Metal error checkers

Checking for memory problems

```c
on null_checker {
  decl { scalar } sz; // match any scalar
  decl { const int } retv; // match const int
  decl { any_ptr } vt; // match any ptr
  // 'true' specifies 'v' will have a state
  state DECL { any_ptr } v;

  // Associate allocated memory with unknown
  // state until compared to null.
  start, v.all:
  // set v's state on true path to "null",
  // on false path to "not_null"
  { (v = (async_malloc(sz)) == 0) }
    => truev.not_null, falsev.not_null
  // vice versa
  | { (v = (async_malloc(sz)) != 0) }
    => truev.not_null, falsev.not_null
  // unknown state until observed.
  | { v = (async_malloc(sz)) } => v.unknown;

  // Kill comparisons on variables in
  // states "unknown", "null", and "not_null."
  v.unknown, v.null, v.not_null:
  (v == 0) =>
  true = v.null, false = v.not_null;

  // Catch error path leaks by warning when
  // a non-null, non-freed variable gets to a
  // return of a negative integer.
  v.unknown, v.not_null: { return retv; } =>
    { if(mk_int_cst(retv) < 0) 
      err("Error path leak!");
    }

  // No dereferences of null or unknown pths.
  v.null, v.unknown: { *(any *v) } =>
    { err("Using ptr illegally!");
    }

  // Allow free of all non-freed variables.
  v.unknown, v.null, v.not_null:
    { free(v); } => v.freed;

  // Check for double free and use after free.
  v.freed:
    { free(v); } => { err("Dup free!");
      | { v } => { err("Use-after-free!");
    }

  // Overwriting v's value kills its state
  v.all: { v = vt } => v.ok;
}
```
Memory allocation

<table>
<thead>
<tr>
<th>Violation</th>
<th>Linux</th>
<th></th>
<th>OpenBSD</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bug</td>
<td>False</td>
<td>Bug</td>
<td>False</td>
</tr>
<tr>
<td>No check</td>
<td>79</td>
<td>9</td>
<td>49</td>
<td>2</td>
</tr>
<tr>
<td>Error leak</td>
<td>44</td>
<td>49</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Use after Free</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Underflow</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>132</td>
<td>61</td>
<td>52</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2: Error counts for Linux and OpenBSD. The checker was applied 4268 times in Linux and 464 times in OpenBSD.

Improper blocking

<table>
<thead>
<tr>
<th>Check</th>
<th>Local</th>
<th>Global</th>
<th>False Pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupts</td>
<td>18</td>
<td>42</td>
<td>4</td>
</tr>
<tr>
<td>Spin Lock</td>
<td>21</td>
<td>42</td>
<td>4</td>
</tr>
<tr>
<td>Module</td>
<td>22</td>
<td>~53</td>
<td>~2</td>
</tr>
<tr>
<td>Total</td>
<td>61</td>
<td>~137</td>
<td>~10</td>
</tr>
</tbody>
</table>

Table 3: Results for checking if kernel routines block (1) with interrupts disabled (“Interrupts”), (2) while holding a spin lock (“Spin Lock”), or (3) in a way that causes a module race (“Module”). We divide errors into whether they needed local or global analysis. Local errors were due to direct calls to blocking functions; global errors reached a blocking routine via a multi-level call chain. The global analysis results for Module are marked as approximate since they have not been manually confirmed.
Linux synchronization checker

<table>
<thead>
<tr>
<th>Condition</th>
<th>Applied</th>
<th>Bug</th>
<th>False Pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holding lock</td>
<td>~5400</td>
<td>29</td>
<td>113 (90)</td>
</tr>
<tr>
<td>Double lock</td>
<td>-</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Double unlock</td>
<td>-</td>
<td>1</td>
<td>20 (18)</td>
</tr>
<tr>
<td>Intr disabled</td>
<td>~5800</td>
<td>44  (43)</td>
<td>63 (54)</td>
</tr>
<tr>
<td>Bottom half</td>
<td>~180</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>Bogus flags</td>
<td>~3200</td>
<td>4</td>
<td>49 (24)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>-</td>
<td>83  (82)</td>
<td>260 (201)</td>
</tr>
</tbody>
</table>

Table 4: Results of running the Linux synchronization primitives checker on kernel version 2.3.99. The **Applied** column is an estimate of the number of times the check was applied. We skipped twelve warnings that were difficult to classify. The parenthesized numbers show the changes when the two files with the most false positives are ignored.

Errors found by Metal

<table>
<thead>
<tr>
<th>Check</th>
<th>Errors</th>
<th>False Positives</th>
<th>Uses</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Side-effects(§ 4.1)</td>
<td>14</td>
<td>2</td>
<td>199</td>
<td>25</td>
</tr>
<tr>
<td>Static assert(§ 4.2)</td>
<td>5</td>
<td>0</td>
<td>1739</td>
<td>100</td>
</tr>
<tr>
<td>Stack check(§ 4.3)</td>
<td>10+</td>
<td>0</td>
<td>332K</td>
<td>53</td>
</tr>
<tr>
<td>User-per(§ 5.1)</td>
<td>18</td>
<td>15</td>
<td>187</td>
<td>68</td>
</tr>
<tr>
<td>Allocation(§ 5.2)</td>
<td>184</td>
<td>64</td>
<td>4732</td>
<td>60</td>
</tr>
<tr>
<td>Block(§ 6.2)</td>
<td>123</td>
<td>8</td>
<td>-</td>
<td>131</td>
</tr>
<tr>
<td>Module(§ 6.3)</td>
<td>~75</td>
<td>2</td>
<td>-</td>
<td>133</td>
</tr>
<tr>
<td>Miner(§ 7)</td>
<td>52</td>
<td>201</td>
<td>14K</td>
<td>64</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>~311</td>
<td>~292</td>
<td>-</td>
<td>669</td>
</tr>
</tbody>
</table>

Table 6: The results of MC-based checkers summarized over all checks. **Error** is the number of errors found, **False Positives** is the number of false positives, **Uses** is the number of times the check was applied, and **LOC** is the number of lines of metal code for the extension (including comments and whitespace).
Interrupt checking

// Mark paths containing non-returning function as dead.
em_call_backs local:
    decl any_free_call;
    decl any_arg arg;

start: ( calls langs ) mono
    call vs = to_identifier(all);
    if ( vs -> no_interrupts() )
        no_set_psudo_lb(all);
    else if ( vs -> (int, pmag) )
        set_psudo_lb(all);

// Include useful macros.

// Extension data and code: accessible from EN
// patterns matching callouts and actions.
em_header ( static int enables, disables; )

em cli_set_consent local:
    decl any_edge flags;

// Run at beginning of each function.
init: ( enable = disables = 0; );

// Run at the end of each function.
final: ( if (enables = 0 && disables = 0)
    emit("CLI_ST1": enable hdl to disable hdl, enables, disables);
    );

// Pattern to match the various ways to
// disable interrupts.
pat disable =
    ( cli( ); )
    | ( _cli( ); )
    | ( localible_disable( ); )
    | ( localible_disable( ); )
    | ( ...
    | ( ... to enable interrupts.
    pat enable =
    ( en( ); )
    | ( ... to enable interrupts.
    pat enable =
    ( en( ); )
    | ( ...

Issues/ideas

- Mini-language for writing checkers
- Evaluate state machines along each edge
- Interprocedural analysis
- False positives
- Ranking of results
Mini-language for writing checkers

• pattern matching against AST
  – before macro expansion?
• Selected variables/patterns can have associated state
• state transitions based on pattern matching
• Can also evaluate arbitrary C code

evaluate state machines along each edge

• enumerate all paths from entry to exit
• Perform DFS of CFG
• Don’t recurse if finite state matches previous state at node
  – also matching information used for impossible paths, ...

...?
Interprocedural analysis

- earlier papers did little or no interprocedural analysis
  - recent addition
- doesn’t seem to handle function pointers or polymorphic method invocation
- Examine all functions and build call graph
- Functions with no callers are roots
  - check all roots
- Refine/restore information across procedure boundaries
  - Don’t track global information?
- Cache and reuse results

False positives

- impossible paths
- killing expressions
  - what happens to tracked state of a[i] when i is modified
- synonyms
Ranking of results

• generic ranking
• statistical ranking of uncertain hypotheses