Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions

Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem
Stanford University
Checking System Rules

• Model checkers or theorem provers
  – Pro: Finds error difficult to detect.
  – Con: Specifications are difficult to construct and do not mirror the code.

• Testing
  – Pro: Simpler and no mirroring problem.
  – Con: Dynamic

• Manual Inspection
Compiler Extension

- Compilers work with the code itself. Specification is not needed.
- Static analysis examines all execution paths.
- Need to extend the compiler to include system specific semantic information
System Rules

- Operations follow a given order
- Operations obey contextual restrictions
- Sample System Rules
  - Never/always do X
  - Always do X before/after Y
  - Never do X before/after Y
  - In situation X do (not do) Y
Metal Overview

```c
#include "linux-includes.h"

sm check_interrupts {
    // Variables used in patterns
    decl { unsigned } flags;

    // Patterns: enable/disable
    pat enable = { sti(); } |
        { restore_flags(flags); };
    pat disable = { cli(); } |

    // States
    is_enabled: disable ==> is_disable
        | enable ==> { err("double enable"); };
    is_disable: enable ==> is_enabled
        | disable ==> { err("double disable"); } |
        $end_of_path$ ==> { err("exiting w/intr disabled"); };
}
```

```c
/* From Linux 2.3.99
drivers/block/raid5.c */

static struct buffer_head * get_free_buffer(
    struct stripe_head *sh,
    int b_size)
{
    struct buffer_head * bh;
    unsigned long flags;

    save_flags(flags);
    cli();
    if ((bh = sh->buffer_pool) == NULL)
        return NULL;
    sh->buffer_pool = bh->b_next;
    bh->b_size = b_size;
    restore_flags(flags);
    return bh;
}
```
Simple Example: assert

- Expression in an assert should not have non-debugging side effects.
- Assert condition should not fail.
Assertion side-effects

```c
#include <assert.h>

sm Assert flow_insensitive {
    decl { any } expr, x, y, z;
    decl { any_call } any_fcall;
    decl { any_args } args;

    start: { assert(expr); } ==> { mgk_expr_recurse(expr, in_assert); };

    in_assert:
        { any_fcall(args) } ==> { err("function call"); }
        | { x = y } ==> { err("assignment"); }
        | { z++ } ==> { err("post-increment"); }
        | { z-- } ==> { err("post-decrement"); }
...
```
Checking assertions statically

- Use data-flow information to track the value of scalar variables.
- Evaluate the assertion expression against these values
- Implemented using 100 lines of Meta
Temporal Ordering

• Many system have requirement in the order of operations.

• Sample Extension
  – Argument copying in kernel system calls
  – Memory management
    • Check pointer after memory allocation
    • Memory not used after free
    • Deallocate memory on error
    • Size allocated is not less than assigned pointer
Global Rules

- Kernel cannot lock with interrupt disabled or while holding spin lock
  - Compute list of possibly blocking routine
  - Check if potentially blocking routine is called with interrupt disabled or while holding lock
- Linux module loading and unloading
## Result Summary

<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</td>
<td>14</td>
<td>2</td>
<td>199</td>
<td>25</td>
</tr>
<tr>
<td>Static assert</td>
<td>5</td>
<td>0</td>
<td>1759</td>
<td>100</td>
</tr>
<tr>
<td>Stack check</td>
<td>10</td>
<td>0</td>
<td>332K</td>
<td>53</td>
</tr>
<tr>
<td>User-ptr</td>
<td>18</td>
<td>15</td>
<td>187</td>
<td>68</td>
</tr>
<tr>
<td>Allocation</td>
<td>184</td>
<td>64</td>
<td>4732</td>
<td>60</td>
</tr>
<tr>
<td>Block</td>
<td>123</td>
<td>8</td>
<td>-</td>
<td>131</td>
</tr>
<tr>
<td>Module</td>
<td>~75</td>
<td>2</td>
<td>-</td>
<td>133</td>
</tr>
<tr>
<td>Mutex</td>
<td>82</td>
<td>201</td>
<td>14K</td>
<td>64</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>~511</td>
<td>~292</td>
<td>-</td>
<td>669</td>
</tr>
</tbody>
</table>
Conclusion

• Many systems have high level system rules that generic compiler cannot check

• Use meta information in the form of compiler extensions to check for these system rules

• Has advantages over verification, testing, and manual inspection.
Questions?