Dynamically diagnosing type errors in unsafe code

Stephen Kell

stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge

A definition

"… dynamically type-safe [means]
the behavior of any program,
correct or not,
can be easily understood in terms of
the source-level language semantics."

A definition

> "… dynamically type-safe [means]
> the behavior of any program,
> correct or not,
> can be easily understood in terms of
> the source-level language semantics."

—Ungar, Spitz and Ausch, *Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment*

A definition

> "… dynamically type-safe [means]
> the behavior of any program,
> **correct or not,**
> can be easily understood in terms of
> the source-level language semantics."

—Ungar, Spitz and Ausch, *Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment*

A definition

> "… dynamically type-safe [means]
> the behavior of any program,
> **correct or not,**
> can be easily understood in terms of
> the source-level language semantics."

—Ungar, Spitz and Ausch, *Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment*

"Type safety" [at run time] is really about *debugging*!

A definition

> "… dynamically type-safe [means]
> the behavior of any program,
> **correct or not,**
> can be easily understood in terms of
> the source-level language semantics."

—Ungar, Spitz and Ausch, *Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment*

"Type safety" [at run time] is really about *debugging*!

- clean error reports are better than corrupting errors
- … would be nice even in *unsafe languages*, like C

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
- . . . for real, idiomatic code in (say) C
- reasonable performance

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
- . . . for real, idiomatic code in (say) C
- reasonable performance

Enter libcrunch, which does the above.

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`

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

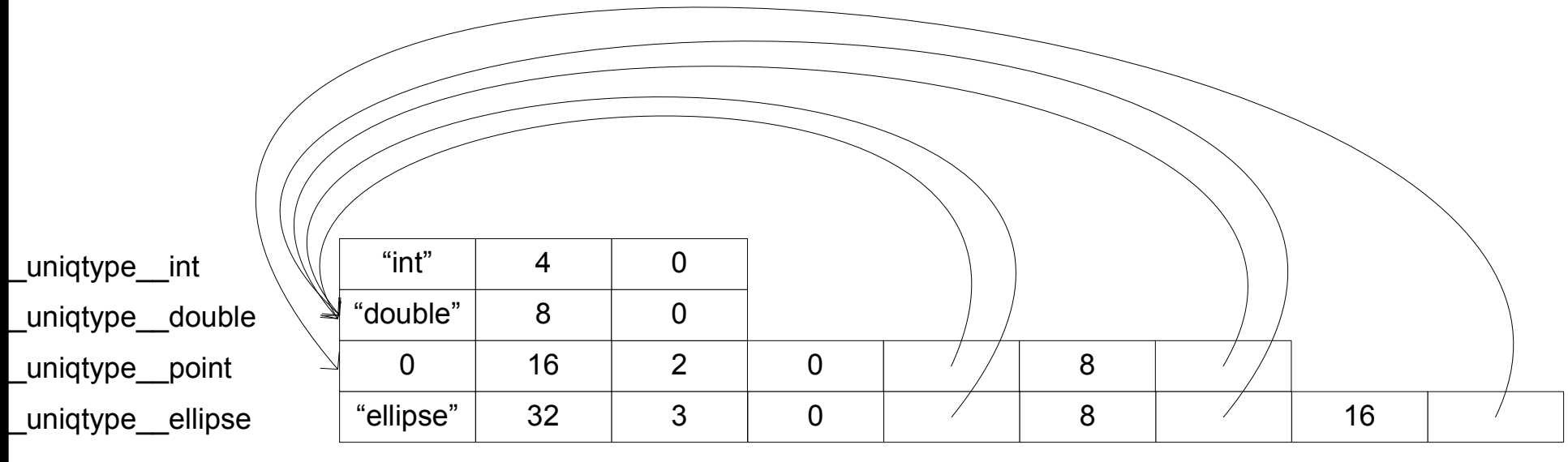
```
if (obj->type == OBJ_COMMIT) {
  if (process_commit(walker,
        (CHECK(__is_a(obj, "struct commit"),
          (struct commit *)obj)))
    return -1;
  return 0;
}
```

Need a runtime which

- provides a fast __is_a() function
- ... and a few other flavours of check
- by efficiently tracking *allocations*
- ... and attaching reified type info

Reified, unique data types (see my Onward! 2015 paper about liballocs)

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



- 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?

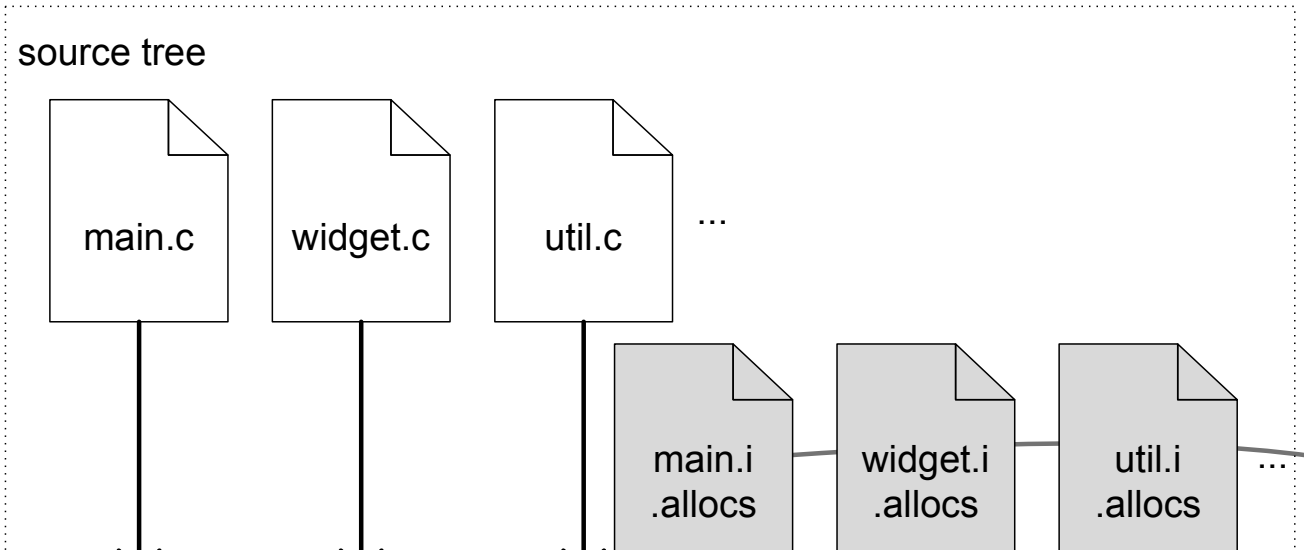
Use intraprocedural "sizeofness" analysis

```
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	3×10^5	14	3
gobmk	0	0	0	0
h264ref	2	27	2	0
hmmer	0	0	0	0
lbm	0	5×10^7	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);}
```

Future work: shopping list for a *safe* implementation of C−ϵ

- check `memcpy()`, `realloc()`, etc..
- add a bounds checker (improve on SoftBound)
- add a GC (precise! improve on Boehm)
- check unions and varargs
- always initialize pointers
- check unsafe writes through `char*`
- safely address-takeable union members (!)

Good prospects for all of the above! (ask me)

Conclusions

Checking pointer casts can be made efficient and helpful

- source- and binary-compatible
- low overhead, convenient to use (e.g. no rebuilds)
- good prospects for extension

Code is here: http://github.com/stephenrkell/libcrunch/

Thanks for your attention. Questions?