Dynamically diagnosing type errors in unsafe code

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

Computer Laboratory
University of Cambridge
“... dynamically type-safe [means] the behavior of any program, correct or not, can be easily understood in terms of the source-level language semantics.”
“... dynamically type-safe [means] the behavior of any program, correct or not, can be easily understood in terms of the source-level language semantics.”

—Ungar, Spitz and Ausch, *Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment*
“… dynamically type-safe [means] the behavior of any program, correct or not, can be easily understood in terms of the source-level language semantics.”

—Ungar, Spitz and Ausch, *Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment*
“... dynamically type-safe [means] the behavior of any program, correct or not, can be easily understood in terms of the source-level language semantics.”

—Ungar, Spitz and Ausch, *Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment*

“Type safety” [at run time] is really about *debugging*!
"… dynamically type-safe [means] the behavior of any program, correct or not, can be easily understood in terms of the source-level language semantics."

—Ungar, Spitz andAusch, *Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment*

“Type safety” [at run time] is really about *debugging*!

- clean error reports are better than corrupting errors
- … would be nice even in *unsafe languages*, like C
 Tool wanted

```c
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;  \(\leftarrow\) \(\rightarrow\)
    return 0;  \(\text{CHECK this}\)
}  \(\text{(at run time)}\)
Tool wanted

```c
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
Tool wanted

```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
- ... for real, idiomatic code in (say) C
- reasonable performance

Enter libcrunch, which does the above.
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

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
Reified, unique data types (see my Onward! 2015 paper about liballocs)

```c
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

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

Sizeofness propagates, a bit like dimensional analysis.

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

```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 *[])
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>
Experience on “correct” code

<table>
<thead>
<tr>
<th>benchmark</th>
<th>compile fixes</th>
<th>run-time false positives</th>
<th>instances</th>
<th>unique (of which...)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>total</td>
<td></td>
<td>unf</td>
</tr>
<tr>
<td>bzip2</td>
<td>0</td>
<td>48</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>gcc</td>
<td>1</td>
<td>$3 \times 10^5$</td>
<td>14</td>
<td>3</td>
</tr>
<tr>
<td>gobmk</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>h264ref</td>
<td>2</td>
<td>27</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>hmmmer</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lbm</td>
<td>0</td>
<td>$5 \times 10^7$</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>mcf</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>milc</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sjeng</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sphinx3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
A “helpful” false positive?

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—ε

- check memcpy(), realloc(), etc..
- 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)
Conclusions

Checking pointer casts can be made efficient and helpful

■ source- and binary-compatible
■ low overhead, convenient to use (e.g. no rebuilds)
■ good prospects for extension

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

Thanks for your attention. Questions?