Run-time Type Checking in C with Clang and Libcrunch

Chris Diamand
University of Cambridge, now ARM
Stephen Kell
Computer Laboratory, University of Cambridge
David Chisnall
Computer Laboratory, University of Cambridge

Overview

- What is libcrunch?
- Instrumenting casts
- Finding allocation sites
- Runtime
- Performance
- Status and todo
- Conclusion

Run-time type checking

...but C is statically-typed!

Run-time type checking

...but C is statically-typed! ...mostly.

```
my_bar = (struct bar *) some_other_pointer;
my_bar->x = 3;
```

Run-time type checking

...but C is statically-typed! ...mostly.

```
my_bar = (struct bar *) some_other_pointer;
my_bar->x = 3;
```

`my_bar` filled with garbage, but may not find out until later...
What's at x's location?

How to catch this?

Clang sanitizers:

- MemorySanitizer - uninitialised reads
- AddressSanitizer - out-of-bounds, use-after-free
- ThreadSanitizer, UndefinedBehaviourSanitizer

Other tools:

- Compiler warnings
- Valgrind (memcheck)

What is libcrunch?

Framework for tracking and checking *types* at run-time.

```
$ clangcrunchcc -o random random.c ...
$ LD_PRELOAD=/path/to/libcrunch.so ./random

random: Failed check __is_a(0x1bf57f0, 0x6056c0
a.k.a. "stat") at 0x4039f7 (randommain+0x16a5);
obj is 0 bytes into an allocation of a heap
sockaddr (deepest subobject: uint$16 at offset 0)
originating at (nil)
```

How does it work?

How does it work?

- Instrument pointer casts:

```
my_bar = (struct bar *) my_foo;
```

$$\Longrightarrow$$

```
my_bar = (warn_if_not(__is_aU(my_foo,
                              &__uniqtype_bar)),
          (struct bar *) my_foo);
```

How does it work?

- Instrument pointer casts:

```
my_bar = (struct bar *) my_foo;
```

$$\Longrightarrow$$

```
my_bar = (warn_if_not(__is_aU(my_foo,
                             &__uniqtype_bar)),
          (struct bar *) my_foo);
```

- Find and analyse allocation sites:

```
... = malloc(200 * sizeof(int));
```

$$\Longrightarrow$$

```
/path/to/test.c 5 malloc __uniqtype__int
```

How does it work?

- Instrument pointer casts:

```
my_bar = (struct bar *) my_foo;
```

$$\Longrightarrow$$

```
my_bar = (warn_if_not(__is_aU(my_foo,
                              &__uniqtype_bar)),
          (struct bar *) my_foo);
```

- Find and analyse allocation sites:

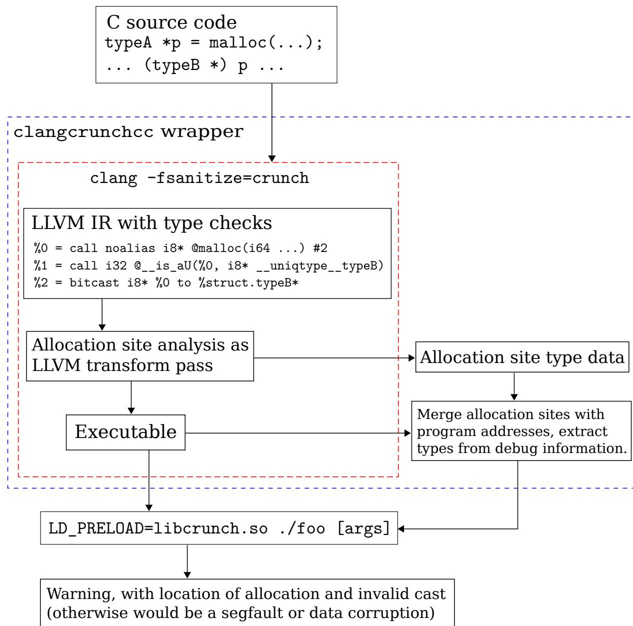
```
... = malloc(200 * sizeof(int));
```

$$\Longrightarrow$$

```
/path/to/test.c 5 malloc __uniqtype__int
```

- Linker magic and run-time.

How does it work?



Instrumenting pointer casts

```
$ clang -fsanitize=crunch ...

%crunch_check =
  call i32 @__is_aU(i8* bitcast
                      (i32* @blah to i8*),
                    i8* bitcast
                      (i8** @__uniqtype__int to i8*))
... ; Warn if check failed
%0 = bitcast i8* bitcast (i32* @blah to i8*) to i32*
```

Statically find allocation types

How do we know the type of an allocation in C?

```
struct foo *ptr =
  (struct foo *) malloc(sizeof(struct foo));
```

Statically find allocation types

How do we know the type of an allocation in C?

```
struct foo *ptr =
  (struct foo *) malloc(sizeof(struct foo));
```

Answer: What it's first assigned to.

Statically find allocation types

How do we know the type of an allocation in C?

```
struct foo *ptr =
  (struct foo *) malloc(sizeof(struct foo));
```

Answer: What it’s first assigned to.

But we could miss exactly the type of bug we’re trying to catch:

```
// WRONG:
struct foo *ptr = malloc(sizeof(struct foo *));
```

A better solution

Look at the allocation's *size*:

A better solution

Look at the allocation's *size*:

```
void *ptr = malloc(sizeof(struct foo));
```

Easy to infer that `ptr` points to a `struct foo`.

A better solution

Look at the allocation's *size*:

```
void *ptr = malloc(sizeof(struct foo));
```

Easy to infer that `ptr` points to a `struct foo`.

But tricky to implement in Clang:

```
size_t size = sizeof(int) * 10;
...
void *ptr = malloc(size);
```

How to find the definition of `size` from the AST?

Use an LLVM analysis

- Clang generates a dummy function call whenever it sees `sizeof`.
- In an LLVM transform pass:
  - Look for uses of all the allocation functions we know about
  - Recurse over operands of the `size` parameter
  - Hope we find a `sizeof` expression

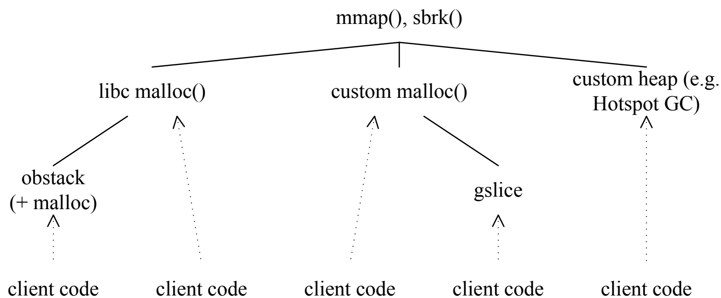
Type ‘arithmetic’

Preserve `sizeof` information through arithmetic operations:

- `sizeof(struct foo) * len`: Array of `foo`s
- `sizeof(struct foo) + len`: A `foo` before a variable-length buffer
- `sizeof(array) / sizeof(*array)`: The number of elements in a constant array

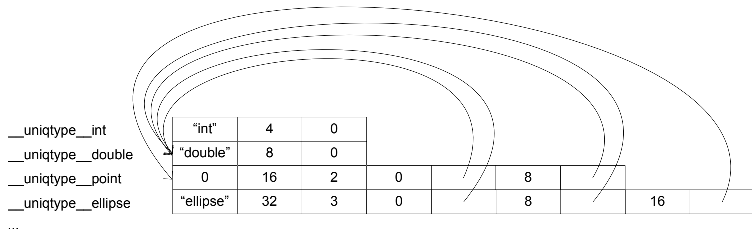
Like dimensional analysis.

Allocations



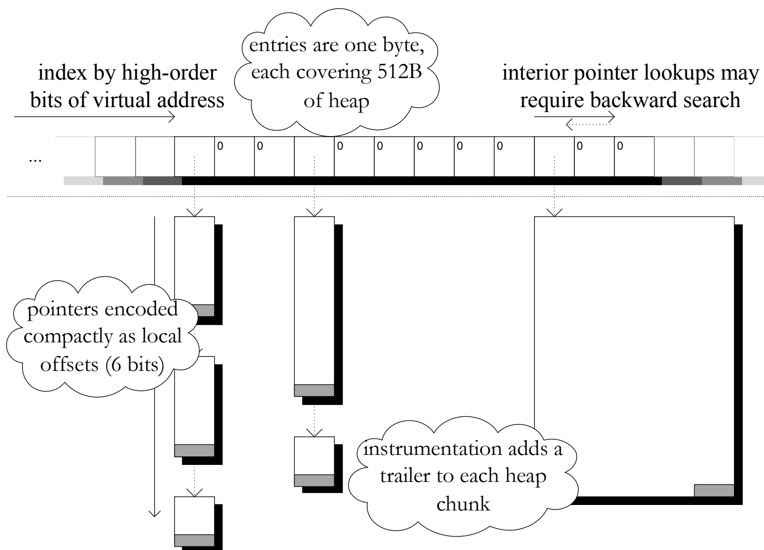
Uniqtypes

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



- Use the linker to keep them unique
- ⇒ 'exact type' test is a pointer comparison
- `__is_a()` is a short search

Memtables



Performance

SPECCPU2006:

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%

Status and wish-list

Status:

- Open-source:
  - https://github.com/chrisdiamand/clangcrunch
  - https://github.com/stephenrkell
- Works! (mostly)
- Could be faster

To-do:

- Eliminate compiler wrapper
- More languages (C++)
- Build system

Contributions welcome!
Questions?