Run-time type checking of whole programs
and other stories

Stephen Kell

stephen.kell@usi.ch

University of Lugano
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!

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;
}
```

(at run time)

But also wanted:

- binary compatible
- source compatible
- reasonable performance
- avoid being C-specific!*

* mostly…
Wanted (truthful version)

- Understand “type-correct” compositions of these static checking too!
- Other “type systems” (a.k.a. classes of specification) too too much to talk about today...
This talk in one slide

I 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)
Why unsafe languages?

■ fine control of resource utilisation
■ talk directly to operating system
■ talk directly to hardware
■ freedom to simulate new language-level abstractions
■ freedom to violate program-level abstractions
■ manual optimisation
■ re-use existing code
State of the art: a straw man

Competitive existing approaches are broadly alike:

- conflate type- with memory-correctness
- demand proof
- reject many correct programs
- break library compatibility
- specific to C
- and/or high run-time overhead

CCured, Deputy, Chandra & Reps ’99, Condit et al ’09,…
Terminology (a minefield)

“type”
- for this talk: a data type (named set of values)
- normally I’m with Pierce, but life’s too short

“safety”
- once upon a time, meant a run-time property
- now has vague meaning

“soundness”
- opposing usages exist

I prefer “type correctness”, “verified type correctness”
Introducing libcrunch

The vision:

- $ ./myprog # runs normally
- $ LD_PRELOAD=libcrunch.so ./myprog # does checks

where

- myprog contains type assertions
- normally “disabled”
- enabled when libcrunch is linked in
- compiler [wrapper] inserts assertions automatically
What is run-time type checking?

Ideally, checks every program operation is “type-correct”

- respects the meaning of its run-time input values
- as ascribed by programmers using data types
- all storage is allocated with a data type

Examples

```c
int a; char b; double f(int, char*);
f(a, &b);  // okay!
f(b, &a);  // not okay
b = f(a, &b);  // okay (implicit conversion inserted)

void *p = get_object (); void *q = (void *)0xdeadbeef;
f(*(int*)p, (char*)q); // depends...
```
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;
}
```

For us, the interesting values are pointers

- even C checks primitive type-correctness
- → limited BCPL support :-(
- C requires casts on “dangerous” pointer ops

Subtle question: what *invariant* do we want to maintain?
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")),
            (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")),
                (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)

/lib/foo
/lib/libxyz.so

debugging information (with allocation site information)

/bin/.debug/foo
/lib/.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"?

true

_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?

\[
\text{program image} \quad \text{\_is\_a}(0x\text{deadbeec}, \text{“Widget”})? \\
\text{\_is\_a} \quad \text{\_is\_a} \quad \text{\_is\_a} \\
\text{true} \\
\]

\[
\begin{align*}
\text{lookup}(\text{“Widget”}) & \quad \text{\_uniqtype\_Widget} \\
\text{lookup}(0x\text{deadbeec}) & \quad \text{\_uniqtype\_Widget} \\
\text{lookup}(0x\text{8901234}) & \quad \text{\_uniqtype\_Window} \\
\text{find} & \quad \text{\_uniqtype\_Window, \_uniqtype\_Widget, 0xc} \\
\text{found} & \quad \text{true} \\
\end{align*}
\]

libdl

heap_index

allocsites

uniqtypes
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
- static case: use debug info
- heap case: hard! might be an *interior* pointer
Looking up object metadata (2)

Why the heap case is difficult:

Native objects are trees; no descriptive headers! Contrast:

VM-style objects: “no interior pointers”
Solution in the heap (difficult) case:

- we’ll need some `malloc()` hooks...
- which keep an index of the heap
- in a memtable—efficient address-keyed associative map
- storing object’s allocation site
- look up corresponding data type later
Indexing chunks

Inspired by free chunk binning in Doug Lea’s `malloc`...
Indexing chunks

Inspired by free chunk binning in Doug Lea’s `malloc`...

... but index *allocated* chunks binned by *address*
How many bins?

Each bin is a linked list of heap chunks
- thread next/prev pointers through allocated chunks...
- also store allocation site addr
- overhead per chunk: one word + two bytes

Finding chunk is $O(n)$ given bin of size $n$
- → want bins to be as small as possible
- Q: how many bins can we have?
- A: lots... really, lots!
Really, how big?

Bin index resembles a linear page table. Exploit

- sparseness of address space usage
- lazy memory commit on “modern OSes” (Linux)

Reasonable tuning for Intel architectures:

- one bin covers 512 bytes of VAS
- each bin’s head pointer takes one byte in the index
- covering $n$-bit AS requires $2^{n-9}$-byte bin index
index by high-order bits of virtual address

instrumentation adds a trailer to each heap chunk

use trailers to search a short list for chunk overlapping looked-up address

table entries are one byte, each covering 1KB of heap

interior pointer lookups may require backward search

large objects are indexed at their base address

pointers encoded compactly as local offsets (7 bits)
More about memtables

Indexing the heap with a memtable is

- more VAS-efficient than shadow space (SoftBound)
- supports > 1 index, unlike placement-based approaches

Memtables are versatile

- buckets don’t have to be linked lists
- can tune size / coverage…
Remind me: what happens at run time?

**Program Image**

`__is_a(0xdeadbeec, “Widget”)?`

**Heap Index**

`lookup(0xdeadbeec)`

- `allocsite`: 0x8901234
- `offset`: 0xc

**Allocsites**

`lookup(0x8901234)`

- `__uniqtype_Window`

**Uniqtypes**

`find(`

- `__uniqtype_Window`

- `__uniqtype_Widget`

- `0xc`

- `found`
__is_a, containment... 

A pointer might satisfy __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

Precompute this for speed (using `make`!)
__is_a is a nominal check, but we can also write

- __like_a – “structural” (unwrap one level)
- __refines – padded open unions (à la sockaddr)
- __named_a – opaque workaround
Notes about memory safety

We do nothing about memory safety! 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
Recap

What we’ve just seen is

- a runtime system for evaluating type assertions
- fast
- flexible
- a “whole program” design
- language-neutral
- binary compatible

What about source compatibility?
libcrunch prototype: C front-end

Who inserts the assertions?

- instrumentation: “one assertion per pointer cast”
- analysis: “what data type is being malloc()’d?”
- … guess from use of sizeof
A quick peek at the full picture

source tree
- main.c
- widget.c
- util.c
- ...

CIL-based compiler front-end
- dump allocation sites (dumpallocs)
- instrument pointer casts
- compile and link (with underlying compiler)

deployed binaries (with data-type assertions)
- /bin/foo
- /lib/libxyz.so

debugging information (without allocation site information)
- /bin/.debug/foo
- /lib/.debug/libxyz.so

precompute unique data types
- /bin/.allocs/foo
- /lib/.allocs/libxyz.so

load, link and run (ld.so)
- loaded dynamically

program image
- heap_index
- uniqtypes
- _is_a
- find
- uniqtypes
- ...

compile and link (with underlying compiler)
- source tree
  - main.i
  - allocs
  - main.i
  - allocs
  - widget.i
  - allocs
  - util.i
  - allocs
  - ...

dump allocation sites (dumpallocs)
- instrument pointer casts
- compile and link (with underlying compiler)

instrumented pointer casts
- CIL-based compiler front-end
- libcrunch...
Complications (2)

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...
Complications (3)

With the C front end (false pos or “intervention required”)

- weird uses of sizeof
- weird avoidance of sizeof
- redefinition of “the same” data type: use __like_a
- casts to incomplete data types: use __named_a
- char special case
- object re-use
- unions (sometimes okay)
- address-taken union arms
How fast is it?

Performance results go here!

Very quick experiment measuring: heap overhead:

- run **gcc** on a large C file
- ... with/without **malloc** instrumentation
- times in seconds (three runs each):
  - **gcc** + no-op hooks: 1.73, 1.76, 1.72
  - **gcc** + **vgHash** index: 1.83, 1.82, 1.85
  - **gcc** + **memtable** index: 1.77, 1.78, 1.77
A story about multiple languages

Imagine a world where

- data structures can be shared
- ... among code in different languages
- object representations are not “owned” by language
- (“How?” is a separate talk.)

Different invariants make sense for different languages...
Enforcing an invariant with libcrunch (1)

Subtle question: what invariant do we want to maintain?

- for C: storage has an allocated data type
- … including pointer contracts

```c
struct blah {
    int something;
    struct foo *my_foo; // always points to a foo! (or null)
};
```

- allows us to do

```c
x = p->q->r;
```

without check.

This is good for C…
Enforcing an invariant with libcrunch (2)

In multi-language scenarios we may not have this invariant

■ e.g. Python sharing data structures with C
■ to be safe, our
  \[ x = p \rightarrow q \rightarrow r; \]
  might need extra checks

Invariants on shared data are a shared concern

■ e.g. C instrumentor must be more paranoid
■ future work to support this
■ an exercise in assume–guarantee between languages
Generality: an assertion about assertions

All “type checks” can be encoded as assertions...

```c
void swap(void *a, void *b, size_t size)
{
    /* Unconstrained parametric polymorphism */
    assert(typeof(a) == typeof(b));
}

... over a suitably augmented program.
```
So much for run time

Type checkers are useful, but suffer some problems:

- specification language is distinct and complex
- sometimes incomprehensible error messages
- inflexible
- hard to reason across languages
- can’t easily add new kinds of check

Can we extend run-time checking towards compile time?
Symbolic execution: the Noddy story (1)

“Let input be $x$; run for all inputs at once.”

```c
int main(int argc, char **argv) { // ... where argv[1] is symbolic
    assume(argc > 1 && strlen(argv[1]) == 2); // for simplicity
    int temp1 = atoi(argv[1]);
}
```

After this function, `temp1` has symbolic value:

```
(Add w32
  (Mul w32 10 (Sub w32 48 (SExt w32 (Read w8 0 argv_1))))
  (Sub w32 48 (SExt w32 (Read w8 1 argv_1))))
```

Execution is

- `deferred`, w.r.t. program values
- `exploratory`, w.r.t. program paths
Symbolic execution: the Noddy story (2)

SE is a general forward \((sp)\) analysis:

- program state includes "symbolic variables"…
- …constrained by branches taken so far
- constraints are SMT formulae
- branching exploration of state space
- usu. maximally path- (and "heap-") sensitive

Popular application: bug finding ("test case generation")

- log feasible failures; solver can generate test input
- usually runs forever…
Symbolic execution tools only find errors that are

- generic, or...
- ...programmer-supplied: `assert()`

libcrunch has expanded our specification language!

- i.e. `assert()` not just over program variables!
- use `__is_a()` to assert about allocation types
Extending libcrunch towards compile time (2)

Use symbolic execution to “accelerate” dynamic checking!

- start from our run-time checker
- adding exploratoriness (path-sensitivity) via SE
- … with tunable proof burden on the user
- (i.e. not-provably-true checks are “left in”)

Challenge: scalability. Need an *abstraction* technique.
Reconstructing type checking

```c
void *expensive(void *arg); // no type signature!
if (runExpensive) { outputStr = expensive(); }
else { outputStr = "(not done)"; }

assert(is_a(outputStr, "char"));
printf("Status:%s\n", outputStr);
```

Want to avoid `expensive()`; how?
- signatures (what type checkers do)
Reconstructing type checking

```c
void *expensive(void *arg); // no type sig, but now have summary...
if (runExpensive) {
    outputStr = expensive();
    typeof(outputStr) = "char"; // from summary
} else {
    outputStr = "(not done)";
    typeof(outputStr) = "char";
}
assert(typeof(outputStr) == "char");
printf("Status:\n%s\n", outputStr);
```

Want to avoid `expensive();`; how?

- signatures (what type checkers have)
- summaries (their generalisation to SE)
Reconstructing type checking

```c
if (runExpensive) {
    typeof(outputStr) = "char";  // from summary
} else {
    typeof(outputStr) = "char"; }
assert(typeof(outputStr) == "char");
// printf () sliced away

Want to avoid expensive(); how?
- signatures (what type checkers have)
- summaries (their generalisation to SE)

To check just the assertions:
- slice on the assertion condition!
```
Recap, conclusions

We’ve seen

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

Challenges for the run-time part:

■ encode more complex specifications (types)
■ make configurable for multiple languages

Challenges for the “towards static” part:

■ make it work, make it scale, …

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

Thanks for listening. Questions?
The story so far

We’ve seen two possibly-fruitful techniques

- symbolic execution + shadow instrumentation
  - SE “accelerates” dynamic analysis
- slicing on assertion conditions
  - slicing abstracts from “program” to “checker”

Neither is clearly static nor dynamic analysis. What’s good:

- programmer-friendly specifications,
- compositional: many shadows can co-exist
- analysability is a property of program, not language
- scope for tuning the precision of analysis
Challenges

Challenges:

■ make it work!
■ make it terminate
■ DSL for specifying new shadow domains?
■ integrating with dynamic compilation infrastructure

Scalability is determined by *dependency structure*

■ i.e. complexity of dependency…
■ … between shadow and program values
■ … not on program or specification *language*!

Thanks for listening. Questions?
Agenda

“Program-level type checking”
■ not “type-level programming”!

Decouple specification (type assertions)
■ from verification algorithm (checker)

Specify a wide range of interesting stuff
■ types are one kind of shadow (≈ ghosts)

The burden of static typing isn’t annotations!
■ it’s syntactic regularity
Type correctness versus memory correctness

Memory correctness

- has some nice recent work (Softbound + CETS)
- per-pointer metadata → slower at run time

My take:

- memory-incorrect programs are also type-incorrect
- converse is not true!
- in this work, I just deal with type correctness
Generalised structural checking

(draw a tree on the whiteboard, please)
Invariants – a ramble-to-self

So far, fully instrumented, what invariant do we get?

- nominal typing
- stored pointers point to their allocated type (or null)
- … precisely! but
- subtyping is a non-issue!
- e.g. encoding subtyping as zero-offset containment “just works”
- and multiple inheritance works, given that adjustment is done explicitly
- the invariant holds as a consequence of the semantics of C!
- pointer loads: may be passed through cast
It's okay if we have just Python, say, and every pointer to int, we would have a harder problem. Allowed pointers declared as char* to actually allocate, so we didn't maintain this invariant, i.e., we

(anyone want to formalise this?)

explore the heap

so, we don't need to do check that chase pointers /

(invariant, i.e., points to a char (or null)
allocated as a char*, it still respects that we have maintained the invariant that if it was that was allocated as a char*

This checks that x points to something:

e.g., consider b = (char**) x;

rest

... and rely on the invariant to guarantee us the immediately

we only check the first level of indirection

... pointers...

on the target of pointers (to pointers (to...
It's a problem if we allow Python to manipulate C-allocated data structures.

C allocated data structures will claim to store an int.  

Python may have updated it to point to a Foo!

E.g., if we allowed C data

they will claim to store an int.