Dynamically checking types, bounds and maybe even more

(or: “some were meant for C”)

Stephen Kell

`stephen.kell@cl.cam.ac.uk`

Computer Laboratory
University of Cambridge

“Tool wanted” (how it all started)

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

“Tool wanted” (how it all started)

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

$\nwarrow \quad \nearrow$

CHECK this

(at run time)

“Tool wanted” (how it all started)

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

$\nwarrow \quad \nearrow$

CHECK this

(at run time)

But also wanted:

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

* mostly…

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`

Fast-forward to 2016

We can do it!

- checking casts works pretty well

Last year I talked about a bounds checker

- also now going pretty well (more shortly)

Other new developments:

- Clang front-end (Chris Diamand)
- generalising the infrastructure to other uses
  - liballocs core library (see Onward! 2015)

Impending tie-ins: Cerberus, CHERI, . . .

State of play c.2015

- libcrunch pretty good at run-time type checking
- supports idiomatic C, source- and binary-compatibly
- *does not check memory correctness*

State of play c.2015

- libcrunch pretty good at run-time type checking
- supports idiomatic C, source- and binary-compatibly
- *does not check memory correctness*

```
struct {int x; float y;} z;
int *x1 =              &z.x;       // ok
int *x2 = (int*)       &z;         // passes check
int *y1 = (int*)       &z.y;       //  fails !
int *y2 =              &z.x + 1;   // use SoftBound
int *y3 =          &((&z.x )[1]);  // use SoftBound
return &z;                         // use CETS
```

State of play c.2015

- libcrunch pretty good at run-time type checking
- supports idiomatic C, source- and binary-compatibly
- *does not check memory correctness*

```
struct {int x; float y;} z;
int *x1 =          &z.x;        // ok
int *x2 = (int*)   &z;          // passes check
int *y1 = (int*)   &z.y;        //  fails  (good)!
int *y2 =          &z.x + 1;    // ***
int *y3 =        &((&z.x )[1]); // ***
return &z;                      // use CETS
```

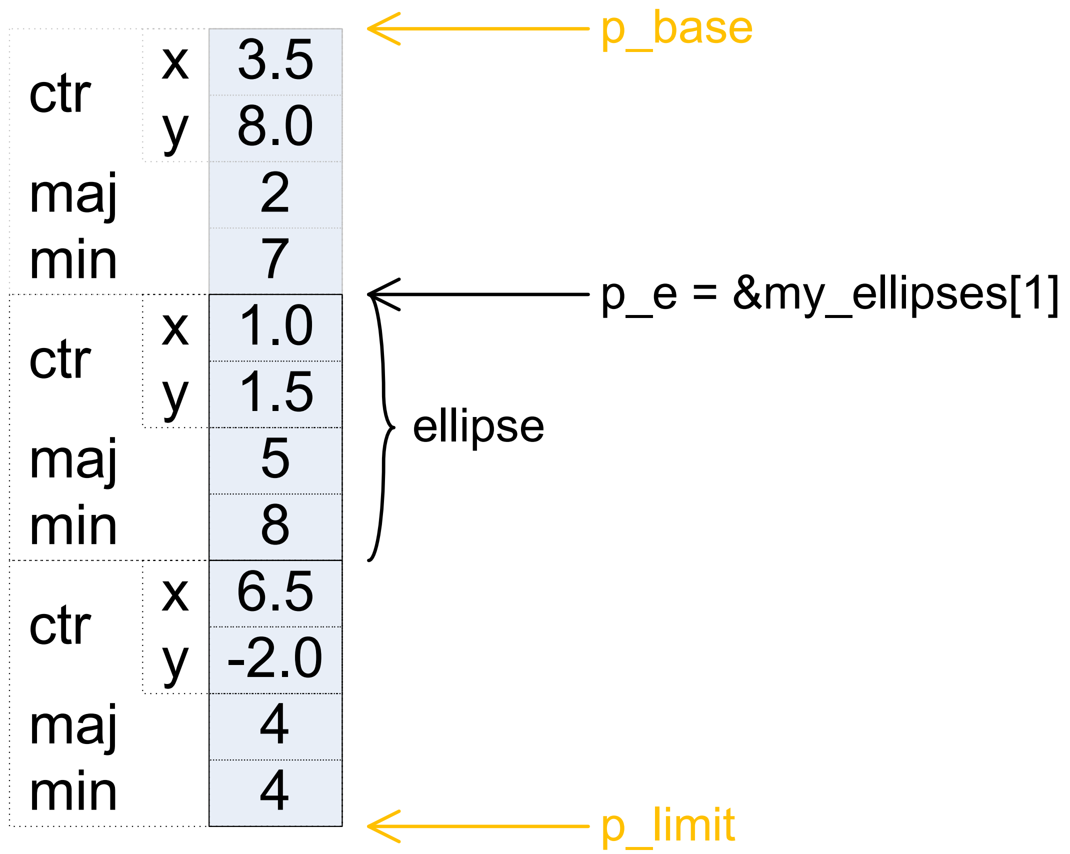
Wanted: a bounds checker people might even leave turned on?!

Must check bounds! But also

- support all common idioms
- be *precise*, not best-effort
- very, very few false positives
- minimise problems with uninstrumented libraries
- *option* to continue after a reported error
- easy to turn on/off
- fast

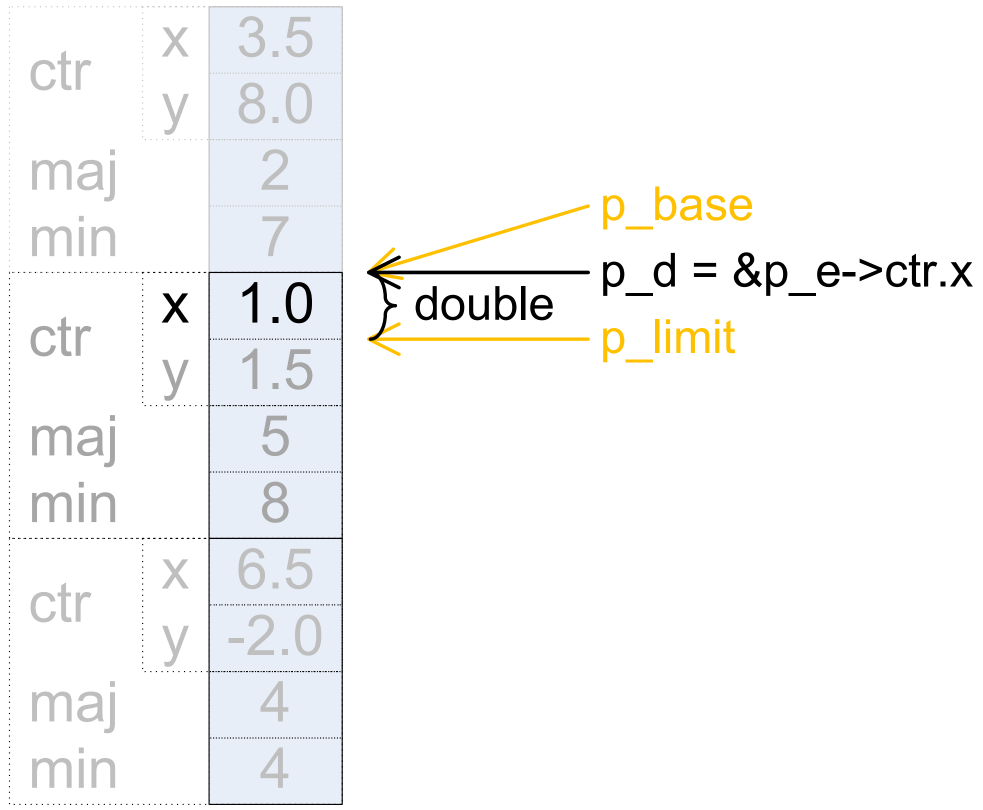
Memcheck, ASan, SoftBound all fail at > 1 of these

Existing bounds checkers use per-pointer metadata



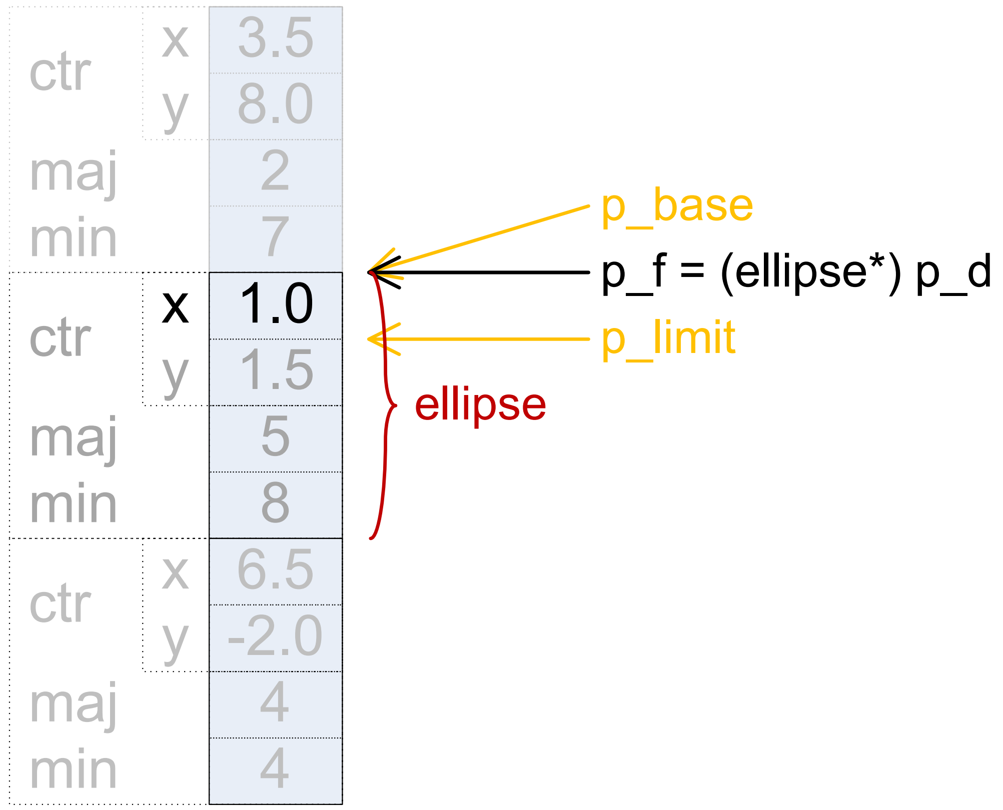
```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Existing bounds checkers use per-pointer metadata



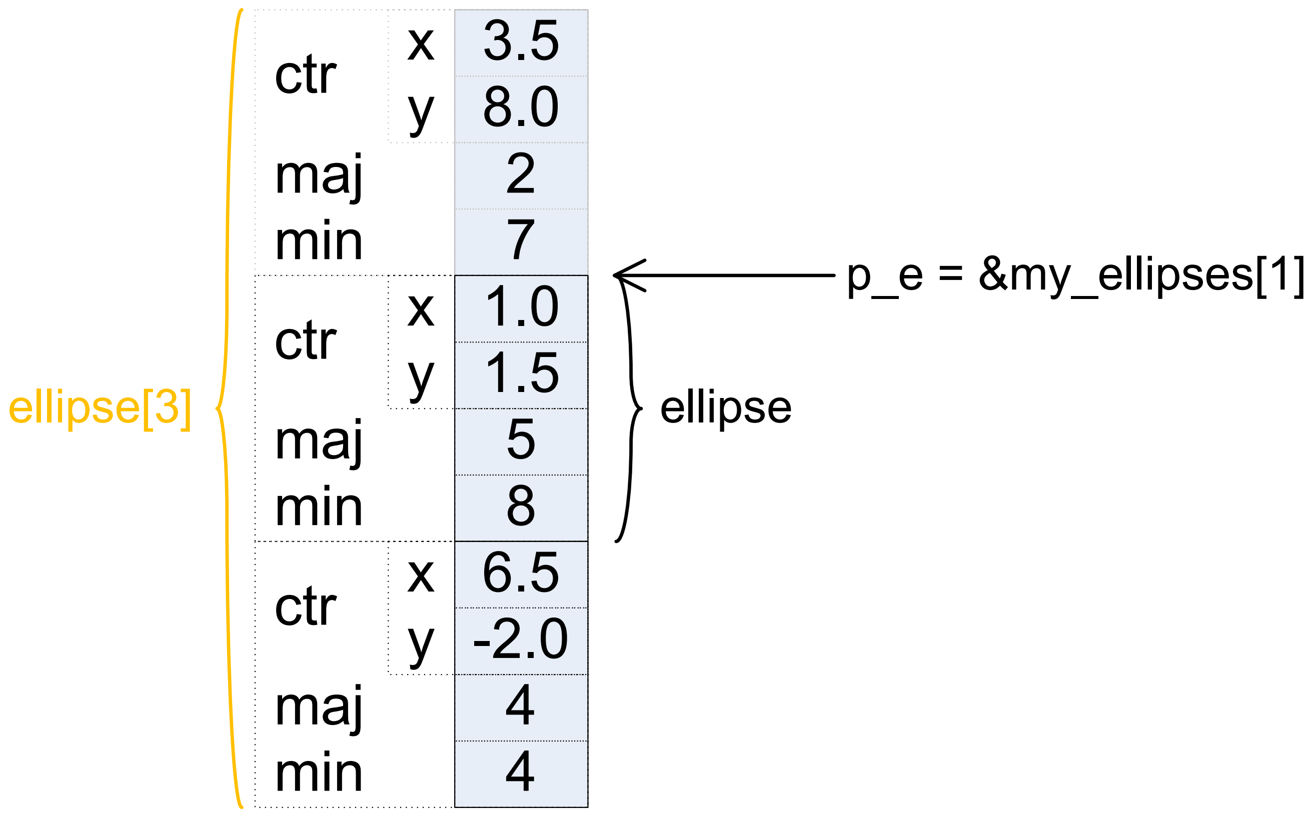
```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Without type information, pointer bounds may lose precision



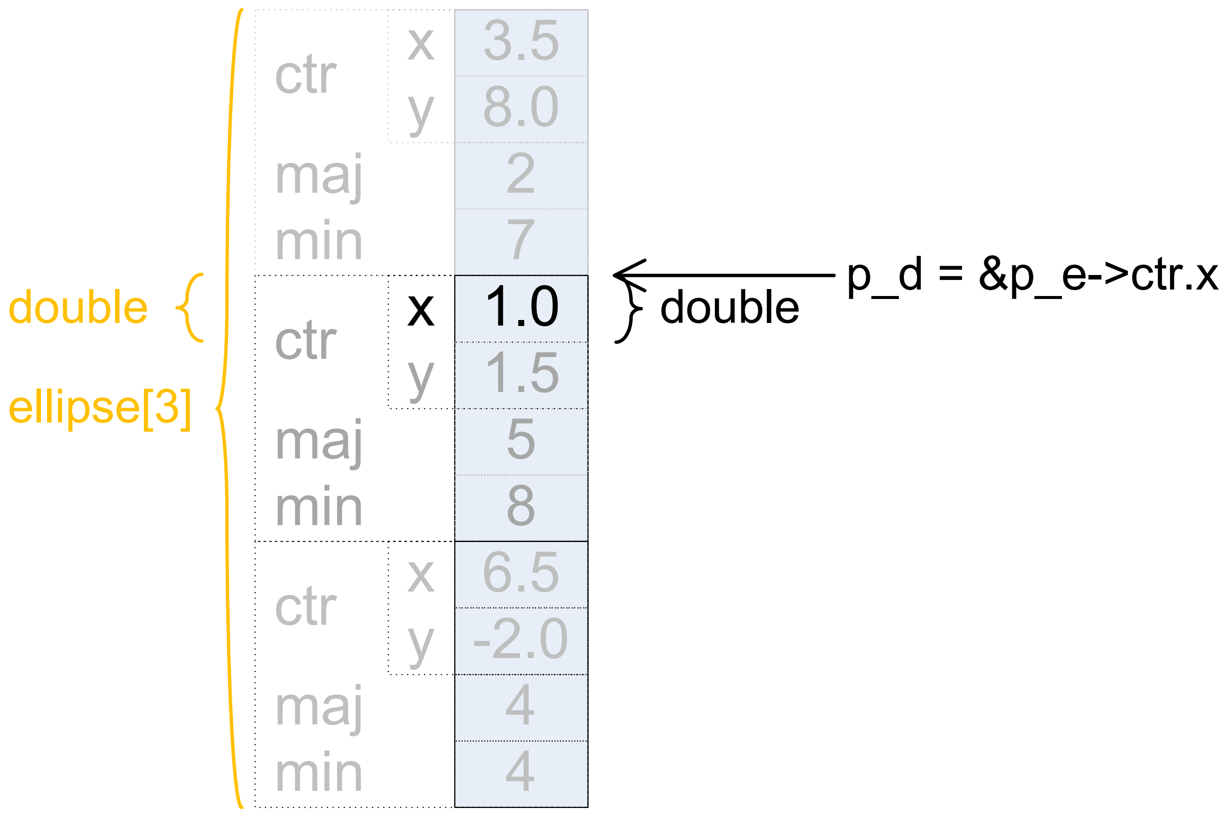
```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Given allocation type and pointer type, bounds are implicit



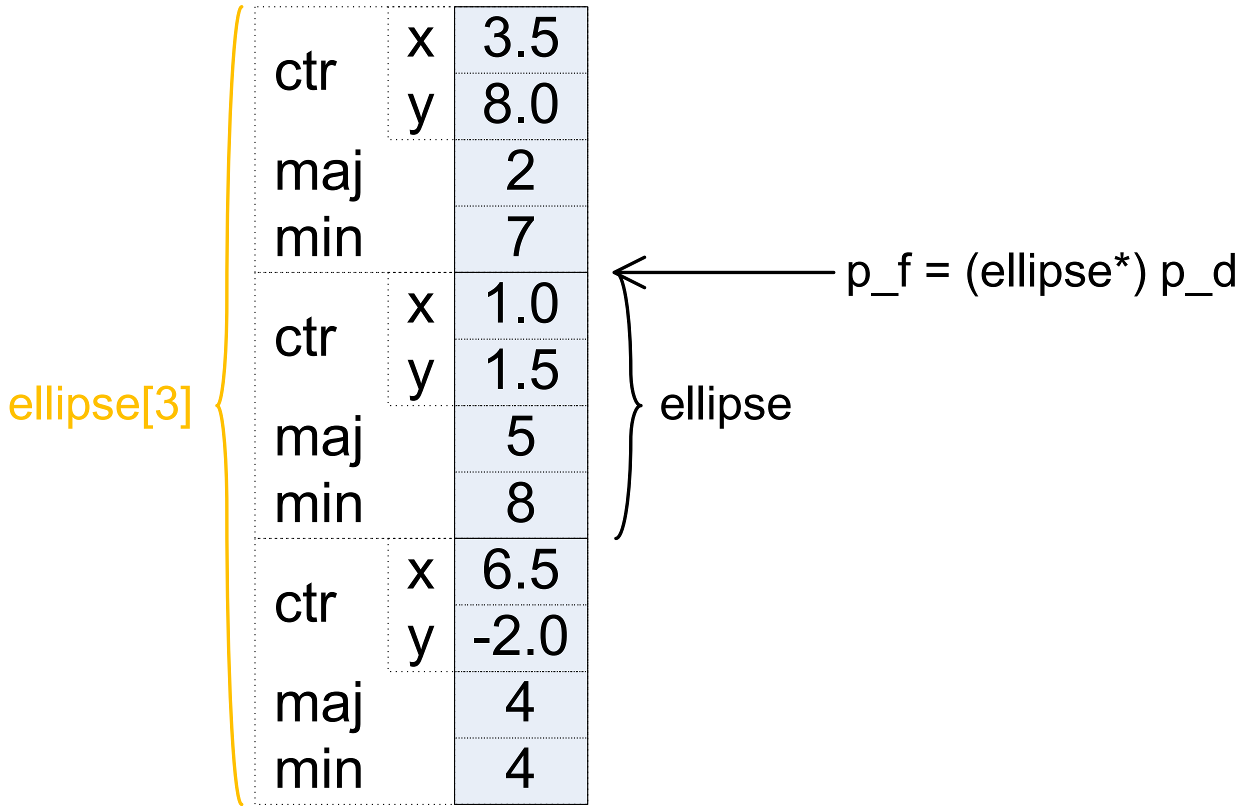
```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Given allocation type and pointer type, bounds are implicit



```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Given allocation type and pointer type, bounds are implicit



```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

The importance of being type-aware (when bounds-checking)

```
struct driver      { /* ... */ } *d = /* ... */;
struct i2c_driver { /* ... */ struct driver driver; /* ... */ };

#define container_of(ptr, type, member) \
 ((type *)( (char *)(ptr) - offsetof(type,member) ))

i2c_drv = container_of(d, struct i2c_driver, driver);
```

The importance of being type-aware (when bounds-checking)

```
struct driver      { /* ... */ } *d = /* ... */;
struct i2c_driver { /* ... */ struct driver driver; /* ... */ };

#define container_of(ptr, type, member) \
 ((type *)( (char *)(ptr) - offsetof(type,member) ))

i2c_drv = container_of(d, struct i2c_driver, driver);
```

SoftBound is oblivious to casts, even though they matter:

- bounds of `d`: just the smaller struct
- bounds of the `char*`: the whole allocation
- bounds of `i2c_drv`: the bigger struct

If only we knew the *type* of the storage!

Idea: a bounds-checker build on per-allocation type metadata

- avoid these false positives
- avoid libc wrappers, ...
- robust to uninstrumented callers/callees

Making it fast:

- cache bounds: make pointers “locally fat, globally thin”
- only check *derivation*, not *use*

```
inline int __check_derive_ptr(const void **p_derived,
    const void *derivedfrom, struct uniqtype *t,
    __libcrunch_bounds_t *opt_derivedfrom_bounds);
```

Lots of hacking later: did it work?

Mostly! But SoftBound-competitive performance requires

- bounds passing via a shadow stack (like SoftBound)
- bounds store/load via a shadow space (like SoftBound)

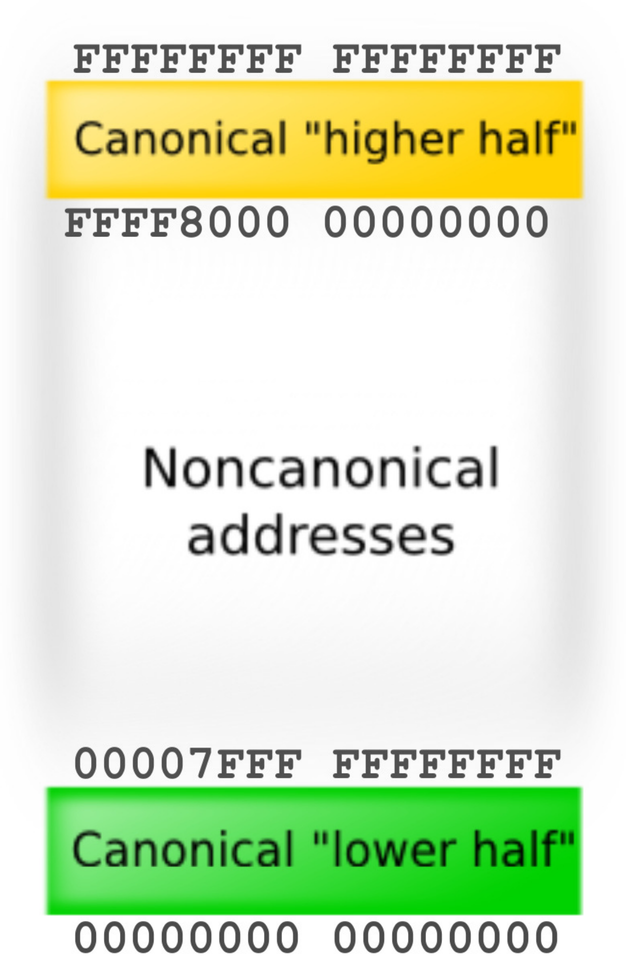
… i.e. still pushing per-pointer metadata around. But!

```
T  t = a[i];                    // derive, then immediately use
T *t = p + n;                   // derive (no use)
T *t = p->next->next->t; // use (x3)
```

Unlike SoftBound, we check pointer *derivations* not uses

- performance implications go here

Trap reps for one-past pointers



Use x86-64's non-canonical addresses

- to represent "one-past" addresses
- trap if used
- de-trap to compare, cast, etc.

Massively useful!

- tolerate some "pointer stuffing"
- (should) support nasty union cases
- (should) help "roaming" char*

Other arches: reserve $\frac{n-1}{n}$ of VAS

(diagram: Vladsinger, CC-BY-SA 3.0)

Other advances on SoftBound

- continuing after an error (!)
- dealing with casts
- staying precise even with uninstrumented libraries
- performance on linked-structure-based programs
  - TBC! good benchmarks, anyone?

Next: repetition and reproduction studies on SoftBound

- repeating SoftBound results (same code): tricky
- *reproducing* SoftBound results
  - do SoftBound-identical checks with libcrunch
  - disjoint infrastructure $\rightarrow$ reproduction interest

Emerging: a safe C that people might actually use?!

Likely forthcoming research tie-ins:

- Cerberus: formally state what’s being checked
- CHERI: multiple bounds checking “personalities”
- syscall spec work: syscalls need bounds checks!

Safety gap-plugging to do:

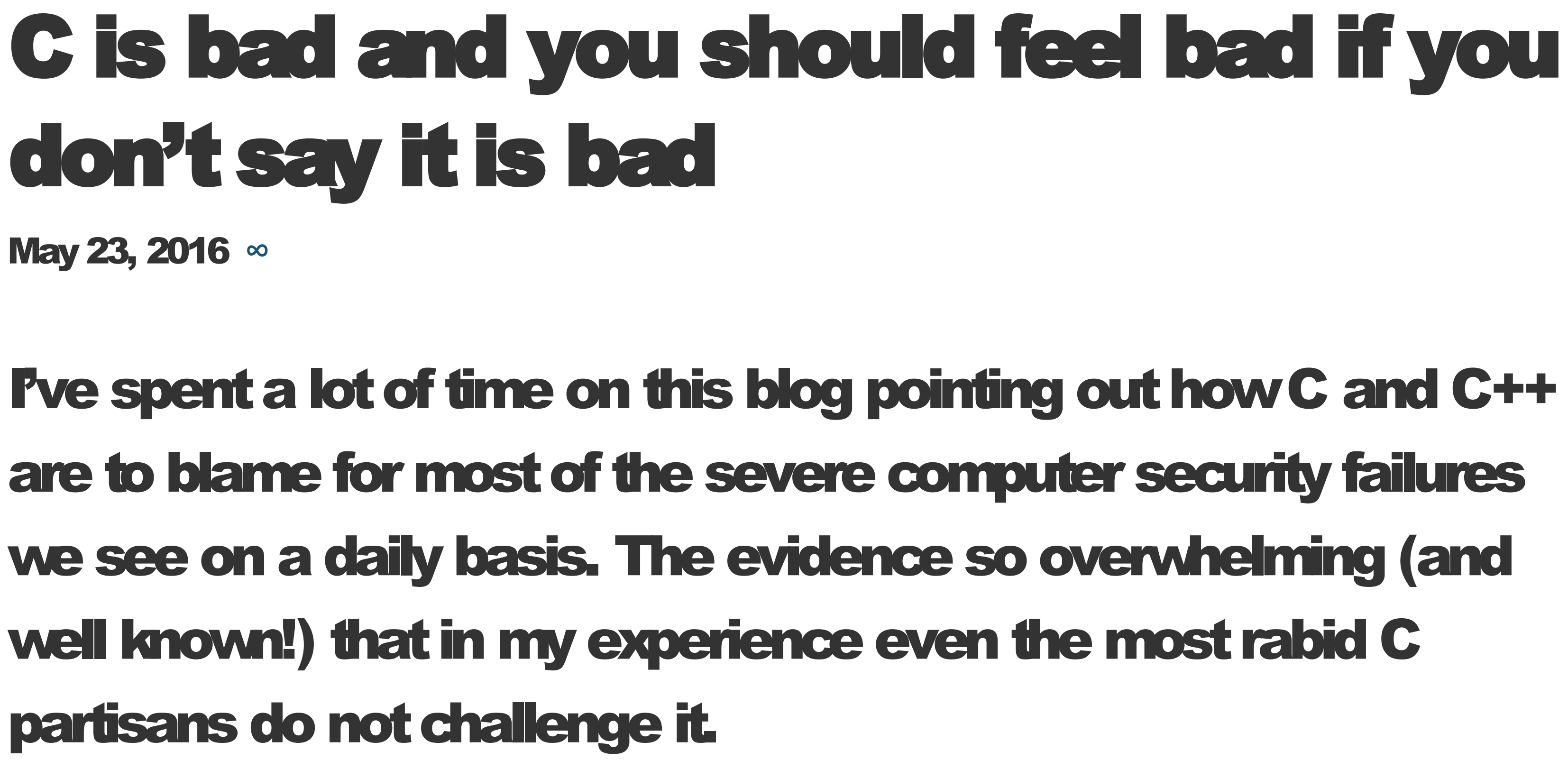
- easy-ish: unions, memcpy, link-time check
- more work: temporal safety (GC, initialization)
- roaming pointers, . . .

Development:

- in Clang; in-kernel, other arch/OSes, `make world`. . .

How not to feel bad (1)

A common view among language-y people:



C is bad and you should feel bad if you don’t say it is bad

May 23, 2016 ∞

I’ve spent a lot of time on this blog pointing out how C and C++ are to blame for most of the severe computer security failures we see on a daily basis. The evidence so overwhelming (and well known!) that in my experience even the most rabid C partisans do not challenge it.

How not to feel bad (2)

… but this view confuses *languages* with *implementations*!

What the world really needs is

- a safe implementation of C!   (and C++ and…)
- *not* (just) new safe languages or dialects

Preserve all of C, including the *real* good bits

- communicating with “aliens”, through memory
- it’s not [just] about manual memory management
- it’s not really about performance at all

“Conclusions”

```
$ git clone https://github.com/stephenrkell/liballocs.git
$ cd liballocs
$ git submodule init && git submodule update
$ make -C contrib
$ ./autogen.sh && . contrib/env.sh
$ ./configure --prefix=/usr/local && make
$ cd ..; export LIBALLOCS=`pwd`/liballocs
$ git clone https://github.com/stephenrkell/libcrunch.git
$ cd libcrunch && make
$ frontend/c/bin/crunchcc -o hello /path/to/hello.c
$ LD_PRELOAD=`pwd`/lib/libcrunch_preload.so ./hello
```

Thanks for listening. Please consider trying it out!