How to Shadow Every Byte of Memory Used by a Program

Nicholas Nethercote – National ICT Australia
Julian Seward – OpenWorks LLP
• Shadow every byte of memory with another value that describes it

• This talk:
  – Why shadow memory is useful
  – How to implement it well
## Examples

<table>
<thead>
<tr>
<th>Tool(s)</th>
<th>Shadow memory helps find…</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bugs</strong></td>
<td></td>
</tr>
<tr>
<td>Memcheck, Purify</td>
<td><strong>Memory errors</strong></td>
</tr>
<tr>
<td>Eraser, DRD, Helgrind, etc.</td>
<td>Data races</td>
</tr>
<tr>
<td>Hobbes</td>
<td>Run-time type errors</td>
</tr>
<tr>
<td>Annelid</td>
<td>Array bounds violations</td>
</tr>
<tr>
<td><strong>Security</strong></td>
<td></td>
</tr>
<tr>
<td>TaintCheck, LIFT, TaintTrace</td>
<td>Uses of untrusted values</td>
</tr>
<tr>
<td>“Secret tracker”</td>
<td>Leaked secrets</td>
</tr>
<tr>
<td><strong>Properties</strong></td>
<td></td>
</tr>
<tr>
<td>Redux</td>
<td>Dynamic dataflow graphs</td>
</tr>
<tr>
<td>DynCompB</td>
<td>Invariants</td>
</tr>
<tr>
<td>pinSEL</td>
<td>System call side-effects</td>
</tr>
</tbody>
</table>
Shadow memory is difficult

• Performance
  – Lots of extra state, many operations instrumented

• Robustness

  original values  shadow values

  squeeze!

  address space

• Trade-offs must be made
An example tool: Memcheck
Memcheck

• Three kinds of information:
  – A (“addressability”) bits: 1 bit / memory byte
  – V (“validity”) bits: 1 bit / register bit, 1 bit / memory bit
  – Heap blocks: location, size, allocation function

• Memory information:

original memory byte  \[0110 0101\]
shadow memory \[VVVVVVVV\]  A

V bits only used if A bit is “addressable”
A simple implementation
Basics (I)

SM₁

<table>
<thead>
<tr>
<th>0</th>
<th>VVVVVVV</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VVVVVVV</td>
<td>A</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>65535</td>
<td>VVVVVVV</td>
<td>A</td>
</tr>
</tbody>
</table>

NoAccess DSM

<table>
<thead>
<tr>
<th>0</th>
<th>------0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>0</td>
</tr>
</tbody>
</table>

SM₂

<table>
<thead>
<tr>
<th>VVVVVVV</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>VVVVVVV</td>
<td>A</td>
</tr>
</tbody>
</table>

PM

<table>
<thead>
<tr>
<th>0KB</th>
<th>64KB</th>
<th>128KB</th>
<th>...</th>
<th>3904KB</th>
<th>3968KB</th>
<th>4032KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>0001</td>
<td>FFFF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Basics (II)

• Multi-byte shadow accesses:
  – Combine multiple single-byte accesses
  – Complain if any unaddressable bytes accessed
  – Values loaded from unaddressable bytes marked as defined

• Range-setting (**set_range**)
  – Loop over many bytes, one at a time

• Range-checking
  – E.g.: `write(fd, buf, n)` -- check n bytes in buf

• Slow-down: 209.6x
Complications

- Corruption of shadow memory
  - Possible with a buggy program
  - Originally used x86 segmentation, but not portable
  - Keep original and shadow memory far apart, and pray

- 64-bit machines
  - Three- or four-level structure would be slow
  - Two level structure extended to handle 32GB
  - Slow auxiliary table for memory beyond 32GB
  - Better solution is an open research question
Four optimisations
#1: Faster loads and stores

- Multi-byte loads/stores are very common
  - N separate lookups accesses is silly (where N = 2, 4, or 8)

- If access is aligned, fully addressable
  - Extract/write V bits for N shadow bytes at once
  - Else fall back to slow case: 1 in a 1000 or less

- Slow-down: 56.2x
  - 3.73x faster
#2: Faster range-setting

- Range-setting large areas is common
  - Vectorise `set_range`
  - 8-byte stride works well

- Replacing whole SMs
  - If marking a 64KB chunk as `NoAccess`, replace the SM with the `NoAccess` DSM
  - Add `Defined` and `Undefined` DSMs
  - Large read-only code sections covered by `Defined` DSM

- Slow-down: 34.7x
  - 1.62x faster, 1.97x smaller
#3: Faster SP updates

- Stack pointer (SP) updates are very common
- Inc/dec size often small, statically known
  - E.g. 4, 8, 12, 16, 32 bytes
- More specialised range-setting functions
  - Unrolled versions of `set_range()`
- Slow-down: 27.2x
  - 1.28x faster
#4: Compressed V bits

* Partially-defined bytes (PDBs) are rare
  - Memory: 1 A bit + 8 V bits → 2 VA bits
  - Four states: NoAccess, Undefined, Defined, PartDefined
  - Full V bits for PDBs in secondary V bits table
  - Registers unchanged -- still 8 V bits per byte

* Slow-down: 23.4x
  - 4.29x smaller, 1.16x faster

* Obvious in hindsight, but took 3 years to identify
Discussion

• Optimising principles:
  – Start with a simple implementation
  – Make the common cases fast
  – Exploit redundancy to reduce data sizes

• Novelty?
  – First detailed description of Memcheck’s shadow memory
  – First detailed description of a two-level table version
  – First detailed evaluation of shadow memory
  – Compressed V bits
Evaluation
Robustness

• Two-level table is very flexible
  – Small shadow memory chunks, each can go anywhere

• Earlier versions required large contiguous regions
  – Some programs require access to upper address space
  – Some Linux kernels have trouble mmap’ing large regions
  – Big problems with Mac OS X, AIX, other OSes

• Memcheck is robust
  – Standard Linux C and C++ development tool
  – Official: Linux, AIX; experimental: Mac OS X, FreeBSD
SPEC 2000 Performance

<table>
<thead>
<tr>
<th>Tool</th>
<th>Slow-down</th>
<th>Relative improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>No instrumentation</td>
<td>4.3x</td>
<td></td>
</tr>
<tr>
<td>Simple Memcheck</td>
<td>209.6x</td>
<td></td>
</tr>
<tr>
<td>+ faster loads/stores</td>
<td>56.2x</td>
<td>3.73x faster</td>
</tr>
<tr>
<td>+ faster range-setting</td>
<td>34.7x</td>
<td>1.62x faster, 1.97x smaller</td>
</tr>
<tr>
<td>+ faster SP updates</td>
<td>27.2x</td>
<td>1.28x faster</td>
</tr>
<tr>
<td>+ compressed V bits</td>
<td>23.4x</td>
<td>1.16x faster, 4.29x smaller</td>
</tr>
<tr>
<td>Overall improvement</td>
<td></td>
<td>8.9x faster, 8.5x smaller</td>
</tr>
</tbody>
</table>

- Shadow memory causes about half of Memcheck’s overhead
Performance observations

• Performance is a traditional research obsession
  “The subjective issues are important – ease of use and robustness, but performance is the item which would be most interesting for the audience.” (my emphasis)

• Users: slowness is #1 survey complaint
  – But most user emails are about bugs or interpreting results
  – Zero preparation is a big win

• Cost/benefit
  – People will use slow tools if they are sufficiently useful
Alternative implementation

- “Half-and-half”
  - Used by Hobbes, TaintTrace, (with variation) LIFT

<table>
<thead>
<tr>
<th>Original Memory</th>
<th>Shadow Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a'</td>
</tr>
<tr>
<td>b</td>
<td>b'</td>
</tr>
<tr>
<td>c</td>
<td>c'</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>z</td>
<td>z'</td>
</tr>
</tbody>
</table>

- Compared to two-level table
  - Faster
  - Not robust enough for our purposes
If you remember nothing else...
Take-home messages

• Shadow memory is powerful
• Shadow memory can be implemented well
• Implementations require trade-offs