Fast, precise dynamic checking of types and bounds "in C"

Stephen Kell

stephen.kell@cl.cam.ac.uk

Computer Laboratory

University of Cambridge

Some years ago: a tool is born

```
if (obj->type == OBJ_COMMIT) {
  if (process_commit(walker, (struct commit *)obj))
    return -1;
  return 0;
}
```

Some years ago: a tool is born

```
if  (obj->type == OBJ_COMMIT) {
  if  (process_commit(walker, (struct commit *)obj))
    return -1;
  return 0;
}
```

↖ ↗

CHECK this

(at run time)

Some years ago: a tool is born

```
if (obj->type == OBJ_COMMIT) {
  if (process_commit(walker, (struct commit *)obj))
    return -1;
  return 0;
}
```

↖ ↗

CHECK this

(at run time)

Also wanted:

- binary- and source-compatible
- for real, idiomatic C code
- reasonable performance

Some years ago: a tool is born

```
if (obj->type == OBJ_COMMIT) {
  if (process_commit(walker, (struct commit *)obj))
    return -1;
  return 0;
}
```

↖ ↗ CHECK this (at run time)

Also wanted:

- binary- and source-compatible
- for real, idiomatic C code
- reasonable performance

Enter libcrunch, which does this and (ever) more…

Some years ago: a tool is born

```
if (obj->type == OBJ_COMMIT) {
  if (process_commit(walker, (struct commit *)obj))
    return -1;
  return 0;
}
```

↖ ↗ CHECK this (at run time)

Also wanted:

- binary- and source-compatible
- for real, idiomatic C code
- reasonable performance

Enter libcrunch, which does this and (ever) more...

- medium-term: a ‘safe’ implementation of C

Some years ago: a tool is born

```
if (obj->type == OBJ_COMMIT) {
  if (process_commit(walker, (struct commit *)obj))
    return -1;
  return 0;
}
```

↖ ↗ CHECK this (at run time)

Also wanted:

- binary- and source-compatible
- for real, idiomatic C code
- reasonable performance

Enter libcrunch, which does this and (ever) more…

- medium-term: a 'safe' implementation of C
- yes, really

Outline of this talk

- a bit about run-time type information
- a bit about using it for dynamic type checking
- a bit about using it for dynamic bounds checking

But first…

Interlude: isn't this boring?

Checking stuff about C code, blah blah

- legacy blah
- security blah
- performance blah

Interlude: isn't this boring?

~~Checking stuff about C code, blah blah~~

- ~~legacy blah~~
- ~~security blah~~
- ~~performance blah~~

PL implementation has got stuck in a local minimum!

- Unix-like host environment – 'low-level'
- 'VMs' packaging specific languages – 'high-level'

Interlude: isn't this boring?

~~Checking stuff about C code, blah blah~~

- ~~legacy blah~~
- ~~security blah~~
- ~~performance blah~~

PL implementation has got stuck in a local minimum!

- Unix-like host environment – 'low-level'
- 'VMs' packaging specific languages – 'high-level'

Results

- language balkanization
- 'cool new things', not consolidated progress
- endless {rewrites, firefighting, 'legacy'}

Don't 'disrupt'; evolve!

> Instead of having VMs suppland Unix processes, is it practical to make Unix processes *become* VMs? This means evolving the basic interfaces offered by a Unix process so that they subsume those of a VM, while remaining backward-compatible. At first glance this appears implausible…

—from my Onward! 2015 paper. Punchline: it's not!

Key idea: evolve Unix to offer 'modern' affordances!

- don't 'leave behind' the 'legacy'!

Gaps to plug:

- VMs have meta-info, Unix… doesn't?
- C/Unix offer an *address space* abstraction; VMs not?

Who says Unix doesn’t have metadata? (1)

```
$ cat /proc/self/maps
    00400000-    0040c000 r-xp 00000000 08:01 89694        /bin/cat
    0060b000-    0060c000 r--p 0000b000 08:01 89694        /bin/cat
    0060c000-    0060d000 rw-p 0000c000 08:01 89694        /bin/cat
    0190c000-    0192d000 rw-p 00000000 00:00 0           [heap]
7f44459a8000-7f44459ca000 rw-p 00000000 00:00 0
7f44459ca000-7f4445b5f000 r-xp 00000000 08:01 81543        /lib/x86
7f4445b5f000-7f4445d5e000 ---p 00195000 08:01 81543        /lib/x86
7f4445d5e000-7f4445d62000 r--p 00194000 08:01 81543        /lib/x86
7f4445d62000-7f4445d64000 rw-p 00198000 08:01 81543        /lib/x86
7f4445d64000-7f4445d68000 rw-p 00000000 00:00 0
7f4445d68000-7f4445d8b000 r-xp 00000000 08:01 81444        /lib/x86
7f4445da1000-7f4445f86000 r--p 00000000 08:05 1524484      /usr/lib
7f4445f86000-7f4445f8b000 rw-p 00000000 00:00 0
7f4445f8b000-7f4445f8c000 r--p 00023000 08:01 81444        /lib/x86
7f4445f8c000-7f4445f8d000 rw-p 00024000 08:01 81444        /lib/x86
```

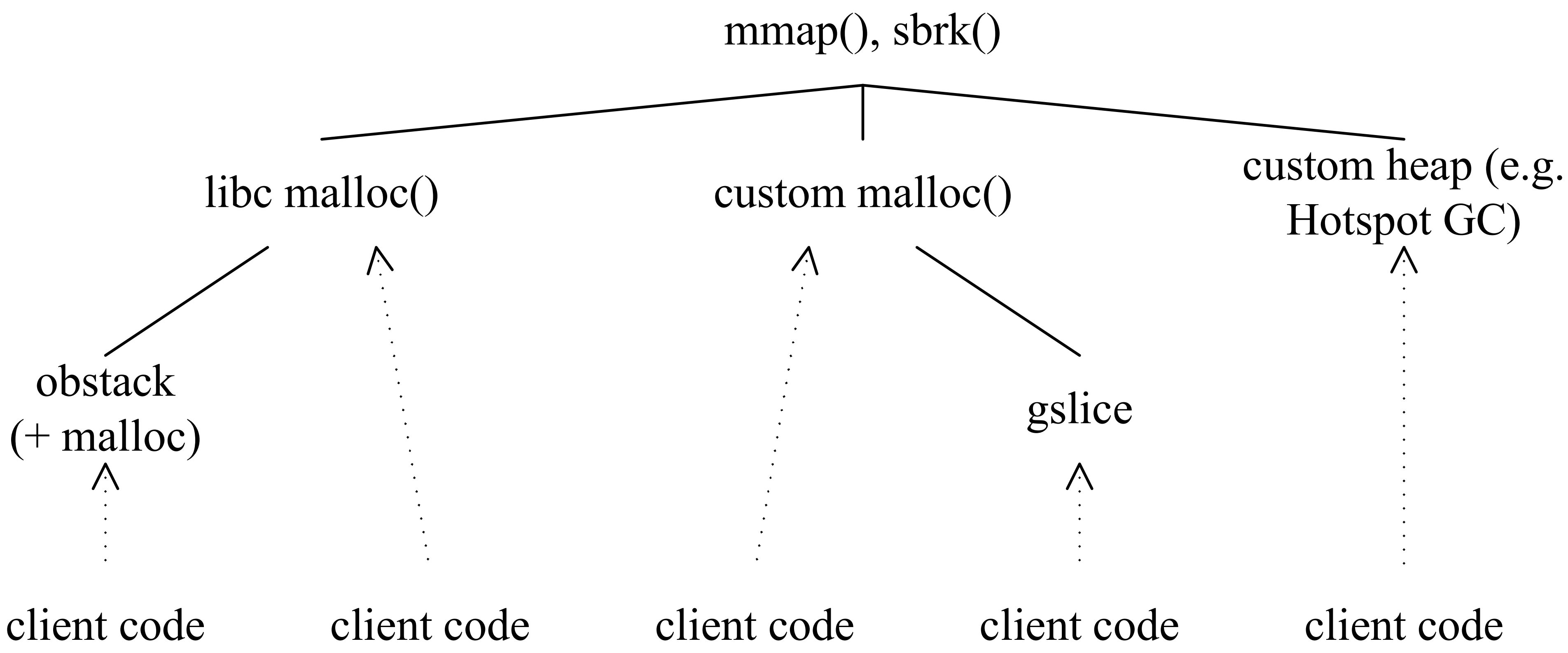
Who says Unix doesn’t have metadata? (2)

```
$ cc -g -o hello hello.c && readelf -wi hello | column
 <b>:TAG_compile_unit                <7ae>:TAG_pointer_type
      AT_language   : 1 (ANSI C)           AT_byte_size: 8
      AT_name       : hello.c              AT_type      : <0x2af>
      AT_low_pc     : 0x4004f4       <76c>:TAG_subprogram
      AT_high_pc    : 0x400514             AT_name      : main
 <c5>: TAG_base_type                       AT_type      : <0xc5>
        AT_byte_size : 4                   AT_low_pc    : 0x4004f4
        AT_encoding  : 5 (signed)          AT_high_pc   : 0x400514
        AT_name      : int            <791>: TAG_formal_parameter
 <2af>:TAG_pointer_type                     AT_name      : argc
        AT_byte_size: 8                     AT_type      : <0xc5>
        AT_type     : <0x2b5>               AT_location : fbreg - 20
 <2b5>:TAG_base_type                  <79f>: TAG_formal_parameter
        AT_byte_size: 1                     AT_name      : argv
        AT_encoding : 6 (char)              AT_type      : <0x7ae>
        AT_name     : char                  AT_location : fbreg - 32
```

Adding a meta-level to Unix (2)

A new meta-level abstraction: *typed allocations*

- allocations: the hierarchical structure of memory



- types: borrow from DWARF debugging information

A meta-level API

```
struct uniqtype;                                      /* type descriptor  */
struct allocator;                           /* heap, stack, static, etc */
allocator * alloc_get_allocator (void *obj); /* which one? (at leaf) */
uniqtype * alloc_get_type       (void *obj);  /* what type?          */
void *     alloc_get_site       (void *obj);  /* where allocated?    */
void *     alloc_get_base       (void *obj);  /* base address?       */
void *     alloc_get_limit      (void *obj);  /* end address?        */
Dl_info    alloc_dladdr         (void *obj);  /* dladdr-like         */
// more calls go here...
```

A meta-level API

```
struct uniqtype;                                          /* type descriptor     */
struct allocator;                                 /* heap, stack, static, etc */
allocator * alloc_get_allocator (void *obj); /* which one? (at leaf) */
uniqtype  * alloc_get_type      (void *obj);      /* what type?          */
void *      alloc_get_site      (void *obj);      /* where allocated?    */
void *      alloc_get_base      (void *obj);      /* base address?       */
void *      alloc_get_limit     (void *obj);      /* end address?        */
Dl_info     alloc_dladdr        (void *obj);      /* dladdr-like         */
// more calls go here...
```

Each allocator implements [most of] these

- static, stack, malloc, custom...

Slowdown, step 1: with typeinfo-maintaining hooks

bench	normal/s	liballocs %
bzip2	4.91	+2.9%
gobmk	14.2	+2.8%
h264ref	10.1	+5.0%
hmmer	2.09	+8.6%
lbm	2.10	+0.9%
mcf	2.36	(−0.4%)
milc	8.54	(−3.0%)
sjeng	3.22	+0.6%
sphinx3	1.54	+7.7%
gcc	0.985	+88 %
perlbench	3.57	+23 %

Back to our dynamic checking problem

```
if (obj->type == OBJ_COMMIT) {
  if (process_commit(walker,

                (struct commit *)obj))
    return -1;
  return 0;
}
```

Back to our dynamic checking problem

```
if (obj->type == OBJ_COMMIT) {
  if (process_commit(walker,
        (CHECK(__is_a(obj, &__uniqtype__commit)),
            (struct commit *)obj)))
    return -1;
  return 0;
}
```

Back to our dynamic checking problem

```
if (obj->type == OBJ_COMMIT) {
  if (process_commit(walker,
        (CHECK(__is_a(obj, &__uniqtype__commit)),
            (struct commit *)obj)))
    return -1;
  return 0;
}
```

Easy to implement __is_a() in terms of alloc_get_type()

Back to our dynamic checking problem

```
if (obj->type == OBJ_COMMIT) {
  if (process_commit(walker,
        (CHECK(__is_a(obj, &__uniqtype__commit)),
          (struct commit *)obj)))
    return -1;
  return 0;
}
```

Easy to implement __is_a() in terms of alloc_get_type()

Not so easy, but doable: accommodating wackier C idioms

- see OOPSLA 2016 paper

Slowdown, step 2: use the type info to check casts

bench	normal/s	crunch %
bzip2	4.95	+6.8%
gobmk	14.6	+11 %
h264ref	10.1	+3.9%
hmmer	2.16	+8.3%
lbm	3.42	+9.6%
mcf	2.48	+12 %
milc	8.78	+38 %
sjeng	3.33	+1.5%
sphinx3	1.60	+13 %
gcc	0.983	+160 %
perlbench		

So what about that ‘safe C implementation’?

- add a bounds checker (improve on SoftBound)
- add a temporal checker
- … and/or a GC (precise! improve on Boehm)
- always initialize pointers
- type-check memcpy(), realloc(), etc..
- bounds-check syscalls (e.g. read())
- type-check against mmap’d file data (why not?)
- check unions and varargs
- check unsafe writes through char*
- do something about address-takeable union members
- do something about threads (anybody?)

So what about that 'safe C implementation'?

- **add a bounds checker (improve on SoftBound)**
- add a temporal checker
- . . . and/or a GC (precise! improve on Boehm)
- always initialize pointers
- type-check memcpy(), realloc(), etc..
- bounds-check syscalls (e.g. read())
- type-check against mmap'd file data (why not?)
- check unions and varargs
- check unsafe writes through char*
- do something about address-takeable union members
- do something about threads (anybody?)

What were we checking, again?

```
typedef struct {int x[2]; char y[4];} blah;
blah z = { {0, 0}, "foo" };

int *x1 =          z.x;  // ok
int *x2 = (int*) &z;     // ok -- check passes
int *x3 = (int*)  z.y;   // type error -- check fails

int i2  = *(z.x + 2);    // need a bounds check

blah *p = malloc(sizeof z);
free(p); *p;             // need a temporal check or GC
return &z;               // ditto
```

A new example

ctr	x	3.5
	y	8.0
maj		2
min		7
ctr	x	1.0
	y	1.5
maj		5
min		8
ctr	x	6.5
	y	-2.0
maj		4
min		4

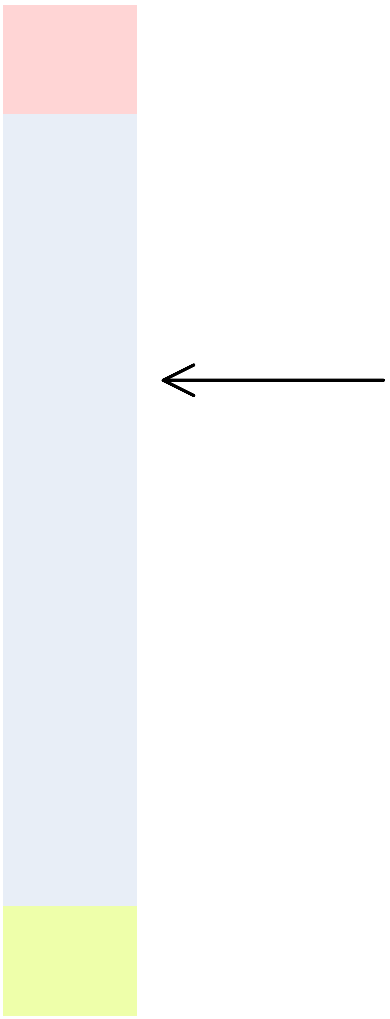
```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double min;
  double maj;
} my_ellipses[3];
```

Existing bounds checkers redux, part 1

‘Object table’ à la Jones & Kelly, mudflap, baggy, lowfat, . . .

- allowing object lookup by address
- on each `p[i]` or `p + i`, check ‘same object’

Object-table checkers are coarse-grained



```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double min;
  double maj;
} my_ellipses[3];

p = &my_ellipses[1];

pd = &my_ellipses[1].ctr.x;
```

Existing bounds checkers redux, part 1

Good about object tables:

- doesn't need ABI changes
- tolerates casts via integers

Bad:

- doesn't catch *subobject overflows*
- doesn't tolerate way-out-of-bounds intermediates

Ugly:

- one-past-the-end pointers; pointers into the stack. . .

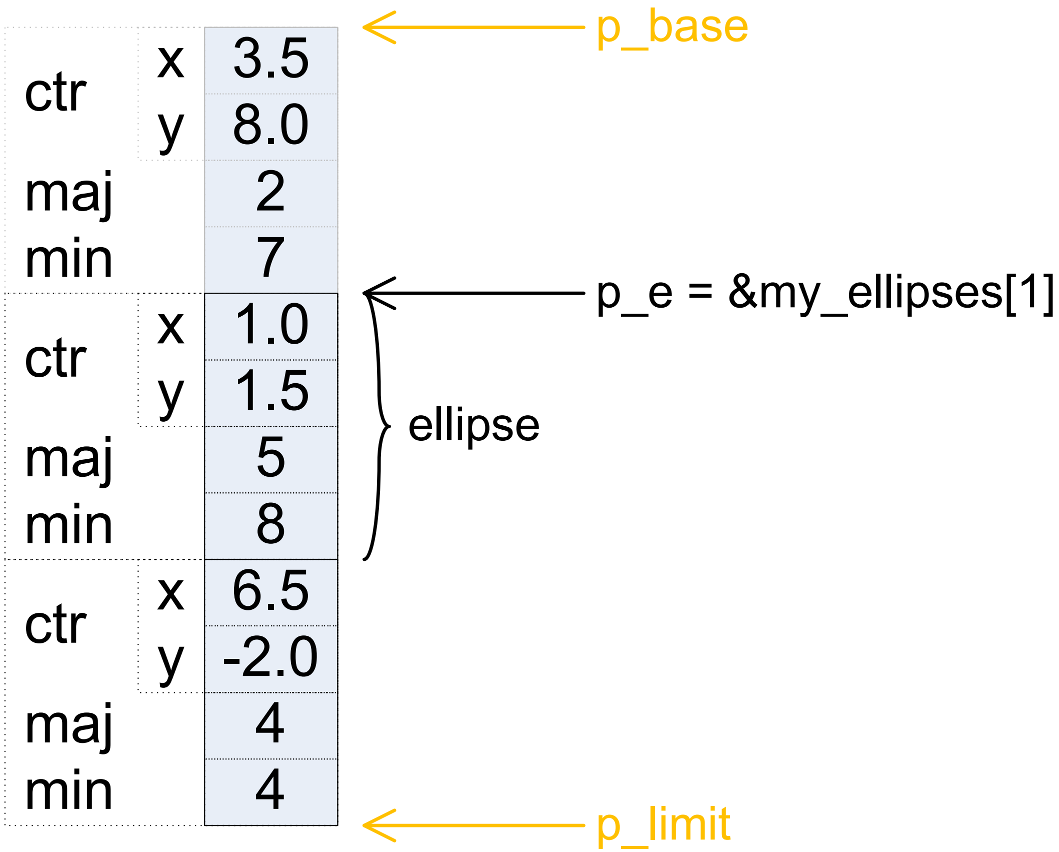
Also: fairly slow!

Existing bounds checkers redux, part 2

'Fat pointers' à la Kendall, Austin et al, SoftBound, …

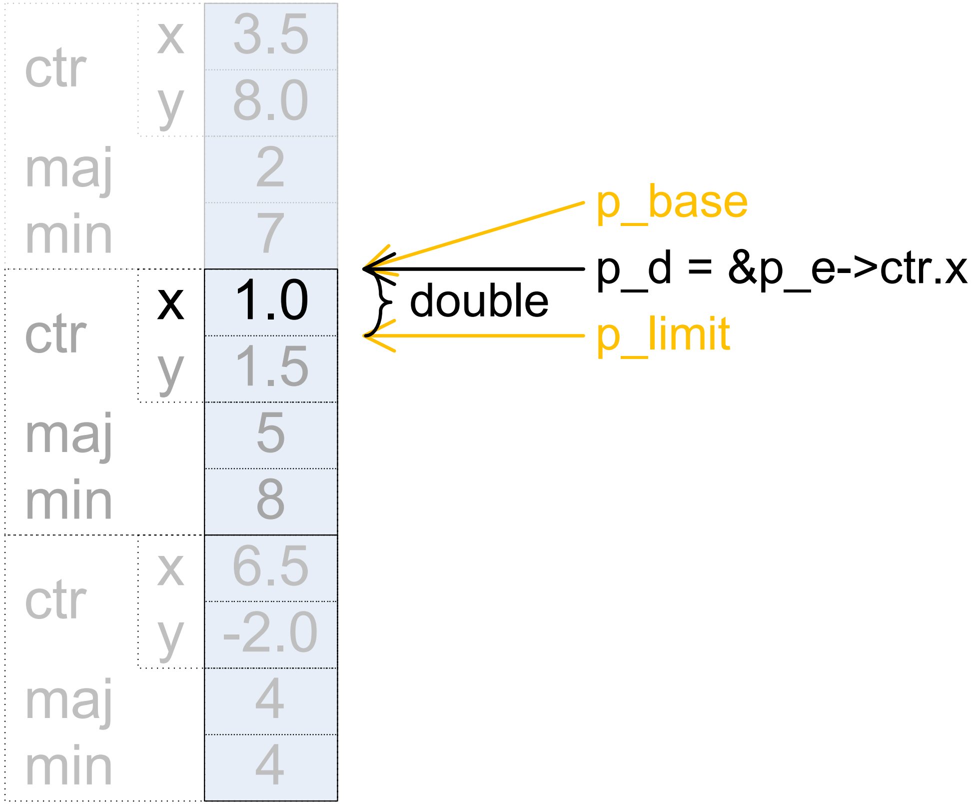
- make pointers bigger: $\{addr, base, limit\}$
- on each `p[i]` or `*p`, check 'within bounds'

Using fat pointers and provenance, can narrow bounds to subobjects




```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Using fat pointers and provenance, can narrow bounds to subobjects




```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Existing bounds checkers redux, part 2

Good about fat pointers:

- tolerates all out-of-bounds pointers
- can catch subobject overflows by *pointer provenance*

Bad:

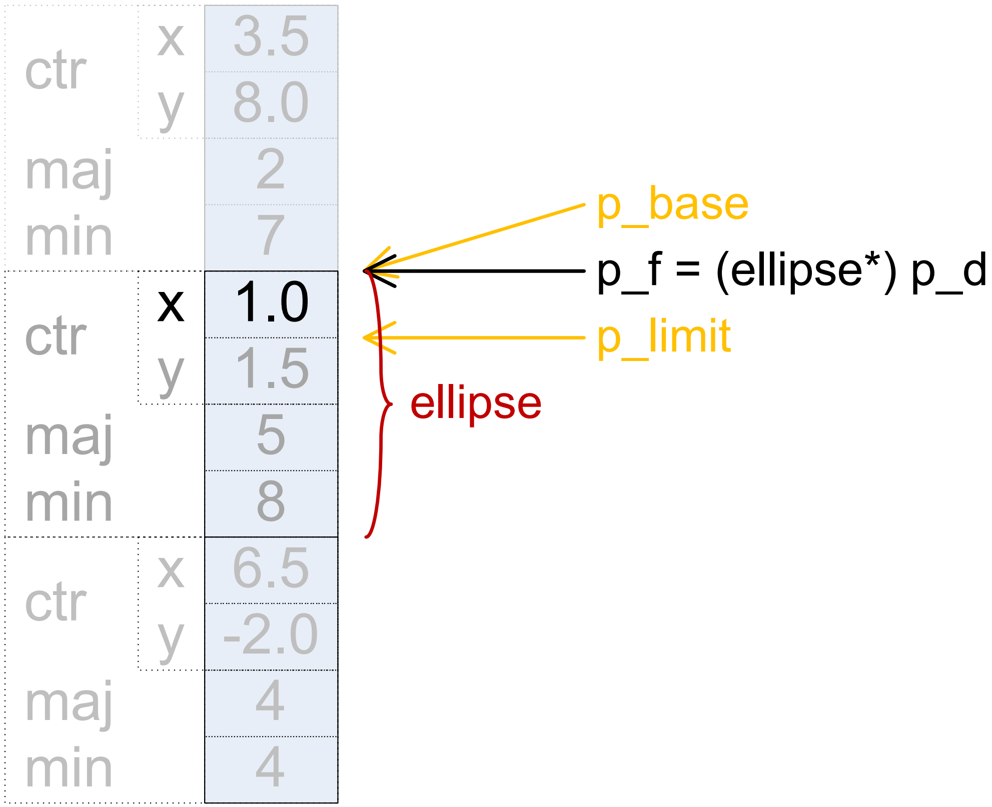
- needs ABI changes or disjoint metadata
- false positives: casts which would *widen* bounds
- false negatives: casts which would *narrow* bounds
- give up: casts from integers, unwrapped libraries, …

Also: fairly slow!

A zoo of difficult-to-check bounds errors and non-errors

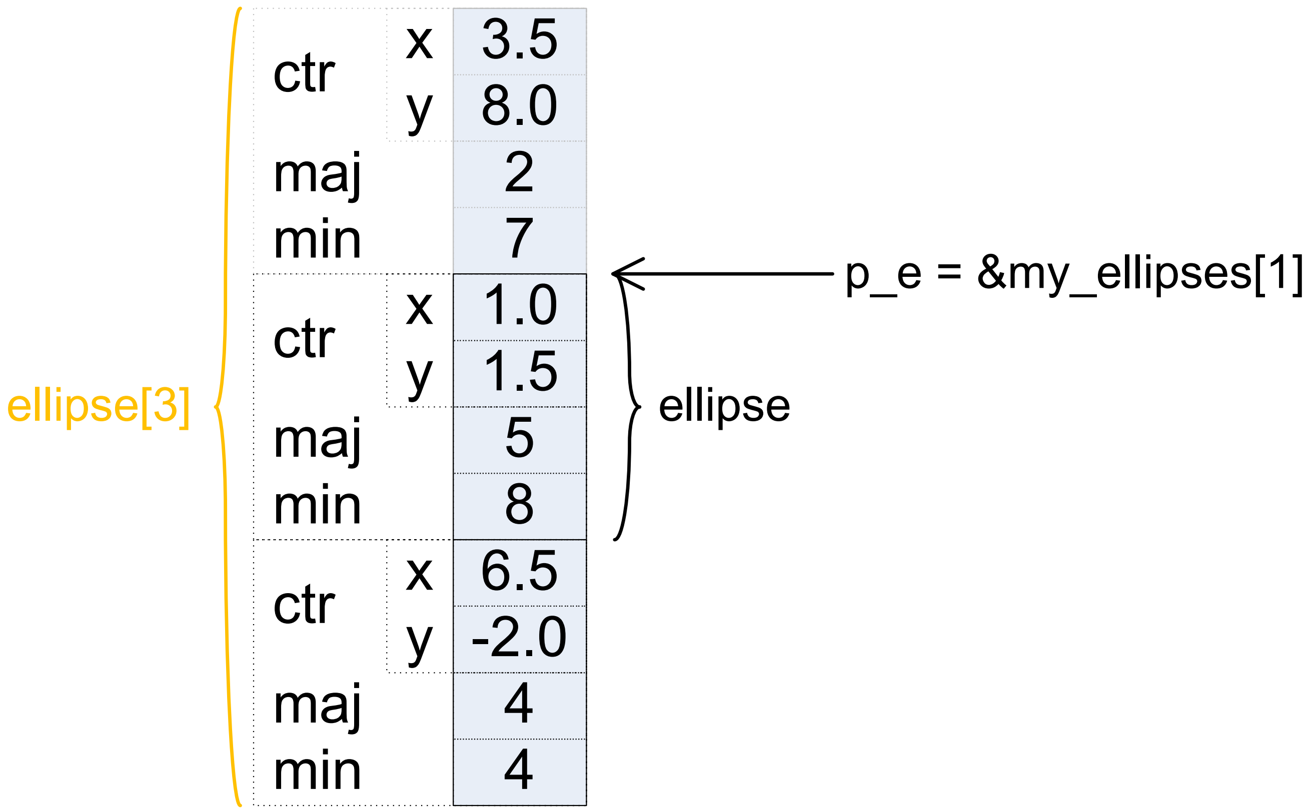
```
typedef struct {int x[2]; char y[4];} blah;
blah z = { {0, 0}, "foo" };

*(z.x + 42);              // error: top-level overflow
*(z.x + 2);               // error: subobject overflow

((int*) &z)[2];           // error: after bounds-narrowing cast

*((z.x + 42) - 42);       // non-error: via invalid (OOB) intermediate
((blah *) z.x)->y;        // non-error: after bounds-widening cast
*(int*)(intptr_t)z.x;     // non-error: via integer
*strfry(z.y);             // non-error: after uninstrumented code
```

Without type information, pointer bounds lose precision



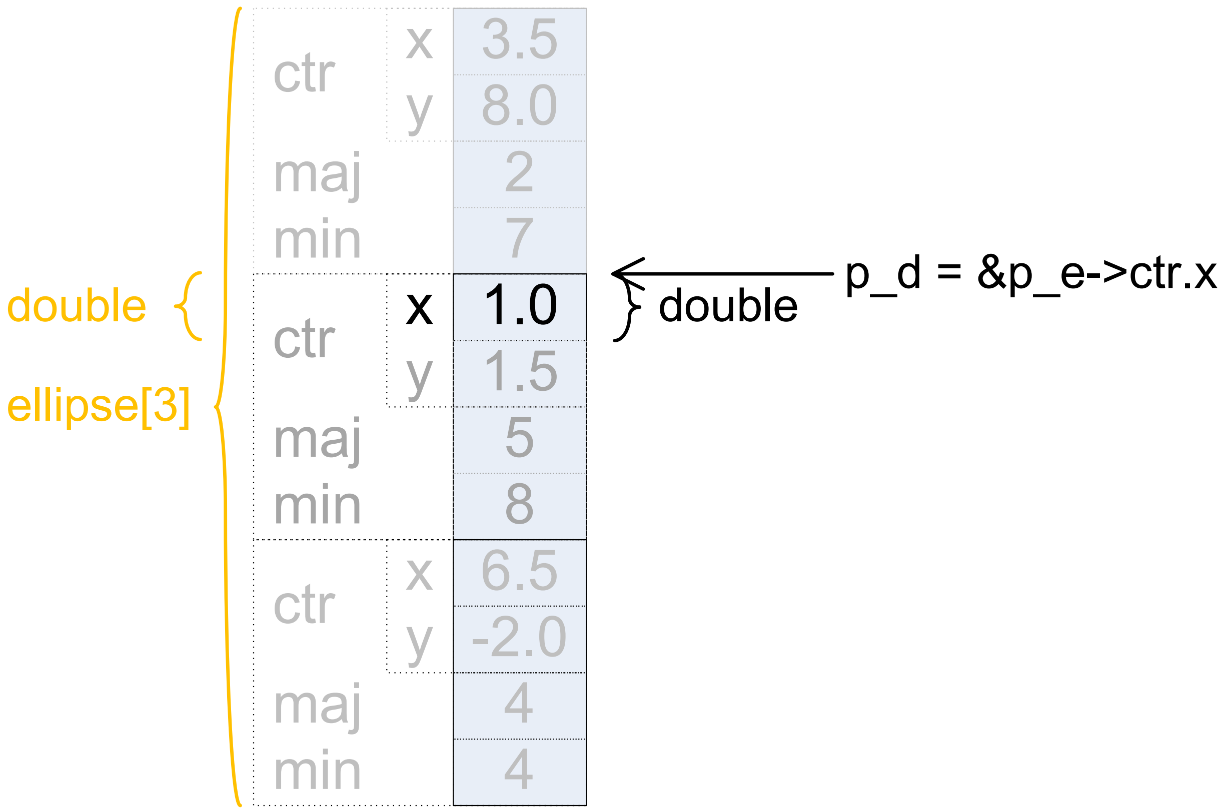

```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Given allocation type and pointer type, bounds are implicit



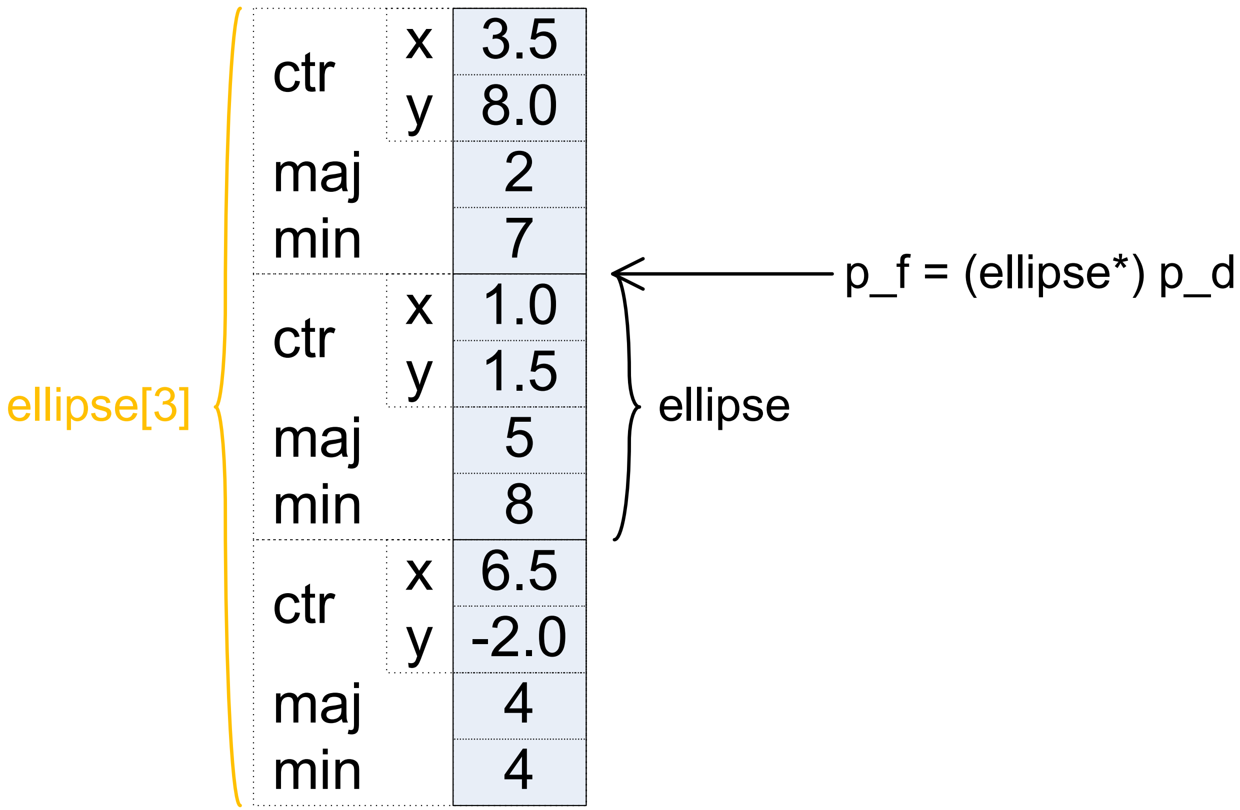

```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Given allocation type and pointer type, bounds are implicit




```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Given allocation type and pointer type, bounds are implicit




```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Widening and narrowing bounds: the pointed-to type matters!

```
struct driver     { /* ... */ } *d = /* ... */;
struct i2c_driver { /* ... */ struct driver driver; /* ... */ };

#define container_of(ptr, type, member) \
  ((type *)( (char *)(ptr) - offsetof(type,member) ))

i2c_drv = container_of(d, struct i2c_driver, driver);
```

Widening and narrowing bounds: the pointed-to type matters!

```
struct driver     { /* ... */ } *d = /* ... */;
struct i2c_driver { /* ... */ struct driver driver; /* ... */ };

#define container_of(ptr, type, member) \
  ((type *)( (char *)(ptr) - offsetof(type,member) ))

i2c_drv = container_of(d, struct i2c_driver, driver);
```

Here we first *widen* and then *narrow* the bounds.

- bounds of d: just the smaller struct
- bounds of the char*: the whole allocation
- bounds of i2c_drv: the bigger struct

The *type* of the storage matters! If only we knew it!

Making a checker that uses run-time type information

- casts matter! → can't rely on pointer provenance
- not an object table
- ... but more! logically a 'typed object' table

Use `alloc_get_type(p)` to write `__fetch_bounds(p)`!

- avoids SoftBound-style false positives and negatives
- avoids `libc` wrappers much of the time
- some extra cost – per-cast instrumentation

Existing bounds checkers redux, part 3 – a.k.a. ‘the competition’

Noting slowness and false positives… be even coarser!

- AddressSanitizer a.k.a. ASan
- … in both gcc and clang: `-fsanitize=address`

Uses best-effort techniques:

- shadow memory to catch small heap overflows
- *redzones* to catch stack and static overflows

Slowdown is mostly 50–250% – considered good!

Making it fast

Basic approach

- either ‘mostly fat pointers’ underneath
- *or* make typeinfo lookup blazing fast, somehow
- goal: competitive with (best of) ASan and SoftBound

Making it fast

Basic approach

- either **‘mostly fat pointers’ underneath**
- *or* make typeinfo lookup blazing fast, somehow
- goal: competitive with (best of) ASan and SoftBound

Making it fast

Basic approach

- either **‘mostly fat pointers’ underneath**
- *or* make typeinfo lookup blazing fast, somehow
- goal: competitive with (best of) ASan and SoftBound

To do better: can we ‘think like a VM’?

- avoiding deref checks
- speculate...
- goal: ‘as fast as Java’ (eventually)

Status: almost got to ASan…

Some bleeding-edge, very rough numbers (changing every day!)

bench	baseline/s	crunchpb/s	crunchSoft/s	ASan/s
lbm	11.3	12.2	11.6	13.0
sjeng	10.4	25.9	18.0	22.1
hmmer	6.5	27.8	20.8	12.6
mcf	10.8	25.1	$\times$	19.4
milc	33.3	164	46.0	52.4
gobmk	46	157	122	132
bzip2	22.6	45.8	35.1	38.4
sphinx3	4.31	12.6	8.11	7.09
h264ref	27.1	132	$\times$	73.2
gcc	–	–	–	7.11
perlbench	4.6	–	$\times$	$\times$

Thinking like a VM: how many checks should this code do?

```
int ret = 0;
for (int i = 0; i < n; ++i)
{ struct list_node *p = malloc(sizeof (struct list_node ));
  p->next = head;
  head = p;
}
for (int i = 0; i < m; ++i)
{ unsigned out = 0;
  for (struct list_node *p = head; p; p = p->next)
  { out += p->x; }
  ret += out;
}
return ret;
```

Don't check derefs; we have an MMU for that!

Just use a trapping representation for any bad pointers…

Trap pointers: problems

Initially, adding trap pointers made things *slower*!

- derefs are faster, but
- array/arith needs many more instructions!
  - set trap on OOB, unset on back-in-bounds
- I-cache footprint …

Trap pointers: solutions

Fast/slow path optimisation!

- clone function bodies at instrumentation time
- instrument ‘top half’ to handle common cases
- complex cases and check failures fall…
- … into bottom half, doing ‘full version’
- … including trap pointer manipulations

Thinking like a VM

Fast/slow split is a case of *speculative optimisation*

- how dynamic languages go fast!
- … without slowing down common case
- another use in our case: *continue past error*

Perf goal: ‘as fast as Java’

- very similar checks needed in both cases
- next: speculation to hoist checks out of loops
- will need lightweight dynamic compilation…
- use run-time type information to help

Conclusions

Unix-like abstractions can and should be evolved!

- e.g. with type info $\rightarrow$ type & bounds checking
- can be binary-compatible, mostly C-source-compatible
- good prospects for extension
- next: pointer metadata to enable fast+precise GC

Code is here:

- http://github.com/stephenrkell/liballocs/
- http://github.com/stephenrkell/libcrunch/

Thanks for your attention. Questions?