Dynamically checking type-correctness of whole programs
(work newly in-progress).

Stephen Kell
stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker, (struct commit *)obj))
        return -1;
    return 0;
}
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker, (struct commit *)obj))
        return -1;
    return 0;
}

Wanted (naive version): check this!

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

CHECK this (at run time)
Wanted (naive version): check this!

```c
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…
I will describe libcrunch, which is

- an infrastructure for run-time type checking
- encodes type checks as assertions
- no guarantee of “safety” (but...)
- support idiomatic unsafe code
- checks inserted by per-language front-ends
- no binary interface changes
- no source changes, usually*

(* but sometimes out-of-band guidance helps)
Introducing libcrunch

The user’s view:

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

where

- myprog contains *type assertions* (we’ll see how)
- normally “disabled”
- enabled when libcrunch is linked in
- compiler [wrapper] inserts assertions automatically
What is run-time type checking?

Check every program operation is “type-correct”, i.e.

- program state is a collection of stored values
- … allocated as instances of some “data type”
- data types signify meaning
- operations consume and produce stored values…

More precise definition wanted…

- for C, plan to use Cerberus to create formal definition
What checks are we interested in?

Recall the example:

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

Primitive errors are not our concern

- even C compilers check primitive type-correctness

First-order and up

- all about pointers
- first cut: check casts (& implicit strengthenings) in C
How it works, in a nutshell

```c
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker,
                      (struct commit *)obj))
        return -1;
    return 0;
}
```
How it works, in a nutshell

```c
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker,
                        (assert(__is_a(obj, "struct_commit")), // or something like this
                         (struct commit *)obj)))
        return -1;
    return 0;
}
```
How it works, in a nutshell

```c
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker,
        (assert(__is_a(obj, "struct_commit")), // or something like this
            (struct commit *)obj))
            return -1;
    return 0;
}
```

To make this work, we need:

- type information on every allocation in program
- efficient run-time representation of types
- fast __is_a function
- something to write these assertions for us
Idealised view of libcrunch operation

deployed binaries
(with data-type assertions)

/bin/foo
/lib/libxyz.so

debugging information
(with allocation site information)

/bin/...debug/...libxyz.so

precompute unique data types

/libcrunch.so
/bin/uniqtyp/foo.so

load, link and run (ld.so)

program image

heap_index

0xdeadbeef, "Widget"?

__is_a

uniqtypes
Type info for each allocation

Type info for allocation is reasonable because

- ... to allocate, you need a size
- three kinds of allocations: static, stack, heap
- assume all heap allocators are instrumented...

Assume we have debug info

- handles stack and static cases
What happens at run time?

program image

`__is_a(0xdeadbeec, “Widget”)?`

true

libdl

heap_index

allocsites

uniqtypes

lookup(“Widget”)

&__uniqtype_Widget

lookup(0xdeadbeec)

allocsite: 0x8901234,
offset: 0xc

lookup(0x8901234)

&__uniqtype_Window

find(
&__uniqtype_Window,
&__uniqtype_Widget,
0xc)

found
Looking up object metadata (1)

Recall: need info about an arbitrary object’s allocation

- … given an arbitrary pointer

Stack case

- walk the stack + use debug info for locals/args

Static case

- use debug info

Heap case

- hard! might be an interior pointer
- use clever virtual memory-based data structure (ask me)
is a containment...

A pointer might satisfy \textit{\_is\_a} > 1 way

Consider “what is”

- \&my\_ellipse
- \&my\_ellipse.ctr
- ...

(Subclassing is usually implemented this way.)
Efficiently reifying data types at run time

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

Reify data types *uniquely*, describing *containment*

- uniqueness → “exact type” test is a pointer comparison
- `__is_a()` is a simple, fast search through this structure
__is_a is a nominal check, but we can also write

- __like_a – “1-structural” (unwrap one level)
- __phys_a – “*-structural” (unwrap maximally)
- __refines – may instantiate padding (à la sockaddr)
- __named_a – opaque workaround
Notes about memory correctness

We (currently) do nothing about memory correctness! E.g.

```c
void f() {
    int a;
    int bs[2];
    for (int *p = &bs[0]; p <= 2; ++p) {
        /* ... */
    }
}
```

- bug-finding, not verification, not security...
- faster! avoid per-pointer (cf. per-object) metadata
- most memory-incorrect programs are type-incorrect...
- could “force a cast” after pointer arithmetic

SoftBound + CETS do a pretty good job

- we could replicate them...
Recap

What we’ve just seen is

- a runtime system for evaluating type assertions
- fast (biggest slowdown seen 20%; often <10%)
- (by design) flexible
- a “whole program” language-neutral design
- binary compatible

What about source compatibility?
Who inserts the assertions?

- instrumentation: “one assertion per pointer cast”
- analysis: “what data type is being `malloc()`’d?”
- … guess from use of `sizeof`

![Diagram showing source tree, CIL-based compiler front-end, and allocation sites](image)
Complications (1)

With metadata

- dynamic loading (merge uniqtypes)
- non-standard alloc functions (explicit support)

With compilers (currently false pos/negs)

- address-taken temporaries (fix compiler for debug info)
- varargs actuals
- alloca()

+ assert() usually isn’t quite what you want...
With the C front end (false pos or “intervention required”)

- very weird uses of sizeof
- weird avoidance of sizeof
- char special case
- object re-use
- unions (but mostly doable! three cases; ask me)
- some cases of multiple indirection cause false pos
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];}
}

Client then does (making libcrunch print a warning)

neighbor = (int **) calloc (NDIRS, sizeof(int *));
/* ... */
sort_eight_special ((void **) neighbor );

Question: is this valid C?
What’s in it for REMS

Check “agreement” between libcrunch and cerberus

- inclusion, for the relevant subset of complaints

Tool for exploring behaviour of real programs

- good at turning up “dodgy” code (oft also “correct”!)

Representative of a wider set of tools...

- insight for bridging between source and run-time worlds
- linking tie-in...
Recap, conclusions

We’ve seen

- a runtime infrastructure for fast checking
- a prototype C front-end

Remaining challenges for the run-time part:

- finish the paper...
- multi-language story
- support more complex specifications ("types")

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

Thanks for listening. Questions?