Using System-Specific Compiler Extensions to Find Errors in Systems Code

Dawson Engler
Ben Ches, Andy Chou, Seth Hallem
Stanford University

Another approach

- Observation: rules can be checked with a compiler scan source for "relevant" acts check if they make sense E.g., to check "disabled interrupts must be re-enabled:" scan for calls to disable/enable(), check that they match, not done twice
- Main problem: compiler has machinery to automatically check, but not knowledge implementor has knowledge but not machinery
- Meta-level compilation (MC): give implementors a framework to add easily-written, system-specific compiler extensions

Meta-level compilation (MC)

- Implementation:
  Extensions dynamically linked into GNU gcc compiler
  Applied down all paths in program source
  E.g. 64 line extension to check disable/disable (82 bugs)
- Static detection of real errors in real systems:
  600+ bugs in Linux, OpenBSD, FLASH, Xok exokernel
  most extensions < 100 lines, written by system outsiders

A bit more detail

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

am chk_interrups()
    am { unsigned } flags;
    // named patterns
    pat enable = { sti(); };
        | { restore_flags(flags); }; 
    pat disable = { cli(); }; 

    // states
    is_enabled: disable -> is_disabled 
    | enable -> { err("double enable")}; 
    is_disabled: enable -> is_enabled 
    | disable -> { err("double disable"); } 
    | $end_of_path: -> 
    | { err("exiting w/intr disabled!"); } 
    ; }
```

"X before Y" rule: system call pointers

- Applications are evil OS much check all input pointers before use one missing check = security hole
- MC checker:
  Bind syscall ptr's to "tainted" state tainted vars only touched w/ "safe" routines
  or: explicit check to make "clean"

"X before Y" rule: system call pointers

- Applications are evil OS much check all input pointers before use one missing check = security hole
- MC checker:
  Bind syscall ptr's to "tainted" state tainted vars only touched w/ "safe" routines
  or: explicit check to make "clean"

Deriving specification from common usage

- Problem: difficult to specify all user pointers so see what code usually does, deviations probably errors if ever pass ptr to paranoid routine, make sure always do
- Found 5 security errors in Linux
  Canonical example: hole in an "ioctl" routine for some obscure device driver.
  /* drivers/usb/evdev.c */
  static int evdev_ioctl(..., unsigned long arg) {
    ... switch (cmd) {
    case EVIOCGVERSION:
      return put_user(EV_VERSION, ____u32 * ) arg);
    case EVIOCSETBACK:
      return copy_to_user( & dev_id, void /* arg, sizeof(struct input_id) */);
    }
  }

Kernel alloc/dealloc rules

- Must check that alloc succeeded
- Must allocate enough space
- Must not use after free()
- Must free alloc'd object on error:
  /* from drivers/char/tsa6300.c */
  client = kmalloc(sizeof *client, GFP_KERNEL);
  if (client) return -ENOMEM;
  ...
  tea = kmalloc(sizeof *tea, GFP_KERNEL);
  if (!tea) return -ENOMEM;
  ...
  MOD_INC_USE_COUNT; /* bonus bug: kmalloc could sleep */

Stripped-down kernel malloc/free checker

decl { scalar } sz; // match any scalar
decl { const int } retv; // match const ints
state decl { any_ptr } v; // match any pointer, can bind to a state

// Bind malloc results to "unknown" until observed
start: { v = (any)malloc(sz) } = = > v.unknown
  { (free v) } = = > v.freed;
// can compare in states unknown, null, not null
v.unknown, v.null, v.not_null:
  ... { (v == 0) } = = > true = v.null, false = v.not_null
  ... { (v != 0) } = = > true = v.not_null, false = v.null;
// Cannot reach error path with unknown or not-null
v.unknown, v.not_null: { return retv; } = = >
  { (if (mgk_int cat(retv) < 0) err("Error path leak!");
// No dereferences of null, freed, or unknown ptrs.
  v.null, v.freed v.unknown:
    { (any w) } = = > err("Using ptr illegally!");

"In context Y, don't do X": blocking

- Linux: if interrupts are disabled, or spin lock held, do not call an operation that could block.
- MC checker:

  Compute transitive closure of all potentially blocking fn's
  Hit disable/lock: warn of any calls
  123 errors, 8 false pos

  /* drivers/net/pmcie/wavelan rx.c */
  spin_lock_irqsave ( &lock, flags); /* 1889 */
  switch (cmd) ...
  case SIDOCENPRIV:
    ...
  if (copy_to_user(vq->v.data pointer,...) ) /* 2305 */
    ret = -EFAULT;

Some amusing bugs

- No check (130 errors, 11 false pos). Worse case (many uses):
  /* include/linux/coda_linux.h: CODA_ALLOC */
  ptr = (cast)malloc((unsigned long) size);...
  if (ptr == 0) printk("kernel malloc returns 0\n");
  memset( ptr, 0, size );
- use after free (14 errors, 3 false pos): 5 cut&paste of
  /* drivers/isanp/chip/chip_init_dev */
  kmalloc((unsigned char*)dev->sh_mem);
  release_mem_region(dev->ph_mem, ...);
- wrong size (2 errors)
  /* drivers/parport/daisy.c: add_dev */
  newdev = kmalloc (GFP_KERNEL, sizeof(struct daisysdev));

Example: statically checking assert

- Assert(x) used to check "x" at runtime. Abort if false compiler oblivious, so cannot analyze statically
- Use MC to build an assert-aware extension

msg.len = 0;
... assert(msg.len <= 0)
  line 211: assert failure!

- Result: found 5 errors in FLASH
- Common: code cut&paste from other context
- Manual detection questionable: 300-line path explosion between violation and check
- General method to push dynamic checks to static
Result overview

<table>
<thead>
<tr>
<th>Check</th>
<th>Errors</th>
<th>False pos</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static assert</td>
<td>5</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Stack check</td>
<td>10+</td>
<td>0</td>
<td>53</td>
</tr>
<tr>
<td>Allocation</td>
<td>184</td>
<td>64</td>
<td>60</td>
</tr>
<tr>
<td>Blocking</td>
<td>123</td>
<td>8</td>
<td>131</td>
</tr>
<tr>
<td>Module race</td>
<td>~75</td>
<td>2</td>
<td>133</td>
</tr>
<tr>
<td>Mutex</td>
<td>82</td>
<td>201</td>
<td>64</td>
</tr>
<tr>
<td>FLASH</td>
<td>34</td>
<td>69</td>
<td>553</td>
</tr>
<tr>
<td>Total (others)</td>
<td>~545</td>
<td>~226</td>
<td>~1100</td>
</tr>
</tbody>
</table>

Conclusions

- **MC goal:** make programming much more powerful
  - How: Raise compilation from level of programming language to the "meta level" of the systems implemented in that language
  - MC works well in real, heavily tested systems
  - We found bugs in every system we've looked at.
  - Over 600 bugs in total, many capable of crashing system
  - Easily written by people unfamiliar w/ checked system
  - Currently: making correctors, using domain-knowledge to extract verifiable specs, deriving errors by usage deviations, performing meta-level optimization...

Conclusions

- **Meta-level compilation:**
  - Make compilers aggressively system-specific
  - Easy: digest sentence fragment, write checker/optimizer
  - Result: Static, precise, immediate error diagnosis
  - As outsiders found errors in every system looked at
  - Over 600 bugs, many capable of crashing system
  - Currently: making correctors, using domain-knowledge to extract verifiable specs, deriving errors by usage deviations, performing meta-level optimization...

Bugs as deviant behavior

- **One way to find bugs:** have a deep understanding of code semantics, detect when code makes no sense. Hard.
- **Easier:** see what code usually does: deviations probably bugs
  - x protected by lock(a) 1000 times, by lock(b) once, probably an error
  - lock(a); lock(a); lock(a); lock(b); lock(a); x++; x++; x++; x++; x++;
  - unlock(a); unlock(a); unlock(a); unlock(a); unlock(b); unlock(a);
  - Find inverses by looking for common pairings
  - More general: derive temporal orderings. Use machine learning to derive more sophisticated patterns?

What to do when static analysis too weak?

- **Static analysis works in some cases, not well in others**
  - hit undecidable problems with loop termination conditions, data values, pointers,...
- **Alternative:**
  - use domain-specific slicing to extract spec from code
  - run through verifier
- **Main lever:** a little domain knowledge goes a long way
  - e.g., strip out Linux TCP finite-state-machine by keying off of variable "sk-state"
  - Real example: checking FLASH code

Extracting specs from FLASH code

- **Embedded sw for cache coherence in FLASH machine**
  - errors crash or deadlock machine: can take week to track typical protocol: 18K lines of hairy C code
- **Extract specifications from source by simple slicing**
  - found 9 errors in code despite 5+ years of heavy testing and formal verification!
- **How?**
  - Given list of data structure fields and message operations, slice out all relevant operations
  - Compose with specification (manual) boilerplate
  - run through Murphi model checker
  - Levers: aliasing and globals, but in a stylized way that we can mostly ignore. 4 loops in code.
FLASH vs Murphi

FLASH

HANDLER_GLOBALS(header.nl:len) = Len_Cacheline;
if (! HANDLER_GLOBALS(h.hl.Pending)) {
    if (! HANDLER_GLOBALS(h.hl.Dirty)) {
        PI_SEND(P_DATA, P_FREE, P_SWAP, ...);
        HANDLER_GLOBALS(h.hl.Local) = 1;
    } else {
        assert(! HANDLER_GLOBALS(h.hl.List));
        assert(! HANDLER_GLOBALS(h.hl.RealPtrs));
    }
} else {
    assert(! HANDLER_GLOBALS(h.hl.Pending));
    assert(! HANDLER_GLOBALS(h.hl.Dirty));
    PI_SEND(P_DATA, P_FREE, P_SWAP, ...);
    HANDLER_GLOBALS(h.hl.Local) = 1;
} / * ... deleted 14 lines */

Murphi

nh.len := len_cacheline;
if (! (DN.Pending = 0)) then
    if (! (DN.Dirty = 0)) then
        assert(nh.len != len_nodeata);
        mbResult := pi_send_func(src, PI_Put);
        DH.Local := 1;
    else
        assert((DN.List = 0));
        assert((DN.RealPtrs = 0));

Checkers into Correctors

- Problem: big system, lots of bugs may not be your system or take too long to fix manually
- Can turn some classes of checkers into correctors:
  - "Do not allocate large variables on kernel stack": if you hit a violation, rewrite code to dynamically allocate var
  - "Do not call blocking memory alloc with interrupts disabled": hoist allocation out
  - "On error paths, rollback side-effects": dynamically track what these are, and reverse.
- Interesting: trade dynamic checks for simplicity

MC optimization

- Optimization rules similar to checking:
  - "to save an instruction when setting a message opcode, xor it with the new and old (msg opcode = (new ^ old)); replace quicksort with radix sort when sorting integers"
- Common rule: "In situation X, do Y rather than Z":
  - "if a variable is not modified, protect using read locks"
  - "modifying q with read lock"

MC analysis vs. traditional compiler analysis

- Meaning more apparent + domain-specific knowledge
  - Bigint a, b, c;
  - set(a, 3);
  - mul(b, a, c);
    print("B1");
  - mul(c, b);
    print("B2", bigint_to_str(c));
- Easier to bound side-effects: use knowledge of abstract state to ignore many concrete actions
- Allowing less of a problem
  - typical: opaque handles vs normal mess of pointers
- Operations more coarse grain
  - read/write() vs load/store; matrix ops vs +/-