Adding run-time type information to the GNU toolchain and glibc

Stephen Kell

stephen.kell@cl.cam.ac.uk

Computer Laboratory

University of Cambridge

How it all started: “tool wanted”

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

How it all started: “tool wanted”

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

CHECK this

(at run time)

How it all started: “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, where possible
- build general-purpose infrastructure, where possible

Outlines of this talk

I've "done" it!

- published research papers, given talks, ...

Here to find out from you:

- is there a {will, way} to tech-transfer it?

Will cover:

- a case for run-time type info as a general facility
- overview of my implementation
- steps towards improving and integrating the code

Please interrupt with questions!

A sketch of how to do it

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

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

A sketch of how to do it

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

A sketch of how to do it

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

Must augment toolchain + runtime with power to

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

A research prototype

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

A research prototype

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

A research prototype

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

A research prototype

- `$ 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`

A research prototype

- `$ 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`

Naming note:

- liballocs + allocscc: the generic part
- libcrunch + crunchcc: C type-checking specifically
- various support libraries have other names

What do I mean by "run-time type information"?

Roughly same content as DWARF type entries...

- ... but available at run time, efficiently

+ query API to access it:

- e.g. "what's on the end of this pointer?"
- ... for any *allocation* in a process's address space

It's mostly *not*

- replacement e.g. for C++ typeinfo (but...)
- for specifying higher-order behaviours (but...)

Let's see some applications (besides crunchcc)...

Precise debugging

```
(gdb) print obj
$1 = (const void *) 0x6b4880 # unknown type!
```

Precise debugging

```
(gdb) print obj
$1 = (const void *) 0x6b4880 # unknown type!
(gdb) print __liballocs_get_alloc_type (obj)
$2 = (struct uniqtype *) 0x2b3aac997630
 <__uniqtype___InputParameters>
```

Precise debugging

```
(gdb) print obj
$1 = (void *) 0x6b4880
(gdb) print __liballocs_get_alloc_type (obj)
$2 = (struct uniqtype *) 0x2b3aac997630
 <__uniqtype___InputParameters>
(gdb) print *(struct InputParameters *) $2
$3 = {ProfileIDC = 0, LevelIDC = 0, no_frames = 0,
    ... }
```

Better debugger integration is desirable...

- note how types exist as symbols in the inferior...
  - (more later)
- ... but gdb doesn't grok the connection

Scripting without FFI

```
$ ./node
```

Scripting without FFI

```
$ ./node   # <-- ... with liballocs extensions
> process.lm.printf("Hello, world!\n")
Hello, world!
14
```

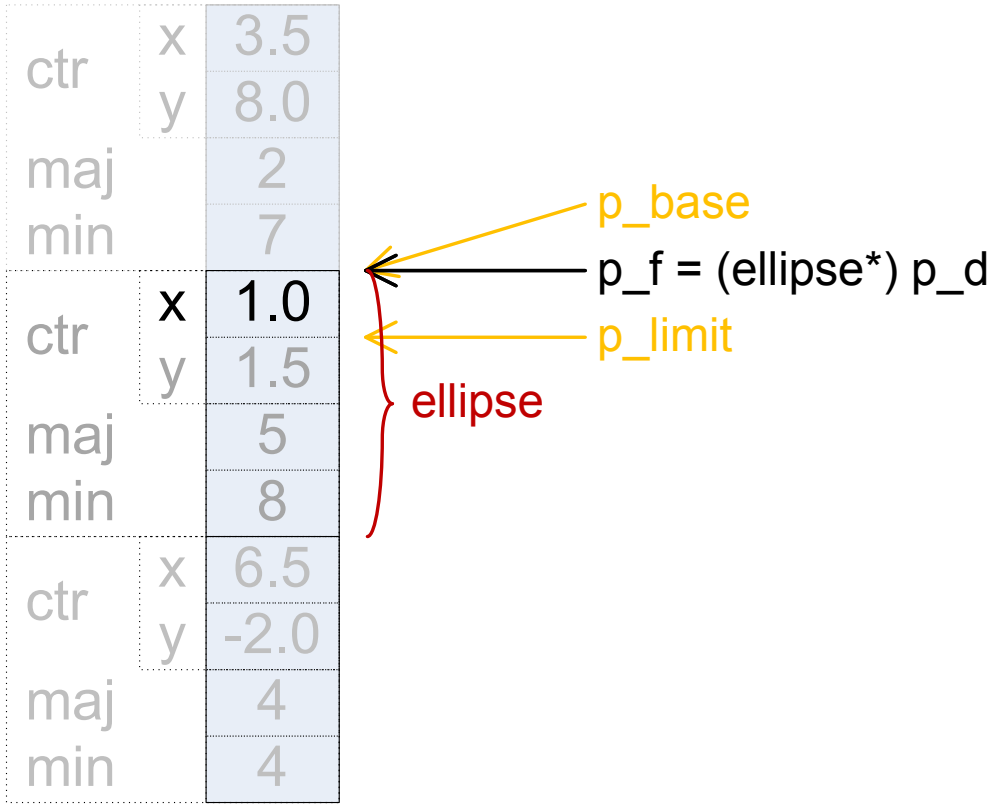
Scripting without FFI

```
$ ./node # with liballocs extensions
> process.lm.printf("Hello, world!\n")
Hello, world!
14
> require('-lXt');
```

Scripting without FFI

```
$ ./node # with liballocs extensions
> process.lm.printf("Hello, world!\n")
Hello, world!
14
> require('-lXt')
> var toplvl = process.lm.XtInitialize(
    process.argv[0], "simple", null, 0,
    [process.argv.length], process.argv);
  var cmd = process.lm.XtCreateManagedWidget(
    "exit", commandWidgetClass, toplvl, null, 0);
  process.lm.XtAddCallback(
    cmd, XtNcallback, process.lm.exit, null);
  process.lm.XtRealizeWidget(toplvl);
  process.lm.XtMainLoop();
```

Non-tyrannical bounds checking




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

More exotic stuff

- memory-mapped files with type info
- checking ABI type info for shared-memory objects
- checking ABIs at dynamic load time
- run-time metaprogramming in C / C++
- better garbage collection?
- fast & flexible DSU system?
- . . . your idea here!

Sounds nice; how does it work?

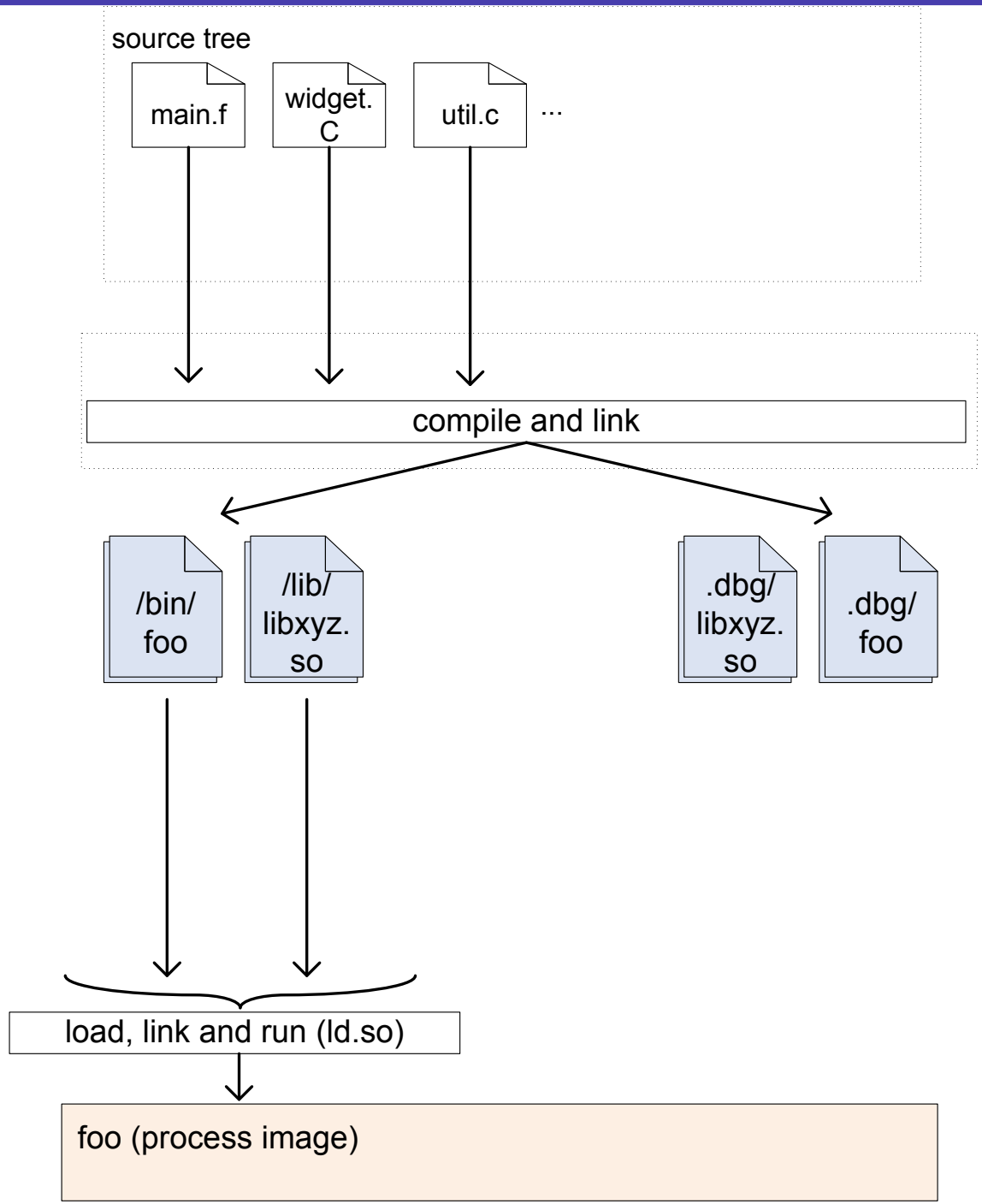
Key design point: *separable*, optional

- minimal overheads if not used
- can easily skip / turn off
- a bit like Dwarf debug info

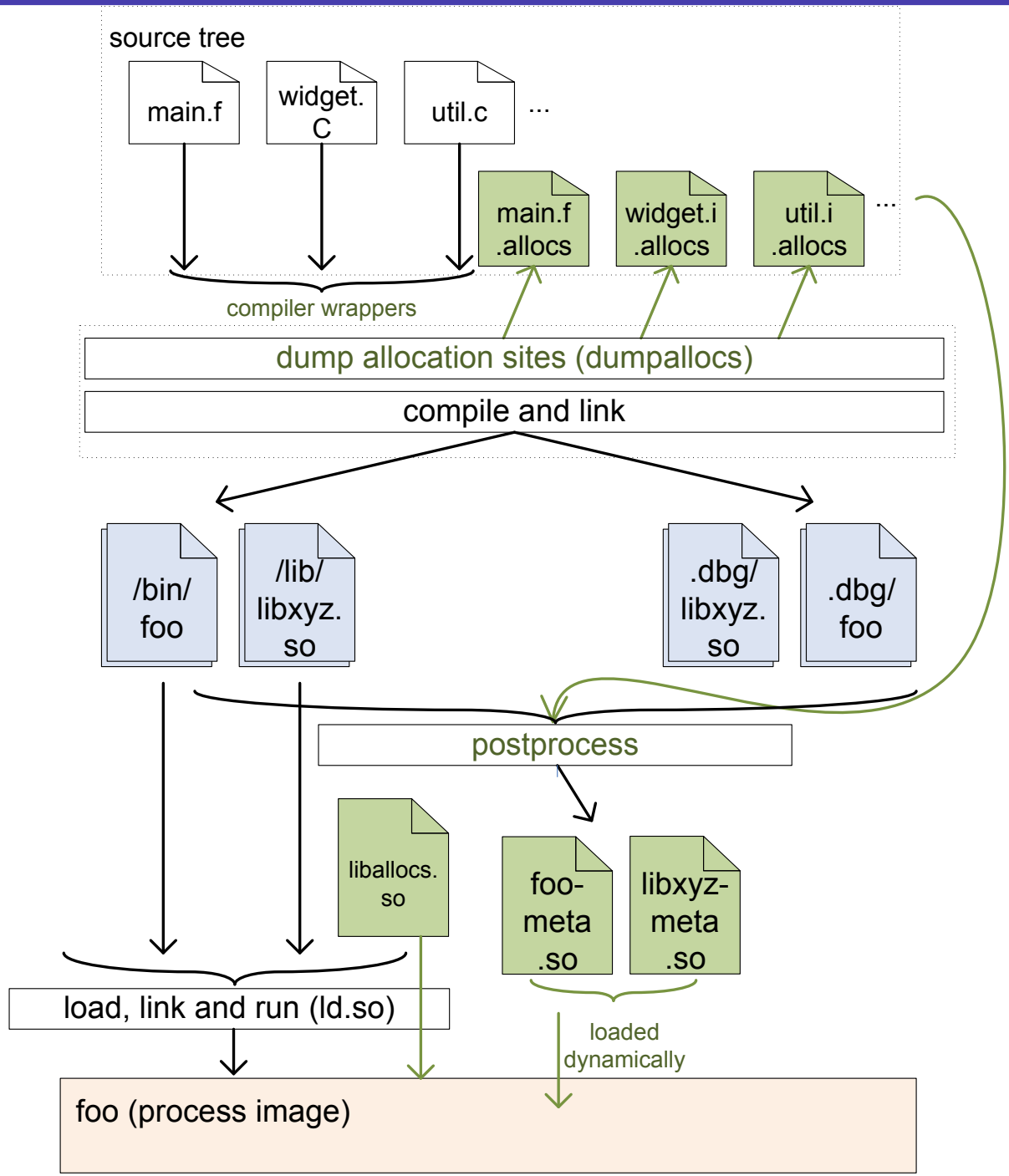
Three different "implementation states" in mind

- prototype (what works now)
- mostly sane, mostly out-of-tree ("in progress")
- fully integrated in glibc and gcc ("eventually"?)

Unmodified toolchain



Augmented toolchain



Key design points

Taken care to be *separable* / *optional*

- a bit like DWARF debug info
- can easily skip / strip / turn off type info
- minimal run-time overheads if not used

Taken care to be ABI-compatible

- no changes to layouts of anything
- only corner-case interventions at compile and link
- freely mix code built with/without extended toolchain

Key additions to toolchain and runtime

At/before compile time

- allocation site analysis + generate metadata
- tweak compiler options, mess with alloca(), . . .

At link time

- hook allocator functions
- generate deduplicated type info (mostly from DWARF)

At run time

- hook loader events → load metadata
- hook allocation events
- answer queries (e.g. “is this cast okay?”)

Problem 1: what type is being malloc()'d?

Use intraprocedural "sizeofness" analysis

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

Sizeofness propagates, a bit like dimensional analysis.

Problem 1: what 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))
```

Problem 1: what type is being malloc()'d?

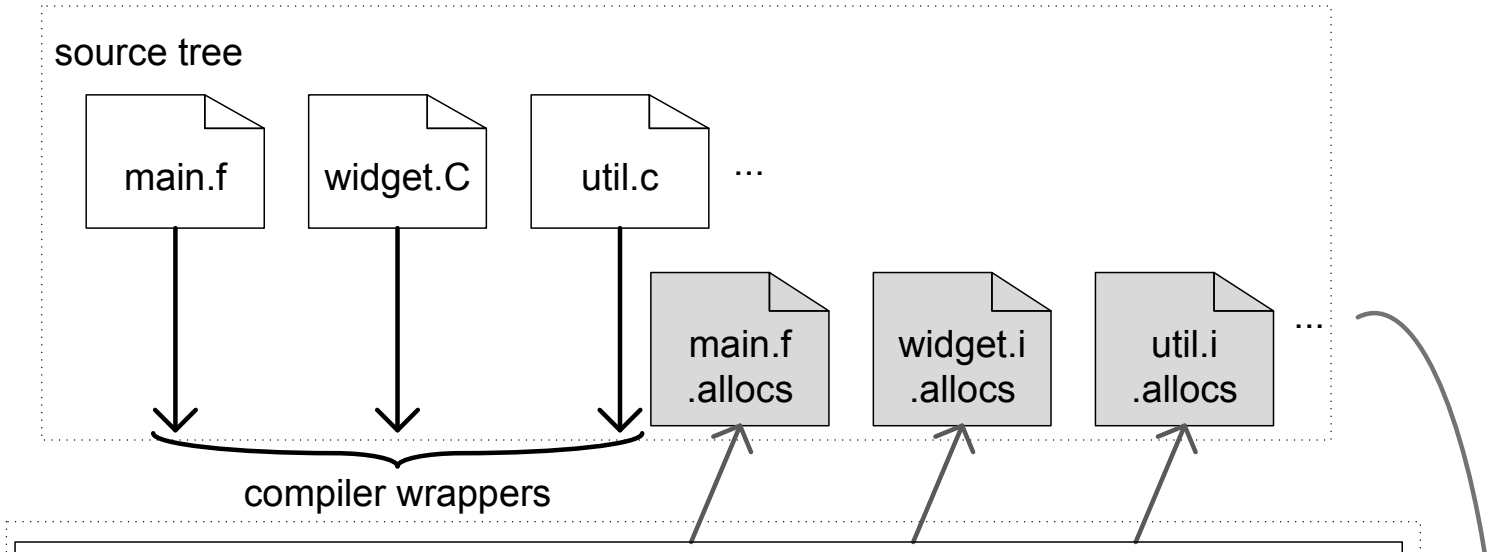
Use intraprocedural "sizeofness" analysis

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

Sizeofness propagates, a bit like dimensional analysis.

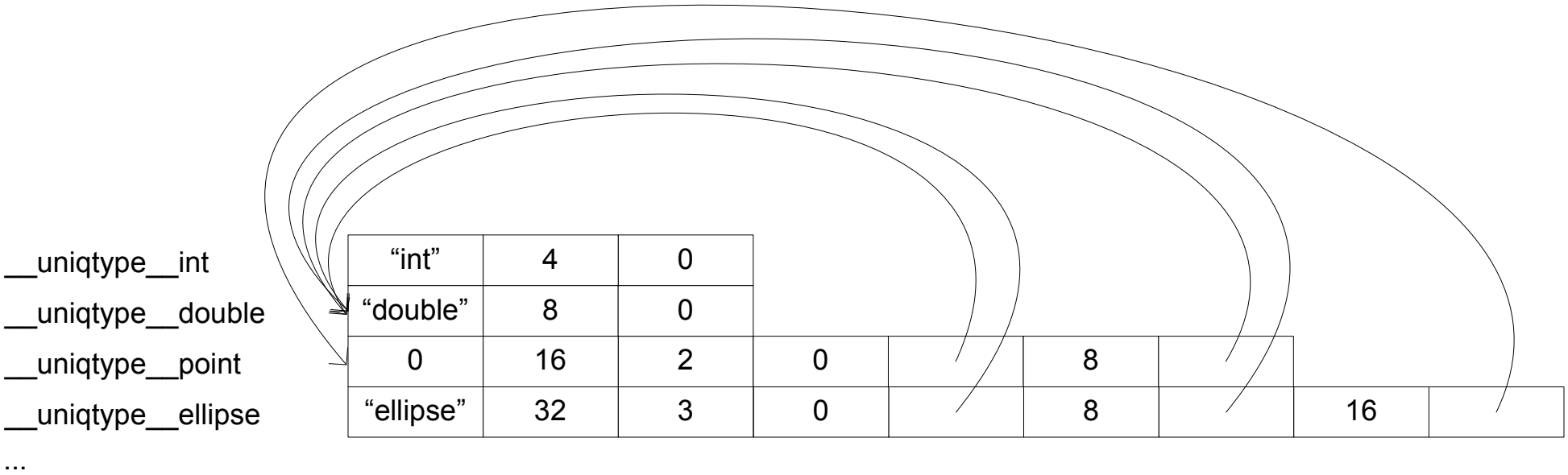
```
malloc(sizeof (Blah) + n * sizeof (struct Foo))
```

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



Problem 2: what should type info look like at run time?

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

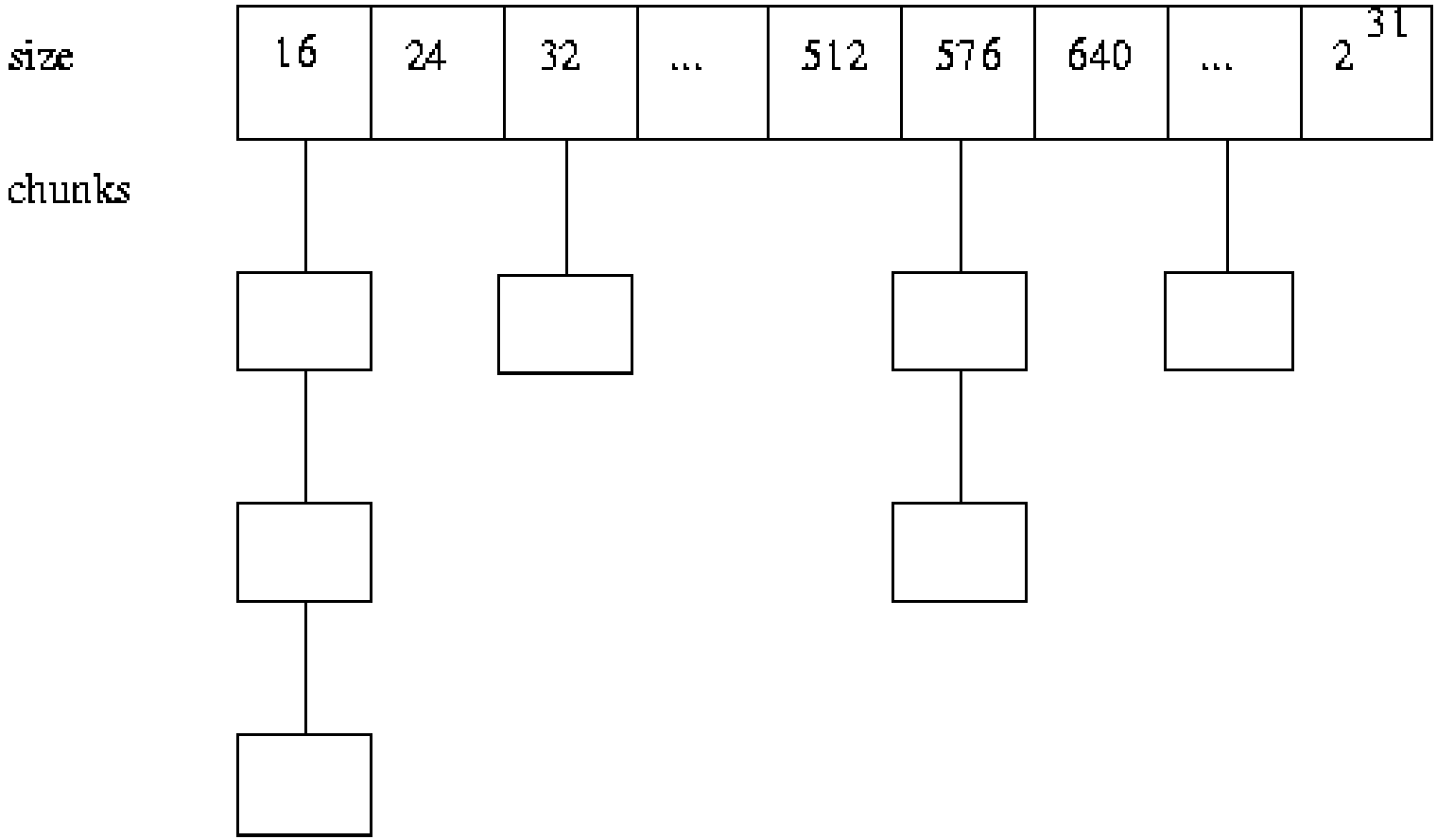


+ many cases not shown (functions, unions, named fields. . .)

- types are COMDAT'd globals → uniqued at link time
- "hash code" to distinguish aliased defs

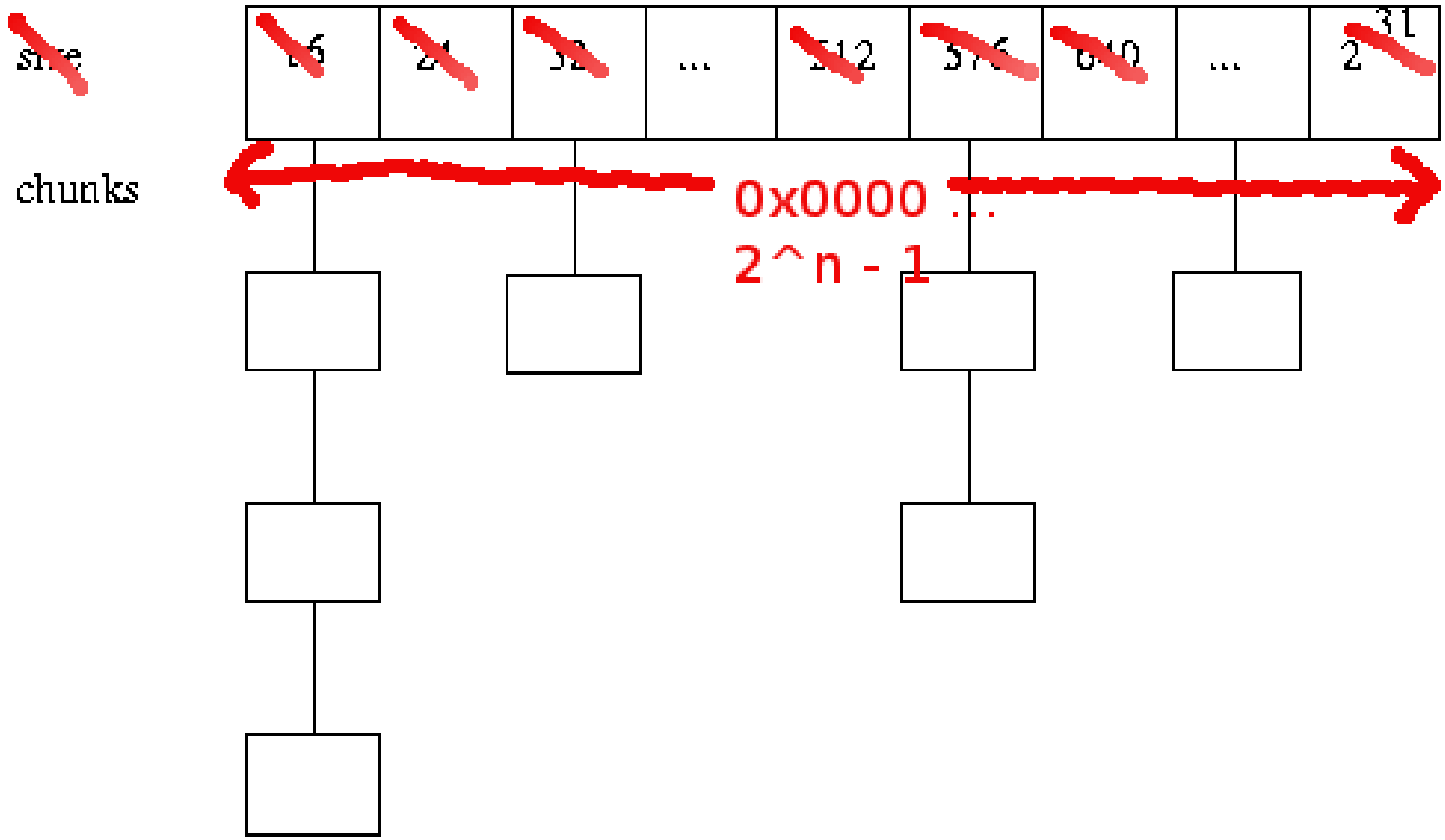
Problem 3: querying the malloc heap

- each malloc chunk gets one word of metadata
- track chunks: any range-queryable associative structure



Problem 3: querying the malloc heap

- each malloc chunk gets one word of metadata
- track chunks: any range-queryable associative structure



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

Problem 3: querying the malloc heap

- each malloc chunk gets one word of metadata
- track chunks: any range-queryable associative structure

… huge linear lookup in virtual memory, mostly unmapped

Problem 4: stack frames + stack walking

Stack frames get uniqtypes much like structs/unions

- via non-trivial DWARF postprocessing
- different uniqtypes for different vaddr ranges
- run-time lookup maps vaddr $\rightarrow$ frame uniqtype

Walking the stack

- can use libunwind
- usually faster to turn on frame pointers

Problem 4: custom allocators

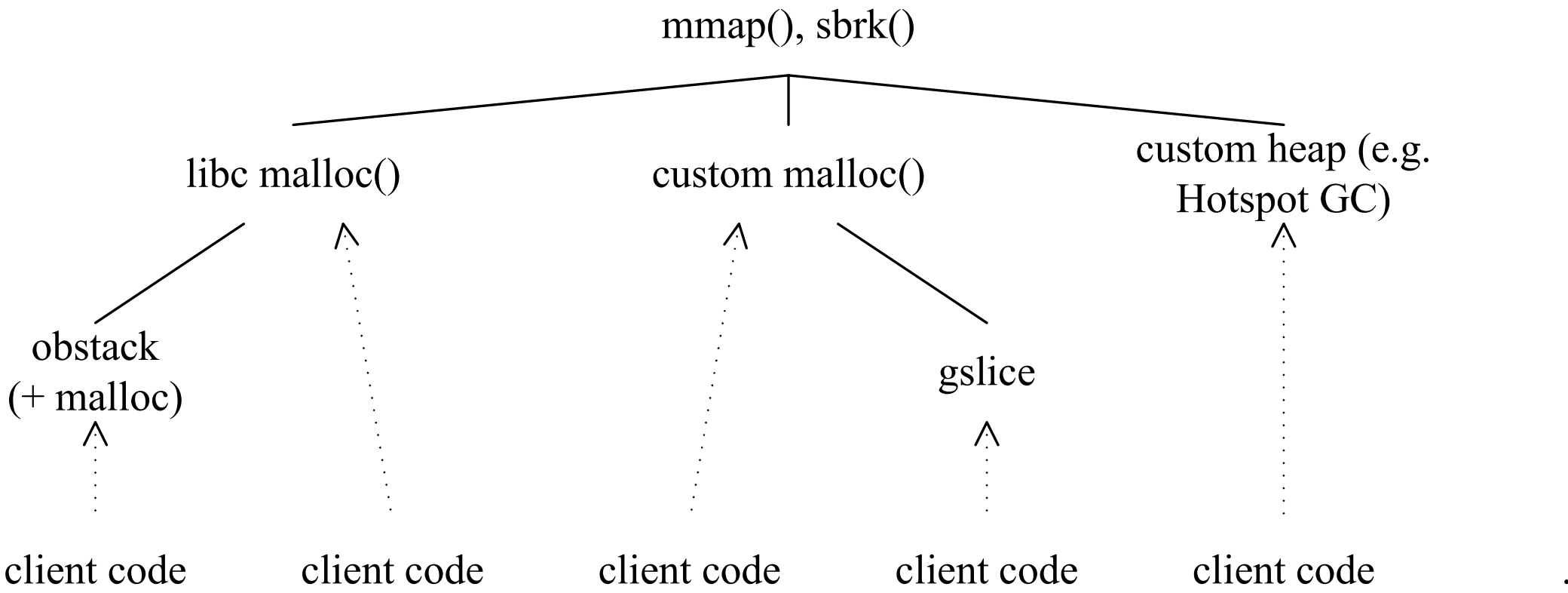
Superficial solution: “tell me your allocation functions”

```
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
```

Deep solution: “it’s all allocators, man”

- run-time model of *allocators*
- includes mmap, static, stack, auxv, alloca, . . .
- query interface is a “meta-allocation protocol”

Allocation hierarchy



Meta-level protocol (roughly)

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

Each allocator has a vtable-like structure of these calls

- top-level API dispatches to “deepest allocator”

Problem 5: hooking mmap()

Necessary for robust tracking of memory-mapped regions

- overriding libc's mmap misses a lot
- mmap table is perf-critical → must be up-to-date

Solution is hairy: a trap-and-emulate layer (libsystrap)

- rewrite syscall instrs that "might do mmap()"
  - … as ud2, on Intel
- do the mmap() in SIGILL handler
- update metadata

Overkill? But has proved useful also e.g. in bounds checker

Performance numbers from SPEC CPU2006

bench	normal/s	liballocs/s	liballocs %	no-load
bzip2	4.91	5.05	+2.9%	+1.6%
gcc	0.985	1.85	+88 %	– %
gobmk	14.2	14.6	+2.8%	+0.7%
h264ref	10.1	10.6	+5.0%	+5.0%
hmmer	2.09	2.27	+8.6%	+6.7%
lbm	2.10	2.12	+0.9%	(−0.5%)
mcf	2.36	2.35	(−0.4%)	(−1.7%)
milc	8.54	8.29	(−3.0%)	+0.4%
perlbench	3.57	4.39	+23 %	+1.6%
sjeng	3.22	3.24	+0.6%	(−0.7%)
sphinx3	1.54	1.66	+7.7%	(−1.3%)

Some remaining problems

- slow for custom non-malloc()-like allocators
- build slowdown
- limited support for C++ or other languages
- occasional CIL bugs/omissions
- must rebuild!
  - even though work is mostly metadata-gathering
  - (… if no custom allocators, no alloca())
- only Linux/x86-64 runtime for now
  - some FreeBSD code…
- quite a few ugly hairy hacks
  - to avoid modifying gcc, ld, glibc, ld-linux.so, …

Where next? Recall stages:

1. "current" prototype
   - build via wrapper scripts + helpers
   - source passes using CIL
   - preloadable runtime
2. mostly sane, mostly out-of-tree
   - still use CIL; tiny wrapper (gcc -B/path/to/it)
   - other helper logic in gold plugin
   - still preload; fix worst uglinesses (patched glibc...)
3. fully integrated
   - source-level stuff in gcc
   - runtime stuff integrated in glibc (somehow)

Currently working towards 2; some thoughts on 3.

What uglinesses, you ask?

- separate .i.allocs files, not DW_TAG_alloc_site
- dwarfidl to deal with funky malloc() "types"
- "allocation functions link differently"
- hooking malloc() et al. in glibc
- hooking libdl functions (for meta-object loading)
- trap-and-emulate to catch mmap() et al.
- libdlbind: API for dynamically creating DSOs(!)
- hacks for getting at program headers, auxv, . . .
- /usr/lib/meta hierarchy only (or. . .)
- reentrancy avoidance measures (e.g. fake_dlsym())
- . . . probably others I'm forgetting

Selected uglinesses (1): allocation functions link differently

```
LIBALLOCS_ALLOC_FNS="default_bzalloc(pIi)p"
```

Selected uglinesses (1): allocation functions link differently

`LIBALLOCS_ALLOC_FNS="default_bzalloc(pIi)p"`

. . . means compiler wrapper will link with

- `--wrap default_bzalloc`
- and generate *caller* wrapper to latch the caller address

Selected uglinesses (1): allocation functions link differently

`LIBALLOCS_ALLOC_FNS="default_bzalloc(pIi)p"`

. . . means compiler wrapper will link with

- `--wrap default_bzalloc`
- and generate *caller* wrapper to latch the caller address
- . . . passed to *callee* hook in preloaded calloc()

Selected uglinesses (1): allocation functions link differently

`LIBALLOCS_ALLOC_FNS="default_bzalloc(pIi)p"`

. . . means compiler wrapper will link with

- `--wrap default_bzalloc`
- and generate *caller* wrapper to latch the caller address
- . . . passed to *callee* hook in preloaded calloc()

Oh, but default_bzalloc is static so `--wrap` is no-op

- globalize it via `objcopy`

Selected uglinesses (1): allocation functions link differently

`LIBALLOCS_ALLOC_FNS="default_bzalloc(pIi)p"`

. . . means compiler wrapper will link with

- `--wrap default_bzalloc`
- and generate *caller* wrapper to latch the caller address
- . . . passed to *callee* hook in preloaded calloc()

Oh, but default_bzalloc is static so `--wrap` is no-op

- globalize it via `objcopy`
- avoid intra-section calls: `-ffunction-sections`
- "unbind" intra-CU calls: via hacked `objcopy`

Selected uglinesses (1): allocation functions link differently

`LIBALLOCS_ALLOC_FNS="default_bzalloc(pIi)p"`
. . . means compiler wrapper will link with

- `--wrap default_bzalloc`
- and generate *caller* wrapper to latch the caller address
- . . . passed to *callee* hook in preloaded calloc()

Oh, but default_bzalloc is static so `--wrap` is no-op

- globalize it via `objcopy`
- avoid intra-section calls: `-ffunction-sections`
- "unbind" intra-CU calls: via hacked `objcopy`

Allocators in executables. . .

- can't callee-hook using LD_PRELOAD ☹
- want two wrappers! but `--wrap` doesn't compose. . .

Selected uglinesses (2): libdlbind + syscall hackery

Sometimes need to create type info at run time…

- (ask me why, but later)
- want uniformity of linkage, w.r.t. other type info

Selected uglinesses (2): libdlbind + syscall hackery

Sometimes need to create type info at run time...

- (ask me why, but later)
- want uniformity of linkage, w.r.t. other type info

libdlbind: dynamically build an ELF object

```
/* Create a new shared library in this address space. */
void *dlcreate(const char *libname);
/* Allocate a chunk of space in the file . */
void *dlalloc (void *lib , size_t sz, unsigned flags);
/* Create a new symbol binding. */
void *dlbind(void *lib , const char *symname, void *obj, size_t len, Elf64_Word typ
```

Selected uglinesses (2): libdlbind + syscall hackery

Sometimes need to create type info at run time...

- (ask me why, but later)
- want uniformity of linkage, w.r.t. other type info

libdlbind: dynamically build an ELF object

```
/* Create a new shared library in this address space. */
void *dlcreate(const char *libname);
/* Allocate a chunk of space in the file . */
void *dlalloc(void *lib, size_t sz, unsigned flags);
/* Create a new symbol binding. */
void *dlbind(void *lib, const char *symname, void *obj, size_t len, Elf64_Word typ
```

Need dlopen() to MAP_SHARED, not MAP_PRIVATE!

- do it by abusing the syscall trap-and-emulate layer

Selected uglinesses (3): /usr/lib/meta hierarchy

```
$ allocscc -o myprog myprog.c
```

- creates `/usr/lib/meta/path/to/myprog-meta.so`

```
$ mv myprog /another/path/
```

- . . . the metadata is no longer in the right place!

Instead of "separate meta-DSO", want to bundle in myprog

- meta-DSO packaged as non-allocated ELF section
- (yes, ELF file within an ELF file)
- identify with magic ELF phdr in myprog
- load with ld.so monster hackery

90% of a fix: provide dl_open_from_fd?

Fixing the uglinesses

A lot of it comes down to doing *hooks* more/better:

- a better version of ld –wrap
- in-glibc hooks for mmap()? (avoid trap-and-emulate)

Maybe also some ld.so functionality

- auto-loading the meta-DSOs?
- loading from file descriptor?
- dlbind() done sanely?

Also want conventions for metadata

- maybe additional DWARF, e.g. DW_TAG_alloc_site
- meta-DSO formats, filesystem locations, etc..

Would these be useful to anyone else (or am I insane)?

Tentative plan

Code is here: https://github.com/stephenrkell

- liballocs is the main repo
- submodules + contrib/Makefile for dependencies
- following README "should" give clean build

Currently working on "mostly sane, mostly out-of-tree":

- gold plugin to replace compiler wrapper
- speeding up DWARF postprocessing
- Debian packaging everything

Even more tentative plan + conclusions

Could easily work on

- patches to lessen hooking ugliness etc.… if welcome?
- gcc-based source passes? (some Clang work already)
- other progress towards “full integration”

Or perhaps I’m insane for wanting any of this?

- you can be honest!

Thanks for listening!

- code link again: https://github.com/stephenrkell
- my web page: http://www.cl.cam.ac.uk/users/srk31%7esrk31