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


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


## Checking systems software

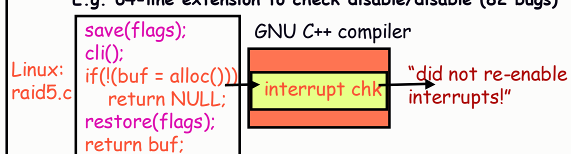
- Systems software has many ad-hoc restrictions:
  - "acquire lock L before accessing shared variable X"
  - "do not allocate large variables on 6K kernel stack"
- Error = crashed system. How to find errors?
  - Formal verification
    - + rigorous
    - - costly + expensive. *Very* rare to do for software
  - Testing:
    - + simple, few false positives
    - - requires running code: doesn't scale & can be impractical
  - Manual inspection
    - + flexible
    - - erratic & doesn't scale well.
  - What to do??

## 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 g++ compiler
  - Applied down all paths in input program source
  - E.g. 64-line extension to check disable/disable (82 bugs)

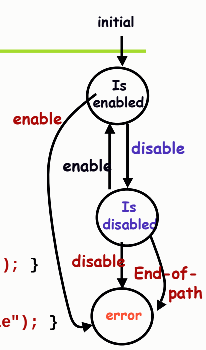



- Static detection of real errors in real systems:
  - 600+ bugs in Linux, OpenBSD, FLASH, Xok exokernel
  - most extensions < 100 lines, written by system outsiders

## A bit more detail

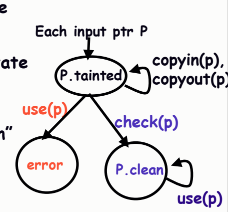
```
{ #include "linux-includes.h" }
sm chk_interrupts {
 decl { unsigned } flags;
 // named patterns
 pat enable = { sti(); }
       | { restore_flags(flags); };
 pat disable = { cli(); };

 // 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"




```
/* from sys/kern/disk.c */
int sys_disk_request(…
  struct buf *reqbp, u_int k) {
  ...
  /* bypass for direct scsi commands */
  if (reqbp->b_flags & B_SCSICMD)
     return sys_disk_scsicmd (sn, k, reqbp);
```

## 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 EVIOCGID: /* copy_to_user(to, from)! */
         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/tea6300.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_cst(retv) < 0)  err("Error path leak!"); };
// No dereferences of null, freed, or unknown ptrs.
v.null, v.freed v.unknown:
          { *(any *)v } ==>  { err("Using ptr illegally!"); };
```

## Some amusing bugs

- No check (130 errors, 11 false pos). Worse case (many uses):

```
/* include/linux/coda_linux.h:CODA_ALLOC */
ptr = (cast)vmalloc((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

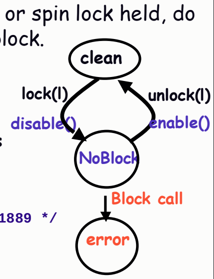
```
/* drivers/isdn/pcbit:pcbit_init_dev */
kfree(dev);
iounmap((unsigned char*)dev->sh_mem);
release_mem_region(dev->ph_mem, 4096);
```

- wrong size (2 errors)

```
/* drivers/parport/daisy.c:add_dev:50 */
newdev = kmalloc (GFP_KERNEL,sizeof(struct daisydev));
```

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

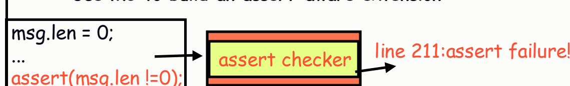
- Linux: if interrupts are disabled, or spin lock held, do not call an operation that could block.
- MC checker:
  - **Compute transitive closure of all potentially blocking fn's**
  - **Hit disable/lock: warn of any calls**
  - **123 errors, 8 false pos**




```
/* drivers/net/pcmcia/wavelan_cs.c */
spin_lock_irqsave (&lp->lock, flags); /* 1889 */
switch(cmd)
...
   case SIOCGIWPRIV:
        ...
        if(copy_to_user(wrq->u.data.pointer, …))  /* 2305 */
                ret = -EFAULT;
```

## 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**




- 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

| Check | Errors | False pos | LOC |
|---|---|---|---|
| Static assert | 5 | 0 | 100 |
| Stack check | 10+ | 0 | 53 |
| Allocation | 184 | 64 | 60 |
| Blocking | 123 | 8 | 131 |
| Module race | ~75 | 2 | 133 |
| Mutex | 82 | 201 | 64 |
| FLASH | 34 | 69 | 553 |
| Total(+others) | ~545 | ~226 | ~1100 |

## 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(a); | lock(b); | lock(a); |
| x++; | 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.nh.len) = LEN_CACHELINE;
if (! HANDLER_GLOBALS(h.hl.Pending)) {
    if (! HANDLER_GLOBALS(h.hl.Dirty)) {
        ASSERT(!HANDLER_GLOBALS(h.hl.IO));
        PI_SEND(F_DATA, F_FREE, F_SWAP,…,…);
        HANDLER_GLOBALS(h.hl.Local) = 1;
        /* ... deleted 14 lines */
    } else {
        ASSERT(!HANDLER_GLOBALS(h.hl.List));
        ASSERT(!HANDLER_GLOBALS(h.hl.RealPtrs));
```

Murphi
```
nh.len := len_cacheline;
if ((DH.Pending = 0)) then
    if ((DH.Dirty = 0)) then
        assert(nh.len != len_nodata);
        mbResult := pi_send_func(src, PI_Putt);
        DH.Local := 1;
    else
        assert((DH.List = 0));
        assert((DH.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 allocator with interrupts disabled": hoist allocation out

```
cli();                          tmp = malloc(sizeof *p);
p = malloc(sizeof *p);   --->   cli();
sti();                          p = tmp;
                                sti();
```

  - "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:
  - "if data is not shared with interrupt handlers, protect using spin locks rather than interrupt disable/enable
  - "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"

```
lock(q->lock);              read_lock(q->lock);
head = q->head;    --->     head = q->head;
n = q->nelem;               n = q->nelem;
unlock(q->lock);            read_unlock(q->lock);
```

  - and with a few lines: change opt into checker:

```
read_lock(q->lock);   --->  "modifying q with read lock!"
q->head = h;
```

## MC analysis vs. traditional compiler analysis

- Meaning more apparent + domain-specific knowlege

```
Bigint a, b, c;
set(a, 3);
mul(b, a, a);         --->      printf("81");
mul(c, b, b);
printf("%s", bigint_to_str(c));
```

- Easier to bound side-effects: use knowledge of abstract state to ignore many concrete actions
- Aliasing less of a problem
  - typical: opaque handles vs normal mess of pointers
- Operations more coarse grain
  - read()/write() vs load/store; matrix ops vs +/-