RacerX: effective, static detection of race conditions and deadlocks

Dawson Engler and Ken Ashcraft
Stanford University

The problem.

- **Big picture:**
  - Races and deadlocks are bad.
  - Hard to get w/ testing: depend on low-probability events.
  - Want to get rid of them.
  - Main games in town have problems.

- **Language:** Mesa, Java, various type systems.
  - Forced to use language; still have errors

- **Tools:**
  - Dynamic (Eraser&co): must execute code; no run, no bug.
  - Static (ESC, Warlock): High annotation overhead.
  - Static & dynamic high false positive rates.

RacerX: lightweight checking for big code

- **Goal:**
  - As many bugs as possible with as little help as possible

  - Works on real million line systems
  - Low annotation overhead (<100 lines per system)
  - Aggressively infers checking information.

  - Unusual techniques to reduce false positives.

Talk Overview

- **Context**
- **RacerX overview**

  - Context-sensitive, flow-sensitive lockset analysis.

  - Deadlock checking
  - Race detection.

  - Conclusion.

The RacerX experience

- **How to use:**
  - List locking functions & entry points. Small:
    - Linux: 18 + 31, FreeBSD: 30 + 36, System X: 50 + 52

  - Emit trees from source code (2x cost of compile)

  - Run RacerX over emitted trees
    - Links all trees into global control flow graph (CFG)
    - Checks for deadlocks & races
    - ~2-20 minutes for Linux.

  - Post-process to rank errors (most of IQ spent here)

  - Inspect

Lockset analysis

- **Lockset:** set of locks currently held [Eraser]

  - For each root, do a flow-sensitive, inter-procedural DFS traversal computing lockset at each statement

    ```
    initial  lockset = {}
    lock(l)  lockset = lockset U {l}
    unlock(l) lockset = lockset - {l}
    ```

  - Speed: If stmt s was visited before with lockset ls, stop.

- **Inter-procedural:**

  - Routine can exit with multiple locksets: resume DFS w/ each after callsite.

  - Record (in-ls, (out-ls)) in fn summary. If is in summary, grab cached out-ls’s and skip fn body.
**Lockset**

```c
class A {
    a
    b
    c
}

class B {
    a
    b
    c
}

int main() {
    A a;
    B b;
    c
}
```
False positive trouble.

- Most FPs from bogus locks in lockset
  - Typically caused by mishandled data dependencies
- Oversimplified typical example
  - Naive analysis will think four paths rather than two, including false one that holds lock a at line 5.

```c
1: if(x) ()
2: lock(a); (a)
3: if(x) (a)
4: unlock(a); (a) "a+b"
5: lock(b);

Inter-procedural analysis makes this much worse.
Could add path-sensitivity, but undecidable in general
```

Unlockset analysis

- Observations:
  - In practice, all false positives due to the A in "A+B", most because A goes "too far"
  - We had unconsciously adapted pattern of inspecting errors where there was an explicit unlock of "A" after "A+B" since that strongly suggested "A" was held.

```c
// 2.5.62/drivers/char/rtc.c
rtc_register(rtc_task_t *task) {
spin_lock_irq(&rtc_lock);
//...
spin_lock(&rtc_task_lock);
if (rtc_callback()) {
spin_unlock_irq(&rtc_task_lock);
spin_unlock_irq(&rtc_lock);
}
```

Unlockset implementation sketch

- Essentially compute reaching definitions
- Run lockset analysis in reverse from leaves to roots
- Unlockset holds all locks that will be released
  - lock() -> unlockset = unlockset - { }
  - unlock() -> unlockset = unlockset U { }
  - s.unlockset = s.unlockset U unlockset
- During lockset analysis:
  - lockset = intersect(s.unlockset, lockset);
- Main complication: function calls.
  - Different locks released after different call sites. Don’t want to mix these up (context sensitivity)

Unlockset analysis

At statement 5 remove any lock L from lockset if there exists no successor statement S’ reachable from S that contains an unlock of L.

```c
1: if(x) ()
2: lock(a); (a)
3: if(x) (a)
4: unlock(a); (a) X ()
5: lock(b);

Key: lockset holds exactly those locks the analysis can handle. Scales with analysis sophistication.
Without this we just can’t check FreeBSD.
```

Deadlock results

<table>
<thead>
<tr>
<th>System</th>
<th>Confirmed</th>
<th>Unconfirmed</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>System X</td>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Linux 2.6.2</td>
<td>4</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Total</td>
<td>8</td>
<td>14</td>
<td>19</td>
</tr>
</tbody>
</table>

- A bit surprised at the low bug counts
  - Main reason seems to be not that many locks held simultaneously
  - < 1000 unique constraints, only so many chances for error.

The most surprising error

```c
// Entered holding scsiLock
int FindHandle(int handleID) {
prevIRQ = SP_LockIRQ(handleArrayLock ...);
Validate(handle);
...}
int Validate(handle) {
ASSERT(SP_IsLocked(&scsiLock));
while (adapter->openInProgress) {
CpuSched_Wait(&adapter->openInProgress,
CPSCHED_WAIT_SCSI, &scsiLock);
SP_Lock(&scsiLock);
}

T1 enters FindHandle with scsiLock, calls Validate, calls
cpuSched_wait (rel scsiLock, sleep w/ handleArrayLock)
T2 acquires scsiLock and calls FindHandle. Boom.
```
Talk Overview

- Context
- RacerX overview
- Static inter-procedural lockset analysis
- Deadlock checking
- Race detection
- Conclusion

The big picture: race detection

- Three modes
  - Simple: flag globals accessed w/ empty lockset
  - Simple statistical: flag non-globals accessed w/ empty
  - Precise statistical: flag shared accessed with wrong lockset

```c
int x;
contrived(int *p) {
  *p ++;
lock(a);
  foo();
unlock(a);
}
```

- Ranking
  - Bulk of effort devising heuristics for probable races
  - Each error message falls under several. Need to order.
  - The usual trick: use a scoring function to map non-numeric attributes to a numeric value. Sort by value.

What's important to know

- Is lockset valid?
  - Roughly same as for deadlock
- Is code multithreaded?

- Does X have to be protected (by lock l)?

Does X have to be protected?

- Naive: flag any access to shared state w/o lock held.
  - Way too strong: 1000s of unprotected accesses. Only a few errors.
- The right definition:
  - Race = concurrent access that violates app invariant
- Problem:
  - No one tells us invariants
  - Diagnosing race requires understanding app...
- General approach: belief analysis [sosp'01]
  - Analyze if programmer seems to "believe" X must be protected

Infer if coder believes X needs locking

- If X "often" protected, flag when not.
  - lock(); lock(); lock(); lock(); lock(); // error!
  - foo(); foo(); foo(); foo(); foo();
  - unlock(); unlock(); unlock(); unlock(); unlock();
- Two modes:
  - Simple: count how often protected (S) versus not (F)
  - More precise: count how often protected by "most common" lock L (S) versus not (F)
  - Use "z-test statistic" to rank based on S and F counts
  - Intuition: the more protected (S/(S+F)), and the more samples (S+F), the higher the score.

Infer if coder believes X needs locking

- Coders generally don't do spurious concurrency ops
- If X is only object in critical section
  - Almost certainly protected (by L)
  - lock(); // error
  - foo();
  - unlock();
- Similar (but weaker) if first or last
  - lock();
  - bar();
  - foo();
  - unlock();
- Most important ranking feature
  - Almost always look at these errors first
Combined belief analysis example

- serial_out-info pair:
  - First statement in section 11 times & last 17 times.

```
// Ex1: drivers/char/esp.c
cll();
serial_out(info, ...);
restore_flags(flags);
```

```
// Ex2: drivers/char/esp.c
cll();
info->IER & = ~UART_EIR_RDI;
serial_out(info, ...);
serial_out(info, ...);
```

Obvious bug, trivial to diagnose.
```
restore_flags(flags); // re-enable interrupts
... //ERR: calling serial_out-info w/o cll
serial_out(info, ...);
```

Race results

<table>
<thead>
<tr>
<th>System</th>
<th>Confirmed</th>
<th>Unconfirmed</th>
<th>Minor</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>System X</td>
<td>7</td>
<td>4</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>Linux 2.6.2</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

Many more un inspected results. Races "very" hard to inspect: 10 minutes+ rather than 10 seconds.

Summary

- **RacerX**
  - Few annotations: 100 or less for > million lines of code
  - Takes an hour to setup for new system
  - Finds bugs
  - Reasonable false positive rate

- **Main tricks**
  - Belief analysis is a big win.
  - Unlockset analysis kills many false positives.
  - Ranking heuristics: other tools should be able to use.
  - Much more in paper...

- Lots of work left to do.

Some high-probability unsafe operations

- **Non-atomic writes (> 32-bits, bitfields):**
  - easy to diagnose, almost certainly bad.
  - st r1, 0x103
  - st r2, 4x0 (Read here = bizarre value)

- **Many vars modified in "non-critical section"**
  - 1 variable on unprotected path, almost certainly going to result in an inconsistent world-view.
  - shared int x, y;
  - x = i;
  - y = j;
  - Read x,y here = bizarre values

- **Data shared with interrupt handler.**
  - Bug on uniprocessor.

- Many others...

An illustrative race

```
/* ERROR:RACE: unprotected access to
  [logLevelPtr-__loglevel_offset_vmm, 
   *(theIOSpace).enabledPassThroughPorts,
   *(theIOSpace).enabledPassThroughWords] 
  [nvars=4] (modified=1) [has_locked=1] */
LOG(2, "IOspaceEnablePassThrough 0x%x count=%d\n", 
    theIOSpace->resumeCount));
theIOSpace->enabledPassThroughPorts = TRUE;
theIOSpace->enabledPassThroughWords |= (1<<word);
```

- **High rank:**
  - Modified (modified=1)
  - Four variables in non-critical section (nvars=4)
  - Concurrency operations in calichain (has_locked)

Multithreaded inference

- Infer if coder "believes" code is multithreaded.

Programmers generally don’t do spurious concurrency ops
- Any such op implies belief code is multithreaded.

RacerX marks function F as multithreaded if concurrency ops occur (1) in F’s body or (2) above it in callchain.

```
int x; 
threaded() {
  bar();
  atomic_inc(&x);
}
```

```
non_threaded() {
  x++;
  threaded();
}
```

Note: concurrency ops in caliche do not nec imply caller multithreaded
Programmer-written annotators

- Use coder knowledge to automatically mark code as:
  - Multithreaded or interrupt handlers (errors promoted)
  - Ignore or single-threaded (elided)

```c
// mark all system calls as multithreaded
for (struct fn *f = fn_list; f; f = fn.next(f))
  if(strcmp(f->name, "sys..", 4) == 0)
    f->multithreaded = 1;
```

Big win: small fixed cost → many annotations (100-1000)

- Function pointer equivalence
  - Functions assigned to same fp -> have same interface
  - If one annotated, automatically annotate others

Main limitations

- Very weak alias analysis:
  - Pointers to locals and parameters named by type.
    - `struct foo *f` → `struct:foo:local`
- Limited function pointer analysis
  - Record all functions assigned to fp (static or explicitly)
  - Assume call using that fp type can call any of them.
  - Miss: functions passed as arguments and then assigned.

- Main speed problem:
  - Deep fn class in many places with different locksets.
  - Will cause RacerX to re-analyze each time. Expensive.
  - Skips any fn when more than > 100 different locksets.

The problem with rendezvous semaphores

- Two conflated semaphore uses
  - Sometimes as locks (dep)
    - down(a);
     lock(b);
     up(o);

  - Sometimes for signaling (no dependency)
    - // Producer
     up(o); // signal
    - // Consumer
     down(a); // wait
     lock(b);

  - If not separated cause lots of false positives. Many.
  - Use behavioral analysis to automatically eliminate

Behavioral analysis

- Does s behave more like lock or more like semaphore?
  - Lock: (1) many down-up pairings, (2) few spurious ups
    - down(a);
    - up(b);
    - down(a);
    - up(b);
  - Scheduling: (1) few down-up pairs, (2) many spurious ups
    - down(s);
    - up(s);
    - down(s);
    - up(s)

- Use statistical analysis to calculate which s behaves like

Statistical classification sketch

- Foreach semaphore s, compute:
  - Ratio of paired down(s)/up(s)
  - Ratio of spurious up(s) to total down(s) calls
  - Baseline ratios using known spin-lock functions
  - Compare s’s ratio against baseline using “z-test statistic”
  - “Very improbable”? classify s as scheduling sem.

<table>
<thead>
<tr>
<th>name</th>
<th>down</th>
<th>up</th>
<th>spurious up</th>
</tr>
</thead>
<tbody>
<tr>
<td>PQPCT &amp; Aocomplete</td>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>event_exit</td>
<td>2</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>thread_exit</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>us_data_sem</td>
<td>8</td>
<td>28</td>
<td>2</td>
</tr>
<tr>
<td>nlm_struct_sem</td>
<td>141</td>
<td>208</td>
<td>2</td>
</tr>
</tbody>
</table>

Example scoring

- X first, last, or only object in critical section.
  - +4 if only object > 1 times, +2 if 1 time.
  - +1 if first, last object > 0 times
- Count protected vs unprotected, rank using z-test
  - +2 if z > 2; -2 if non-global and z < -2.
  - Writes:
    - Unprotected vars in non-csection: +2 n > 2; +1 if n > 1
    - Non-atomic write: +1
    - Written by interrupt handler: +2, in general: +1.
    - Modified by > 2 roots: +2
- Rank
  - Cases with concurrency op in callchain above not.
  - Order same score by callchain depth and conditionals