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.

```c
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:
  
  ```c
  my_bar = (struct bar *) my_foo;
  
  ⇒
  
  my_bar = (warn_if_not(__is_aU(my_foo, &__uniqtype_bar)),
            (struct bar *) my_foo);
  ```
How does it work?

- **Instrument pointer casts:**
  
  \[
  \text{my\_bar} = (\text{struct bar *}) \text{ my\_foo}; \\
  \Rightarrow \\
  \text{my\_bar} = (\text{warn\_if\_not(__is\_aU(my\_foo, \\
    &__uniqtype\_bar)),} \\
  (\text{struct bar *}) \text{ my\_foo});
  \]

- **Find and analyse allocation sites:**
  
  \[
  \ldots = \text{malloc} (200 \times \text{sizeof(int)}); \\
  \Rightarrow \\
  /\text{path/to/test.c} 5 \text{ malloc __uniqtype__int}
  \]
How does it work?

- Instrument pointer casts:
  
  ```c
  my_bar = (struct bar *) my_foo;
  
  ⇒
  
  my_bar = (warn_if_not(__is_aU(my_foo, &__uniqtype_bar)),
            (struct bar *) my_foo);
  ```

- Find and analyse allocation sites:
  
  ```c
  ... = malloc(200 * sizeof(int));
  
  ⇒
  
  /path/to/test.c 5 malloc __uniqtype__int
  ```

- Linker magic and run-time.
How does it work?

C source code
```
typeA *p = malloc(...);
... (typeB *) p ...
```

c clangcrunchcc wrapper
```
clang -fsanitize=crunch
```

LLVM IR with type checks
```
%0 = call noalias i8* @malloc(i64 ...) #2
%1 = call i32 @__is_aU(%0, i8* __uniqtype__typeB)
%2 = bitcast i8* %0 to %struct.typeB*
```

Allocation site analysis as LLVM transform pass

Allocation site type data

Executable

Merge allocation sites with program addresses, extract types from debug information.

LD_PRELOAD=libcrunch.so ./foo [args]

Warning, with location of allocation and invalid cast (otherwise would be a segfault or data corruption)
$ 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?

```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?

```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?

```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:

```c
// 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:

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

Easy to infer that `ptr` points to a `struct foo`. 
Look at the allocation’s size:

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

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

But tricky to implement in Clang:

```c
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 foos
- `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

- mmap(), sbrk()
- libc malloc()
- custom malloc()
- custom heap (e.g. Hotspot GC)
- obstack (+ malloc)
- gslice
- client code

Diagram:

```
   Allocations
       ├── mmap(), sbrk()
       │    └── libc malloc()
       │         ├── obstack (+ malloc)
       │         │    └── client code
       │         ├── custom malloc()
       │         │    └── gslice
       │         │         └── client code
       │         └── custom heap (e.g. Hotspot GC)
       │                             └── client code
       └── client code
```
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

index by high-order bits of virtual address

entries are one byte, each covering 512B of heap

interior pointer lookups may require backward search

pointers encoded compactly as local offsets (6 bits)

instrumentation adds a trailer to each heap chunk
Performance

SPECCPU2006:

<table>
<thead>
<tr>
<th>bench</th>
<th>normal /s</th>
<th>crunch</th>
<th>nopreload</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>4.95</td>
<td>+6.8%</td>
<td>+1.4%</td>
</tr>
<tr>
<td>gcc</td>
<td>0.983</td>
<td>+160%</td>
<td>−%</td>
</tr>
<tr>
<td>gobmk</td>
<td>14.6</td>
<td>+11%</td>
<td>+2.0%</td>
</tr>
<tr>
<td>h264ref</td>
<td>10.1</td>
<td>+3.9%</td>
<td>+2.9%</td>
</tr>
<tr>
<td>hmer</td>
<td>2.16</td>
<td>+8.3%</td>
<td>+3.7%</td>
</tr>
<tr>
<td>lbm</td>
<td>3.42</td>
<td>+9.6%</td>
<td>+1.7%</td>
</tr>
<tr>
<td>mcf</td>
<td>2.48</td>
<td>+12%</td>
<td>(0.5%)</td>
</tr>
<tr>
<td>milc</td>
<td>8.78</td>
<td>+38%</td>
<td>+5.4%</td>
</tr>
<tr>
<td>sjeng</td>
<td>3.33</td>
<td>+1.5%</td>
<td>(1.3%)</td>
</tr>
<tr>
<td>sphinx3</td>
<td>1.60</td>
<td>+13%</td>
<td>+0.0%</td>
</tr>
</tbody>
</table>
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?