Dynamically checking types and bounds with libcrunch

Stephen Kell

stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge

Tool wanted

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

Tool wanted

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

↖ ↗
CHECK this
(at run time)

Tool wanted

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

↖ ↗ CHECK this (at run time)

But also wanted:

- binary-compatible
- source-compatible
- reasonable performance
- avoid being C-specific!*

* mostly…

The user's-eye view

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

The user's-eye view

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

The user's-eye view

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

The user's-eye view

- `$ 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`

The user's-eye view

- `$ 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`

```
struct {int x; float y;} z;
int *x1 =          &z.x;          // ok
int *x2 = (int*)   &z;            // check passes
int *y1 = (int*)   &z.y;          // check fails!
int *y2 =        &((&z.x )[1]);   // use SoftBound
return &z;                        // use CETS
```

How it works for C code, in a nutshell

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

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

How it works for C code, in a nutshell

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

How it works for C code, in a nutshell

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

Want a runtime with the power to

- tracking *allocations*
- with type info
- efficiently
- → fast __is_a() function

The invariant for C

To enforce "all memory accesses respect allocated type":

- every live pointer respects its *contract* (pointee type)
- must also check unsafe loads/stores *not* via pointers
  - unions, varargs

Most contracts are just "points to declared pointee"

- void** and family are subtler (not void*)

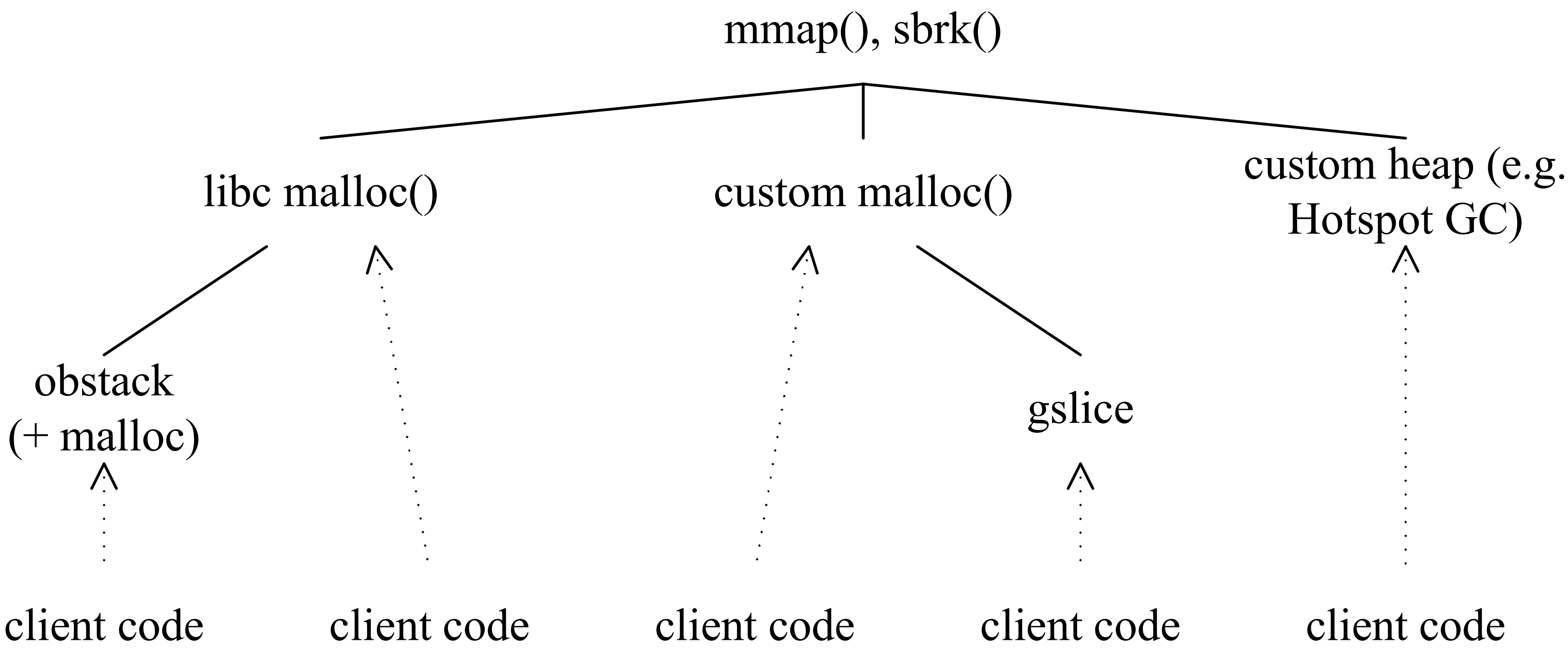
Type info for each allocation

What is an allocation?

- static memory
- stack memory
- heap memory
  - returned by `malloc()` – "level 1" allocation
  - returned by `mmap()` – "level 0" allocation
  - (maybe) memory issued by user allocators...

Runtime keeps *indexes* for each kind of memory...

Hierarchical model of allocations



A small departure from standard C

6 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

6 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:

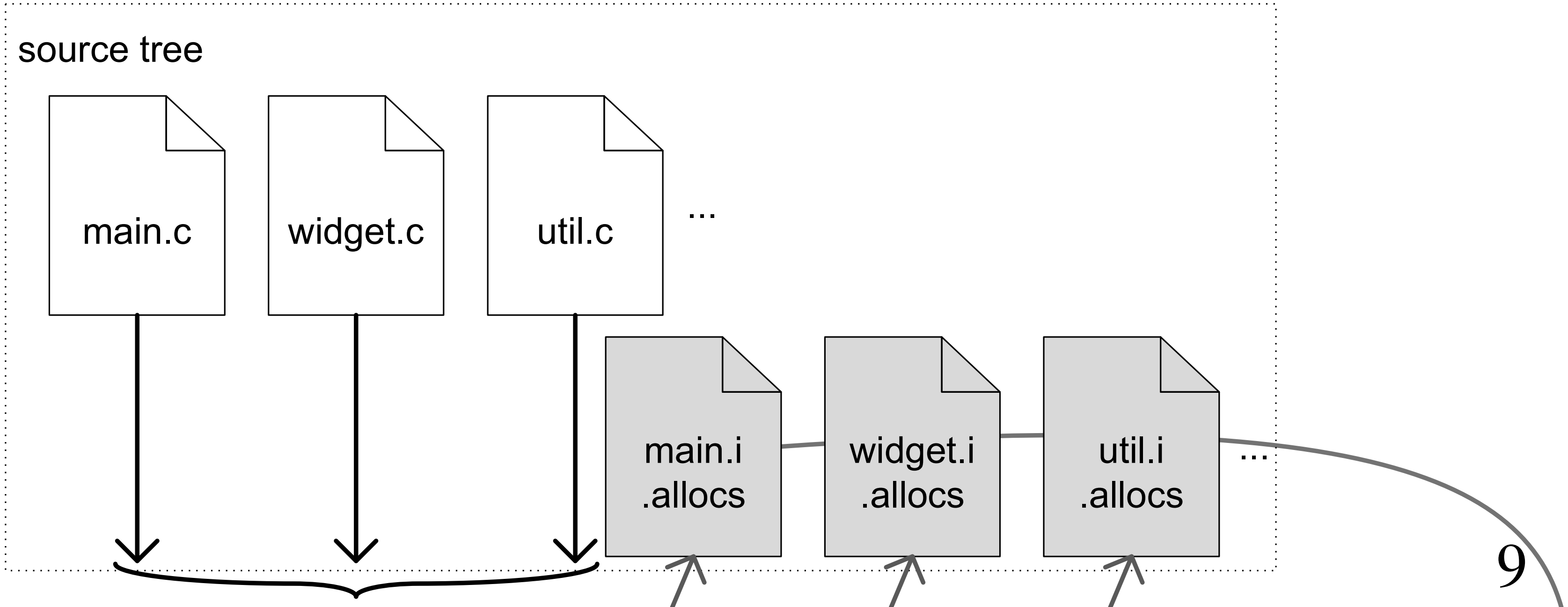
- all allocations have ≤ 1 effective type
- stack, locals / actuals: use declared types
- heap, alloca(): use *allocation site* (+ finesse)
- trap memcpy() and reassign type

What data type is being malloc()'d?

- … infer from use of sizeof
- dump *typed allocation sites* from compiler

Inference: intraprocedural "sizeofness" analysis

- e.g. size_t sz = sizeof (struct Foo); /* ... */; malloc(sz);
- some subties: e.g. malloc(sizeof (Blah) + n * sizeof (Foo))



Challenges

- typed stack storage
- typed heap storage
- support custom heap allocators
- support nested heap allocators
- fast run-time metadata
- robustness to basic C idiom e.g. integer $\leftrightarrow$ pointer
- polymorphic allocation sites (e.g. `sizeof (void*)`)
- subtler C features (function pointers, varargs, unions)
- understanding the invariant ("no bad pointers, *if*...")
- relating to C standard

Performance data: C-language SPEC CPU2006 benchmarks

bench	normal/s	crunch %	nopreload	onlymeta
bzip2	4.95	+6.8%	+1.4%	+2.6%
gcc	0.983	+160 %	– %	+14.9%
gobmk	14.6	+11 %	+2.0%	+4.1%
h264ref	10.1	+3.9%	+2.9%	+0.9%
hmmer	2.16	+8.3%	+3.7%	+3.7%
lbm	3.42	+9.6%	+1.7%	+2.0%
mcf	2.48	+12 %	(−0.5%)	+3.6%
milc	8.78	+38 %	+5.4%	+0.5%
sjeng	3.33	+1.5%	(−1.3%)	+2.4%
sphinx3	1.60	+13 %	+0.0%	+8.7%
perlbench				

State of play

- libcrunch is now pretty good at run-time type checking
- supports idiomatic C, source- and binary-compatibly
- *does not check memory correctness*

State of play

- libcrunch is now pretty good at run-time type checking
- supports idiomatic C, source- and binary-compatibly
- *does not check memory correctness*

```
struct {int x; float y;} z;
int *x1 =          &z.x;          // ok
int *x2 = (int*)   &z;            // check passes
int *y1 = (int*)   &z.y;          // check fails !
int *y2 =        &((&z.x )[1]);   // use SoftBound
return &z;                        // use CETS
```

State of play

- libcrunch is now pretty good at run-time type checking
- supports idiomatic C, source- and binary-compatibly
- *does not check memory correctness*

```
struct {int x; float y;} z;
int *x1 =          &z.x;          // ok
int *x2 = (int*)   &z;            // check passes
int *y1 = (int*)   &z.y;          // check fails !
int *y2 =        &((&z.x )[1]);   // ***
return &z;                        // use CETS
```

Plenty of existing tools do bounds checking

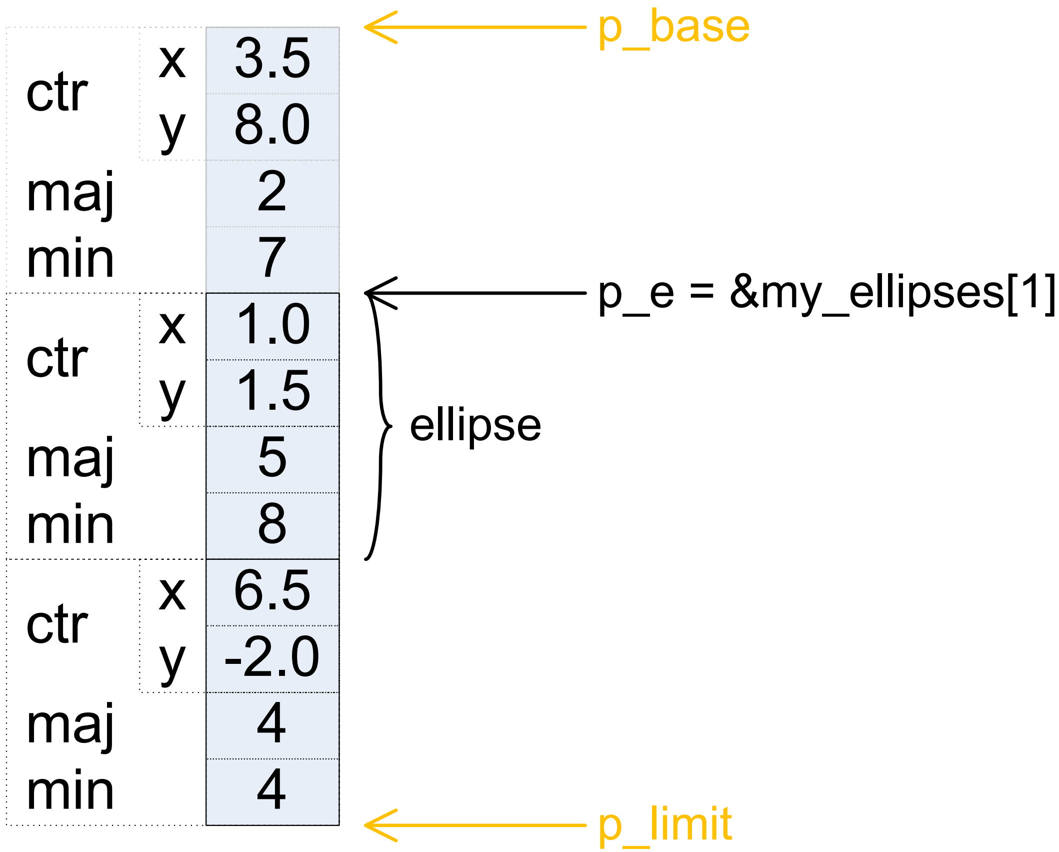
Memcheck (coarse), ASan (fine-ish), SoftBound (fine) . . .

- detect out-of-bounds pointer/array use
- first two also catch some temporal errors
- can run under libcrunch and [then] . . .

Problems remaining:

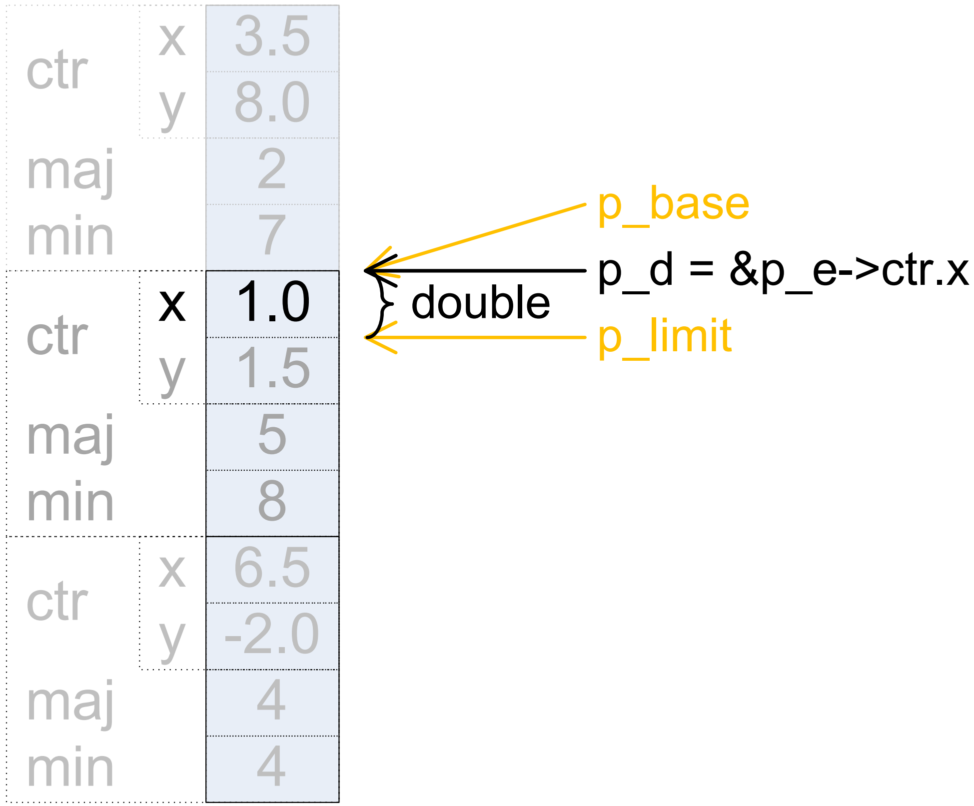
- overhead at best 50–100% (ASan & SoftBound)
- problems mixing uninstrumented code (libraries)
- *false positives for some idiomatic code!*

Existing bounds checkers use per-pointer metadata



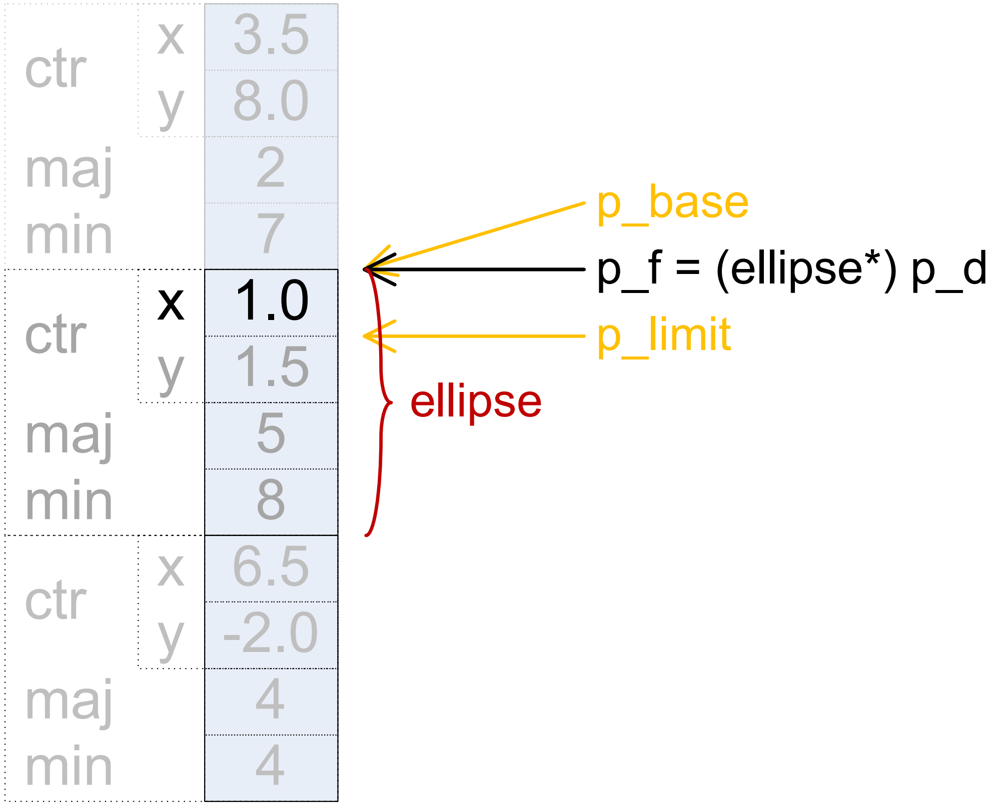

```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Existing bounds checkers use per-pointer metadata



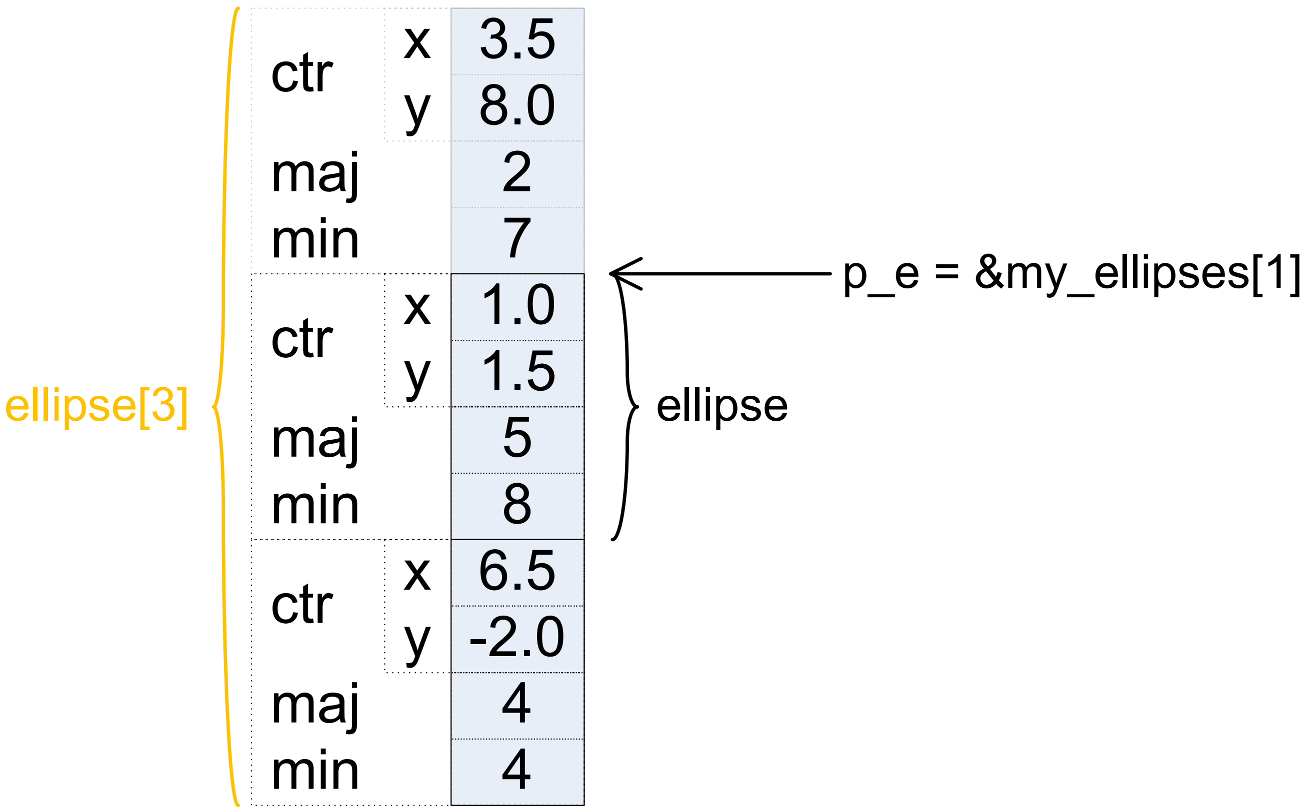

```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Without type information, pointer bounds lose precision



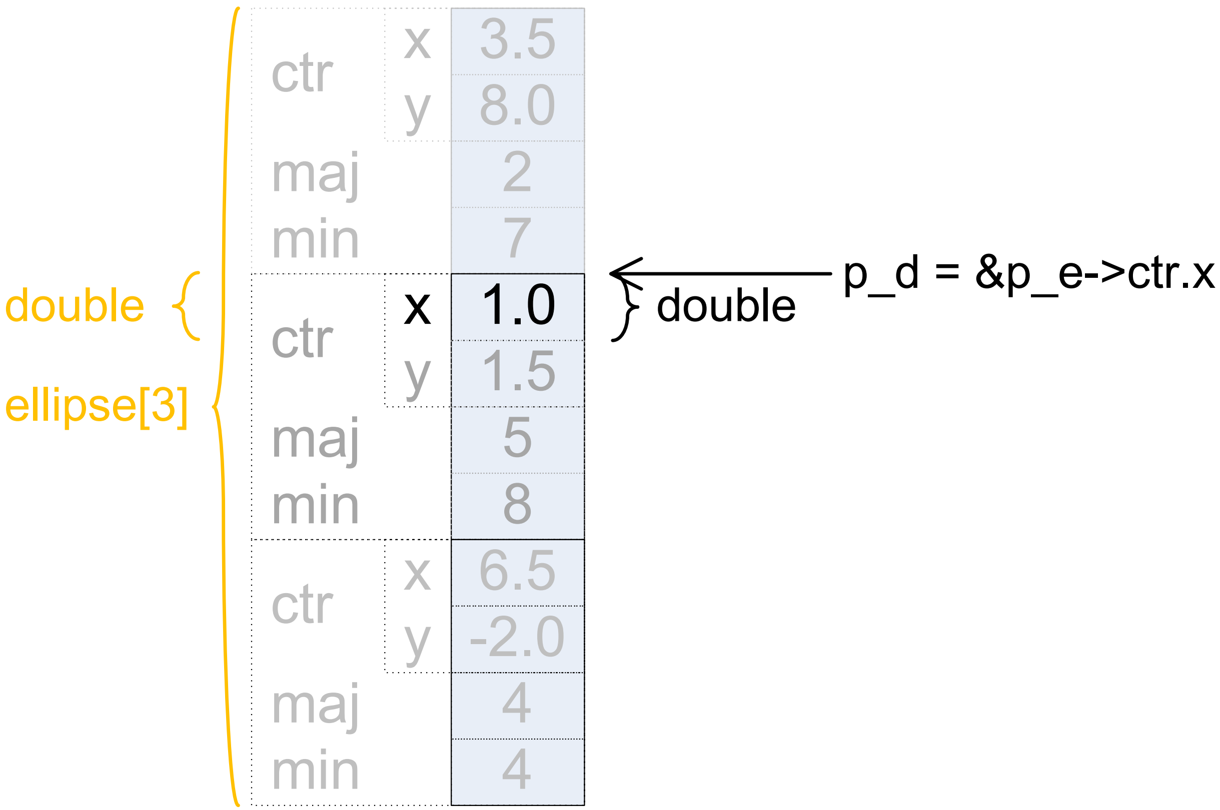

```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Given allocation type and pointer type, bounds are implicit



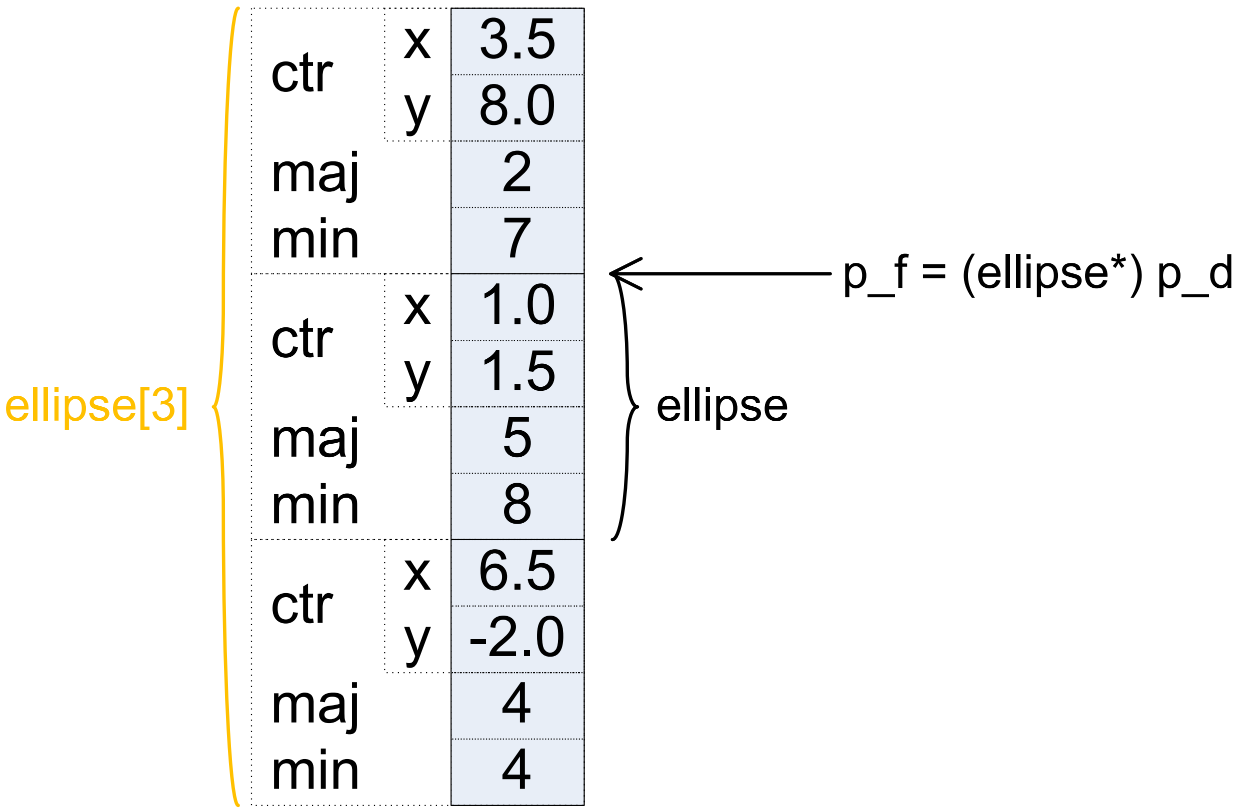

```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

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 (when bounds-checking)

```c
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 (when bounds-checking)

```
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 is oblivious to casts, even though 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!

Idea

Write a bounds-checker consuming per-allocation metadata

- avoid these false positives
- avoid libc wrappers, ...
- robust to uninstrumented callers/callees
- performance?

Making it fast:

- cache bounds: make pointers "locally fat, globally thin"
- only check *derivation*, not *use*

```
inline int __check_derive_ptr(const void **p_derived,
    const void *derivedfrom, struct uniqtype *t,
    __libcrunch_bounds_t *opt_derivedfrom_bounds);
```

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

On x86-64, use noncanonical addresses as trap reps

(ask me!)

Status of the bounds checking extension

Does it work?

- yes! . . . modulo a few bugs right now
- several to-dos to make it fast (caching)

How fast will it be?

- no idea yet, but hopeful it can be competitive (or. . .)
- checks per-derive less frequent than per-deref

Extra ingredients for a *safe* implementation of C$-\epsilon$

- check union access
- check variadic calls
- always initialize pointers
- protect {code, pointers} from writes through char*
- check memcpy(), realloc(), etc..
- allocate address-taken locals on heap not stack
- add a GC (improve on Boehm)

Code remaining unsafe:

- *reflection* (e.g. stack walkers)

Surprisingly perhaps, allocators are not inherently unsafe

Conclusions

- libcrunch tracks per-allocation types
- checking casts is the "obvious" application
- good basis properties for checking bounds too!

Hypothesis: *unsafety* is a property of C implementations

- most code can do without inherently unsafe features
- "fast enough, safe enough" impl. should be doable

Thanks for your attention. Questions?

Memory-correctness vs type-correctness

Related properties checked by existing tools

- spatial m-c – bounds (SoftBound, Asan)
- temporal_1 m-c – use-after-free (CETS, Asan)
- temporal_2 m-c – initializedness (Memcheck, Msan)
- oblivious to data types!

Slow!

- metadata per {value, pointer}
- check on use

Memory-correctness vs type-correctness

Related properties checked by existing tools

- spatial m-c – bounds (SoftBound, Asan)
- temporal_1 m-c – use-after-free (CETS, Asan)
- temporal_2 m-c – initializedness (Memcheck, Msan)
- oblivious to data types!

~~Slow!~~ Faster:

- metadata per ~~{value, pointer}~~ allocation
- check on ~~use~~ create

```
// a check over object metadata... guards creation of the pointer
(assert(__is_a(obj, "struct␣commit")), (struct commit *)obj)
```

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”

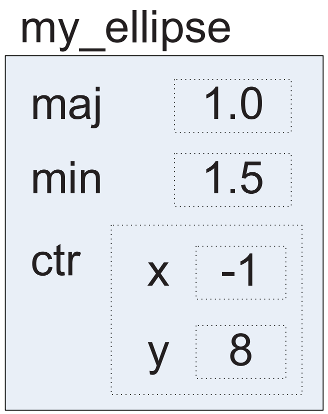
- instantiate = allocate
- concerns storage
- “correct”: reads and writes respect allocated data type
- cf. *memory*-correct (spatial, temporal)

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
```

Non-difficulties




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

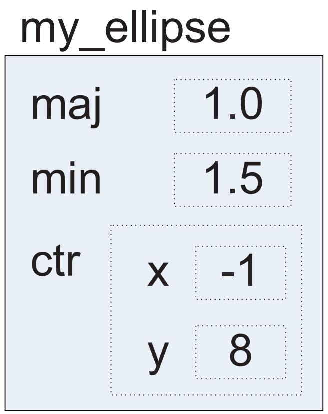
- function pointers (most of the time)
- void pointers, char pointers
- integer ↔ pointer casts
- custom allocators, memory pools etc.

Give up on:

- address-taken union members
- non-procedurally abstracted object allocation/re-use

__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)

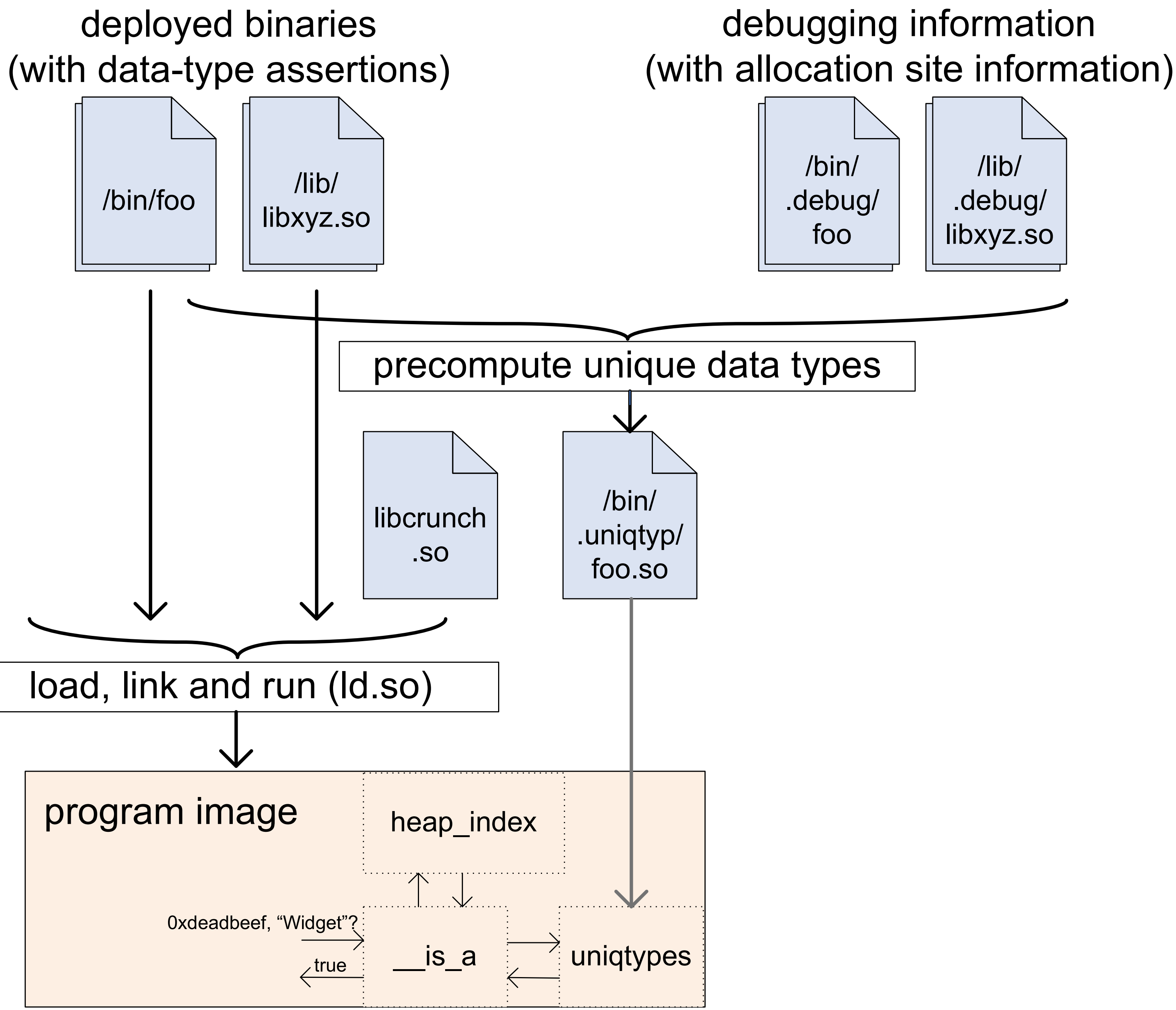
Remaining awkwards

- 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



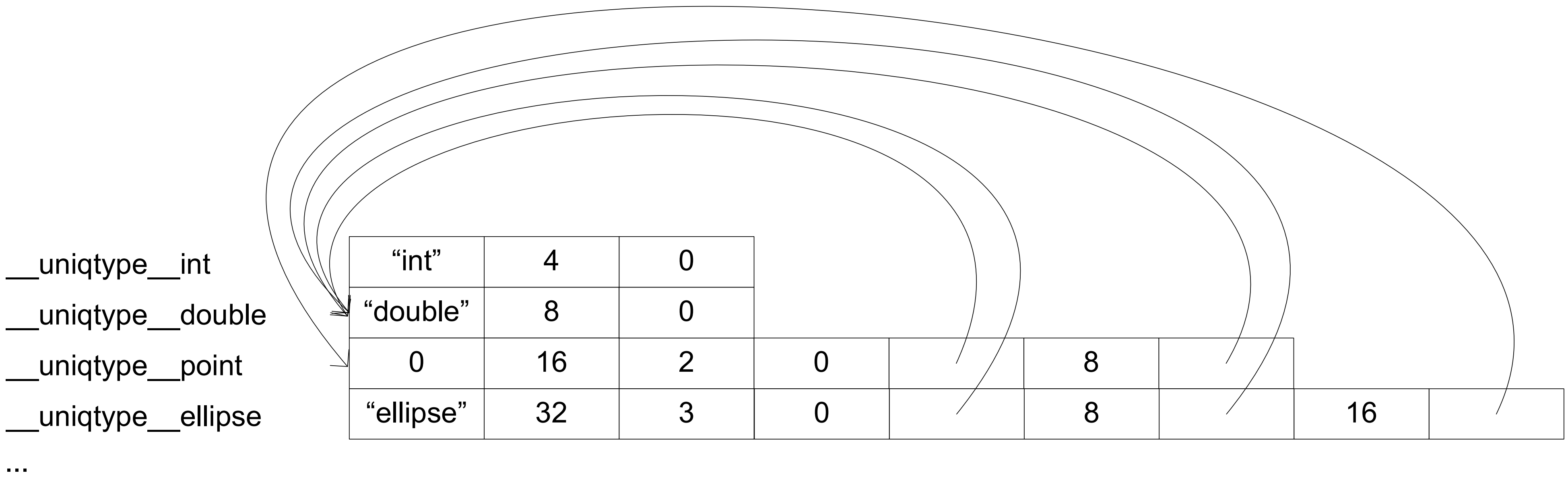
A model of data types: DWARF debugging info

```
$ 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
```

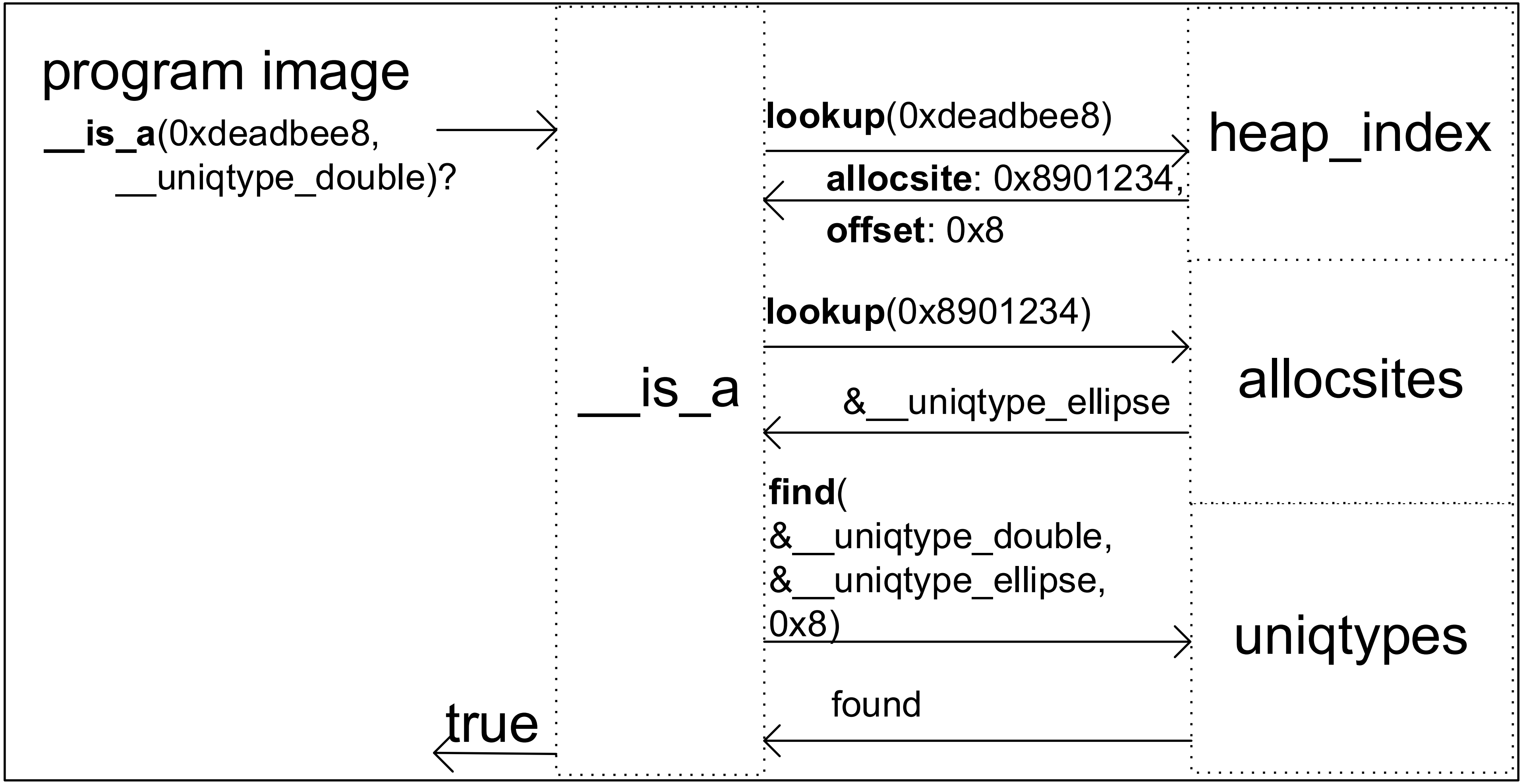
Representation of data types

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



- use the linker to keep them unique
- → “exact type” test is a pointer comparison
- __is_a() is a short search

What happens at run time?



Getting from objects to their metadata

Recall: binary & source compatibility requirements

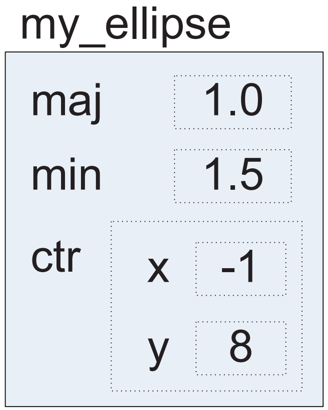
- can't embed metadata into objects
- can't change pointer representation
- $\rightarrow$ need out-of-band ("disjoint") metadata

Pointers can point anywhere inside an object

- which may be stack-, static- or heap-allocated

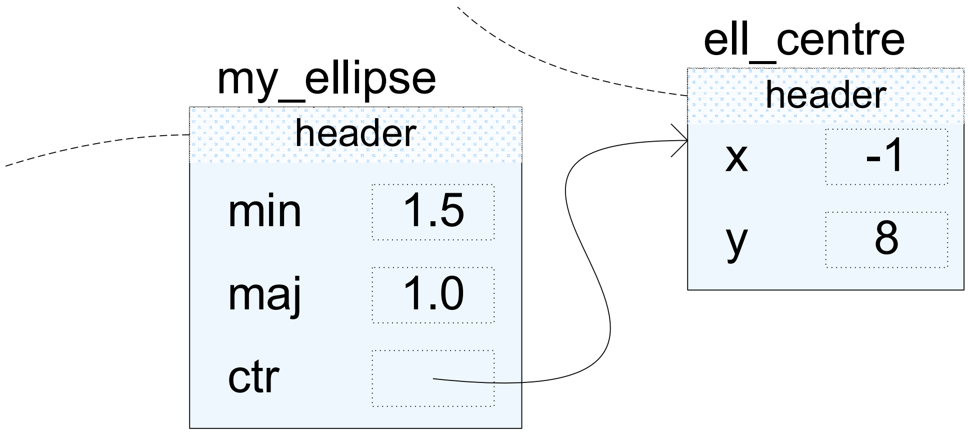
Why the heap case is difficult, cf. virtual machine heaps

Native objects are trees; no descriptive headers!




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

VM-style objects: “no interior pointers”



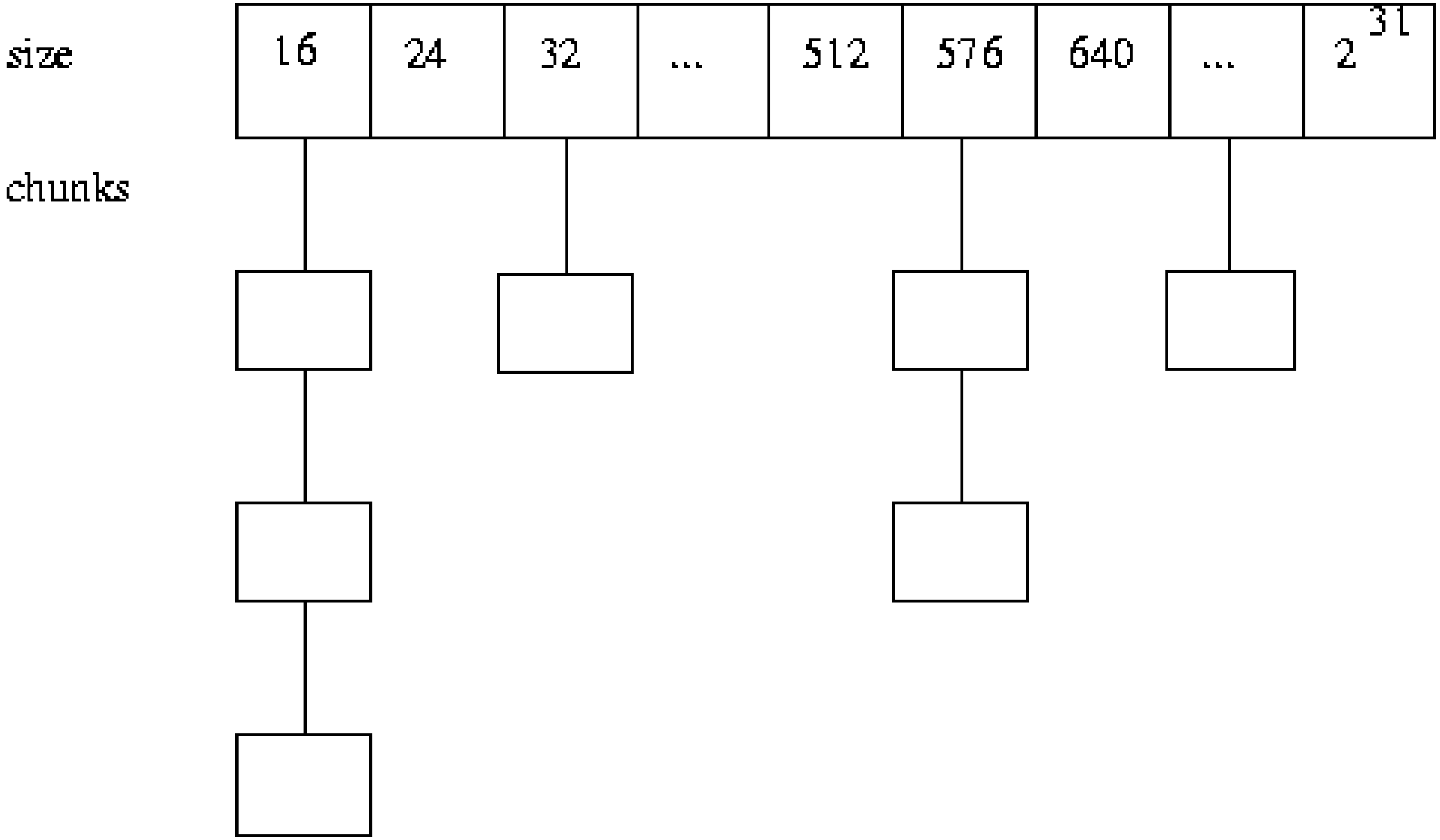
To solve the heap case…

- 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

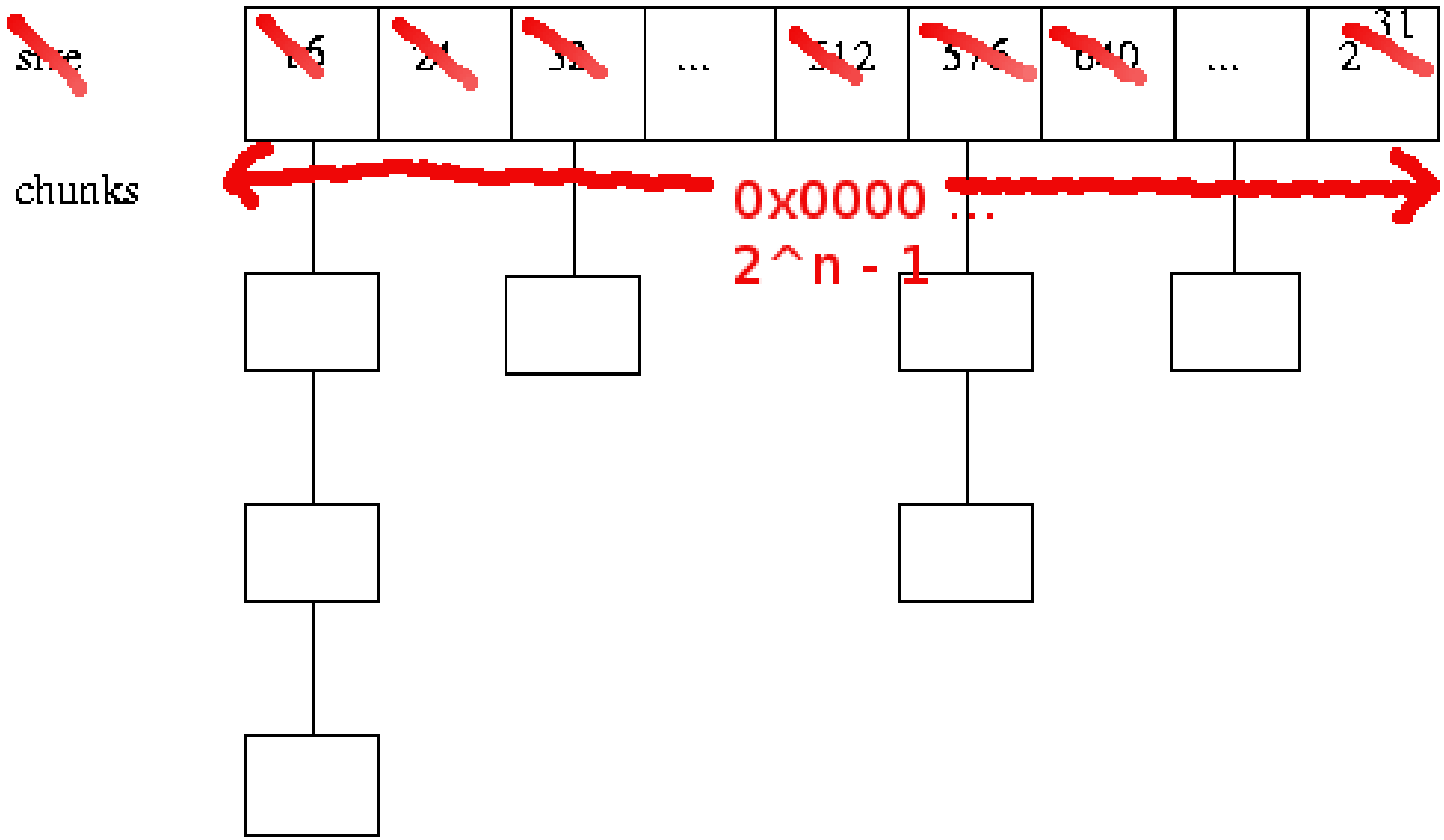
Indexing heap chunks

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?

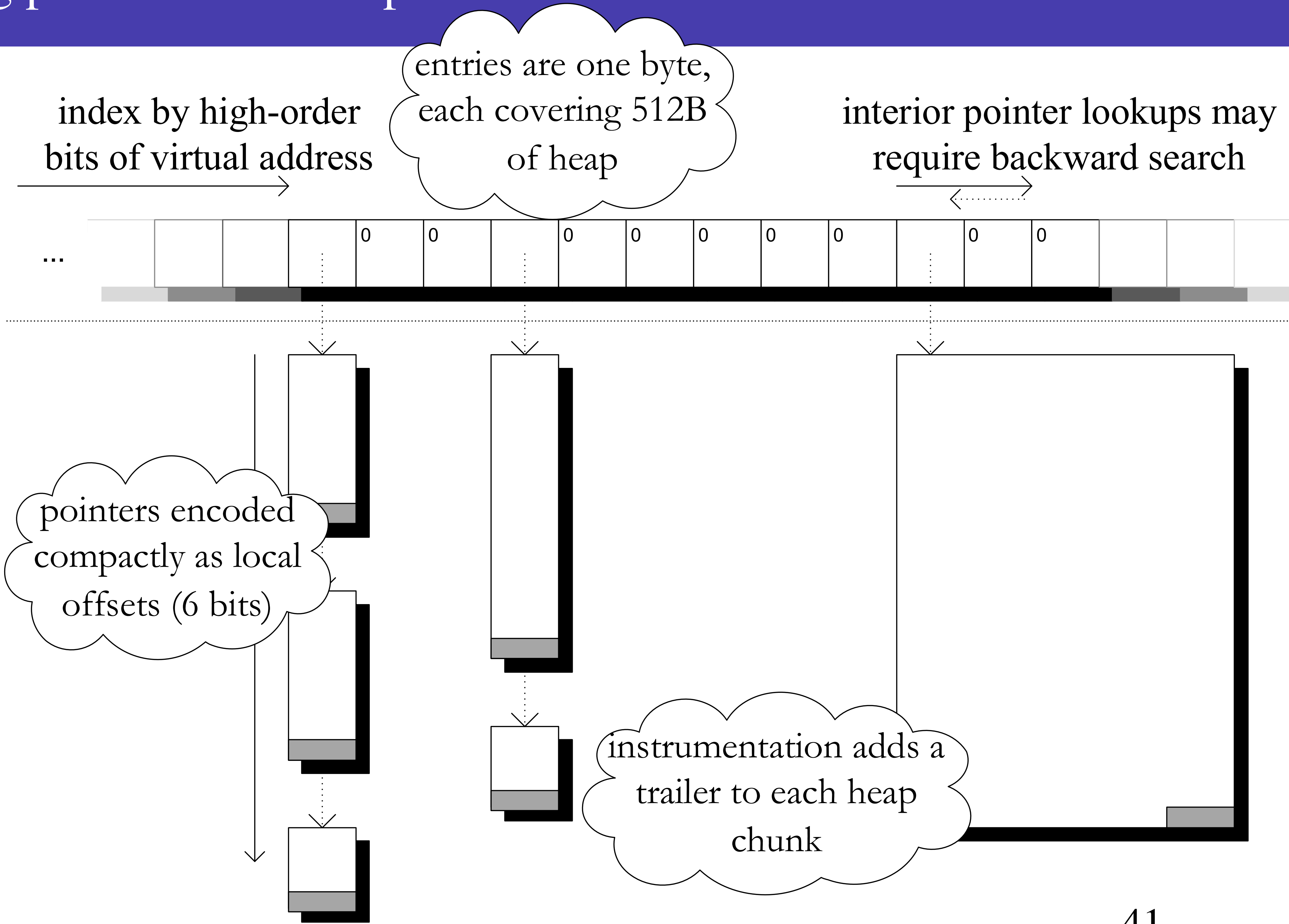
Bin index resembles a linear page table. Exploit

- 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

Big picture of our heap memtable



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)

Other flavours of check

__is_a is a nominal check, but we can also write

- __like_a – "structural" (unwrap one level)
- __refines – padded open unions (à la sockaddr)
- __named_a – opaque workaround

. . . or invent your own!

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) 
LIBCRUNCH_LAZY_HEAP_TYPES="__PTR_void"
```