The inevitable death of VMs

a [work-in-]progress report

Stephen Kell

stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge
Fragmentation by VM

photo: AngMoKio
Language VMs: an implementation concept whose time has gone

‘Rewrite the world’ was sane in the 70s; not now!

- no language has won
- haven’t dislodged C, or Unix-like abstractions generally
- fragmentation, escalating complexity...

What to do instead?

- ‘one VM to rule them all’? same problems!

We really want a sweet spot...

- retain per-language(-implementation) innovation
- enable inter-implementation communication
Why I don’t like VMs, Unix: what should work, but doesn’t

Why I don’t like VMs: ‘isolationism’! Can’t easily have

- calling between languages
- calling between implementations
- tools that ‘see across’ implementations

Why I don’t like Unix: lack of dynamic services!

- reflection
- serialization
- dynamic update
- garbage collection

Goal: fix all of the above. (One step at a time.) But how?
My approach: extending Unix-like operating system...

- ... to *embrace* and *integrate* language impls
- hypothesis: possible without throwing away VMs

This means *adding a meta-level to Unix*

- type information
- memory management primitives...
- ... but primitives for *description*, not implementation!

Don’t ‘define’; ‘circumscribe’ at the meta-level.

- data models, memory managers, ... plural!
- caveat: handle the common cases first (grow!)
A pluralist, whole-process meta-level

Need a new meta-level abstraction: *typed allocations*

- allocations: the hierarchical structure of memory
  
  - `mmap()`, `sbrk()`
  
  - `libc malloc()`
  
  - `custom malloc()`
  
  - `custom heap (e.g. Hotspot GC)`
  
  - `obstack (+ malloc)`
  
  - `gslice`
  
- types: borrow from DWARF debugging information
- details: Onward! ’15 paper
$ allocscc -g -o hello-allocs hello.c  # or use make-meta (wip)
$ LD_PRELOAD=liballocs_preload.so gdb -q --args ./hello-allocs

(gdb) break main
(gdb) run
main (argc=1, argv=0x7fffffffd0c8) at hello.c:5
5 printf ("Hello, world!
");
(gdb) print __liballocs_get_alloc_type (argv)
$1 = (struct uniqtype *) 0x7fff5a77760 <__uniqtype___ARR2___PTR_signed_char$8>
(gdb) print __liballocs_get_alloc_type (printf)
$2 = (struct uniqtype *) 0x7fff5a525c0 <__uniqtype___FUN_FROM___ARG0___PTR_signed_char$8>
(gdb) print *__liballocs_get_memory_mapping((void*) 0x401554, (void*)0)
$3 = {
    begin = 0x400000,
    end = 0x402000,
    prot = 5,
    flags = 2,
A quick demo, maybe
A sketch of how things should integrate (1)

cmp [ebx,<class offset>],<cached class>; test
jne <inline cache miss> ; miss? bail
mov eax,[ebx, <cached x offset>] ; hit
xor ebx,<allocator mask> ; get allocator

cmp ebx,<cached allocator prefix> ; test

jne <allocator miss> ; miss? bail

cmp [ebx,<class offset>],<cached class>; test class

jne <cached cache miss> ; miss? bail

mov eax,[ebx, <cached x offset>] ; hit
Some challenges

- pointer identification
- a better story on garbage collection
- making queries fast
Some challenges

- **pointer identification**
- → a better story on garbage collection
- making queries fast
Folklore says: ‘VMs know where the pointers are’

Can we find out where the pointers are on Unix?

- how to even begin?
- how to make it robust?
- how to make it fast?
### Identifying pointers: the easy part

```
cc -g -o hello hello.c && readelf -wi hello | column
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT_language</td>
<td>1 (ANSI C)</td>
</tr>
<tr>
<td>AT_name</td>
<td>hello.c</td>
</tr>
<tr>
<td>AT_low_pc</td>
<td>0x4004f6</td>
</tr>
<tr>
<td>AT_high_pc</td>
<td>0x400516</td>
</tr>
<tr>
<td>AT_encoding</td>
<td>5 (signed)</td>
</tr>
<tr>
<td>AT_name</td>
<td>int</td>
</tr>
<tr>
<td>AT_type</td>
<td>&lt;0x2af&gt;</td>
</tr>
<tr>
<td>AT_low_pc</td>
<td>0x4004f6</td>
</tr>
<tr>
<td>AT_high_pc</td>
<td>0x400516</td>
</tr>
<tr>
<td>AT_name</td>
<td>argc</td>
</tr>
<tr>
<td>AT_type</td>
<td>&lt;0x2b5&gt;</td>
</tr>
<tr>
<td>AT_location</td>
<td>fbreg - 20</td>
</tr>
<tr>
<td>AT_encoding</td>
<td>6 (char)</td>
</tr>
<tr>
<td>AT_name</td>
<td>argv</td>
</tr>
<tr>
<td>AT_type</td>
<td>&lt;0x7ae&gt;</td>
</tr>
<tr>
<td>AT_location</td>
<td>fbreg - 32</td>
</tr>
</tbody>
</table>
What comes out of `dumptypes`

```c
/* uniqtype for stack frame main_0x4004f6_0x400516 */
struct uniqtype __uniqtype_main_0x4004f6_0x400516 = {
    32 /* size */,
    { composite: { COMPOSITE, /* nmemb */ 2 } },
    /* members */ {
        { &__uniqtype____PTR___PTR_signed_char$8,
            /* offset */ 0 }, /* argv (size 8) */ ,
        { &__uniqtype__int$32,
            /* offset */ 12 } /* argc (size 4) (HOLE of 4B) */
    }
}

... like a struct, but describing the stack frame.

Is that hole harmless?
Actually not so easy...

Instead of

\[ fbreg = 20 \]

Compiler will sometimes say

\[ rbx = 12 \]

\[ \ldots \text{if it knows that } rbx = \text{frame base} + 8 \]

Solution:
- also consume DWARF frame information
- \ldots to recover fixed-offset relationships between regs
- solve as a graph-search problem
Some sources of holes in stack maps

- alignment (but...)  
- key ABI details: saved IP, SP  
- more ABI: use of callee-saved registers

We can fill these gaps in!
A better tool, filling in some gaps: dumpptrs

```c
/* for stack frame main_vaddrs_0x4004f6_0x400516 */

struct stored_ptrs __ptrs_hello_c_main_vaddrs_0x4004f6_0x400516
0x4004f6, 0x400516, 2 /* nstored */, /* stored */ {
    { .what = LOCAL,
        .what_info = { local: { "argv" } },
        .where = STACK,
        .where_info = { stack: { -32 } } }
    },
    { .what = CALLER_REG,
        .what_info = { caller_reg: { 6 /* register rbp */ } }
        .where = STACK,
        .where_info = { stack: { -16 } } }
} }
```

Work in progress: plugging more gaps in this tool
Some harder sources of holes in stack maps

- temporaries introduced by optimisation
- compiler whim?

i.e. things are easy at -O0, then get harder...
Compilers: a pragmatic attitude

Ideal-world fix: compilers describe their work perfectly!
- some interest, e.g. in LLVM, on debuginfo quality
- still not focused on high-assurance

More practical: binary analysis
- to *check* compiler description is correct
- to fill in details not captured (e.g. initializedness)
- ... Linux’s *objtool*
A two-pronged approach to compiler metadata

Yes, improve compilers; but also

- build tools to *identify* & *quantify* coverage gaps
- to work around unfixed gaps, dial down optimisation

Joint work ongoing with Francesco Zappa Nardelli

- **objtool-like** checking against binary instructions
- checking frame info only, initially

More to do!

- feedback loop for selective dialing down
More subtleties

1. saved registers – are they pointers or not?
2. initializedness – need a def/use analysis
3. duplicated values
4. conflated (superimposed) values
5. pointer-derived integers – are they pointers?
Subtlety 1: is it a pointer or is it an int?

/* for stack frame main_vaddrs_0x4004f6_0x400516 */

struct stored_ptrs __ptrs_hello_c_main_vaddrs_0x4004f6_0x400516

0x4004f6, 0x400516, 2 /* nstored */, /* stored */
{
    { .what = LOCAL,
      .what_info = { local: { "argv" } },
      .where = STACK,
      .where_info = { stack: { -32 } } }
    },
    { .what = CALLER_REG,
      .what_info = { caller_reg: { 6 /* register rbp */ } }
      .where = STACK,
      .where_info = { stack: { -16 } }
    } } 

‘Caller’s rbp’ could be a pointer, or not! But it’s okay...
Subtlety 2: initializedness

Just because it’s a pointer doesn’t mean it’s initialized

- def/use analysis needed in \texttt{objtool}-like tool
- pointers may be defined (have been written)
- pointers may be live (will be read)
- live but not defined $\rightarrow$ bad user code (check!)
- defined but not live $\rightarrow$ debuggers might care...

No tool yet, but on my list.
Subtlety 3: what about duplicates?

Suppose I copy from a stack slot to a register

… then break execution

Both copies might be live!

Debug info will only give us one

This also breaks the debugger’s `set var expr`
Subtleties 4 and 5: overlap and pointer-derived ints

/* uniqtype for stack frame 0x209a6b_0x556040_0x5562dc */

struct uniqtype __uniqtype___explow_cil_c_vaddrs_0x556040_0x5562dc

72 /* size */,

{ composite: { COMPOSITE, 4 } }, /* related */ {
    { memb: { &__uniqtype____PTR_rtx_def, 0 } } } /* (anonymous) */

    { memb: { &__uniqtype__uint$64, 8 } } } /* (anonymous) @209a8f

    { memb: { &__uniqtype__int$64, 16 } } } /* (anonymous) @209aff

    { memb: { &__uniqtype____PTR_rtx_def, 16 } } } /* (anonymous)

}

Hypothesis: this is less bad than it seems.

- + ask me about static analysis of pointer-derived ints
Not shown, but ...
Subtlety: what about conflations?

If two program values have the same bit-pattern at run time they may be conflated by the compiler.

Even if one of them is a pointer and the other an int!
A more sophisticated tool: dumpptrs

```c
/* for stack frame main_vaddrs_0x4004f6_0x400516 */

struct stored_ptrs __ptrs_hello_c_main_vaddrs_0x4004f6_0x400516

 0x4004f6, 0x400516, 2 /* nstored */, /* stored */ {
    { .what = LOCAL,
      .what_info = { local: { "argv" } },
      .where = STACK,
      .where_info = { stack: { -32 } } }
  },
  { 
    .what = CALLER_REG,
    .what_info = { caller_reg: { 6 /* register rbp */ } }
    .where = STACK,
    .where_info = { stack: { -16 } } 
 } 

Work in progress: plugging more gaps in this tool
```
Another example (1)

```c
const int n = 100;
const int m = 1000;

int *f(void)
{
    int *arr = calloc(n * m, sizeof(int));
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            arr[i * m + n] = rand();
        }
    }
    return arr;
}
```
Another example (2)

40055a: callq 400430 <calloc@plt>
40055f: mov %rax,%r13
400562: mov $0x0,%ebx
400567: jmp 40058d <f+0x47>
400569: movslq %ebx,%rdx
40056c: lea 0x0(%r13,%rdx,4),%rbp
400571: callq 400440 <rand@plt>
400576: mov %eax,0x0(%rbp)
400579: add $0x1,%ebx
40057c: cmp %r12d,%ebx
40057f: jne 400569 <f+0x23>
400581: mov %r12d,%ebx
400584: cmp $0x186a0,%r12d
40058b: je 400593 <f+0x4d>
Things to try:

- force-enabling debug info for temporaries (if pointers?)
- selectively disabling optimisation passes
- … and (hackaround) re-doing as source-to-source?

Things we’ll need

- catch pointer-derived integers
- formal rules about how pointers may be computed from integers
Some things missing from Unix:

- fine-grained *object-like* notion
- semantic metadata ("type info")
- efficient binding from object to metadata
Some things missing from Unix:

- fine-grained *object-like* notion
- semantic metadata ("type info")
- efficient binding from object to metadata

... so I’ve built liballocs, which adds them!

- core abstraction: *allocations*
- implements ≈ a meta-object protocol *process-wide*
- "type" metadata
- based on existing VM-like *rudiments* within Unix
  → very compatible, very general
Compatibility with existing language implementations

Two cases!

Unix-style compilation toolchains (C, C++, Fortran, ...)

- **augment** the toolchain
- mostly generic, + a little per-language effort
- mostly done, working (esp. for C)

Existing language VMs

- retrofit onto liballocs APIs
- hypothesis: small changes only
- mostly future work
Retrofitting a VM: the shopping list

Generate uniqtypes

- need not be 1:1 with “type” in the language

Implement liballocs meta-protocol

- . . . some pre-fab options available

Notify dynamic loader of JITted code

- extra goodie: libdlbind can do this

Whole-process binding. . .

- slow path: just use the meta-protocol
- fast path: no change! i.e. affinity for own objects

p. 38
Core runtime and toolchain extensions work well

- really!

Next step: actually retrofit one or more VMs

- whole-process reflection
- whole-process tools (debuggers, profilers, . . .)
- interop without FFIs (improving the node use-case)

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

- please get in touch

Thanks for listening... questions?