C provenance semantics: slides (extracts from N2363)

Peter Sewell, Kayvan Memarian, Victor B. F. Gomes, Jens Gustedt, Martin Uecker

30 April 2019

with thanks to many others: Frédéric Besson, Richard Biener, Chandler Carruth, David Chisnall, Pascal Cuoq, Hal Finkel, Chung-Kil Hur, Ralf Jung, Robbert Krebbers, Chris Lattner, Juneyoung Lee, Xavier Leroy, Nuno Lopes, Justus Matthiesen, Paul McKenney, Santosh Nagarakatte, John Regehr, Martin Sebor, Kostya Serebryany, Richard Smith, Hubert Tong, Freek Wiedijk, Steve Zdancewic, other WG14 colleagues, EuroLLVM and GNU Cauldron attendees, and survey respondents.

WG14, London, 2019-04-29 – 2019-05-03

ISO/IEC JTC1/SC22/WG14 N2378

Context: previous discussions (selected)

WG14 Pittsburgh meeting, 2018-10

n2294: C Memory Object Model Study Group: Progress Report
n2263: Clarifying Pointer Provenance v4

WG14 Brno meeting, 2018-04 (CMOM SG created)

n2223: Clarifying the C Memory Object Model: Introduction to N2219 - N2222
n2219: Clarifying Pointer Provenance (Q1-Q20) v3
n2220: Clarifying Trap Representations (Q47) v3
n2221: Clarifying Unspecified Values (Q48-Q59) v3
n2222: Further Pointer Issues (Q21-Q46)

WG14 Pittsburgh meeting, 2016-10

n2089: Clarifying Unspecified Values (Draft Defect Report or Proposal for C2x)
n2090: Clarifying Pointer Provenance (Draft Defect Report or Proposal for C2x)
n2091: Clarifying Trap Representations (Draft Defect Report or Proposal for C2x)

WG14 London meeting, 2016-04

n2012: Clarifying the C memory object model
n2013: C Memory Object and Value Semantics: The Space of de facto and ISO Standards
n2014: What is C in Practice? (Cerberus Survey v2): Analysis of Response
n2015: What is C in practice? (Cerberus survey v2): Analysis of Responses - with Comments

Academic papers

Exploring C Semantics and Pointer Provenance (in POPL 2019, and as n2311)
Into the depths of C: elaborating the de facto standards (in PLDI 2016)

Elsewhere and Previously

WG21 p0137r1: Core Issue 1776: Replacement of class objects containing reference members (in C++17) (2016-06)
WG21 p0593r3: Implicit creation of objects for low-level object manipulation (2019-01)
In OOPSLA 2018: Reconciling High-level Optimizations and Low-level Code in LLVM (2018-11)
n1818 / DR451: Defect Report 451 (2014-04)
n1637: Subtleties of the ANSI/ISO C standard (2012-09)
DR260: indeterminate values and identical representations (2004-09)
mail 9350: What is an Object in C Terms? (2001-09)

Study group activity

Activity:

- teleconferences, roughly every two weeks

  (Hal Finkel, Jens Gustedt, Victor Gomes, Kayvan Memarian, Martin Sebor, Peter Sewell, Hubert Tong, Martin Uecker,...)
- mailing list: `https://lists.cam.ac.uk/mailman/listinfo/cl-c-memory-object-model`
- discussion with C++, Clang, and GCC folk:
  - `gcc@gcc.gnu.org`, `ub@isocpp.open-std.org`, `parallel@lists.isocpp.org`
  - keynote at EuroLLVM 2018
  - talk at GNU Tools Cauldron 2018

Focus:

Pointer provenance

Subobject provenance and effective types

Uninitialised reads, trap representations, and padding bytes

Further pointer issues

This meeting (WG14 London, 2019-04)

Well-developed proposal for pointer provenance:

- **Examples:** [these slides are extracts from this]
  n2363: C provenance semantics: examples
- **Proposed standard text diff:**
  n2362: Moving to a provenance-aware memory object model for C
  n2328: Introduce the term storage instance
- **Mathematical semantics:**
  n2364: C provenance semantics: detailed semantics (for PNVI-plain, PNVI address-exposed, PNVI address-exposed user-disambiguation, and PVI models)
- **Executable Web-GUI semantics in Cerberus:**
  `http://cerberus.cl.cam.ac.uk/cerberus`

Also:

- n2369: Pointer lifetime-end zap

Basic pointer provenance

Recall: pointers are typically simple concrete addresses at runtime, but compilers do *provenance-based alias analysis*:

```c
// provenance_basic_global_yx.c
#include <stdio.h>
#include <string.h>
int y=2, x=1;
int main() {
  int *p = &x;
  int *q = &y;
  p=p+1;
  printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11;  // does this have undefined behaviour?
    printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
  }
}
```

Clang 6.0 -O2	x=1 y=11 *p=11 *q=11
GCC 8.1 -O2	x=1 y=2 *p=11 *q=2
ICC 19 -O2	x=1 y=2 *p=11 *q=11 (with x and y swapped)

Basic pointer provenance

To make that GCC and ICC compiler optimisation legal w.r.t. the standard, this program has to be deemed to have UB, so we have to recognise that the `p` in `*p=11` is a one-past pointer, even though it has the same numeric address as `&y`.

DR260 CR (2001) hints at this:

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

but it was never incorporated in the standard text, and it gives no more detail. That leaves unclear whether some programming idioms are allowed or not, and what compiler alias analysis and optimisation are allowed to do.

Our proposal

Our proposal in n2362/n2363/n2364 clarifies this. It reconciles existing C programming practice, compiler implementation practice, and the standard text, as best we can, with a well-defined and reasonably simple semantics.

We aim to be conservative with respect to all those – as far as possible, the proposal is capturing the status quo in the specification. The proposal doesn't involve any new features or change to the language syntax.

Our proposal: the basic idea

We associate a *provenance* @i with every pointer value in the abstract machine, identifying the original storage instance it's derived from
(if any, or `@empty` otherwise).

- On every allocation (for static, thread, automatic, and allocated storage durations), the abstract machine chooses a fresh storage instance ID i (unique across the entire execution), and the resulting pointer value carries that as its provenance @i.
- Provenance is preserved by pointer arithmetic that adds or subtracts an integer to a pointer.
- At any access via a pointer value, its numeric address must be consistent with its provenance, with undefined behaviour otherwise

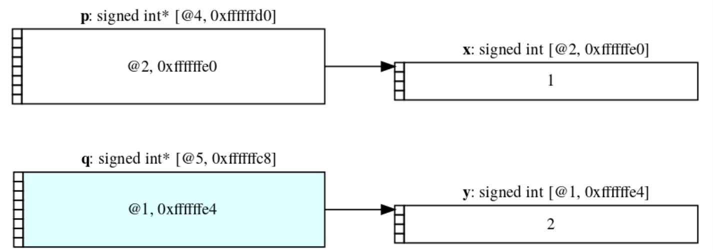
Our proposal: the basic idea

Cerberus ▾ | provenance_basic_global_yx.c ▾ | File ▾ | Model ▾ | Views ▾ | Step: 12 | Forward | Back | Restart | Search ▾ | Comp

provenance_basic_global_yx.c

```c
#include <stdio.h>
#include <string.h>
int y=2, x=1;
int main() {
  int *p = &x;
  int *q = &y;
  p=p+1;
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11;  // does this have UB?
    printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
  }
}
```

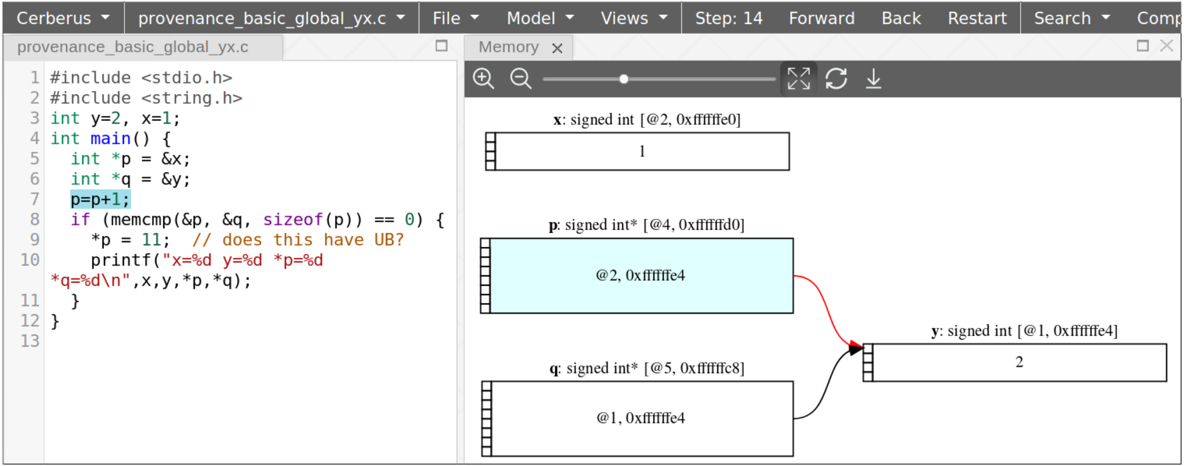
Memory ×



Note the storage instance IDs `@i` of the allocations and as part of the pointer values.

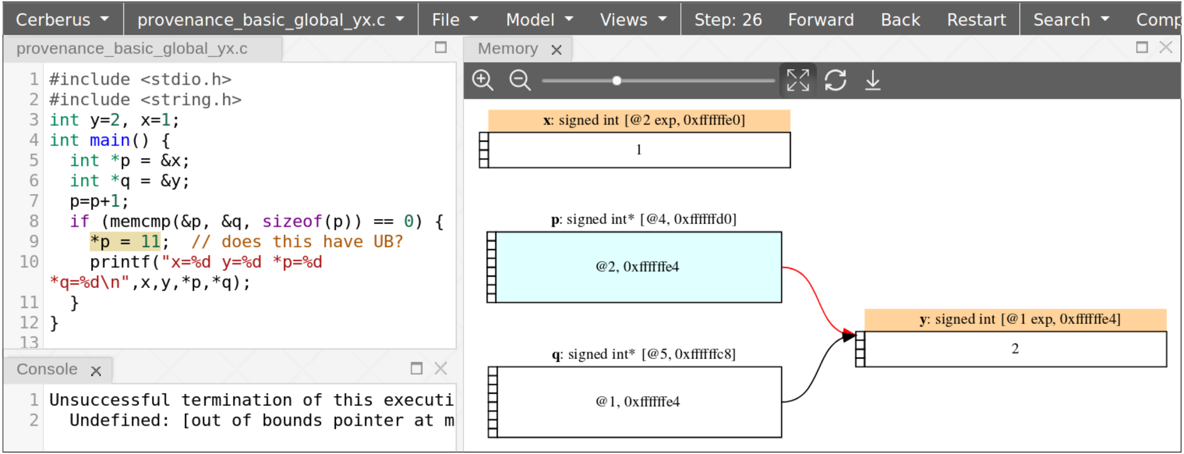
(try it live at `https://cerberus.cl.cam.ac.uk/cerberus?defacto/provenance_basic_global_yx.c`)

Our proposal: the basic idea



after the p=p+1, p has the address of y (in this execution), but it still has the provenance (@2) of x

Our proposal: the basic idea



at the $*$p=11 access, the address is not within the footprint of the allocation with that provenance, so the access is UB, as required

So far so good, but...

C provides many other ways to construct pointer values:

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

We have to address all these, and the impact on optimisation.

Design options

- **PVI**: track provenance via integer computation (n2090, n2263)
  Complex, poor algebraic properties, not good fit with implementation

Design options

- **PVI**: track provenance via integer computation (n2090, n2263)
  Complex, poor algebraic properties, not good fit with implementation

- **PNVI-plain**: don't track provenance via integers. Instead, at integer-to-pointer cast points, check whether the given address points within a live object and, if so, recreate the corresponding provenance.

Design options

- **PVI**: track provenance via integer computation (n2090, n2263)
  Complex, poor algebraic properties, not good fit with implementation

- **PNVI-plain**: don't track provenance via integers. Instead, at integer-to-pointer cast points, check whether the given address points within a live object and, if so, recreate the corresponding provenance.

- **PNVI-exposed-address (PNVI-ae)**: allow integer-to-pointer casts to recreate provenance only for storage instances that have previously been *exposed*, by a cast of a pointer to it to an integer type, by a read (at non-pointer type) of the representation of such a pointer, or by an output of such a pointer using `%p`.

Design options

- **PVI**: track provenance via integer computation (n2090, n2263)
  Complex, poor algebraic properties, not good fit with implementation

- **PNVI-plain**: don't track provenance via integers. Instead, at integer-to-pointer cast points, check whether the given address points within a live object and, if so, recreate the corresponding provenance.

- **PNVI-exposed-address (PNVI-ae)**: allow integer-to-pointer casts to recreate provenance only for storage instances that have previously been *exposed*, by a cast of a pointer to it to an integer type, by a read (at non-pointer type) of the representation of such a pointer, or by an output of such a pointer using `%p`.

- **PNVI exposed-address user-disambiguation (PNVI-ae-udi)**: a further refinement to support roundtrip casts, pointer to integer and back, of one-past pointers. Our preferred option.

Next

- **Idioms:** checking that various desirable idioms work
- **Implications for optimisation:** checking that various cases are UB, e.g. that function arguments can't alias its local variables
- **PNVI-plain vs PNVI-ae-*:** is the "exposed" machinery needed?
- **PNVI-ae vs PNVI-ae-udi:** what about one-past integers?
- **Experimental checks:** running the examples in an executable model and in GCC/Clang/ICC
- **The proposed text diff (Jens)** (n2362)
- **Precise semantics** (n2364)

More details and examples in n2363

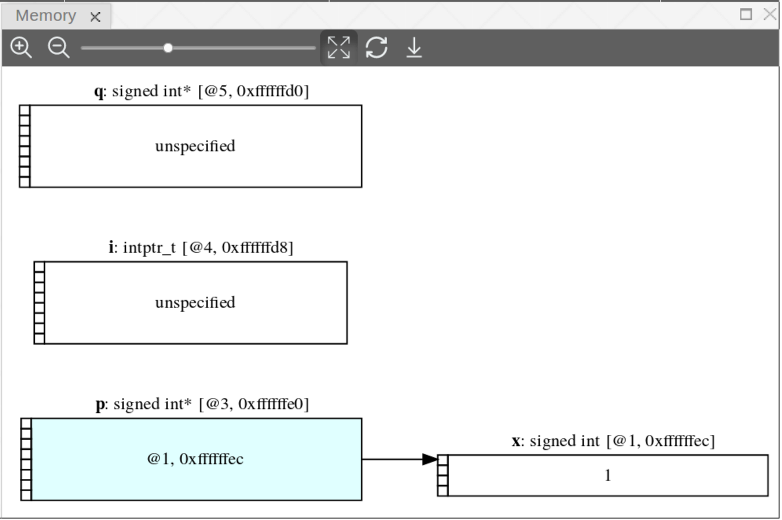
Idioms

Pointer/integer casts

Cerberus | provenance_roundtrip_via_intptr_t.c | File | Model | Views | Step: 9 | Forward | Back | Restart | Search

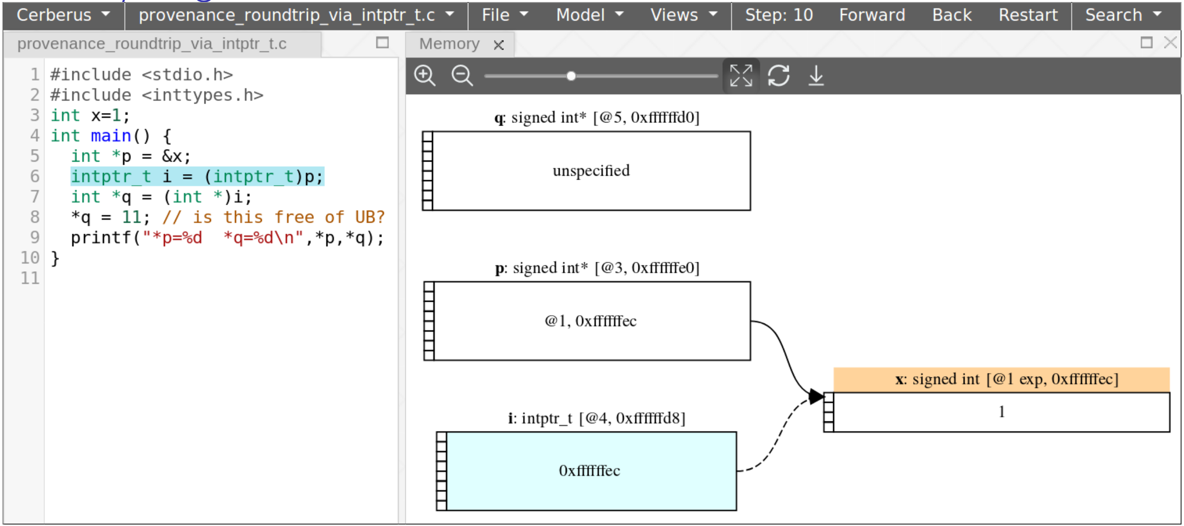
provenance_roundtrip_via_intptr_t.c

```c
#include <stdio.h>
#include <inttypes.h>
int x=1;
int main() {
  int *p = &x;
  intptr_t i = (intptr_t)p;
  int *q = (int *)i;
  *q = 11; // is this free of UB?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```



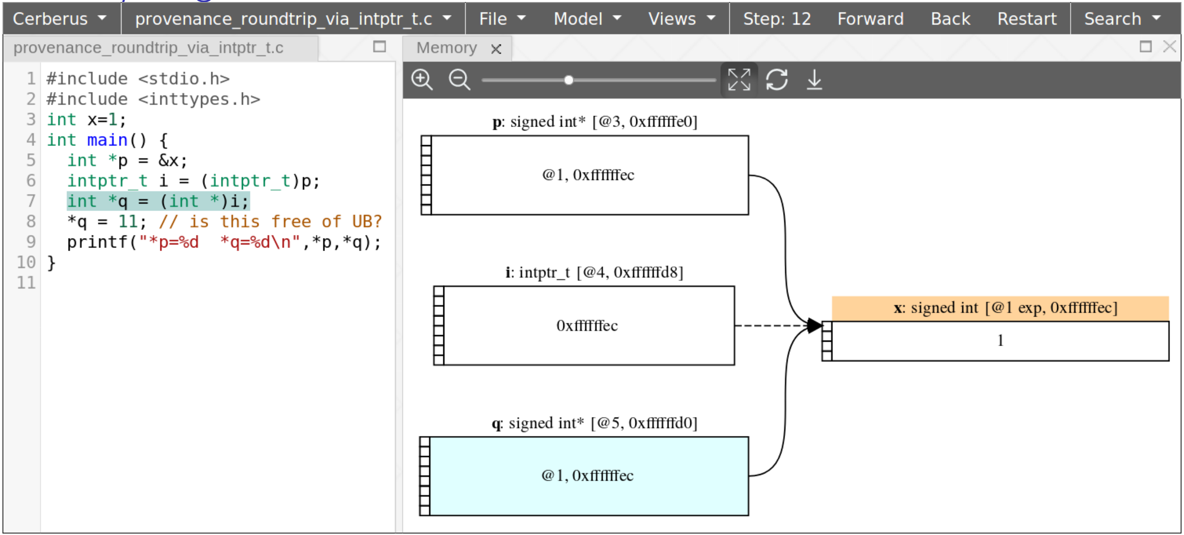
This is a simple pointer-to-integer-to-pointer roundtrip; the result should be usable for access.

Pointer/integer casts



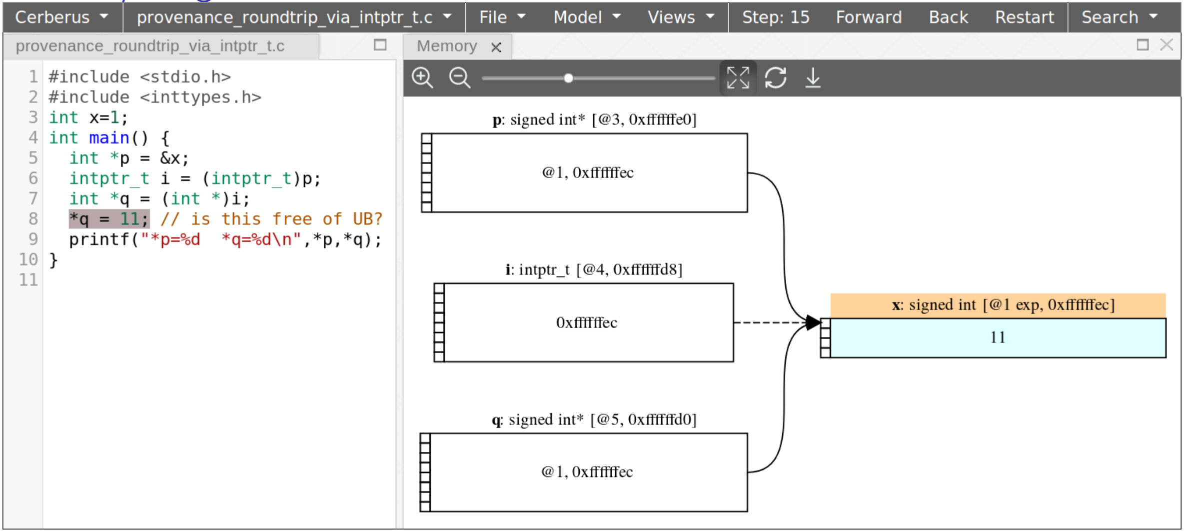
After the pointer-to-integer cast `(intptr_t)p`, the `x` allocation is marked as exposed.

Pointer/integer casts



so the integer-to-pointer cast `(int*)i`, of an integer within the footprint of `x`, will recover the provenance (`@1`) of `x`

Pointer/integer casts



and the access $*$q=11 is defined behaviour.

Pointer provenance for pointer bit manipulations

Common in practice. For example, assuming **`int`** has alignment at least 4, the low-order pointer bits are unused, and the implementation-defined pointer/integer conversions are as expected:

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

As before, `(uintptr_t)x` will expose `x`, so the `(int*)j` cast will recover the correct provenance, making the access `*r=11` legal.

Inter-object integer arithmetic

Can one move between objects with pointer arithmetic? No.

Can one move between objects with integer arithmetic? Debatable whether this must be supported – we get conflicting reports as to how important it is in practice, e.g. for XOR linked lists.

PNVI-* naturally allows it (if the implementation-defined pointer/integer conversions do).

Inter-object integer arithmetic

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

As before: the cast `(uintptr_t)&y` marks `y` as exposed, so the cast `p=(int*)(ux+offset)` can recover the provenance of `y` and make the access `*p=11` legal.

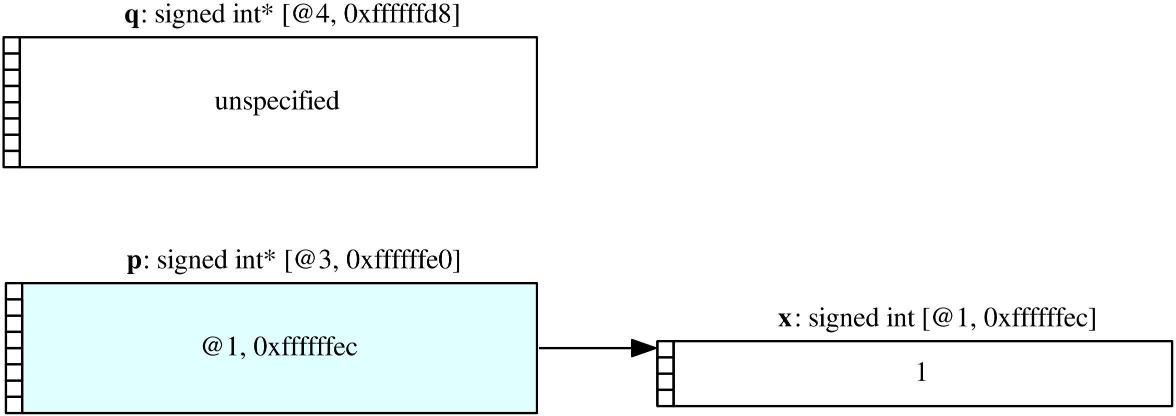
Copying pointer values bytewise, with user-memcpy

C supports manipulation of object representations, e.g. as in the following naive user implementation of `memcpy`:

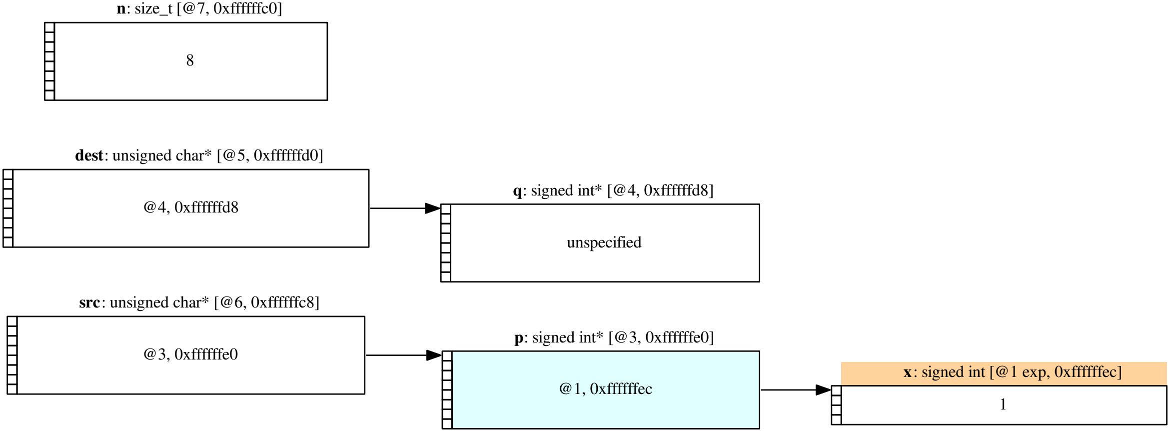
```c
// pointer_copy_user_dataflow_direct_bytewise.c
#include <stdio.h>
#include <string.h>
int  x=1;
void user_memcpy(unsigned char* dest,
                 unsigned char *src, size_t n) {
  while (n > 0)  {
    *dest = *src;
    src += 1; dest += 1; n -= 1;
  }
}
int main() {
  int *p = &x;
  int *q;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

which constructs a pointer value from copied bytes. This too should be allowed.

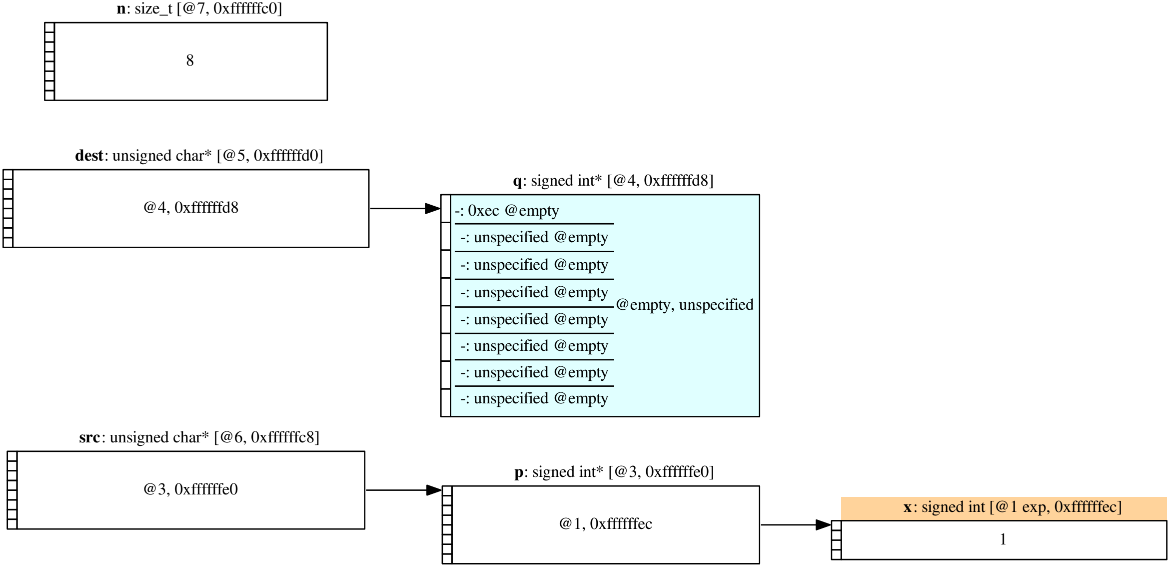
Copying pointer values bytewise, with user-memcpy



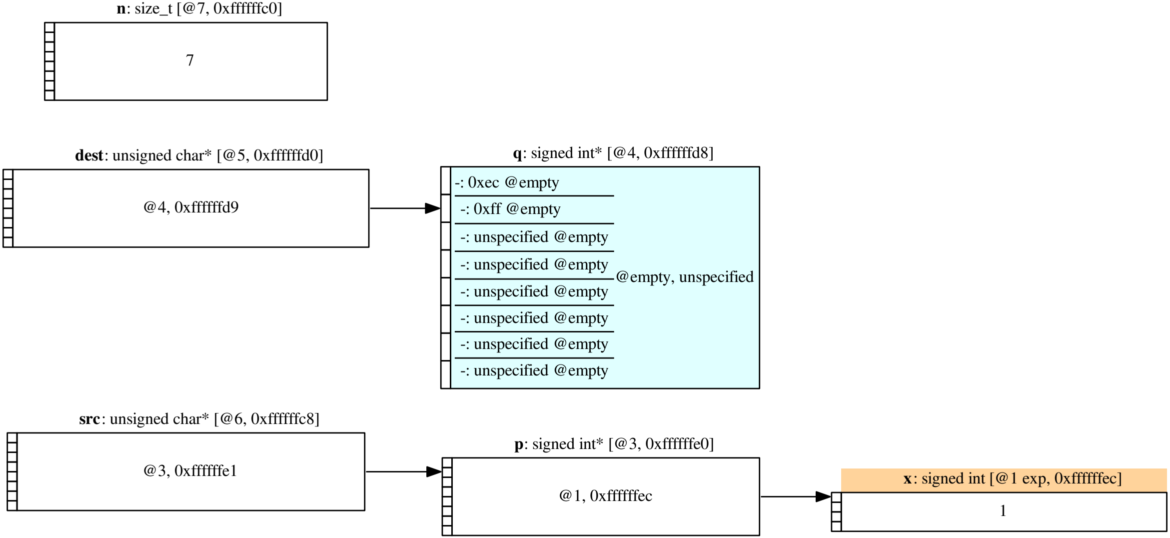
Copying pointer values bytewise, with user-memcpy



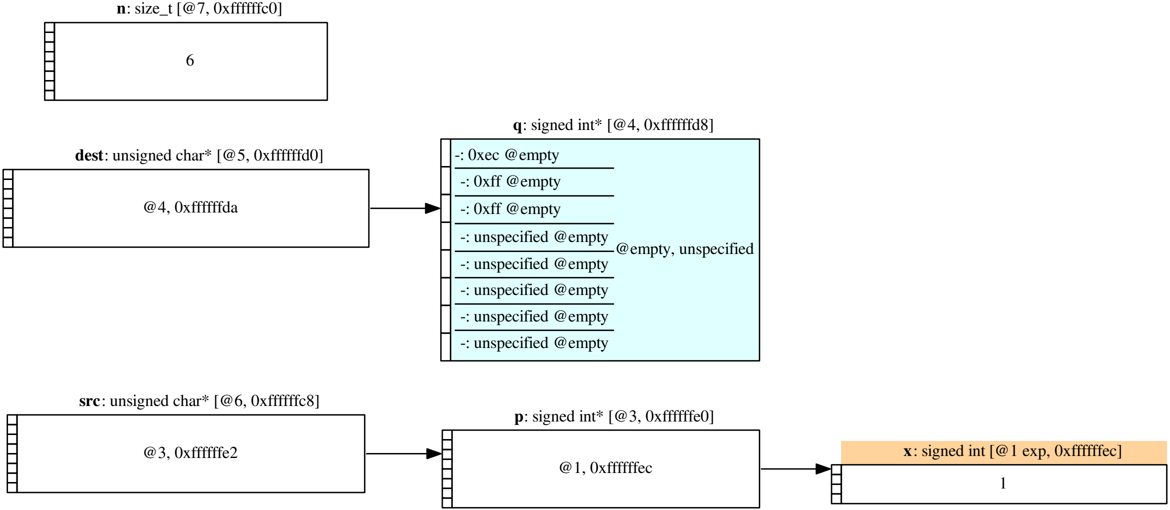
Copying pointer values bytewise, with user-memcpy



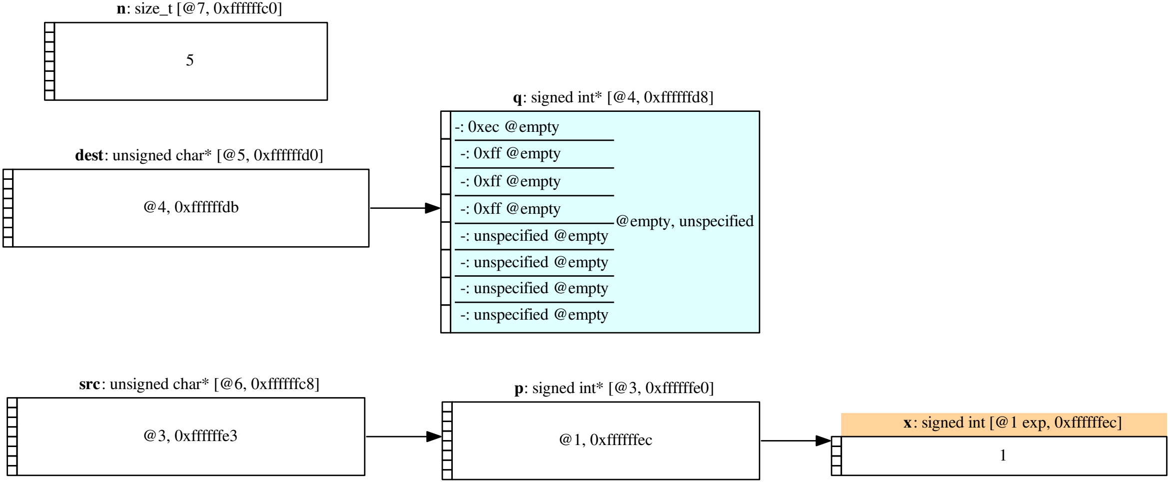
Copying pointer values bytewise, with user-memcpy



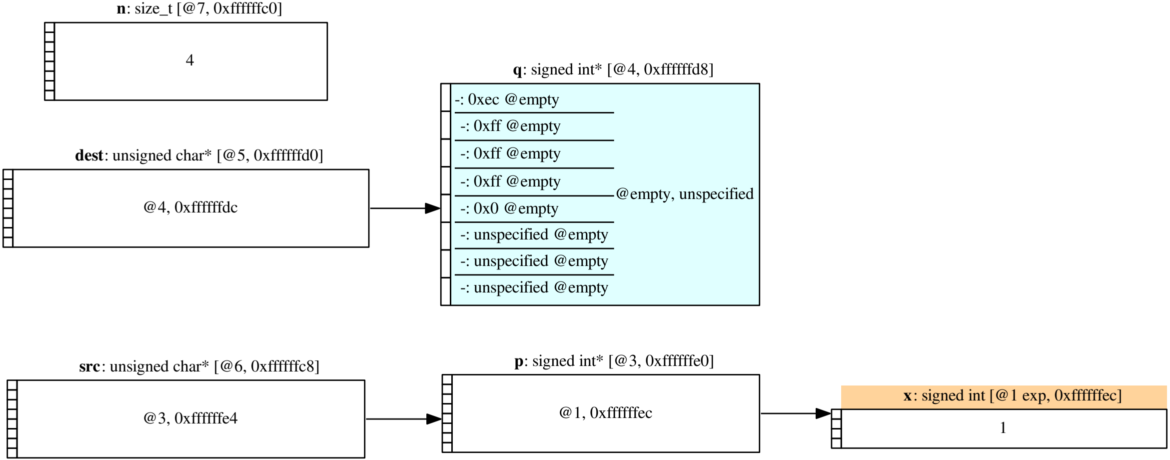
Copying pointer values bytewise, with user-memcpy



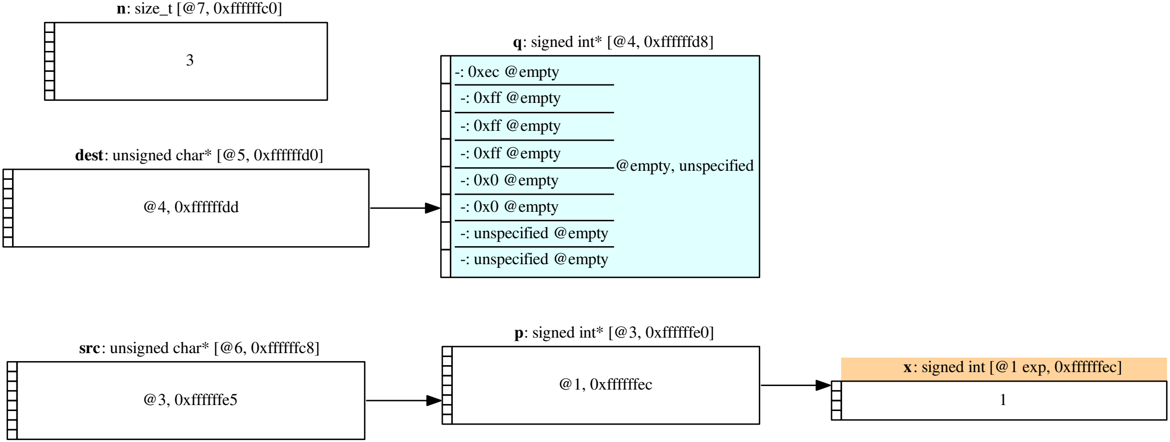
Copying pointer values bytewise, with user-memcpy



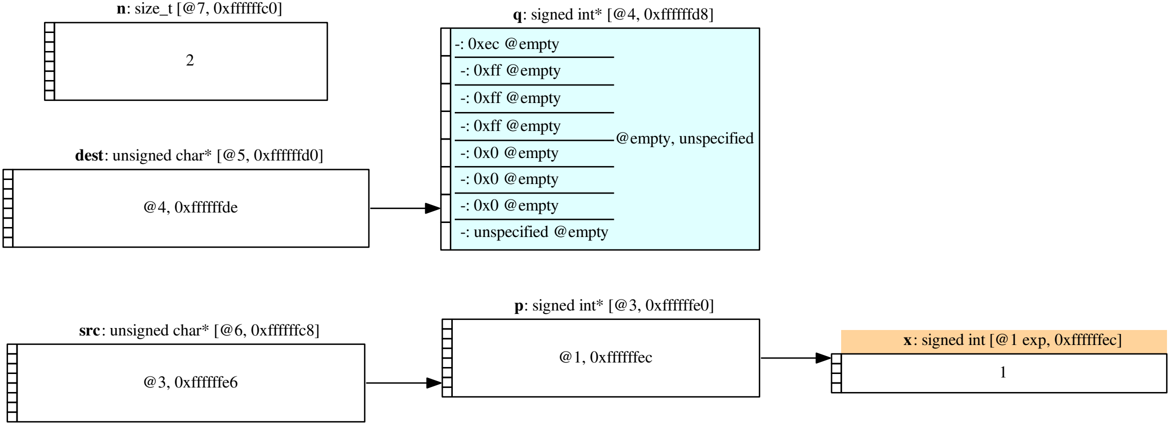
Copying pointer values bytewise, with user-memcpy



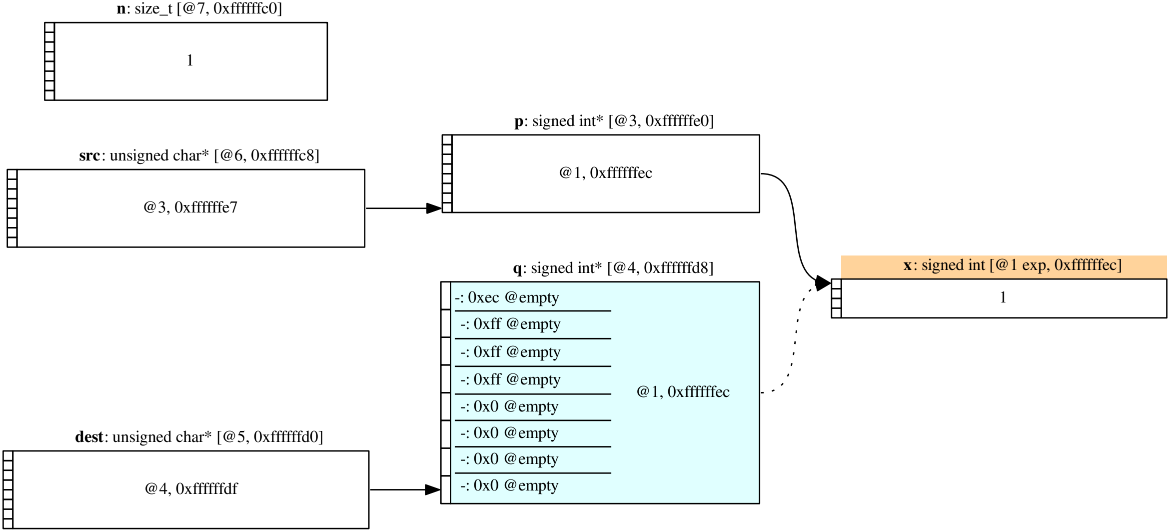
Copying pointer values bytewise, with user-memcpy



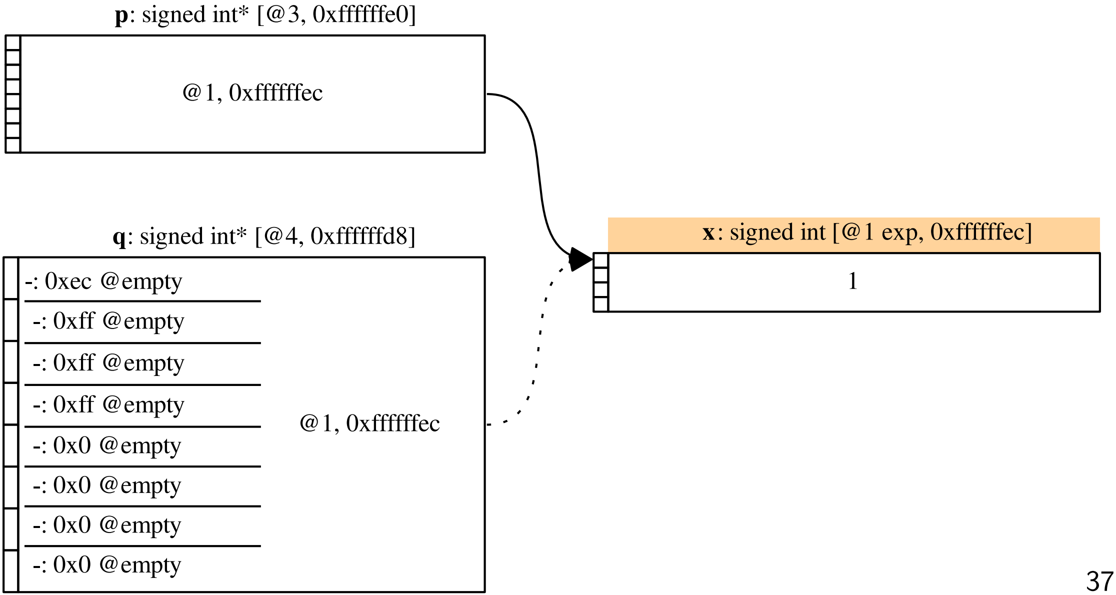
Copying pointer values bytewise, with user-memcpy



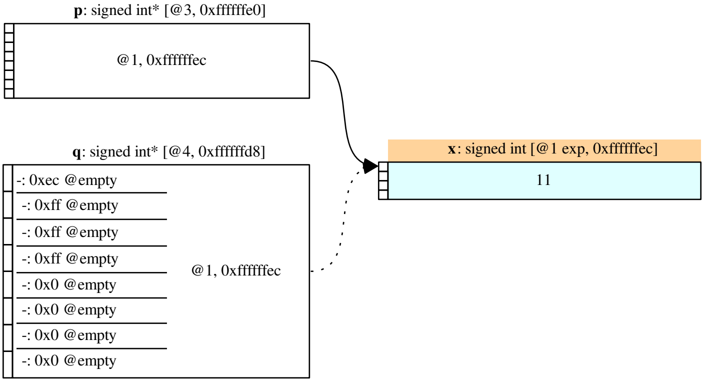
Copying pointer values bytewise, with user-memcpy



Copying pointer values bytewise, with user-memcpy



Copying pointer values bytewise, with user-memcpy



The first read of a `p` pointer byte marked `x` as exposed, then the final `*q=11` access follows the integer-to-pointer cast semantics when reading a pointer value from the memory bytes, recovering the provenance `@1` that the concrete address is within.

Pointer provenance and union type punning

Pointer values can also be constructed by type punning, e.g. writing an **`int*`** union member, reading it as a `uintptr_t` union member, and then casting back to a pointer type.

(The example assumes the object representations of the pointer and the result of the cast to integer are identical. This is not guaranteed by the standard, but holds for many implementations.)

```c
// provenance_union_punning_3_global.c
#include <stdio.h>
#include <string.h>
#include <inttypes.h>
int x=1;
typedef union { uintptr_t ui; int *up; } un;
int main() {
  un u;
  int *p = &x;
  u.up = p;
  uintptr_t i = u.ui;
  int *q = (int*)i;
  *q = 11;  // does this have UB?
  printf("x=%d *p=%d *q=%d\n",x,*p,*q);
  return 0;
}
```

The same semantics as for representation-byte reads also permits this: `x` is deemed exposed by the read of the provenanced representation bytes by the non-pointer-type read. The integer-to-pointer cast then recreates the provenance of `x`.

Pointer provenance via IO

Three versions:

- using `fprintf`/`fscanf` and the `%p` format (which the standard says should work),
- using `fwrite`/`fread` on the pointer representation bytes, and
- converting the pointer to and from `uintptr_t` and using `fprintf`/`fscanf`.

The first gives a syntactic indication of a potentially escaping pointer value; the others do not.

Exotic, but used in practice.

In our proposal, these just work: we mark the storage instance as exposed on the `%p printf`, pointer representation-byte read, or cast, and use the same semantics as integer-to-pointer casts at input-, read-, or cast-time to recover the original provenance.

Implications for optimisation

Can a function argument alias its local variables? (1/3)

This should be forbidden:

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

`main()` guesses the address of `f()`'s local variable `j`, passing it in as a pointer, and `f()` checks it before using it for an access. Here GCC -O0 optimises away the **`if`** and the write `*p=7`, even when `ADDRESS_PFI_1PG` is the same as `&j`. That compiler behaviour should be permitted, so this program should be deemed UB. In other words, code should not normally be allowed to rely on implementation facts about the allocation addresses of C variables.

Can a function argument alias its local variables? (1/3)

This should be forbidden:

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

Our PNVI-* proposals correctly deems this to be UB: at the point of the `(int*)i` cast the `j` storage instance does not yet even exist, so that cast gives a pointer with empty provenance; any execution that goes into the `if` would thus flag UB, so the program as a whole is UB.

Can a function argument alias its local variables? (2/3)

Varying to do the `(int*)` cast after the `j` allocation:

```c
// pointer_from_integer_1ig.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(uintptr_t i) {
  int j=5;
  int *p = (int*)i;
  if (p==&j)
    *p=7;
  printf("j=%d &j=%p\n",j,(void*)&j);
}
int main() {
  uintptr_t j = ADDRESS_PFI_1IG;
  f(j);
}
```

This is still forbidden in PNVI-ae-*, as `j` is not exposed. It would be allowed in PNVI-plain, but perhaps that would also be acceptable – it would just require compilers to be conservative about the results of integer-to-pointer casts where they cannot see the source of the integer, which we imagine is a rare case.

Can a function argument alias its local variables? (3/3)

Varying again to remove the conditional guard and make `j` exposed:

```
// pointer_from_integer_1ie.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(uintptr_t i) {
  int j=5;
  uintptr_t k = (uintptr_t)&j;
  int *p = (int*)i;
  *p=7;
  printf("j=%d\n",j);
}
int main() {
  uintptr_t j = ADDRESS_PFI_1I;
  f(j);
}
```

Executions in which `&j == ADDRESS_PFI_1I` would be ok, but, because the standard does not and should not constrain allocation addresses (beyond alignment and non-overlapping properties), there are always (unless the address space is almost exhausted) other executions in which `ADDRESS_PFI_1I` does not match any allocation. So this is still (correctly) deemed UB.

Can a function argument alias its local variables? (3/3)

Varying again to remove the conditional guard and make j exposed:

```
// pointer_from_integer_1ie.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(uintptr_t i) {
  int j=5;
  uintptr_t k = (uintptr_t)&j;
  int *p = (int*)i;
  *p=7;
  printf("j=%d\n",j);
}
int main() {
  uintptr_t j = ADDRESS_PFI_1I;
  f(j);
}
```

In other words: the fact that programmers cannot assume anything about allocation addresses licenses the desired compiler optimisation. That's expressed in the abstract machine simply by making allocation addresses nondeterministic.

Can a function access local variables of its parent? (1/2)

This too should be forbidden in general.

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

Here `f()` guesses the address of `main()`'s local variable `j`.

This is similarly UB by allocation-address nondeterminism: the abstract machine allows executions in which the guess is correct, but also executions in which it is incorrect, where the `*p=7` flags UB. So the program is UB.

Can a function access local variables of its parent? (1/2)

This too should be forbidden in general.

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

Here `f()` guesses the address of `main()`'s local variable `j`.

(In PNVI-ae-*, `j` is not exposed, so all executions flag UB, but the previous argument applies even if `j` is exposed.)

Can a function access local variables of its parent? (2/2)

Varying to guard the call to `f()` with an address check:

```
// pointer_from_integer_2g.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f() {
  uintptr_t i=ADDRESS_PFI_2G;
  int *p = (int*)i;
  *p=7;
}
int main() {
  int j=5;
  if ((uintptr_t)&j == ADDRESS_PFI_2G)
    f();
  printf("j=%d &j=%p\n",j,(void*)&j);
}
```

This is allowed in PNVI-*, but the guard necessarily involves `&j`, so compilers should be able to deem this escaped. In other words, while we don't think this example needs to be allowed, it should be ok to make it allowed.

Optimisations based on equality tests

In any provenance-aware semantics, `p==q` can hold in some cases where `p` and `q` are not interchangeable (e.g. `*p` is defined but `*q` UB).

(Otherwise, we'd have to require implementations track provenance at runtime for == testing; not usually practical.)

As Lee et al. observe [OOPSLA 2018], that restricts optimisations, e.g. GVN, based on pointer equality tests.

Solution: just don't do those.

(There's no alternative, short of compilers giving up on provenance-based alias analysis altogether, which would be worse.)

PNVI-plain vs PNVI-ae-*

Is the PNVI-ae-* "exposed" machinery necessary?

Debatable. There's not much difference between PNVI-plain and PNVI-ae for these examples (`pointer_from_integer_1ig.c` is allowed in PNVI-plain but forbidden in PNVI-ae-*).

PNVI-plain is simpler, but relies on allocation-address nondeterminism (which some people aren't happy with) for more of the examples than PNVI-ae-*.

PNVI-ae-* is more complex, but makes some of these examples UB just by examining a single execution path. It's also subject to...

The problem with lost address-takens and escapes

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

In PNVI-plain, this is allowed, simply because x exists at the integer-to-pointer cast. Implementations that are conservative w.r.t. all pointers formed from integers would automatically be sound w.r.t. that.

The problem with lost address-takens and escapes

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

In PNVI-ae-*, in the source program `x` is exposed before the integer-to-pointer cast, so this is allowed here too.

But a compiler might optimise (in its intermediate language)...

The problem with lost address-takens and escapes

```c
// provenance_lost_escape_1_optimised.c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include "charon_address_guesses.h"
int x=1; // assume allocation ID @1, at ADDR_PLE_1
int main() {
  int *p = &x;
  uintptr_t i4 =      ADDR_PLE_1;
  int *q = (int *)i4;
  printf("Addresses: p=%p\n",(void*)p);
  uintptr_t i1 = (intptr_t)p;
  if (memcmp(&i1, &i4, sizeof(i1)) == 0) {
    *q = 11;  // does this have defined behaviour?
    printf("x=%d *p=%d *q=%d\n",x,*p,*q);
  }
}
```

and now x is no longer exposed before the cast. If this happens before alias analysis, the results would be wrong.

The problem with lost address-takens and escapes

Solutions: either

- simply be conservative (in alias analysis) w.r.t. all pointers formed from integers, or
- record, in optimisations that occur before alias analysis, any lost exposures, and pass those in as an additional argument to alias analysis.

PNVI-ae vs PNVI-ae-udi

Should we allow one-past integer-to-pointer casts?

We have to decide whether casting a one-past pointer to integer and back gives a usable result.

```
// provenance_roundtrip_via_intptr_t_onepast.c
#include <stdio.h>
#include <inttypes.h>
int x=1;
int main() {
  int *p = &x;
  p=p+1;
  intptr_t i = (intptr_t)p;
  int *q = (int *)i;
  q=q-1;
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

Pro: it's nice for one-past pointers to behave like in-bounds pointers

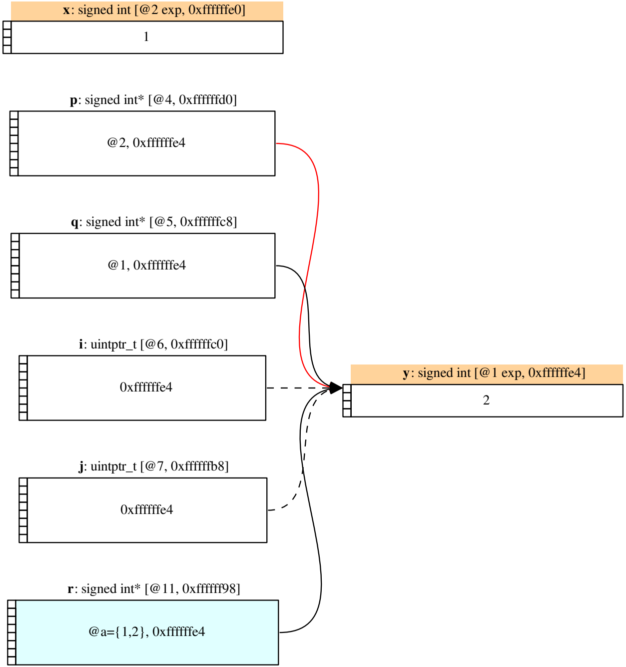
Con: if that's allowed, we have to deal with ambiguous integers, which can be regarded either one-past one object or the start of another.

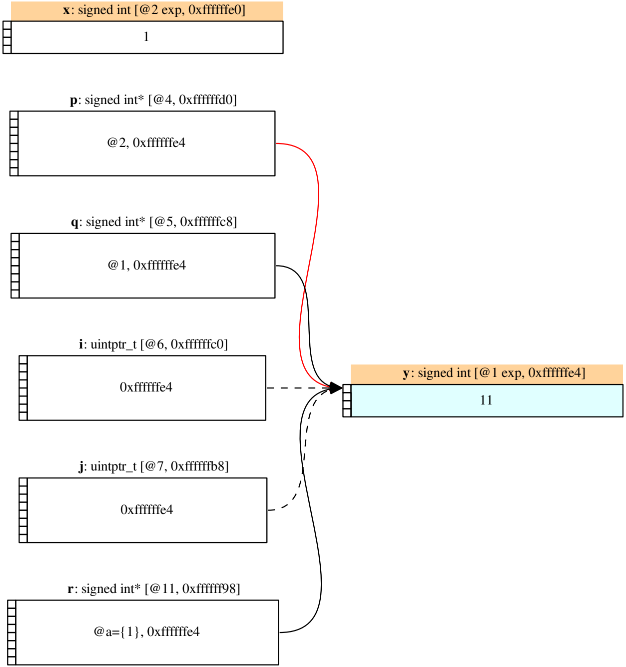
Should we allow one-past integer-to-pointer casts?

We have to decide whether casting a one-past pointer to integer and back gives a usable result.

PNVI-plain and PNVI-ae forbid this: an integer has to be properly within an object for it to be castable to a usable pointer.

PNVI-ae-udi (user disambiguation) permits it: it leaves the provenance of pointer values resulting from such casts unknown until the first operation (e.g. an access, pointer arithmetic, or pointer relational comparison) that disambiguates them. This makes examples that use the result of the cast in one consistent way well defined.


x: signed int [@2 exp, 0xffffffe0]
1
p: signed int* [@4, 0xffffffd0]
@2, 0xffffffe4
q: signed int* [@5, 0xffffffc8]
@1, 0xffffffe4
i: uintptr_t [@6, 0xffffffc0]
0xffffffe4
j: uintptr_t [@7, 0xffffffb8]
0xffffffe4
r: signed int* [@11, 0xffffff98]
@a={1,2}, 0xffffffe4
y: signed int [@1 exp, 0xffffffe4]
2


x: signed int [@2 exp, 0xffffffe0]
1
p: signed int* [@4, 0xffffffd0]
@2, 0xffffffe4
q: signed int* [@5, 0xffffffc8]
@1, 0xffffffe4
i: uintptr_t [@6, 0xffffffc0]
0xffffffe4
y: signed int [@1 exp, 0xffffffe4]
11
j: uintptr_t [@7, 0xffffffb8]
0xffffffe4
r: signed int* [@11, 0xffffff98]
@a={1}, 0xffffffe4

Experimental checks

Testing the example behaviour in Cerberus

We confirmed the examples behave as desired in each model by running them in Cerberus.

test family	test	intended behaviour PNVI-plain	intended behaviour PNVI-ae	intended behaviour PNVI-ae-udi	observed behaviour Cerberus (decreasing allocator) PNVI-plain	observed behaviour Cerberus (decreasing allocator) PNVI-ae	observed behaviour Cerberus (decreasing allocator) PNVI-ae-udi
1	provenance_basic_global_xy.c	UB			not triggered		
	provenance_basic_global_yx.c				UB (line 9)		
	provenance_basic_auto_xy.c				not triggered		
	provenance_basic_auto_yx.c				UB (line 9)		
2	**cheri_03_ii.c**	UB			UB (except with *permissive_pointer_arith* switch)		
3	**pointer_offset_from_ptr_subtraction_global_xy.c**	UB (pointer subtraction)			UB (pointer subtraction) Or UB (out-of-bound store with *permissive_pointer_arith* switch)		
	pointer_offset_from_ptr_subtraction_global_yx.c						
	pointer_offset_from_ptr_subtraction_auto_xy.c						
	pointer_offset_from_ptr_subtraction_auto_yx.c						
4	**provenance_equality_global_xy.c**	defined, nondet			not triggered		
	provenance_equality_global_yx.c				defined (ND except with *strict pointer equality* switch)		
	provenance_equality_auto_xy.c				not triggered		
	provenance_equality_auto_yx.c				defined (ND except with *strict pointer equality* switch)		
	provenance_equality_global_fn_xy.c				not triggered		
	provenance_equality_global_fn_yx.c				defined (ND except with *strict pointer equality* switch)		
5	**provenance_roundtrip_via_intptr_t.c**	defined			defined		
6	**provenance_basic_using_uintptr_t_global_xy.c**	defined			not triggered		
	provenance_basic_using_uintptr_t_global_yx.c				defined		
	provenance_basic_using_uintptr_t_auto_xy.c				not triggered		
	provenance_basic_using_uintptr_t_auto_yx.c				defined		
7	**pointer_offset_from_int_subtraction_global_xy.c**	defined			defined		
	pointer_offset_from_int_subtraction_global_yx.c				defined		
	pointer_offset_from_int_subtraction_auto_xy.c				defined		
	pointer_offset_from_int_subtraction_auto_yx.c				defined		
8	**pointer_offset_xor_global.c**	defined			defined		
	pointer_offset_xor_auto.c				defined		
9	provenance_tag_bits_via_uintptr_t_1.c	defined			defined		
10	**pointer_arith_algebraic_properties_2_global.c**	defined			defined		
11	**pointer_arith_algebraic_properties_3_global.c**	defined			defined		
12	**pointer_copy_memcpy.c**	defined			defined		
13	**pointer_copy_user_dataflow_direct_bytewise.c**	defined			defined		
13	**provenance_tag_bits_via_repr_byte_1.c**	defined			defined		
15	pointer_copy_user_ctrlflow_bytewise.c	defined			defined		
16	**pointer_copy_user_ctrlflow_bitwise.c**	defined			defined		
17	provenance_equality_uintptr_t_global_xy.c	defined			not triggered		
	provenance_equality_uintptr_t_global_yx.c				defined (true)		
	provenance_equality_uintptr_t_auto_xy.c				not triggered		
	provenance_equality_uintptr_t_auto_yx.c				defined (true)		
18	provenance_union_punning_2_global_xy.c	defined	UB (line 16, deref)	UB (line 16, store)	not triggered		
	provenance_union_punning_2_global_yx.c	defined	UB (line 16, deref)	UB (line 16, store)	defined	UB (line 16, deref)	UB (line 16, store)
	provenance_union_punning_2_auto_xy.c	defined	UB (line 16, deref)	UB (line 16, store)	not triggered		
	provenance_union_punning_2_auto_yx.c	defined	UB (line 16, deref)	UB (line 16, store)	defined	UB (line 16, deref)	UB (line 16, store)
19	**provenance_union_punning_3_global.c**	defined			defined		
20	provenance_via_io_percentp_global.c	**filesystem and scanf() are not currently supported by Cerberus**					
	provenance_via_io_bytewise_global.c						
	provenance_via_io_uintptr_t_global.c						
21	**pointer_from_integer_1pg.c**	UB (line 7)			UB in one exec (line 7)		
	pointer_from_integer_1ig.c	defined (j = 7)	UB (line 8)		defined (j = 7)	UB (line 8)	
	pointer_from_integer_1p.c	UB (line 6)			UB (line 6)		
	pointer_from_integer_1i.c	defined (j = 7)	UB (line 7)		defined (j = 7)	UB (line 7)	
	pointer_from_integer_1ie.c	defined (j = 7)			defined (j = 7)		
	pointer_from_integer_2.c	defined (j = 7)	UB (line 7)		defined (j = 7)	UB (line 7)	
	pointer_from_integer_2g.c	defined (j = 7)			defined (j = 7)		
	provenance_lost_escape_1.c	defined			defined		
22	**provenance_roundtrip_via_intptr_t_onepast.c**	UB (line 10)		defined	UB (line 10)		defined
23	**pointer_from_int_disambiguation_1.c**	defined (y = 11)			defined (y = 11)		
	pointer_from_int_disambiguation_1_xy.c				not triggered		
	pointer_from_int_disambiguation_2.c	UB (line 14)		defined	UB (line 14)		defined (x = 11)
	pointer_from_int_disambiguation_2_xy.c				not triggered		
	pointer_from_int_disambiguation_3.c	UB (line 15)		UB (line 15)	UB (line 15)		
	pointer_from_int_disambiguation_3_xy.c				not triggered		

(**bold** = tests mentioned in the document)

green = Cerberus behaviour matches intent

blue = Cerberus behaviour matches intent (witch *permissive_pointer_arith* switch)

grey = Cerberus' allocator doesn't trigger the interesting behaviour

Testing the example behaviour in mainstream C implementations

Our examples are semantic test-cases, not compiler tests, and some compilers have known bugs in this area. But, ignoring that, we show whether the observed behaviour of GCC, Clang, and ICC, at various optimisation levels, is consistent with each model for these tests.

Compilers

		Observed behaviour (compilers), sound w.r.t PNVI-*? (relying on UB or ND?)								
			gcc-8.3			clang-7.0.1			icc-19	
test family	test	PNVI-plain	PNVI-ae	PNVI-ae-udi	PNVI-plain	PNVI-ae	PNVI-ae-udi	PNVI-plain	PNVI-ae	PNVI-ae-udi
1	provenance_basic_global_xy.c		y (n)			y (n)			y (y for O2+)	
	provenance_basic_global_yx.c		y (y for O2+)			not triggered			not triggered	
	provenance_basic_auto_xy.c		y (n)			y (n)			y (y for O2+)	
	provenance_basic_auto_yx.c		y (n)			y (n)			y (y for O2+)	
2	**cheri_03_ii.c**		y (n)			y (n)			y (n)	
3	**pointer_offset_from_ptr_subtraction_global_xy.c**		y (n)			y (n)			y (n)	
	pointer_offset_from_ptr_subtraction_global_yx.c								y (n)	
	pointer_offset_from_ptr_subtraction_auto_xy.c								y (y for O2+)	
	pointer_offset_from_ptr_subtraction_auto_yx.c								y (y for O2+)	
4	**provenance_equality_global_xy.c**		y (n)			y (n)			y (n)	
	provenance_equality_global_yx.c		y (y for O2+)							
	provenance_equality_auto_xy.c		y (y for O2+)							
	provenance_equality_auto_yx.c		y (n)							
	provenance_equality_global_fn_xy.c		y (n)							
	provenance_equality_global_fn_yx.c		y (y for O2+)							
5	**provenance_roundtrip_via_intptr_t.c**		y (n)			y (n)			y (n)	
6	**provenance_basic_using_uintptr_t_global_xy.c**		y (n)			y (n)			n (y)	
	provenance_basic_using_uintptr_t_global_yx.c		n (y)			not triggered			not triggered	
	provenance_basic_using_uintptr_t_auto_xy.c		y (n)			not triggered			n (y)	
	provenance_basic_using_uintptr_t_auto_yx.c		y (n)			y (n)			n (y)	
7	**pointer_offset_from_int_subtraction_global_xy.c**		y (n)			y (n)			y (n)	
	pointer_offset_from_int_subtraction_global_yx.c									
	pointer_offset_from_int_subtraction_auto_xy.c									
	pointer_offset_from_int_subtraction_auto_yx.c									
8	**pointer_offset_xor_global.c**		y (n)			y (n)			y (n)	
	pointer_offset_xor_auto.c									
9	provenance_tag_bits_via_uintptr_t_1.c		y (n)			y (n)			y (n)	
10	**pointer_arith_algebraic_properties_2_global.c**		y (n)			y (n)			y (n)	
11	**pointer_arith_algebraic_properties_3_global.c**		y (n)			y (n)			y (n)	
12	**pointer_copy_memcpy.c**		y (n)			y (n)			y (n)	
13	**pointer_copy_user_dataflow_direct_bytewise.c**		y (n)			y (n)			y (n)	
13	**provenance_tag_bits_via_repr_byte_1.c**		y (n)			y (n)			y (n)	
15	pointer_copy_user_ctrlflow_bytewise.c		y (n)			y (n)			y (n)	
16	**pointer_copy_user_ctrlflow_bitwise.c**		y (n)			y (n)			y (n)	
17	provenance_equality_uintptr_t_global_xy.c		y (n)			y (n)			y (n)	
	provenance_equality_uintptr_t_global_yx.c									
	provenance_equality_uintptr_t_auto_xy.c									
	provenance_equality_uintptr_t_auto_yx.c									
18	provenance_union_punning_2_global_xy.c		y (n)			y (n)		y (y for O2+)	n (y)	
	provenance_union_punning_2_global_yx.c	y (y for O2+)	n (y)			not triggered			not triggered	
	provenance_union_punning_2_auto_xy.c		y (n)			y (n)		y (y for O2+)	n (y)	
	provenance_union_punning_2_auto_yx.c		y (n)					y (y for O2+)	n (y)	
19	**provenance_union_punning_3_global.c**		y (n)			y (n)			y (n)	
20	provenance_via_io_percentp_global.c		NO OPT			NO OPT			NO OPT	
	provenance_via_io_bytewise_global.c									
	provenance_via_io_uintptr_t_global.c									
21	**pointer_from_integer_1pg.c**		y (y for O0+)			y (y for O2+)			y (y for O2+)	
	pointer_from_integer_1ig.c	n (y)	y (y for O2+)		n (y)	y (y for O2+)			n (y for O2+)	
	pointer_from_integer_1p.c					can't test with charon				
	pointer_from_integer_1i.c									
	pointer_from_integer_1ie.c									
	pointer_from_integer_2.c									
	pointer_from_integer_2g.c		y (n)			n (y)			y (n)	
	provenance_lost_escape_1.c		y (n)			y (n)			n (y for O2+)	
22	**provenance_roundtrip_via_intptr_t_onepast.c**		y (n)			y (n)			y (n)	
23	**pointer_from_int_disambiguation_1.c**		n (y)			not triggered			not triggered	
	pointer_from_int_disambiguation_1_xy.c		not triggered			y (n)			n (y for O2+)	
	pointer_from_int_disambiguation_2.c		y (n)			not triggered			not triggered	
	pointer_from_int_disambiguation_2_xy.c		not triggered			y (n)			y (n)	
	pointer_from_int_disambiguation_3.c		y (n)			not triggered			not triggered	
	pointer_from_int_disambiguation_3_xy.c		not triggered			y (n)			y (y for O2+)	

(**bold** = tests mentioned in the document)

Page 2

Details: https://www.cl.cam.ac.uk/~pes20/cerberus/supplementary-material-pnvi-star/generated_html_pnvi_star/

Testing the example behaviour in mainstream C implementations

It doesn't seem possible to make a coherent and useful semantics that admits all the existing observed compiler behaviour – but they do agree in many cases, and it may be that only mild adaptions would be needed.

Pointer equality

Consider pointers `p` and `q` with different provenance. In an execution where they have the same address (same pointer object representation), is `p==q`:

1. required to be true, or
2. allowed to be either true or false, or
3. undefined behaviour?

C18 6.5.9p6 says (1) *"Two pointers compare equal **if and only if** both are [...] or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space"*,

GCC follows (2).

We suspect (3) would break existing code.

Pick one...