Fast, precise dynamic checking of types and bounds “in C”

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;
}

CHECK this (at run time)
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker, (struct commit *)obj))
        return −1;
    return 0;
}

But also wanted:

- binary-compatible
- source-compatible
- ... for real, idiomatic code in (say) C
- reasonable performance
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker, (struct commit *)obj))
        return -1;
    return 0;
}

But also wanted:

- binary-compatible
- source-compatible
- ... for real, idiomatic code in (say) C
- reasonable performance

Enter libcrunch, which does the above.
$ 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

Reminiscent of Valgrind (Memcheck), but different...

- not checking memory definedness, in-boundsness, etc..
- ... in fact, assume correct w.r.t. these!
- provide & exploit run-time type information
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,
        (CHECK(__is_a(obj, "struct_commit")),
        (struct commit *)obj)))
        return -1;
    return 0;
}
Sketch of the instrumentation for C

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

Need a runtime which

- provides a fast `__is_a()` function
- ... and a few other flavours of check
- by efficiently tracking *allocations*
- ... and attaching reified type info

p.4
struct ellipse  {
    double maj, min;
    struct point  { double x, y; } ctr ;
};

also model: stack frames, functions, pointers, arrays, …

unique → “exact type” test is a pointer comparison

_is_a() is a short search over containment edges
Is it really that simple? What about...?

- untyped `malloc()` et al.
- opaque pointers, a.k.a. `void*`
- conversion of pointers to integers and back
- function pointers
- pointers to pointers
- “simulated subtyping”
- `{custom, nested}` heap allocators
- `alloca()`
- “sloppy” (non-standard-compliant) code
- unions, varargs, `memcpy()`
Is it really that simple? What about . . . ?

- **untyped malloc() et al.**
- opaque pointers, a.k.a. `void*`
- conversion of pointers to integers and back
- function pointers
- **pointers to pointers**
- “simulated subtyping”
- `{custom, nested}` heap allocators
- `alloca()`
- “sloppy” (non-standard-compliant) code
- unions, `varargs`, `memcpy()`
What data type is being malloc()’d?

Use intraprocedural “sizeofness” analysis

```c
size_t sz = sizeof(struct Foo);
/* ... */
malloc(sz);
```

Sizeofness propagates, a bit like dimensional analysis.
What data type is being malloc()’d?

Use intraprocedural “sizeofness” analysis

size_t sz = sizeof (struct Foo);
/* ... */
malloc(sz);

Sizeofness propagates, a bit like dimensional analysis.

malloc(sizeof (Blah) + n * sizeof (struct Foo))
What data type is being malloc’ed?

Use intraprocedural “sizeofness” analysis

```c
size_t sz = sizeof (struct Foo);
/* ... */
malloc(sz);
```

Sizeofness propagates, a bit like dimensional analysis.

```c
malloc(sizeof (Blah) + n * sizeof (struct Foo))
```

Dump typed allocation sites from compiler, for later pick-up
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];
    }
}

neighbor = (int **) calloc (NDIRS, sizeof (int *));
sort_eight_special ((void **) neighbor); // <-- must allow!

- solution: tolerate casts from T** to void**...
- and check writes through void**
- ... against the underlying object type (here int *[])

p.8
Performance data: C-language SPEC CPU2006 benchmarks

<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>
<tr>
<td>perlbench</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

p.9
## Experience on “correct” code

<table>
<thead>
<tr>
<th>benchmark</th>
<th>compile fixes</th>
<th>run-time false positives</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>instances</td>
</tr>
<tr>
<td>bzip2</td>
<td>0</td>
<td>48</td>
</tr>
<tr>
<td>gcc</td>
<td>1</td>
<td>$3 \times 10^5$</td>
</tr>
<tr>
<td>gobmk</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>h264ref</td>
<td>2</td>
<td>27</td>
</tr>
<tr>
<td>hmmer</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lbm</td>
<td>0</td>
<td>$5 \times 10^7$</td>
</tr>
<tr>
<td>mcf</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>milc</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sjeng</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sphinx3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
typedef double LBM_Grid[SIZE_Z*SIZE_Y*SIZE_X*N_CELL_ENTRIES];

typedef LBM_Grid* LBM_GridPtr;

#define MAGIC_CAST(v) ((unsigned int*) ((void*) (&(v))))
#define FLAG_VAR(v) unsigned int* const _aux_ = MAGIC_CAST(v)
// ...
#define TEST_FLAG(g,x,y,z,f) \
    ((MAGIC_CAST(GRID_ENTRY(g, x, y, z, FLAGS))) & (f))
#define SET_FLAG(g,x,y,z,f) \
    {FLAG_VAR(GRID_ENTRY(g, x, y, z, FLAGS)); (*_aux_) |= (f);}
Future work: shopping list for a safe implementation of C—$\epsilon$

- check `memcpy()`, `realloc()`, etc..
- check syscalls (e.g. `read()`)
- add a bounds checker (improve on SoftBound)
- add a GC (precise! improve on Boehm)
- check unions and varargs
- always initialize pointers
- check unsafe writes through `char*`
- safely address-takeable union members (!)

Good prospects for all of the above! (ask me)
Future work: shopping list for a *safe* implementation of C—\( \epsilon \)

- check `memcpy()`, `realloc()`, etc..
- check syscalls (e.g. `read()`)
- **add a bounds checker (improve on SoftBound)**
- add a GC (precise! improve on Boehm)
- check unions and varargs
- always initialize pointers
- check unsafe writes through `char*`
- safely address-takeable union members (!)

Good prospects for all of the above! (ask me)
Plenty of existing tools do bounds checking

Memcheck (coarse), ASan (fine-ish), SoftBound (fine) . . .

- detect out-of-bounds pointer/array use
- first two also catch some temporal errors

Problems remaining:

- overhead at best 50–100% (ASan & SoftBound)
- problems mixing uninstrumented code (libraries)
- false positives and negatives
- abort on false positives
typedef struct { int x[2]; char y[2];} blah;

blah z = { {0, 0}, "!" };

*(z.x + 2); // error: subobject overflow

((int*) &z)[2]; // error: after bounds—narrowing cast

**((z.x + 42) - 42); // non-error: via invalid (OOB) intermediate

((blah *) z.x)->y; // non-error: after bounds—widening cast

*(int*)( intptr_t )z.x; // non-error: via integer

*strfry (z.y ); // non-error: after uninstrumented code
Existing checkers using per-pointer metadata

```
struct ellipse {
    struct point {
        double x, y;
    } ctr;
    double maj;
    double min;
} my_ellipses[3];
```
## Existing checkers using per-pointer metadata

<table>
<thead>
<tr>
<th></th>
<th>ctr</th>
<th>maj</th>
<th>min</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>3.5</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>y</td>
<td>8.0</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

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

```
struct point {
    double x, y;
}
```
Without type information, pointer bounds 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

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

p_e = &my_ellipses[1];
```

<table>
<thead>
<tr>
<th>ellipse[3]</th>
<th>ellipse</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctr</td>
<td>x 3.5</td>
</tr>
<tr>
<td></td>
<td>y 8.0</td>
</tr>
<tr>
<td>maj</td>
<td>2</td>
</tr>
<tr>
<td>min</td>
<td>7</td>
</tr>
<tr>
<td>ctr</td>
<td>x 1.0</td>
</tr>
<tr>
<td></td>
<td>y 1.5</td>
</tr>
<tr>
<td>maj</td>
<td>5</td>
</tr>
<tr>
<td>min</td>
<td>8</td>
</tr>
<tr>
<td>ctr</td>
<td>x 6.5</td>
</tr>
<tr>
<td></td>
<td>y -2.0</td>
</tr>
<tr>
<td>maj</td>
<td>4</td>
</tr>
<tr>
<td>min</td>
<td>4</td>
</tr>
</tbody>
</table>
Given allocation type and pointer type, bounds are implicit

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

double p_d = &p_e->ctr.x
```

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctr</td>
<td>3.5</td>
<td>8.0</td>
</tr>
<tr>
<td>maj</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>min</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctr</td>
<td>1.0</td>
<td>1.5</td>
</tr>
<tr>
<td>maj</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>min</td>
<td>6.5</td>
<td>-2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Given allocation type and pointer type, bounds are implicit

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

p_f = (ellipse*) p_d
The importance of being type-aware (when bounds-checking)

```c
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)

```c
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

Write a bounds-checker consuming per-allocation metadata

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

Making it fast:

- cache bounds: make pointers “locally fat, globally thin”
- only check derivation, not use
On x86-64, use noncanonical addresses as trap reps

[Diagram showing canonical and noncanonical addresses]
int ret = 0;
for (int i = 0; i < n; ++i)
{
    struct list_node *p = malloc(sizeof(struct list_node));
    p->next = head;
    head = p;
}

for (int i = 0; i < m; ++i)
{
    unsigned out = 0;
    for (struct list_node *p = head; p; p = p->next)
    {
        out += p->x;
    }
    ret += out;
}

return ret;
Status of the bounds checking

It “works”!

- still turning the handle…

Early indications

- array-based programs: SoftBound-like perf
  - mostly +50–100%
- linked structures: much faster!
  - as expected
- building my own SoftBound-alike to compare…
Conclusions

Run-time type info enables efficient and helpful checking

- source- and binary-compatible
- low overhead type checking
- precise & fast bounds checking
- good prospects for extension

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

Thanks for your attention. Questions?