdash_{\text{SUBSUME}} v \triangleleft_{\text{v}} \ell \ @ \ \text{intptr}(\alpha) <: v \triangleleft_{\text{v}} n \ @ \ \text{int}(\alpha) \ \{G\}}$$

The above subtyping has to be applied before performing an arithmetic operation on ℓ @ intptr(α), which reflects into the type system that arithmetic operations drop provenance in integers.

Going back to the proof sketch of Figure 3, the above subsumption rule is applied to change the type of variable `i` between Line 8 and Line 9, losing provenance information before performing the arithmetic on Line 10.

Round-trip casts. If no arithmetic is performed on a value of type ℓ @ intptr(α), then it can be cast back into a pointer to complete a round-trip cast. This is supported in the operational semantics by the VIP-I-P-CAST-1 reduction rule: casting $Loc(\ell)$ into the pointer ℓ is possible if ℓ is alive and in bounds of (or one past) its allocation. This is reflected by the following typing rule:

$$\frac{\text{alive}(\ell) \wedge G(\ell, \&_{\text{own}}(\text{place}(\ell)))}{\vdash_{\text{EXPR}} \text{Cast}_{\alpha \hookleftarrow \text{void}^*} v \ \text{ when } \ v \triangleleft_{\text{v}} \ell \ @ \ \text{intptr}(\alpha) \ \{v_{res}, \tau_{res}.\ G(v_{res}, \tau_{res})\}}$$

In the premise, the alive(ℓ) assertion ensures that ℓ is alive. Note, however, that no in_bounds(ℓ) precondition is required, since the fact that ℓ satisfied the right bounds condition is already ensured by ℓ @ intptr(α) via a side-condition of the typing rule for pointer-to-integer casts.

Note that the resulting pointer is assigned type $\&_{\text{own}}(\text{place}(\ell))$ in the continuation: it corresponds to an owned pointer without any particular attached ownership for the pointed memory. This type can then be combined in a later step with ownership for the memory pointed to by ℓ to obtain a value of type $\&_{\text{own}}(\tau)$ (where τ is the type of the pointed memory).

Integer-to-Pointer Casts. In case arithmetic has been performed on an integer, it cannot be cast to a pointer using a round-trip cast, and one must instead use VIP's `copy_alloc_id` instruction. Similar to the other operators discussed above, it is straightforward to reflect the reduction rule corresponding to the operation (here VIP-COPY-A-ID) into the type system:

$$\frac{\ell \triangleleft_{\text{v}} \&_{\text{own}}(\tau) \twoheadrightarrow \left(\text{in\_bounds}(\ell.1, n) \wedge \text{alive}(\ell.1, n) \wedge \ G((\ell.1, n), \&_{\text{own}}(\text{place}(\ell.1, n)))\right)}{\vdash_{\text{EXPR}} \text{CopyAllocID}_{\alpha}(v, \ell) \ \text{ when } \ v \triangleleft_{\text{v}} n \ @ \ \text{int}(\alpha) \ \text{ and } \ \ell \triangleleft_{\text{v}} \&_{\text{own}}(\tau) \ \{v_{res}, \tau_{res}.\ G(v_{res}, \tau_{res})\}}$$

As for previous rules, in_bounds(ℓ) and alive(ℓ) are used to ensure that ℓ is in bounds of (or one past) its allocation and that it is alive, as mandated by the operational semantics. The result value of the operation passed to the continuation G is—as expected—a pointer value $(\ell.1, n)$, whose provenance is that of ℓ, and whose address is n.

The proof outline in Figure 3 uses a variant of the above typing rule for tagged pointers on Line 13. Here, ℓ is instantiated with $r.1$ and n with $r.1.2 + t$. The bounds and liveness side-conditions are discharged easily. The store on Line 13 assigns the $\&_{\text{own}}(\text{place}(r.1.1, r.1.2 + t))$ resulting from the rule above to `q`, which then is directly unfolded to asserting that `q` is equal to the location $(r.1.1, r.1.2 + t)$ (see Line 14). Based on this fact, ownership can be transferred from the type assignment for `p` to build a new type assignment for `q` between Line 14 and Line 17.

6.3 Semantic Equality of Pointers

Let us now briefly discuss a difficulty that we encountered when adapting the RefinedC automation to work with integer-pointer casts. In RefinedC, the typing context is populated with type assignments, and the type checker often has to traverse the context to find a type assignment for a specific location. In the original RefinedC [Sammler et al. 2021], each potential location found in the context is *syntactically* matched with the location that is being looked for. Although this simple approach worked well for the programs considered by Sammler et al. [2021], it breaks down in the presence of integer-pointer casts. Indeed, integer-pointer casts allow the programmer to create pointers that are *semantically* the same—but differ syntactically. For example, suppose that we start from a location ℓ, convert it to an integer to perform some arithmetic eventually resulting in $\ell.2 + n$, and then rely on `copy_alloc_id` to build a pointer with the provenance of ℓ. At the end of the process, the resulting pointer will have the shape $(\ell.1, \ell.2 + n)$, but the context may perhaps only contain an assignment for $\mathrm{add\_offset}(\ell, n)$, which is semantically equal but syntactically different. Thus, the RefinedC type checker would not find a type assignment in the context and emit an error. To address this problem, RefinedC-VIP extends the type checker of RefinedC with a solver for semantic equality of locations. The new type checker uses this solver when searching for a type assignment in the context and thus successfully handles the situation described above.

7 CASE STUDIES AND EVALUATION

In this section, we report on two different evaluations of our VIP memory model. First, in §7.1, we use examples verified in RefinedC-VIP to show that VIP is well-suited for verification. Then in §7.2, we empirically evaluate VIP against PNVI-ae-udi using an implementation extending Cerberus [Memarian et al. 2019]. Finally, we evaluate the impact of `copy_alloc_id` on the assembly code generated by mainstream compilers in §7.3. Both RefinedC-VIP and Cerberus, together with all the test programs used for the evaluation, are provided in Lepigre et al. [2021b].

7.1 Verification Case Studies Using RefinedC-VIP

To evaluate RefinedC-VIP, we first verified small programs exercising different forms of integer-pointer casts, with or without `copy_alloc_id` when appropriate. In particular, some of these examples are carefully crafted to show that the system can handle one-past-the-end pointers, either as part of a round-trip cast or using `copy_alloc_id`.

We also used RefinedC-VIP to verify real-world idioms using integer-pointer casts, either directly taken from or inspired by mainstream systems C code. We report on these in the following paragraphs, and Figure 7 additionally gathers more fine-grained evaluation data including (for each verified function): the original numbers of lines of code (when applicable), the amount of changes required in the code, and the number of lines of manually written RefinedC annotations.

#1: Lock-ordering via addresses. To avoid deadlocks, it is a standard idiom to acquire locks in the order of their addresses, as done for example in the Hafnium hypervisor's spinlock library [Hafnium 2020]. In its original version, the code relies on a pointer comparison, which, under PNVI-ae-udi, implies that it should only be used on locks with the same provenance (it is the case in Hafnium: all locks are allocated in a single array). However, one can lift this restriction by performing pointer-to-integer casts and comparing the resulting integer address. Using RefinedC-VIP, we have verified both the original Hafnium function, and the more general version using pointer-to-integer casts.

#2: Small integers and pointers. Another use of integer-pointer casts, found in the OCaml runtime [Leroy and Doligez 1996], is the encoding of both (suitably aligned) pointers and 63-bit

	Verified function	Code	Changes	CopyAId	Spec	Proof	Time
#1	`sl_lock_both` (original)	9	0/0	0	7	0	3.5s
	`sl_lock_both` (with casts)	9	0/1	0	6	0	3.9s
#2	`client`	14	-/-	0	1	1	6.5s
#3	`tag_of`	5	-/-	0	5	1	3.9s
	`tag`	6	-/-	1	4	1	6.4s
	`untag` (using `tag`)	3	-/-	0	4	0	2.1s
	`untag` (direct arithmetic)	4	-/-	1	4	0	4.8s
	`test` (client function)	7	-/-	0	1	0	4.5s
#4	`struct region` (replaces globals)	3	6/3	-	8	-	-
	`hyp_early_alloc_nr_used_pages`	4	0/0	0	4	2	2.9s
	`hyp_early_alloc_contig`	16	0/0	1	6	2	14.6s
	`hyp_early_alloc_page`	4	0/0	0	6	0	2.3s
	`hyp_early_alloc_init`	9	1/4	0	5	1	3.7s

Fig. 7. Evaluation data for the RefinedC-VIP case studies. Code: lines of C code for the (original) function (or the global variables being replaced for `struct region`). Changes: lines of code added / lines of code altered or removed (excluding insertion of `copy_alloc_id`). CopyAId: number of casts replaced by `copy_alloc_id`. Spec: lines of RefinedC annotations for the specification. Proof: lines of RefinedC annotations used for discharging side-conditions. Time: typical measured checking time (average of three runs) for the proof on a laptop with an Intel CORE i7 8th Gen processor and 16GB of RAM.

integers in 64-bit integers (using their first bit to distinguish the two cases). Since no arithmetic is performed in the pointer case, VIP can support this pattern without `copy_alloc_id`. We have successfully verified a small client using this pattern. However, unlike the original, our code uses unsigned integers as we cannot easily guarantee all pointer addresses to be representable in a signed 64-bit integer.

#3: Tagged pointers. As discussed in §2.3, we also verified the pointer tagging library of Figure 1. For this example, we leveraged the extensibility of the RefinedC type system to build the custom `&tagged` type representing a generic notion of tagged pointer. This demonstrates that RefinedC-VIP allows the encoding of specialised integer-pointer cast idioms via custom types, which enable clearer specifications for the user, and additionally ease the verification of the library. In particular, the `&tagged` type ensures that all tagged pointers are in bounds of their allocation (or one past its end), which is used to prove the premise of the typing rule for `copy_alloc_id` shown in §6.2.

#4: Early allocator from pKVM. We verified the page allocator used by the pKVM hypervisor (being developed as part of Linux [pKVM developers 2020]) during its initialisation. The allocator state consists of three global integer variables: (1) `base`, recording the start address of the memory region owned by the allocator, (2) `end`, recording its end, and (3) `cur`, giving the address at which the next allocation is performed. Initially `cur` is equal to `base`, but it is incremented on each allocation. Since the allocator state does not contain pointers, we leverage the ability of VIP to track provenance via integers. In particular, `base` is initialised with a pointer-to-integer cast from a pointer to the

memory backing the allocator, and we can retain the corresponding provenance using the `intptr` type. Since `base` is never modified, the provenance it carries is preserved, and it can be used when a newly allocated pointer is returned. In the original code, the pointer returned by the allocator is created with a cast `(void*) cur`, but it is essentially unusable in VIP since it lacks a provenance. We thus replace the cast by `copy_alloc_id(cur, (void*) base)`, which ensures that the resulting pointer receives the provenance stored in `base`, and that it can be used to access memory.

Introducing `copy_alloc_id` is the only substantial change required to verify the code. However, due to a limitation of RefinedC, we also need to pack the three global variables in a struct, since they could otherwise not be easily tied together in an invariant. This superficial change is orthogonal to the integer-pointer-casts studied in this paper, and it could be avoided with additional engineering work. All ownership reasoning for the example is verified automatically by RefinedC-VIP, but custom automation is required for several pure side-conditions involving shifts on integers.

7.2 Empirical Evaluation of VIP

The PNVI-ae-udi model was implemented as part of the Cerberus C semantics, an executable model of a large fragment of C11 with a parametric memory model. For further validation, we implemented VIP in the same manner, as an OCaml module implementing Cerberus's memory interface, on which the presentation in §3.1 is based. From this, we obtain a VIP version of the Cerberus test oracle. The implementation closely follows the formal presentation we give in this paper, with the exception of the **allocate** event, for which we fix the selection of the new allocation's address with the same deterministic scheme used in the implementation of PNVI-ae-udi.

To validate the behaviour of VIP with respect to PNVI-ae-udi, we have run the test suite given in Gustedt et al. [2020, Annex A.6] consisting of 58 C files between 10 and 30 lines. Of these, we exclude 14 tests whose execution path of interest is not triggered on either model as a result of the allocation scheme. For the tests performing integer-pointer casts, we have additionally run both models on an annotated version where the casts (whose result is typically directly stored) have been replaced by a corresponding call to `copy_alloc_id`. As expected from the refinement proof, for all 44 annotated and unannotated tests, when VIP gives a defined behaviour, so does PNVI-ae-udi. There are 17 unannotated tests deemed undefined by VIP while PNVI-ae-udi gives them a defined behaviour. This is again expected, as VIP has more undefined behaviours. Out of these, 15 tests recover the same defined behaviour as in PNVI-ae-udi when adding `copy_alloc_id` annotations. The 2 other tests cannot be annotated: one involves a convoluted user implementation of `memcpy`, while the other is meant to test compiler optimisation opportunities and is therefore not relevant here. There are 2 tests deemed undefined by both models without annotations, which become defined in both when annotated. The remaining tests or either defined in both models (12), or undefined in both models (13), regardless of whether the integer-pointer casts are annotated.

7.3 Evaluation of the Generated Assembly Code

As we have seen, adapting C code to run under the VIP memory model involves turning integer-to-pointer casts into calls to `copy_alloc_id`. However, this is not without consequences for compilation: `copy_alloc_id` is precisely used to prevent certain optimisations. One may thus wonder: what is the impact of `copy_alloc_id` on the runtime performance of the generated (optimised) assembly?

To evaluate this, we relied again (as in §7.2) on the 44 relevant examples of the test suite given in Gustedt et al. [2020, Annex A.6]. We ruled out 15 of them since their unannotated version has UB under PNVI-ae-udi, and 2 of them since they cannot be annotated to not have UB under VIP. Then, 12 of the remaining examples neither have UB under PNVI-ae-udi nor under VIP, and they also do not need any change for VIP, so they are trivially compiled in the same way. The remaining 15 examples do not have UB in PNVI-ae-udi, and their annotated versions do not have UB in VIP.

We compiled the two versions of each file (with option `-O3`) and compared the generated code. We repeated this for GCC-11, GCC-10, GCC-9, and Clang-12. The generated code is exactly identical on all examples when using GCC (any version we tried), and with Clang-12 there is a difference for only one of the examples (`provenance_basic_using_uintptr_t_auto_yx.c`). The reason for this difference is hard to track in the optimised code, but it seems unrelated: it only involves the setup of a call to `printf`. Regardless, the generated assembly has the same instruction count.

8 RELATED WORK

Over the last few decades, increasingly extensive formal versions of C have been developed [Gurevich and Huggins 1992; Cook and Subramanian 1994; Papaspyrou 1998; Norrish 1998; Leroy 2006; Batty et al. 2011; Ellison and Rosu 2012; Hathhorn et al. 2015; Krebbers and Wiedijk 2015; Krebbers 2015; Memarian et al. 2016, 2019]. These formal semantics focus on a variety of aspects of C, including control flow, expression evaluation, concurrency, and the type system. In this paper, we are concerned primarily with the C memory model, so we focus our discussion on that.

Concrete memory models. As it was not until 2004 that the ISO WG14 C standards committee confirmed the concept of provenance of pointers [WG14 2004], much research on formal C semantics has assumed a concrete memory model. Indeed, recent verification tools—such as those developed in the seL4 project [Tuch et al. 2007; Klein et al. 2009; Greenaway et al. 2013]—still use a concrete memory model. To do that soundly, they restrict the C dialect and incorporate translation validation [Sewell et al. 2013]. Since concrete memory models are close to actual hardware, they are commonly used in verification tools for assembly, *e.g.*, [Chlipala 2011; Jensen et al. 2013; Myreen 2009].

Abstract memory models. Abstract memory models were introduced in the CompCert project [Leroy 2006; Leroy and Blazy 2008] to prove correctness of an optimising C compiler. Leroy et al. [2012] extend the original CompCert memory model with support for byte-level representations of integers and floats, and a permission model. Krebbers et al. [2014] further extend the CompCert memory model with support for user-defined `memcpy` and one-past-the-end pointers. Krebbers [2013, 2015] presents a typed variant of the CompCert memory model with tree-like structures to model the subobject structure needed to support the C standard's notion of effective types. None of these extensions support integer-pointer casts beyond roundtrip casts.

"Goldilocks" memory models: neither too concrete nor too abstract. In addition to PNVI, Memarian et al. [2019] propose the PVI memory model in which pointer provenance is propagated through integer operations. This model has the drawback of falsifying algebraic properties of integer operations [Memarian et al. 2019, p9], and would also make reasoning about integer operations more complicated in a verification tool like RefinedC.

Besson et al. [2014, 2015, 2019] extend CompCert's memory model with symbolic pointer expressions, and adapt the correctness proofs of 12 passes of the CompCert compiler accordingly. When performing an integer operation (*e.g.*, a bitwise operation) on a pointer representation, they keep track of a symbolic expression, which is forced when a concrete block-offset pair is needed (*e.g.*, to perform a load or store). While the use of symbolic expressions avoids the need for concrete addresses, this comes at the expense that the semantics becomes non-trivial to execute (they use SMT solvers for that) and that certain programming idioms are prohibited (such as hashmaps with integer keys, and the use of pointer comparison to determine a lock order). The VIP and PNVI-ae-udi memory models are executable and support the aforementioned programming idioms.

Kang et al. [2015] extend the CompCert memory model with support for concrete addresses, and present a technique for verifying compiler optimisations w.r.t. their model. Blocks in their model initially have an abstract block identifier (which can be seen as a provenance, like in CompCert),

but when performing a pointer-to-integer cast, the block is assigned a concrete address. Their approach simplifies reasoning about address exposure: every memory block with a concrete address is considered to be exposed, and plays an important role in their technique for verifying compiler optimisations (in particular, optimisations that remove allocations). Contrary to PNVI-ae-udi and VIP, their approach prohibits integer-to-pointer casts involving one-past-the-end pointers.

Lee et al. [2018] describe a memory model for LLVM IR that supports integer-pointer casts. The semantics of LLVM is quite different from source-level C, however. An in-depth comparison between the semantics of Lee et al. and PNVI is given in Memarian et al. [2019].

Aside from integer-pointer casts, researchers have developed memory models that give a more realistic account of memory consumption, *e.g.*, [Besson et al. 2019; Wang et al. 2019b] in the context of CompCert. Similar to PNVI-ae-udi, our VIP memory model ignores these issues.

Verification tools for C. There are many tools for C verification (*e.g.*, Rondon et al. [2010]; Stefanescu [2014]; Kirchner et al. [2015]), but most of the tools either use a concrete memory model [Tuch et al. 2007; Cohen et al. 2009; Jacobs et al. 2011], or the papers do not discuss the underpinning memory model or their support for intricate C features like integer-pointer casts at all. There are exceptions, however. Notably, several verification tools built on top of CompCert—such as VST [Cao et al. 2018] and CCAL [Gu et al. 2018]—inherit CompCert's limited support for integer-pointer casts but employ various tricks to get around its limitations. In particular, the verification of an OCaml-style garbage collector by Wang et al. [2019a] extends CompCert with external functions specific for pointer-tagging. The verification of OS kernels and hypervisors by Gu et al. [2019] and Li et al. [2021] sidesteps integer-pointer casts altogether by using array indexing into a global array in which the kernel's allocatable data structures are stored. In contrast, with RefinedC-VIP, we have aimed to develop a verification tool that accounts for the integer-pointer cast idioms found in the wild, while also being sound under a realistic semantics.

ACKNOWLEDGMENTS

We thank our shepherd Ronghui Gu and the anonymous reviewers for their helpful feedback, and Deepak Garg, Neel Krishnaswami, Christopher Pulte, Jean Pichon-Pharabod, and Thomas Sewell for many useful discussions. Additionally, we thank Hong-Seok Kim, Chung-Kil Hur, Jieung Kim, Youngju Song, Ben Laurie, Sarah de Haas, the pKVM development team, and all others involved in the pKVM verification effort for their useful feedback and support.

This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289), in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (AdG grant agreement No 789108, ELVER), in part by the Dutch Research Council (NWO), project 016.Veni.192.259, in part by the EPSRC Programme Grant REMS: Rigorous Engineering of Mainstream Systems (EP/K008528/1), in part by a Google PhD Fellowship for the second author, and in part by generous awards from Android Security's ASPIRE program and from Google Research.

REFERENCES

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL*. 55–66. https://doi.org/10.1145/1926385.1926394

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2014. A Precise and Abstract Memory Model for C Using Symbolic Values. In *APLAS (LNCS, Vol. 8858)*. Springer, 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 *ITP (LNCS, Vol. 9236)*. Springer, 67–83. https://doi.org/10.1007/978-3-319-22102-1_5

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2019. CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics. *J. Autom. Reason.* 63, 2 (2019), 369–392. https://doi.org/10.1007/s10817-018-9496-y

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A separation logic tool to verify correctness of C programs. *JAR* 61, 1-4 (2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

Adam Chlipala. 2011. Mostly-automated verification of low-level programs in computational separation logic. *PLDI* (2011). https://doi.org/10.1145/1993498.1993526

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

Jeffrey Cook and Sakthi Subramanian. 1994. *A Formal Semantics for C in Nqthm.* Technical Report 517D. Trusted Information Systems.

Will Deacon. 2020. Virtualization for the Masses: Exposing KVM on Android. https://www.youtube.com/watch?v=wY-u6n75iXc. KVM Forum Talk.

Jake Edge. 2020. KVM for Android. https://lwn.net/Articles/836693/.

Chucky Ellison and Grigore Rosu. 2012. An executable formal semantics of C with applications. In *POPL*. ACM, 533–544. https://doi.org/10.1145/2103656.2103719

David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. 2013. Don't sweat the small stuff: Formal verification of C code without the pain. *PLDI* (2013). https://doi.org/10.1145/2594291.2594296

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *POPL*. ACM, 595–608. https://doi.org/10.1145/2676726.2676975

Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. 2019. Building certified concurrent OS kernels. *Commun. ACM* 62, 10 (2019), 89–99. https://doi.org/10.1145/3356903

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. *PLDI* (2018). https://doi.org/10.1145/3192366.3192381

Yuri Gurevich and James K. Huggins. 1992. The Semantics of the C Programming Language. In *CSL (LNCS, Vol. 702)*. Springer, 274–308. https://doi.org/10.1007/3-540-56992-8_17

Jens Gustedt, Peter Sewell, Kayvan Memarian, Victor B. F. Gomes, and Martin Uecker. 2020. N2577: A Provenance-aware Memory Object Model for C. Working Draft Technical Specification TS 6010. ISO/IEC JTC1/SC22/WG14 N2577 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2577.pdf.

Hafnium. 2020. Hafnium. https://review.trustedfirmware.org/plugins/gitiles/hafnium/hafnium/+/HEAD/README.md.

Chris Hathhorn, Chucky Ellison, and Grigore Rosu. 2015. Defining the undefinedness of C. In *PLDI*. ACM, 336–345. https://doi.org/10.1145/2737924.2737979

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NFM (LNCS, Vol. 6617)*. 41–55. http://dx.doi.org/10.1007/978-3-642-20398-5_4

Jonas Braband Jensen, Nick Benton, and Andrew Kennedy. 2013. High-level separation logic for low-level code. In *POPL*. ACM, 301–314. https://doi.org/10.1145/2429069.2429105

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the foundations of the Rust programming language. *PACMPL* 2, POPL (2018), 1–34. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269. https://doi.org/10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *POPL* (2015). https://doi.org/10.1145/2676726.2676980

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

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. In *SEFM (LNCS, Vol. 7504)*. 233–247. http://dx.doi.org/10.1007/s00165-014-0326-7

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.* 32, 1 (2014), 2:1–2:70. https://doi.org/10.1145/2560537

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal verification of an OS kernel. In *SOSP*. ACM, 207–220. https://doi.org/10.1145/1629575.1629596

Robbert Krebbers. 2013. Aliasing Restrictions of C11 Formalized in Coq. In *CPP (LNCS, Vol. 8307)*.

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

Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The essence of higher-order concurrent separation logic. In *ESOP (LNCS, Vol. 10201)*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. 2014. Formal C Semantics: CompCert and the C Standard. In *ITP (LNCS, Vol. 8558)*. Springer, 543–548. https://doi.org/10.1007/978-3-319-08970-6_36

Robbert Krebbers and Freek Wiedijk. 2015. A Typed C11 Semantics for Interactive Theorem Proving. In *CPP*. ACM, 15–27. https://doi.org/10.1145/2676724.2693571

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

Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. 2021a. VIP: Verifying Real-World C Idioms Involving Integer-Pointer Casts (Appendix). https://doi.org/10.5281/zenodo.5662349

Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. 2021b. VIP: Verifying Real-World C Idioms Involving Integer-Pointer Casts (Artifact). https://doi.org/10.5281/zenodo.5662349

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*. ACM, 42–54. https://doi.org/10.1145/1111037.1111042

Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert memory model, version 2.* Technical Report RR-7987. Inria.

Xavier Leroy and Sandrine Blazy. 2008. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR* 41, 1 (2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

Xavier Leroy and Damien Doligez. 1996. OCaml runtime representation of values. https://github.com/ocaml/ocaml/blob/trunk/runtime/caml/mlvalues.h.

Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. A Secure and Formally Verified Linux KVM Hypervisor. In *IEEE Symposium on Security and Privacy*.

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. *PACMPL* 3, POPL (2019), 67:1–67:32. https://doi.org/10.1145/3290380

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. (2016), 1–15. https://doi.org/10.1145/2908080.2908081

Magnus Oskar Myreen. 2009. *Formal verification of machine-code programs.* Ph.D. Dissertation. University of Cambridge, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.611450

Michael Norrish. 1998. *C formalised in HOL.* Ph.D. Dissertation. University of Cambridge.

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *CSL (LNCS, Vol. 2142)*. 1–19. https://doi.org/10.1007/3-540-44802-0_1

Nikolaos Papaspyrou. 1998. *A Formal Semantics for the C Programming Language.* Ph.D. Dissertation. National Technical University of Athens.

pKVM developers. 2020. Initial allocator of the pKVM hypervisor. https://github.com/torvalds/linux/blob/master/arch/arm64/kvm/hyp/nvhe/early_alloc.c.

John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. (2002), 55–74. https://doi.org/10.1109/LICS.2002.1029817

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-level liquid types. In *POPL*. 131–144. https://doi.org/10.1145/1706299.1706316

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. (2021), 158–174. https://doi.org/10.1145/3453483.3454036

Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013), 22:1–22:50. https://doi.org/10.1145/2487241.2487248

Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *PLDI*. ACM, 471–482. https://doi.org/10.1145/2491956.2462183

Andrei Stefanescu. 2014. MatchC: A Matching Logic Reachability Verifier Using the K Framework. *Electron. Notes Theor. Comput. Sci.* 304 (2014), 183–198. https://doi.org/10.1016/j.entcs.2014.05.010

Harvey Tuch, Gerwin Klein, and Michael Norrish. 2007. Types, bytes, and separation logic. In *POPL*. ACM, 97–108. https://doi.org/10.1145/1190216.1190234

Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. 2019a. Certifying graph-manipulating C programs via localizations within data structures. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 171:1–171:30. https://doi.org/10.1145/3360597

Yuting Wang, Pierre Wilke, and Zhong Shao. 2019b. An abstract stack based approach to verified compositional compilation to machine code. *Proc. ACM Program. Lang.* 3, POPL (2019), 62:1–62:30. https://doi.org/10.1145/3290375

WG14. 2004. Defect Report #260: Indeterminate values and identical representations. http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm