# Fast, precise dynamic checking of types and bounds “in C”

Stephen Kell

stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge

# What could go wrong?

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

# What could go wrong?

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

unchecked

# What else could go wrong?

```
typedef struct {
  // ...
  int blc_size [8][2];    // block sizes
  int part_size [8][2];   // partition sizes
  // ...
} InputParameters;

... input->blc_size[currMB->mb_type][0] ...
```

# What else could go wrong?

```
typedef struct {
  // ...
  int blc_size [8][2];   // block sizes
  int part_size [8][2];  // partition sizes
  // ...
} InputParameters;   unchecked

... input->blc_size[currMB->mb_type][0] ...

#define P8x8    8
#define I4MB    9
#define I16MB  10
// ...
#define MAXMODE 15
```

# What could go right?

```
arc = (arc_t *) realloc( net->arcs, net->max_m * sizeof(arc_t) );
if( !arc ) { /* fail ... */ }

off = (size_t)arc - (size_t)net->arcs;
net->arcs = arc;

for( /* ... */; node < net->stop_nodes; node++ )
    node->basic_arc = (arc_t *)((size_t)node->basic_arc + off);
```

# What could go right?

```
arc = (arc_t *) realloc( net->arcs, net->max_m * sizeof(arc_t) );
 if( !arc ) { /* fail ... */ }
```

possible use after free?

```
off = (size_t)arc - (size_t)net->arcs;
net->arcs = arc;

for( /* ... */; node < net->stop_nodes; node++ )
    node->basic_arc = (arc_t *)((size_t)node->basic_arc + off);
```

unchecked

# What could go right?

```
arc = (arc_t *) realloc( net->arcs, net->max_m * sizeof(arc_t) );
if ( !arc ) { /* fail ... */ }
```

possible use after free?

```
off = (size_t)arc - (size_t)net->arcs;
net->arcs = arc;

for( /* ... */; node < net->stop_nodes; node++ )
    node->basic_arc = (arc_t *)((size_t)node->basic_arc + off);
```

unchecked

Note: this code is correct! Want *safe*, but also *permissive*

# This talk in one slide

I'll talk about

- run-time type checking in C
- run-time bounds checking in C
  - made precise, but permissive
  - again, using type information

More generally I'll be arguing

- a safe implementation of C is feasible
  - certainly not a contradiction in terms!
- can borrow ideas from 'safe' virtual machines
- but avoid giving up the *address space abstraction*

# This talk in one slide

I'll talk about

- run-time type checking in C
- run-time bounds checking in C
  - made precise, but permissive
  - again, using type information

More generally I'll be arguing

- a safe implementation of C is feasible
  - certainly not a contradiction in terms!
- can borrow ideas from 'safe' virtual machines
- but avoid giving up the *address space abstraction*

Unlike prior work, can deal with the foregoing codebases. . .

# What makes C unsafe? Roughly…

**typedef struct** {**int** x[2]; **char** y[4];} blah; blah z;

# What makes C unsafe? Roughly…

```
typedef struct {int x[2]; char y[4];} blah; blah z;

int i1 = z.x[2];        // no spatial check!
int i2 = *(z.x + 3);    // pointer arith, arrays
```

# What makes C unsafe? Roughly…

```
typedef struct {int x[2]; char y[4];} blah; blah z;

int i1  = z.x[2];        // no spatial check!
int i2  = *(z.x + 3);    // pointer arith, arrays
```

- maybe try checking via object tables or fat pointers?

# What makes C unsafe? Roughly…

```
typedef struct {int x[2]; char y[4];} blah; blah z;

int i1 = z.x[2];        // no spatial check!
int i2 = *(z.x + 3);    // pointer arith, arrays
```

- maybe try checking via object tables or fat pointers?

```
// no temporal checks
blah *pz = malloc(sizeof *pz); ...; free(pz); *pz = ...
return &pz;
```

# What makes C unsafe? Roughly…

```
typedef struct {int x[2]; char y[4];} blah; blah z;

int i1 = z.x[2];        // no spatial check!
int i2 = *(z.x + 3);    // pointer arith, arrays
```

- maybe try checking via object tables or fat pointers?

```
// no temporal checks
blah *pz = malloc(sizeof *pz); ...; free(pz); *pz = ...
return &pz;
```

- maybe add an 'alloc ID' checker? conservative GC?

# What makes C unsafe? Roughly…

```
typedef struct {int x[2]; char y[4];} blah; blah z;

int i1  = z.x[2];       // no spatial check!
int i2  = *(z.x + 3);   // pointer arith, arrays
```

- maybe try checking via object tables or fat pointers?

```
// no temporal checks
blah *pz = malloc(sizeof *pz);  ...;  free(pz);  *pz = ...
return &pz;
```

- maybe add an ‘alloc ID’ checker? conservative GC?

```
int *pi = (int*) &z;    // this is okay
int *pj = (int*) &z.y;  // this probably not
return &pz;
```

# What makes C unsafe? Roughly...

```
typedef struct {int x[2]; char y[4];} blah; blah z;

int i1 = z.x[2];        // no spatial check!
int i2 = *(z.x + 3);    // pointer arith, arrays
```

- maybe try checking via object tables or fat pointers?

```
// no temporal checks
blah *pz = malloc(sizeof *pz); ...; free(pz); *pz = ...
return &pz;
```

- maybe add an ‘alloc ID’ checker? conservative GC?

```
int *pi = (int*) &z;    // this is okay
int *pj = (int*) &z.y;  // this probably not
return &pz;
```

- type checking in C?

# How about *starting* with type-checking?

- `$ crunchcc -o myprog ...     # + other front-ends`

# How about *starting* with type-checking?

- `$ crunchcc -o myprog ...    # + other front-ends`
- `$ ./myprog                         # runs normally`

# How about *starting* with type-checking?

- `$ crunchcc -o myprog ...    # + other front-ends`
- `$ ./myprog                             # runs normally`
- `$ LD_PRELOAD=libcrunch.so ./myprog # does checks`

# How about *starting* with type-checking?

- `$ crunchcc -o myprog ...    # + other front-ends`
- `$ ./myprog                          # runs normally`
- `$ LD_PRELOAD=libcrunch.so ./myprog # does checks`
- `myprog: Failed __is_a_internal(0x5a1220, 0x413560 a.k.a. "uint$32") at 0x40dade, allocation was a heap block of int$32 originating at 0x40daa1`

Reminiscent of Valgrind (Memcheck), but different. . .

- not checking memory definedness, in-boundsness, etc..
- . . . in fact, *assume correct* w.r.t. these!
- provide & exploit *run-time type information*

# Sketch of the instrumentation for C

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

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

# Sketch of the instrumentation for C

```
if (obj−>type == OBJ_COMMIT) {
  if (process_commit(walker,
        (CHECK(__is_a(obj, "struct␣commit")),
          (struct commit *)obj)))
    return −1;
  return 0;
}
```

# Sketch of the instrumentation for C

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

Need a runtime that knows what's on the end of a pointer

- provides a fast `__is_a()` function
- … and a few other flavours of check
- by allowing reflection on *allocations*
- … and attaching reified type info

# Who says native code doesn't do reflection? (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 native code doesn't do reflection? (2)

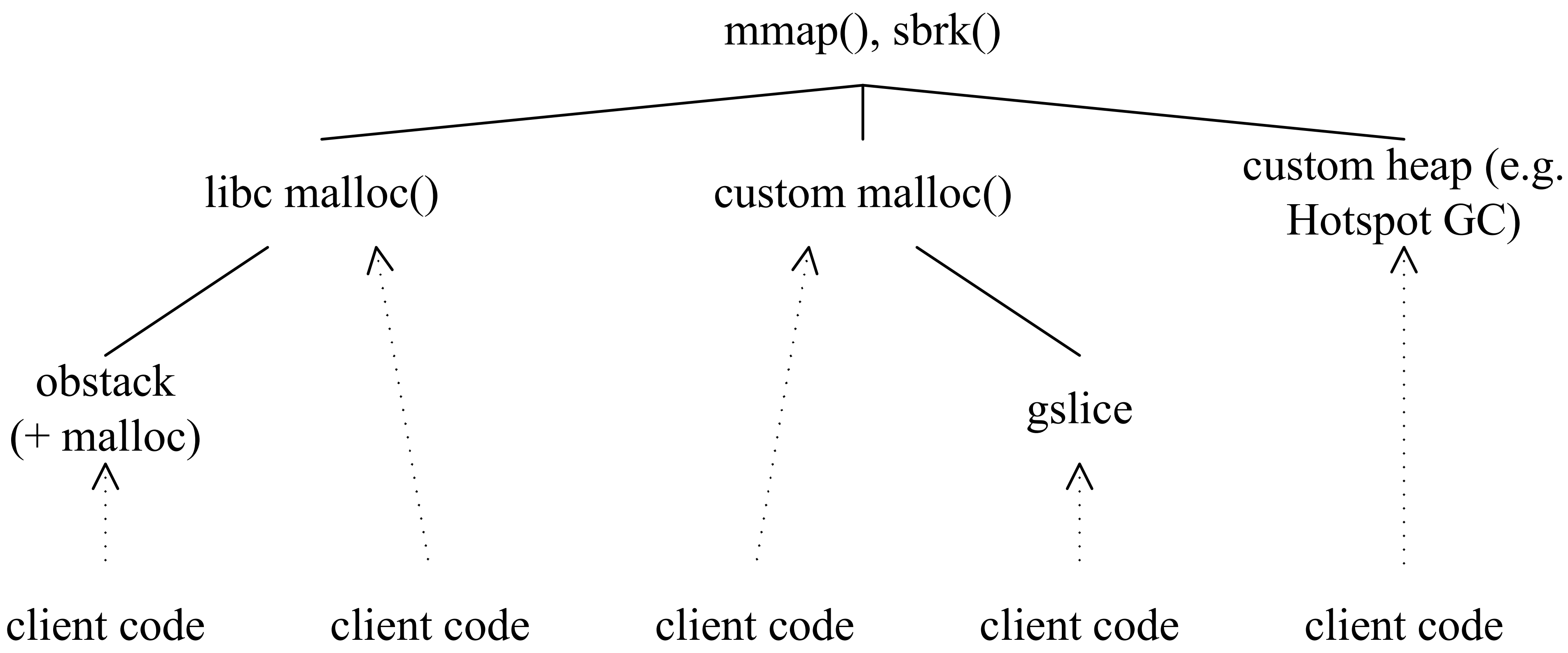
```
$ cc -g -o hello hello.c && readelf -wi hello | column
```

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

# Adding a meta-level to Unix-like processes (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. . .

# A meta-level API

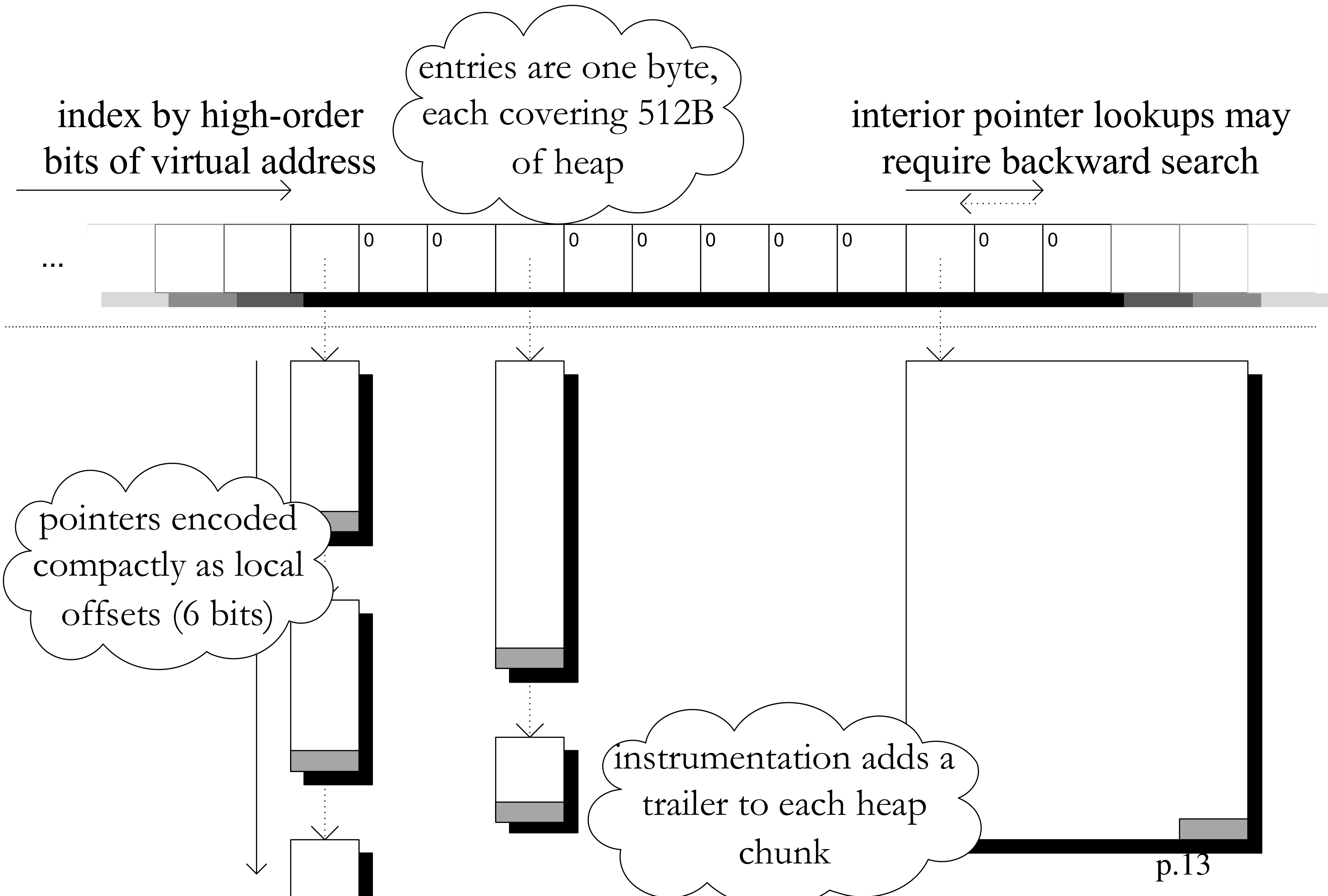
```
struct uniqtype;                                           /* type descriptor    */
struct allocator;                                /* heap, stack, static, etc */
allocator * alloc_get_allocator (void *obj); /* which one? (at leaf) */
uniqtype * alloc_get_type       (void *obj);       /* what type?         */
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...

Overhead of providing this API is usually $< 5\%$

# One way to make a malloc() heap more reflective



# Reified, unique data types

```
struct ellipse {
    double maj, min;
    struct point { double x, y; } ctr ;
};
```

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| _uniqtype__int | “int” | 4 | 0 | | | | | | |
| _uniqtype__double | “double” | 8 | 0 | | | | | | |
| _uniqtype__point | 0 | 16 | 2 | 0 | | 8 | | | |
| _uniqtype__ellipse | “ellipse” | 32 | 3 | 0 | | 8 | | 16 | |

- also model: stack frames, functions, pointers, arrays, . . .
- unique → “exact type” test is a pointer comparison
- **__is_a() is a short search over containment edges**

# Is it really that simple? What about. . . ?

- untyped malloc() et al.
- opaque pointers, a.k.a. void*
- conversion of pointers to integers and back
- function pointers
- pointers to pointers
- “simulated subtyping”
- {custom, nested} heap allocators
- alloca()
- “sloppy” (non-standard-compliant) code
- unions, varargs, memcpy()

# Is it really that simple? What about. . . ?

- **untyped malloc() et al.**
- opaque pointers, a.k.a. void*
- conversion of pointers to integers and back
- function pointers
- **pointers to pointers**
- “simulated subtyping”
- {custom, nested} heap allocators
- alloca()
- “sloppy” (non-standard-compliant) code
- unions, varargs, memcpy()

# What data type is being malloc()'d?

Use intraprocedural "sizeofness" analysis

```
size_t sz = sizeof (struct Foo);
/* ... */
malloc(sz);
```

Sizeofness propagates, a bit like dimensional analysis.

# What data type is being malloc()'d?

Use intraprocedural "sizeofness" analysis

```
size_t sz = sizeof (struct Foo);
/* ... */
malloc(sz);
```

Sizeofness propagates, a bit like dimensional analysis.

```
malloc(sizeof (Blah) + n * sizeof (struct Foo))
```

# What data type is being malloc()'d?

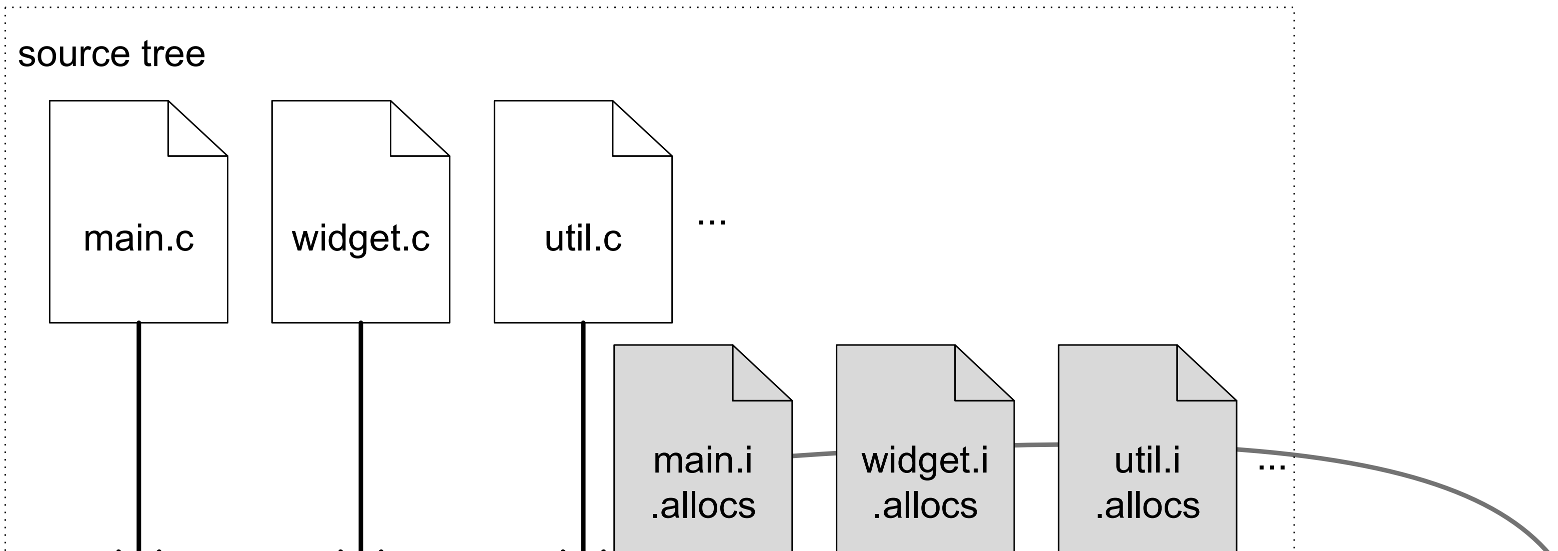
Use intraprocedural "sizeofness" analysis

```
size_t sz = sizeof (struct Foo);
/* ... */
malloc(sz);
```

Sizeofness propagates, a bit like dimensional analysis.

```
malloc(sizeof (Blah) + n * sizeof (struct Foo))
```

Dump *typed allocation sites* from compiler, for later pick-up



# Polymorphism via multiply-indirected void

```
void sort_eight_special (void **pt){
    void *tt [8];
    register int i;
    for(i=0;i<8;i++)tt[i]=pt[i];
    for(i=XUP;i<=TUP;i++){pt[i]=tt[2*i]; pt[OPP_DIR(i)]=tt[2*i+1];}
}
neighbor = (int **)calloc(NDIRS, sizeof(int *));
sort_eight_special ((void **) neighbor ); // ← must allow!
```

- solution: tolerate casts from T** to void**…
- and check *writes through* void**
- … *against the underlying object type* (here int *[])

# Performance data: C-language SPEC CPU2006 benchmarks

| bench | normal/$s$ | crunch % | nopreload |
|---|---|---|---|
| bzip2 | 4.95 | +6.8% | +1.4% |
| gcc | 0.983 | +160 % | – % |
| gobmk | 14.6 | +11 % | +2.0% |
| h264ref | 10.1 | +3.9% | +2.9% |
| hmmer | 2.16 | +8.3% | +3.7% |
| lbm | 3.42 | +9.6% | +1.7% |
| mcf | 2.48 | +12 % | (−0.5%) |
| milc | 8.78 | +38 % | +5.4% |
| sjeng | 3.33 | +1.5% | (−1.3%) |
| sphinx3 | 1.60 | +13 % | +0.0% |
| perlbench | | | |

# Experience on “correct” code

| benchmark | compile fixes | run-time false positives | | |
|---|---|---|---|---|
| | | instances | unique (of which. . . ) | |
| | | | total | unhelpful |
| bzip2 | 0 | 48 | 3 | 3 |
| gcc | 1 | $3 \times 10^5$ | 14 | 3 |
| gobmk | 0 | 0 | 0 | 0 |
| h264ref | 2 | 27 | 2 | 0 |
| hmmer | 0 | 0 | 0 | 0 |
| lbm | 0 | $5 \times 10^7$ | 8 | 0 |
| mcf | 0 | 0 | 0 | 0 |
| milc | 0 | 0 | 0 | 0 |
| sjeng | 0 | 0 | 0 | 0 |
| sphinx3 | 0 | 0 | 0 | 0 |

# A “helpful” false positive?

```
typedef double LBM_Grid[SIZE_Z*SIZE_Y*SIZE_X*N_CELL_ENTRIES];
typedef LBM_Grid* LBM_GridPtr;

#define MAGIC_CAST(v) ((unsigned int*) ((void*) (&(v))))
#define FLAG_VAR(v) unsigned int* const _aux_ = MAGIC_CAST(v)
// ...
#define TEST_FLAG(g,x,y,z,f)   \
    ((*MAGIC_CAST(GRID_ENTRY(g, x, y, z, FLAGS))) & (f))
#define SET_FLAG(g,x,y,z,f)    \
{FLAG_VAR(GRID_ENTRY(g, x, y, z, FLAGS)); (*_aux_) |= (f);}
```

# More helpful false positives

```
item->util = xcalloc(sizeof(struct branch_info), 1);
// calloc arguments inverted
```

```
if (((*array4D) = (short****)calloc(idx,sizeof(short**))) == NULL)
  no_mem_exit("get_mem4Dshort: array4D");
// wrong sizeof
```

```
int length = (len-1) * sizeof (tree) + sizeof (struct tree_vec);
// associativity of '+'...
```

# What about unhelpful?

Two main cases

- ‘effective type’ trickery
- ‘pointer stuffing’ – our check is over-eager

```
if (value->kind > RTX_DOUBLE && value->un.addr.base != 0)
  switch (GET_CODE (value->un.addr.base))
    {
    case SYMBOL_REF:
      /* Use the string's address, not the SYMBOL_REF's address,
         for the sake of addresses of library routines.  */
      value->un.addr.base = (rtx) XSTR (value->un.addr.base, 0);
      break;
    /* ... */
    }
```

# Effective type trickery

```
s->arr1 = BZALLOC( n         * sizeof(UInt32)  );
s->arr2 = BZALLOC( (n+delta) * sizeof(UInt32) );
// ...

s->mtfv = (UInt16*)s->arr1;
```

# Effective type trickery

```
s->arr1 = BZALLOC( n         * sizeof(UInt32) );
s->arr2 = BZALLOC( (n+delta) * sizeof(UInt32) );
// ...

s->mtfv = (UInt16*)s->arr1;
```

The *effective type* of an object for an access to its stored value is the declared type of the object, if any.[87)] If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object having no declared type using **`memcpy`** or **`memmove`**, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.

# Effective type trickery

```
s->arr1 = BZALLOC( n         * sizeof(UInt32) );
s->arr2 = BZALLOC( (n+delta) * sizeof(UInt32) );
// ...
__liballocs_add_type_to_block (s->arr1,
    &__uniqtype__short_unsigned_int);
s->mtfv = (UInt16*)s->arr1;
```

The *effective type* of an object for an access to its stored value is the declared type of the object, if any.[87)] If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object having no declared type using **`memcpy`** or **`memmove`**, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.

# Restricting and extending C a little bit

We are more stringent than “effective types”:

- even heap storage has a declared type!
- must signpost changes explicitly

We are sometimes *looser* than standard C

- e.g. allowed integer $\rightarrow$ pointer conversions
- e.g. (ask me) __like_a() checks

More reasons to be looser than standard C

- out-of-bounds pointer creation…
- … same tricks will also solve ‘pointer stuffing’

# Difficult-to-check bounds errors and non-errors

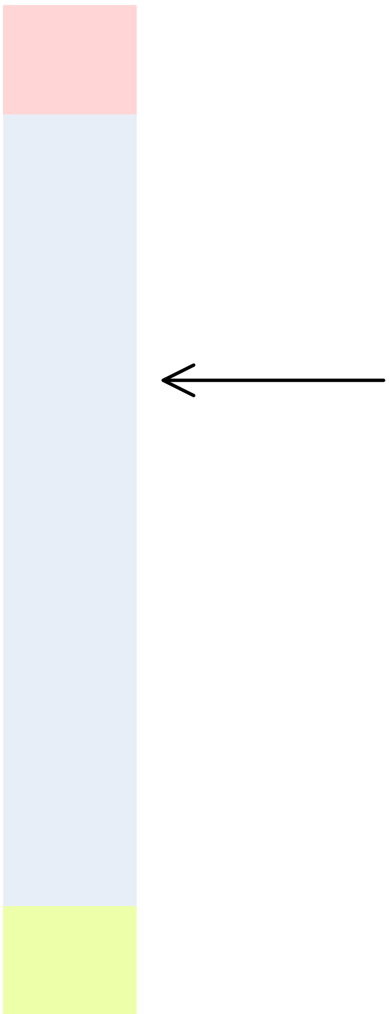
```
typedef struct {int x[2]; char y[2];} blah;
blah z = { {0, 0}, "!" };

*(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
```

# 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...

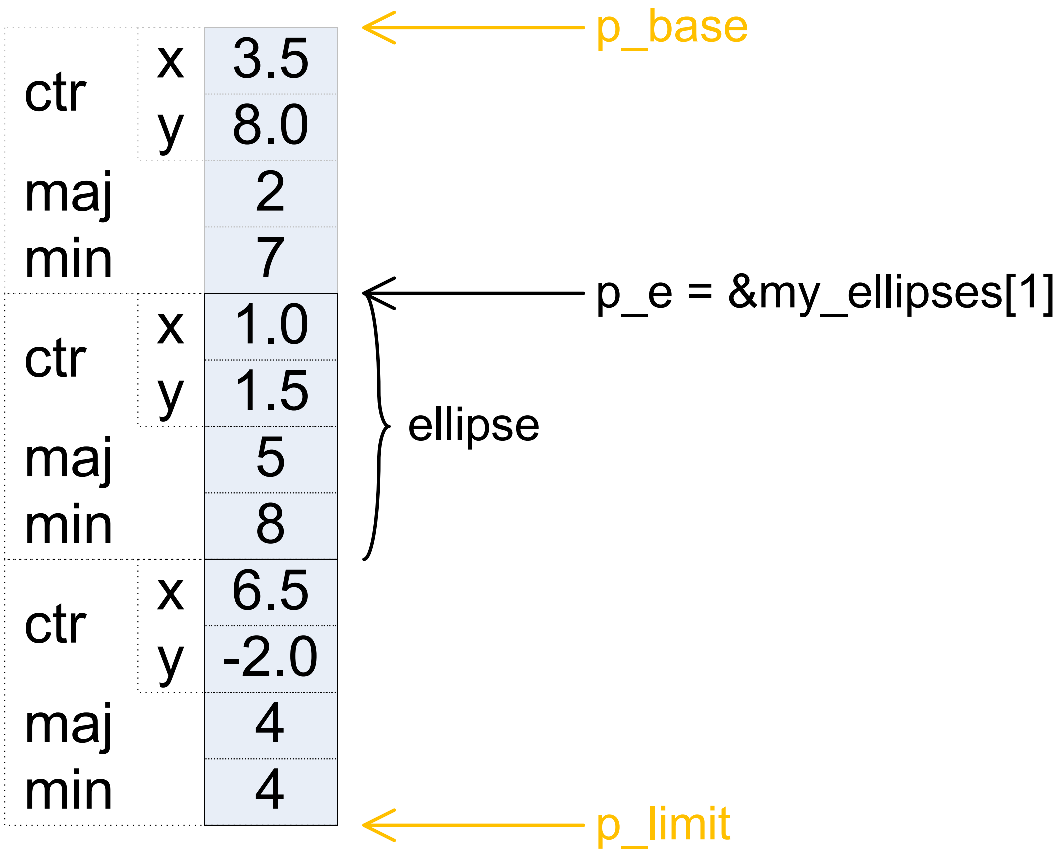
Also: fairly slow!

# Existing bounds checkers redux, part 2

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

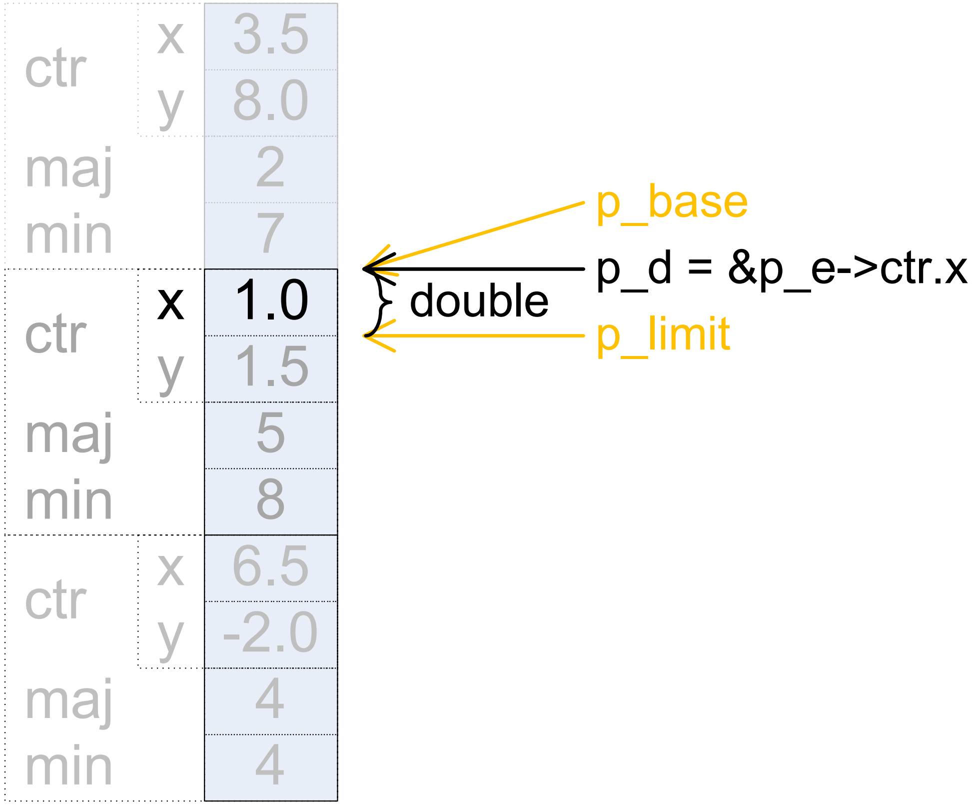
- make pointers bigger: $\{addr, base, limit\}$
- on each `p[i]` or `*p`, check 'within bounds'

# 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:

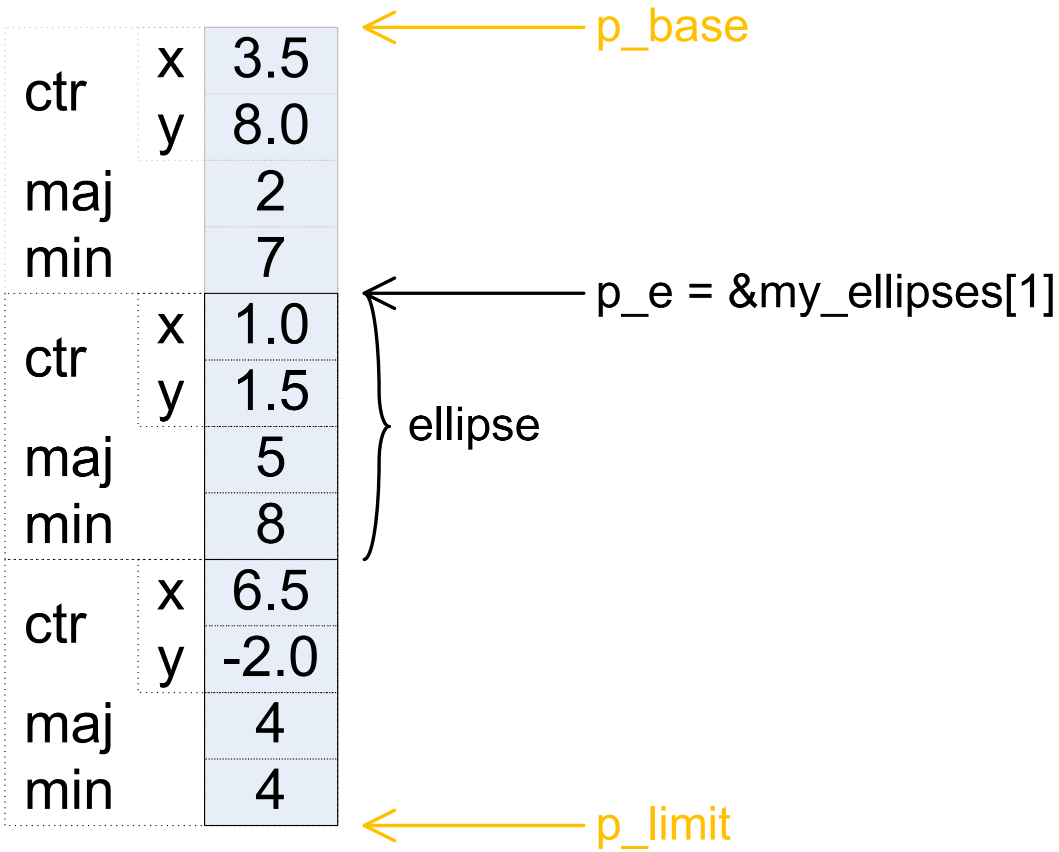
- tolerates all out-of-bounds pointers
- can catch subobject overflows by *pointer provenance*

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, …

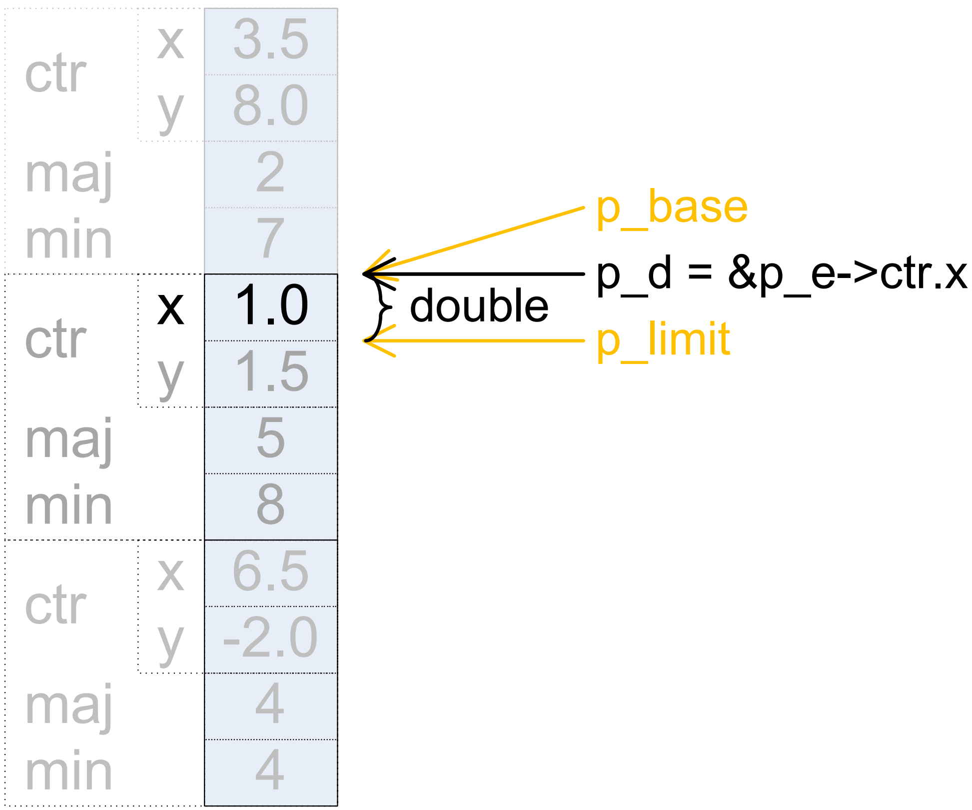
Also: fairly slow!

# Existing checkers using per-pointer metadata



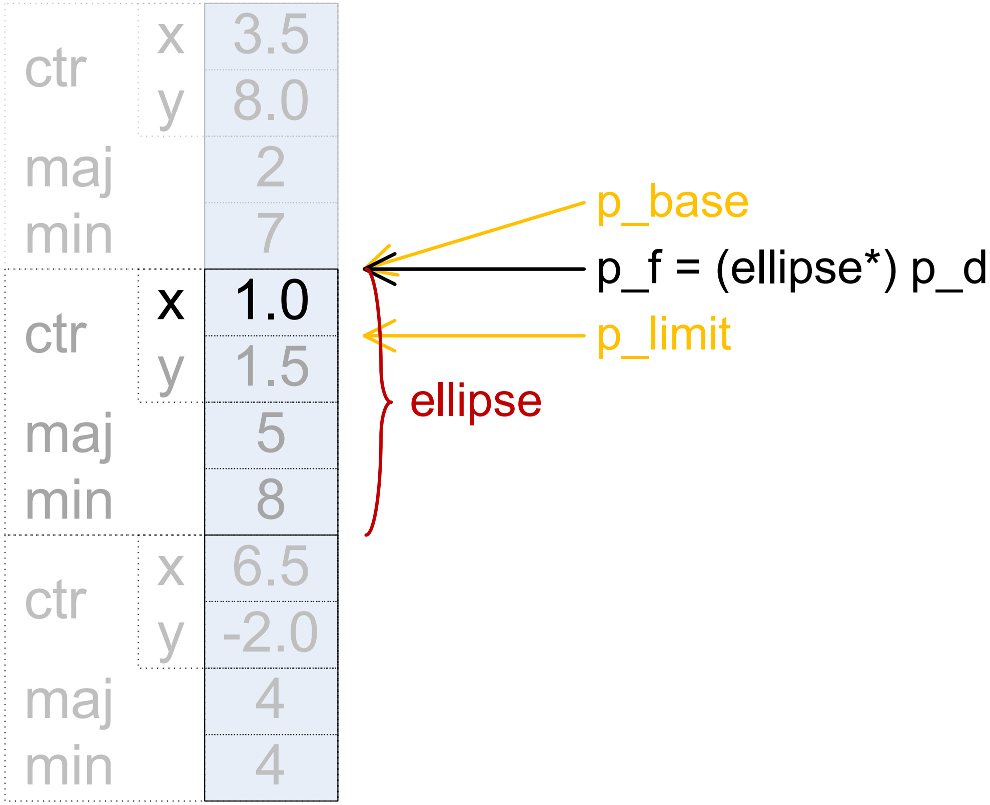

```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

# Existing checkers using per-pointer metadata



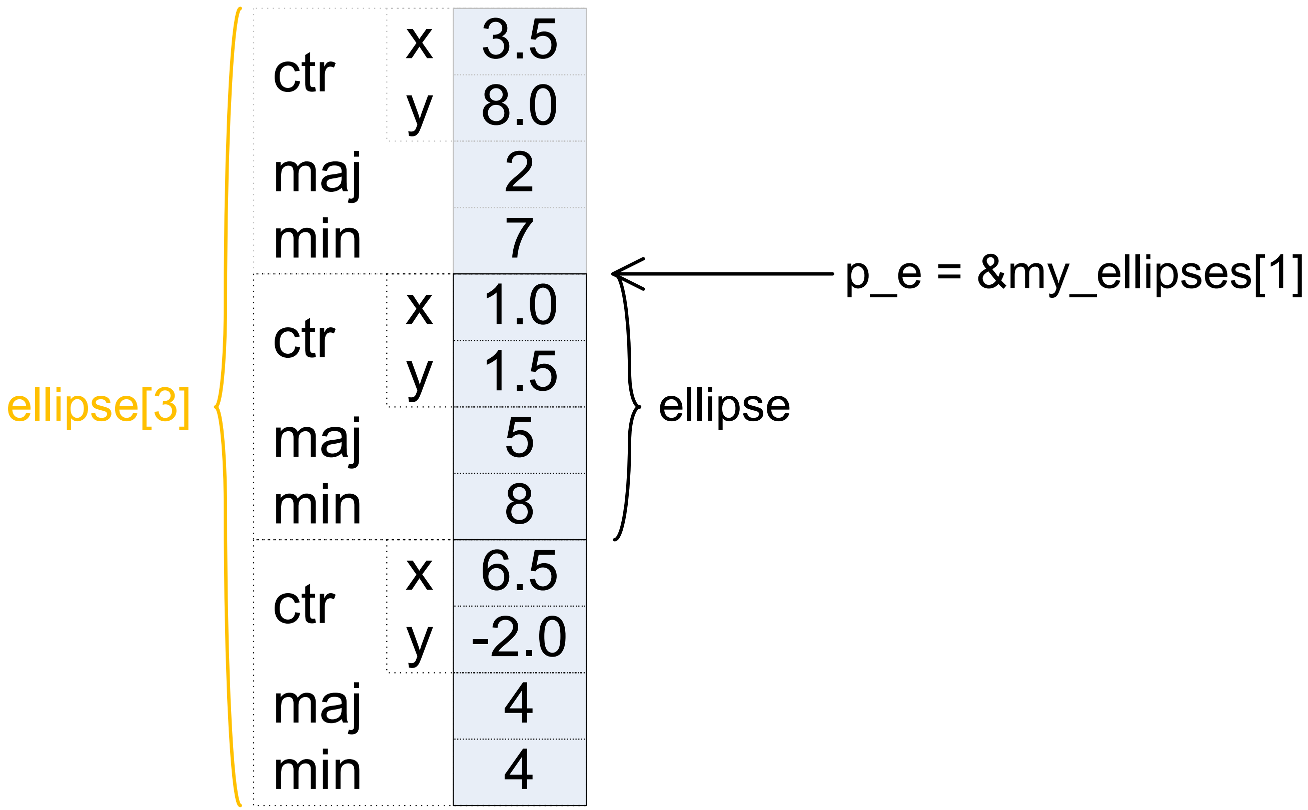

```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

# Without type information, pointer bounds lose precision



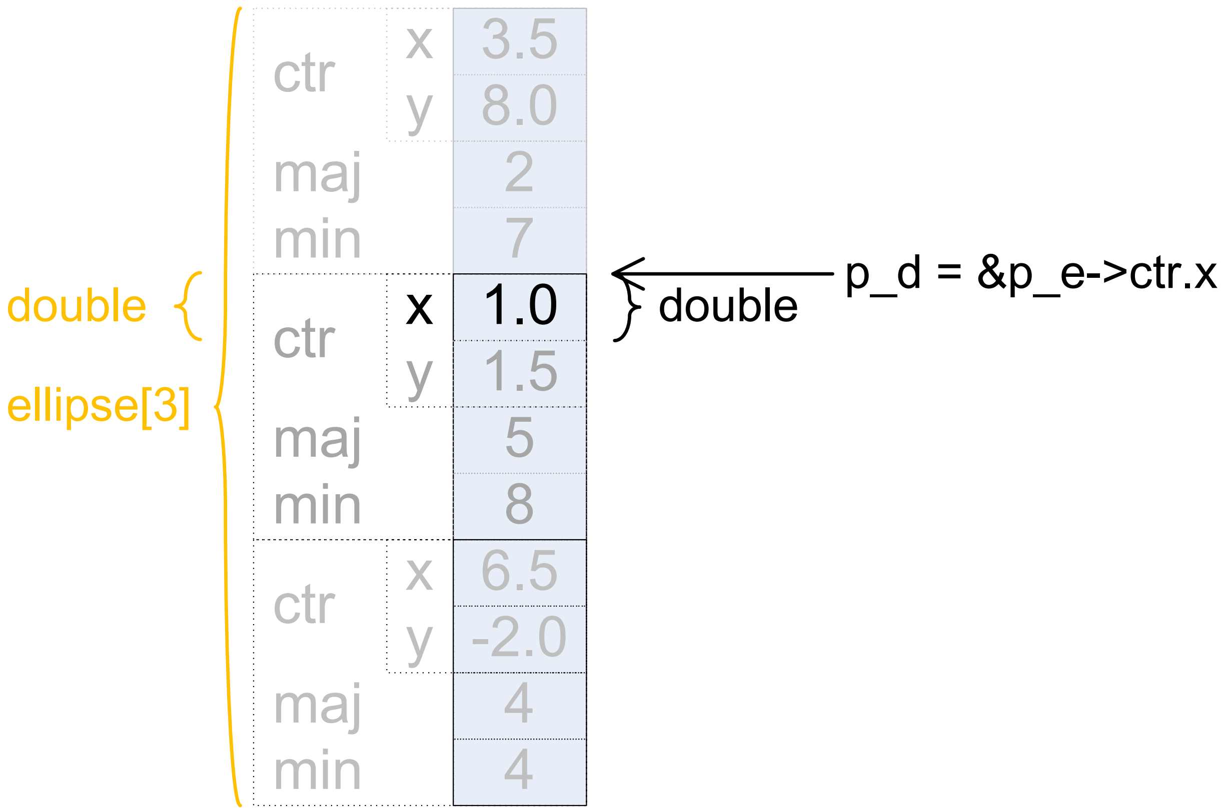

```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

# Given allocation type and pointer type, bounds are implicit



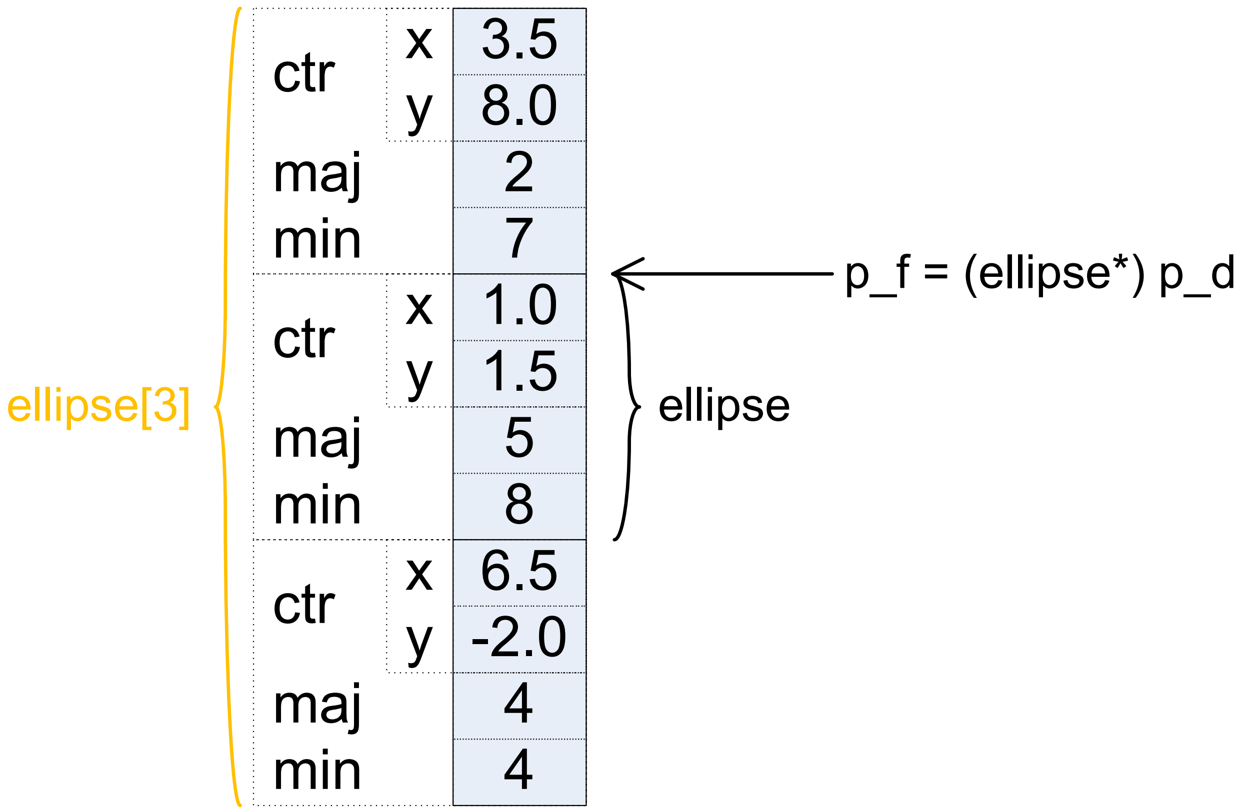

```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

# 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];
```

# The importance of being type-aware

```
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);
```

# The importance of being type-aware

```
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);
```

SoftBound et al. are oblivious to casts, but they matter!

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

If only we knew the *type* of the storage!

# Naïve idea

We can write a precise bounds checker easily

- new version of __is_a() that also returns bounds!
- i.e. how much memory either side has the same type
- look up bounds whenever we index/deref

Good things!

- avoid type-confused false positives
- avoid libc wrappers, ...
- robust to uninstrumented callers/callees

Problem: slow!

- querying liballocs too often is slow

# 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-like performance. . .

- lots more possible-optimisations not tried yet. . .

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

| bench | baseline/$s$ | crunchpb/$s$ | crunchSoft/$s$ | ASan/$s$ |
|---|---|---|---|---|
| lbm | 11.3 | 14.3 | 11.5 | 13.0 |
| sjeng | 10.4 | 23.9 | 17.6 | 22.1 |
| hmmer | 6.5 | 29.9 | 17.7 | 12.6 |
| mcf | 10.8 | 22.0 | × | 19.4 |
| milc | 33.3 | 99.9 | 44.6 | 52.4 |
| gobmk | 46 | 139 | 117 | 132 |
| bzip2 | 22.6 | 41.8 | 34.7 | 38.4 |
| sphinx3 | 4.31 | 12.1 | 8.08 | 7.09 |
| h264ref | 27.1 | 119 | × | 73.2 |
| gcc | – | – | – | 7.11 |
| perlbench | 4.6 | – | × | × |

# Thinking like a VM

Intuition: these checks are very similar to Java-like langs

- array bounds
- cast
- (+ some kind of GC eventually)

We should be roughly as fast! Any tricks we're missing?

- no deref checking
- fast/slow path separation

# Handling one-past pointers (diagram: Vladsinger, CC-BY-SA 3.0)

On x86-64, use noncanonical addresses as trap reps

# 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

# More 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*

Further towards ‘as fast as Java’

- 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?

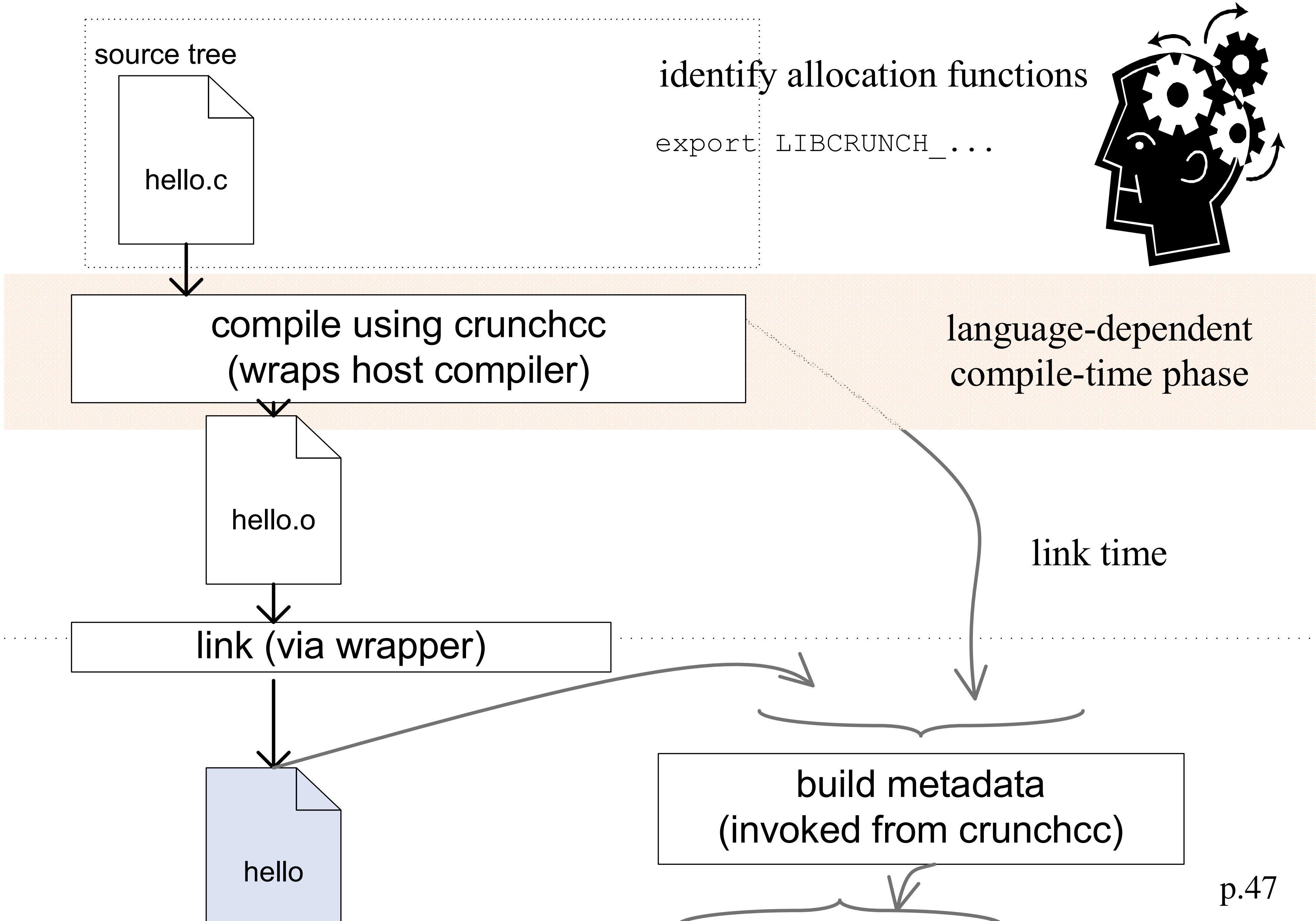
# No more deref checking

```
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;
```

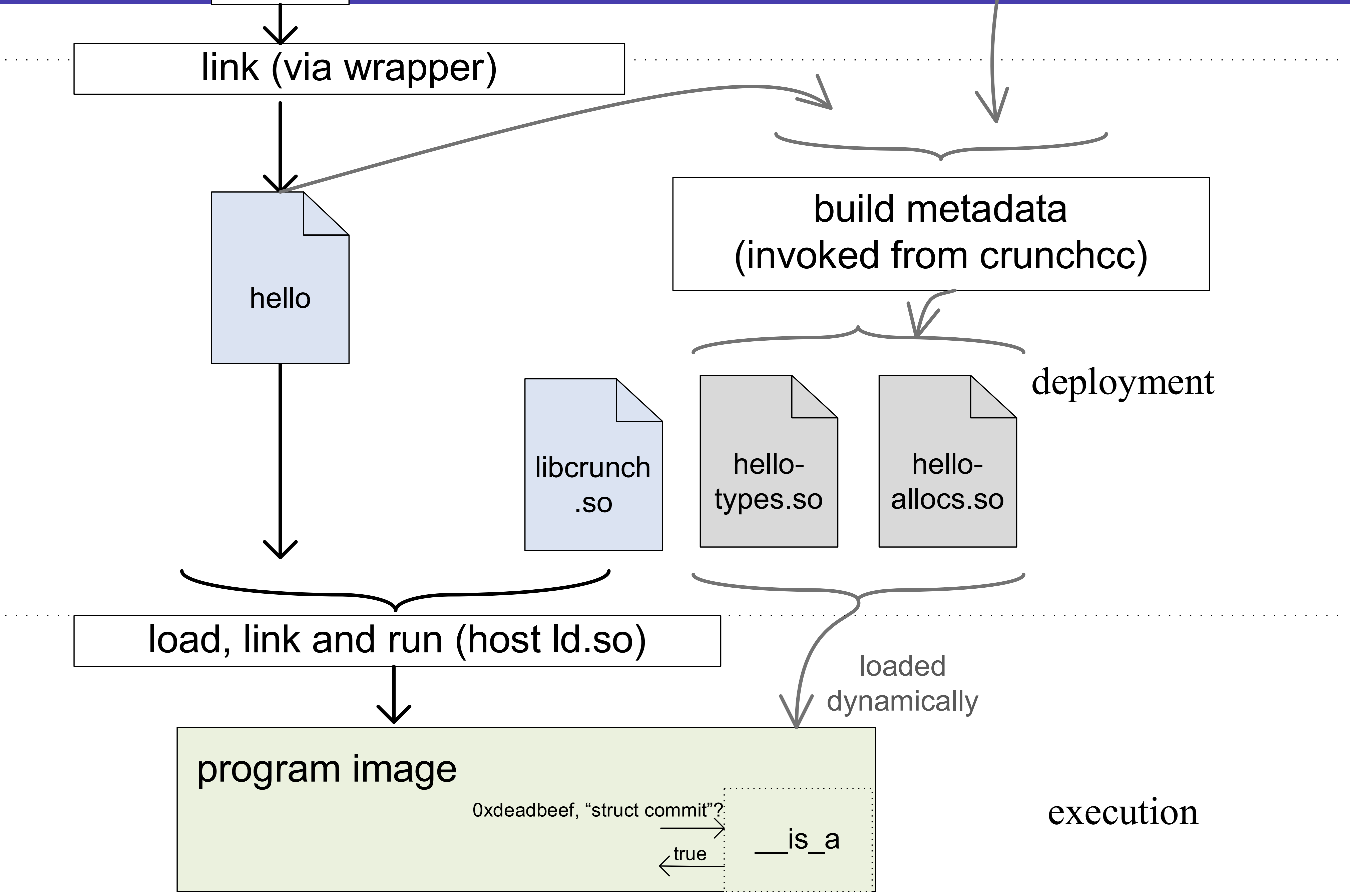
# Why SoftBound can't grok h264ref

```
// order list 0 by PicNum
qsort((void *) listX [0], list0idx , sizeof(StorablePicture*),
      compare_pic_by_pic_num_desc);

// where...
static int compare_pic_by_pic_num_desc( const void *arg1,
      const void *arg2 )
{
   if ( (*(StorablePicture**)arg1)->pic_num <
      (*(StorablePicture**)arg2)->pic_num)
      return 1;
   // ...
}
// no bounds info passed with these pointers! so abort ...
```

# Toolchain – top half



# Toolchain – bottom half



# Arguably helpful false positives (1)

```
#define FUNC_CALL(r) (((AttributeDef*)&(r))->func_call)

typedef struct Sym {
    /* ... */
    long r;      /* associated register */
    /* ... */
} Sym;

func_attr_t *func_call = FUNC_CALL(sym->r);
```

# Arguably helpful false positives (2)

```
typedef int parse_opt_cb(const struct option *,
    const char *arg, int unset);

static int stdin_cacheinfo_callback(struct parse_opt_ctx_t *ctx,
                                    const struct option *opt, int unset)
{ /* ... */ }

struct option options[] = {
    /* ... */,
    {OPTION_LOWLEVEL_CALLBACK, 0, /* ...*/,
        (parse_opt_cb *) stdin_cacheinfo_callback},
    /* ... */
};
```

# Arguably helpful false positives (3)

```
if (value->kind > RTX_DOUBLE && value->un.addr.base != 0)
  switch (GET_CODE (value->un.addr.base))
    {
    case SYMBOL_REF:
      /* Use the string's address, not the SYMBOL_REF's address,
         for the sake of addresses of library routines. */
      value->un.addr.base = (rtx) XSTR (value->un.addr.base, 0);
      break;
    /* ... */
    }
```

# Implementation to-do

Fix remaining holes

- memcpy
- varargs (instrument caller, check in callee)
- unions (instrument writes; instrument reads)
- VLA (like alloca()), VLAIS?

# Generic pointers to pointers to non-generic pointers

```
PUB_FUNC void dynarray_add(void ***ptab, int *nb_ptr, void *data)
{   /* ... */
    /* every power of two we double array size */
    if ((nb & (nb - 1)) == 0) {
        if (!nb) nb_alloc = 1; else nb_alloc = nb * 2;
        pp = tcc_realloc (pp, nb_alloc * sizeof(void *));
        *ptab = pp;
    }
    /* ... */
}
char **libs = NULL;
/* ... */
dynarray_add((void ***) &libs, &nblibs, tcc_strdup(filename));
```

# A small departure from standard C

The *effective type* of an object for an access to its stored value is the declared type of the object, if any.[87)] If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object having no declared type using **`memcpy`** or **`memmove`**, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.

# A small departure from standard C

The *effective type* of an object for an access to its stored value is the declared type of the object, if any.[87)] If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object having no declared type using **`memcpy`** or **`memmove`**, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.

## Instead:

- all allocations have $\leq 1$ effective type
- stack, locals / actuals: use declared types
- heap, alloca(): use *allocation site* (+ finesse)
- trap memcpy() and reassign type

# Handling one-past pointers

```
#define LIBCRUNCH_TRAP_TAG_SHIFT 48
inline void *__libcrunch_trap(const void *ptr, unsigned short tag)
{ return (void *)((( uintptr_t ) ptr)
    ^ ((( uintptr_t ) tag) << LIBCRUNCH_TRAP_TAG_SHIFT));
}
```

Tag allows distinguishing different kinds of trap rep:

- LIBCRUNCH_TRAP_ONE_PAST
- LIBCRUNCH_TRAP_ONE_BEFORE

# What is “type-correctness”?

“Type” means “data type”

- instantiate = allocate
- concerns storage
- “correct”: reads and writes respect allocated data type
- cf. *memory*-correct (spatial, temporal)

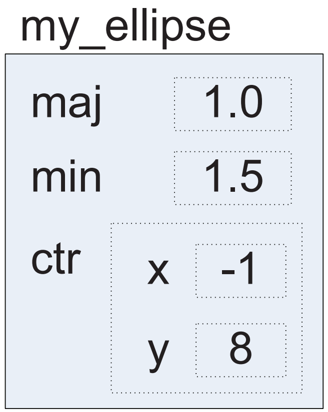
Languages can be “safe”; programs can be “correct”

# Telling libcrunch about allocation functions

```
LIBALLOCS_ALLOC_FNS="xcalloc(zZ)p xmalloc(Z)p xrealloc(pZ)p"
LIBALLOCS_SUBALLOC_FNS="ggc_alloc(Z)p ggc_alloc_cleared(Z)p"
export LIBALLOCS_ALLOC_FNS
export LIBALLOCS_SUBALLOC_FNS
```

# __is_a, containment...

Pointer $p$ might satisfy __is_a($p$, $T$) for $T_0$, $T_1$, ...




```
struct ellipse {
  double maj;
  double min;
  struct point {
    double x, y;
  } ctr;
}
```

- &my_ellipse “is” ellipse and double
- &my_ellipse.ctr “is” point and double
- a.k.a. containment-based “subtyping”

→ libcrunch implements __is_a() appropriately...

# Other solved problems

Structure "subtyping" via prefixing

- relax to __like_a() check

Opaque types

- relax to __named_a() check

"Open unions" like sockaddr

- __like_a() works for these too

# Remaining awkwards

- alloca
- unions
- varargs
- generic use of non-generic pointers (void**, . . . )
- casts of function pointers *to non-supertypes* (of func's t)

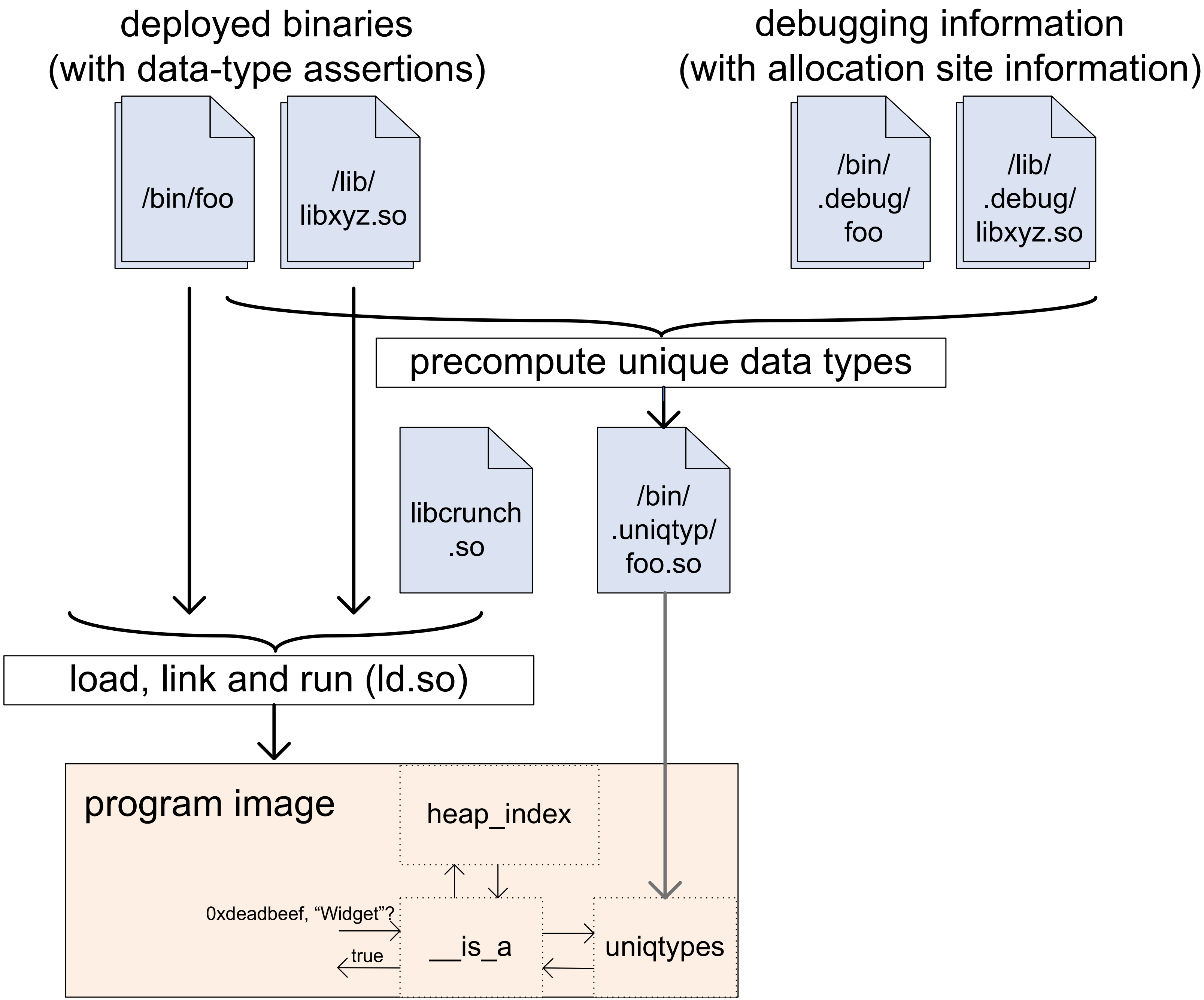
# Remaining awkwards

- alloca
- unions
- varargs
- generic use of non-generic pointers (void**, ...)
- casts of function pointers *to non-supertypes* (of func's t)

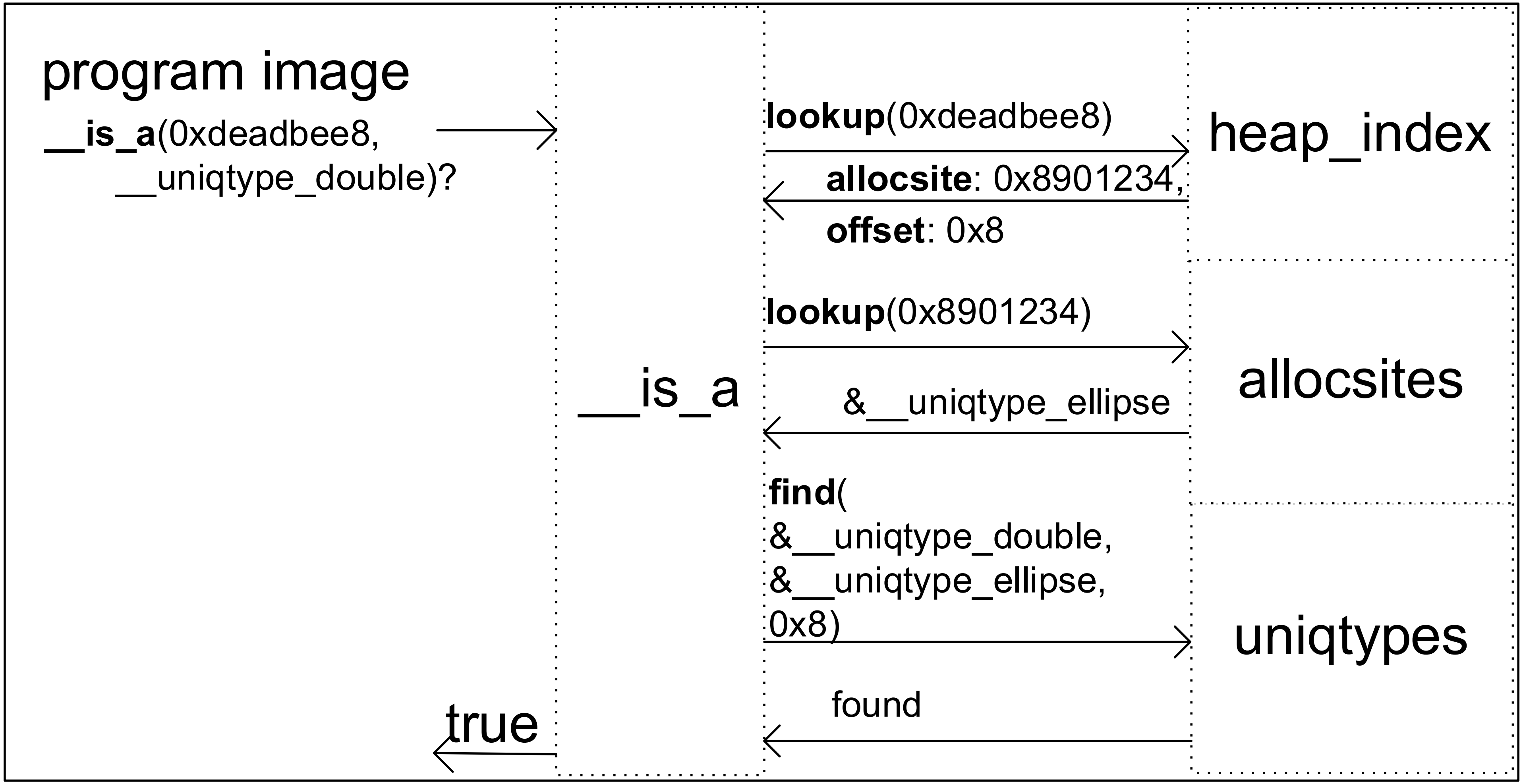
All solved/solvable with some extra instrumentation

- supply our own alloca
- instrument writes to unions
- instrument calls via varargs lvalues; use own va_arg
- instrument writes through void** (check invariant!)
- optionally instr. *all* indirect calls

# Idealised view of libcrunch toolchain



# What happens at run time?



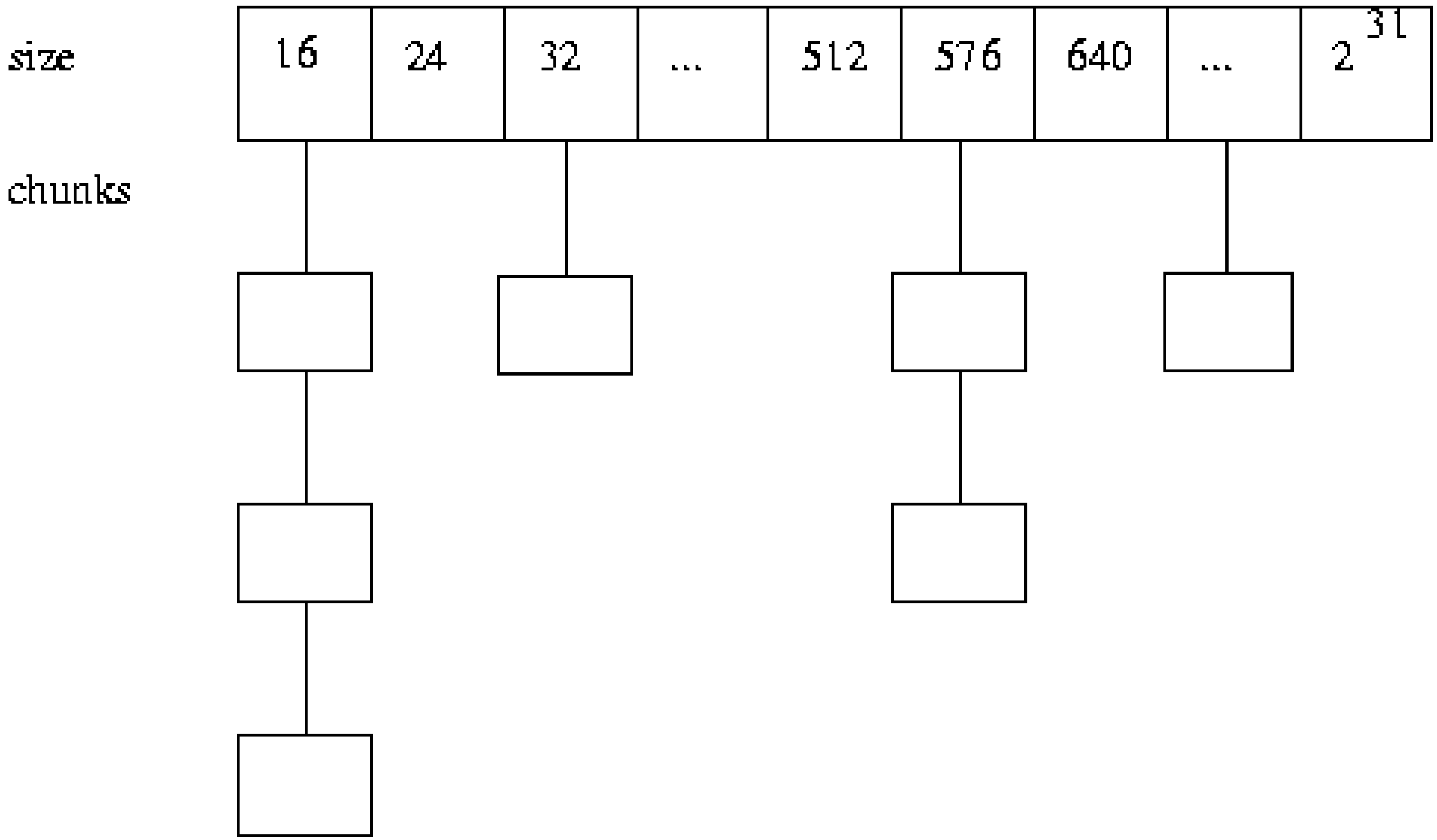
## To solve the heap case. . .

- we'll need some malloc() hooks. . .
- which keep an *index* of the heap
- in a *memtable*
  - efficient *address-keyed* associative map
  - must support (some) range queries
- storing object's metadata

Memtables make aggressive use of virtual memory

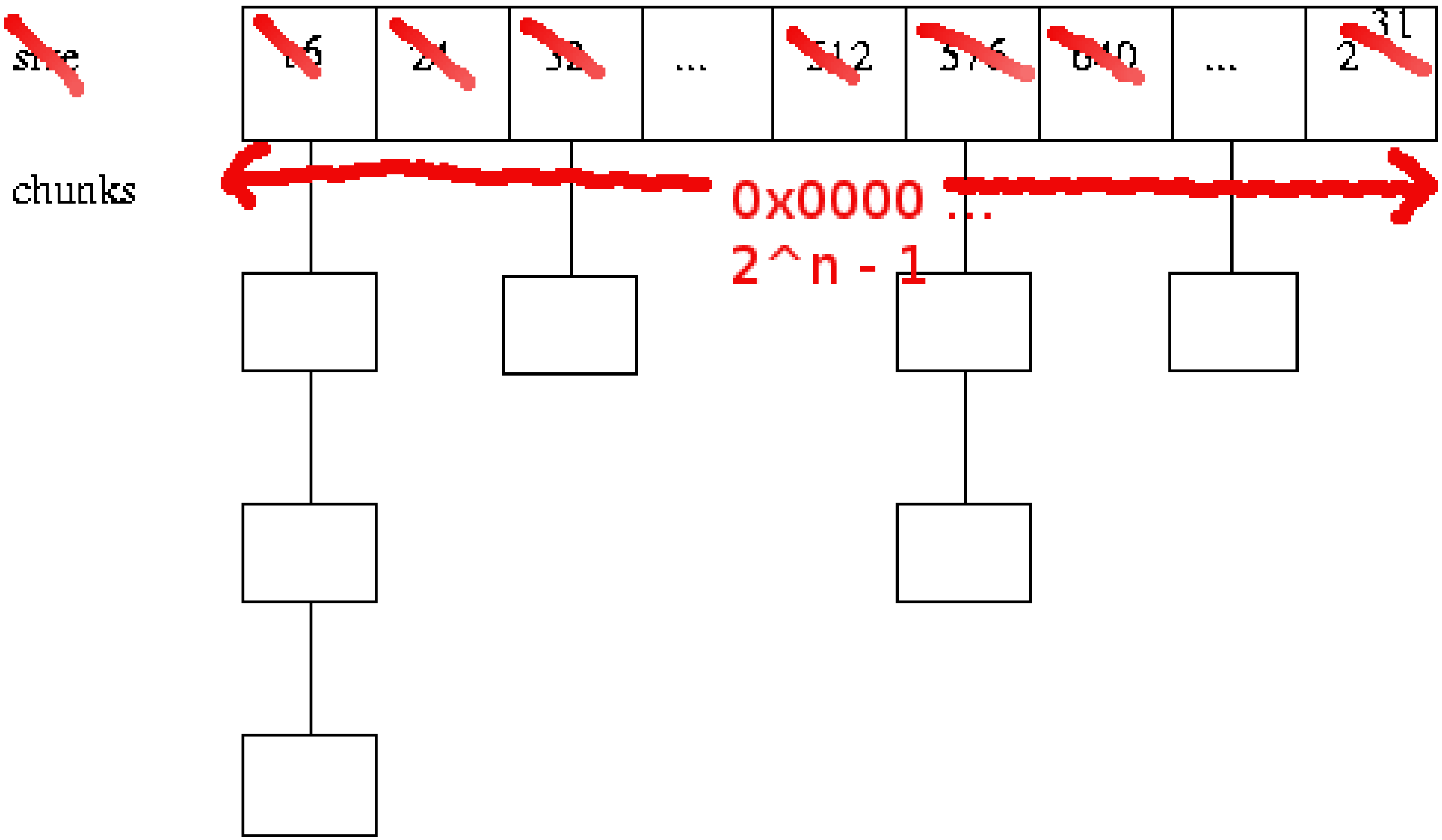
# Indexing heap chunks

Inspired by free chunk binning in Doug Lea's malloc. . .



# Indexing heap chunks

Inspired by free chunk binning in Doug Lea's malloc…




… but index *allocated* chunks binned by *address*

# How many bins?

Each bin is a linked list of heap chunks

- thread next/prev pointers through allocated chunks…
- also store metadata (allocation site address)
- overhead per chunk: one word + two bytes

Finding chunk is $O(n)$ given bin of size $n$

- $\rightarrow$ want bins to be as small as possible
- Q: how many bins can we have?
- A: lots… really, ***lots***!

# Really, how big?

Bin index resembles a linear page table. Exploit

- sparseness of address space usage
- lazy memory commit on "modern OSes" (Linux)


Reasonable tuning for malloc heaps on Intel architectures:

- one bin covers 512 bytes of VAS
- each bin's head pointer takes one byte in the index
- covering $n$-bit AS requires $2^{n-9}$-byte bin index

# Indexing the heap with a memtable is…

- more VAS-efficient than shadow space (SoftBound)
- supports $> 1$ index, unlike placement-based approaches

Memtables are versatile

- buckets don't have to be linked lists
- tunable size / coverage (limit case: bitmap)

We also use memtables to

- index every mapped page in the process ("level 0")
- index "deep" (level 2+) allocations
- index static allocations
- index the stack (map PC to frame uniqtype)

# Link-time interventions

We also interfere with linking:

- link in uniqtypes referred to by each `.o`'s checks
- hook allocation functions
- … distinguishing wrappers from "deep" allocators

Currently provide options in environment variables…

```
LIBCRUNCH_ALLOC_FNS="xcalloc(zZ) xmalloc(Z) xrealloc(pZ) xm
LIBCRUNCH_LAZY_HEAP_TYPES="__PTR_void"
```