RacerX: effective, static detection of race conditions and deadlocks

Dawson Engler and Ken Ashcraft
Stanford University
Originally presented at SOSP 2003

Presented by Yuechao Pan for ECS 289C in class presentation
3 Feb. 2015
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.

S1: pass testing, blows up when shipped. S2: after blows up, you can’t recreate.
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.
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
Talk Overview

- Context
- RacerX overview

- Context-sensitive, flow-sensitive lockset analysis.
- Deadlock checking
- Race detection.

- Conclusion.
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

  \[
  \text{initial} \quad \Rightarrow \quad \text{lockset} = \{ \}
  \]

  \[
  \text{lock}(l) \quad \Rightarrow \quad \text{lockset} = \text{lockset} \cup \{ l \}
  \]

  \[
  \text{unlock}(l) \quad \Rightarrow \quad \text{lockset} = \text{lockset} - \{ l \}
  \]

  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 \(<\text{in}-ls, \{\text{out}-ls}\>> \) in fn summary. If \( ls \) in summary, grab cached out-\( ls \)'s and skip fn body.
Lockset

connect() {
  lock(a); { a }
  open_conn(); { a }
  send(); { a } summary: { a } ➔ ?
}

open_conn() {
  if (x) { a }
  lock(b); { a, b }
  else { a }
  lock(c); { a, c }
}


Lockset

connect() {
    lock(a); { a }
    open_conn(); { a }
    send(); { a, b }, {a, c}
}

open_conn() {
    if (x)
        lock(b); { a, b }
    else
        lock(c); { a, c }
}

summary:
{ a } ➔ { a, b }, {a, c}

{ a, b }, {a, c}
Talk Overview

- Context
- RacerX overview
- Static lockset analysis
- Deadlock checking
- Race detection.
- Conclusion.
Big picture: Deadlock detection

- **Pass 1: constraint extraction**
  
  emit 1-level locking dependencies during lockset analysis

  ```
  lock(a);
  lock(b);
  ```

  
  “a→b”

  ```
  lock(b);
  lock(a);
  ```

  “b→a”

- **Pass 2: constraint solving**

  Compute transitive closure & flag cycles.

  “a→b→a” : T1 acquires a, T2 acquires b, boom.

- **Ranking:**

  Global locks over local

  Depth of callchain & number of conditionals (less better)

  Number of threads involved (fewer MUCH better)
Simplest deadlock example

Constraint extraction emits “rtc_lock $\rightarrow$ rtc_task_lock” and “rtc_task_lock $\rightarrow$ rtc_lock”
Constraint solving flags cycle: T1 acquires rtc_lock, T2 acquires rtc_task_lock. Boom.
Ranked high: only two threads, global locks, local error.
Some crucial improvements

- Unlockset analysis to counter lockset mistakes.
- Automatic elimination of rendezvous semaphores
- Handling lockset mistakes with Summary selection heuristics
  Computing the same result more than one way.
  Pruning false paths based on locking errors
False positive trouble.

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

```
1: if(x)          {}  
2: lock(a);      {a}  
3: if(x)          {a}  
4: unlock(a);     
5: lock(b);       {a}  "a\rightarrow 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 adopted 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(&rtc_task_lock);
        spin_unlock_irq(&rtc_lock);
    }
    rtc_lock→rtc_task_lock
```
Unlockset analysis

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

Key: lockset holds exactly those locks the analysis can handle. Scales with analysis sophistication. Without this we just can't check FreeBSD.
Unlockset implementation sketch

- Essentially compute reaching definitions
  - Run lockset analysis in reverse from leaves to roots
  - Unlockset holds all locks that will be released

  initial \(\rightarrow\) unlockset = \{\}
  lock(l) \(\rightarrow\) unlockset = unlockset - \{l\}
  unlock(l) \(\rightarrow\) unlockset = unlockset U \{l\}
  s.unlockset = s.unlockset U unlockset

  During lockset analysis:
  lockset = intersect(s.unlockset, lockset);

- Main complication: function calls.
  - Different locks released after different callsites. Don't want to mix these up (context sensitivity)
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 2.5.62</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

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

- **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?

- Naïve: 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(l); lock(l); lock(l); lock(l); lock(l); // error!
  foo(); foo(); foo(); foo(); foo();
  unlock(l); unlock(l); unlock(l); unlock(l); unlock(l);

- 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(l);
  foo();
  unlock(l);
  ```
  Similar (but weaker) if first or last.
  ```
  lock(l);
  bar();
  foo();
  unlock(l);
  ```
- Most important ranking feature
  Almost always look at these errors first.
Combined belief analysis example

serial_out-info pair:

First statement in csection 11 times & last 17 times.

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

```
// Ex 2: drivers/char/esp.c
cli();
info->IER &= ~UART_IER_RDI;
serial_out(info, ...);
serial_out(info, ...);
sti();
```

Obvious bug, trivial to diagnose.

```
restore_flags(flags); // re-enable interrupts
...
//ERR: calling <serial_out-info> w/o cli!
serial_out(info,...);
```
Many more uninspected results. Races *very* hard to inspect: 10 minutes+ rather than 10 seconds.

<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 2.5.62</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Total</td>
<td>10</td>
<td>6</td>
<td>15</td>
<td>20</td>
</tr>
</tbody>
</table>
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 fpotr (static or explicitly)
  Assume call using that fpotr type can call any of them.
  **Miss:** functions passed as arguments and then assigned.

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

  \[
  \text{st r1, 0(r3)} \\
  \text{st r2, 4(r3)} \\
  \]
  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.

  \[
  \text{shared int x, y;} \\
  x = i; \\
  y = j; \\
  \]
  Read x,y here = bizarre values

- Data shared with interrupt handler.
  Bug on uniprocessor.

- Many others...
An illustrative race

```c
/* ERROR: RACE: unprotected access to
   [logLevelPtr, _loglevel_offset_vmm,
     (*theIOSpace).enabledPassthroughPorts,
     (*theIOSpace).enabledPassthroughWords]
   [nvars=4] [modified=1] [has_locked=1] */
LOG(2,("IOSpaceEnablePassthrough 0x%lx count=%d\n",
    port, 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 callchain (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.

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

```cpp
bar() {
    x++;
}
```

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

Note: concurrency ops in callee 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)

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

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

```c
// mark all system calls as multithreaded
for(struct fn *f = fn_list; f; f = fn_next(f))
    if(strncmp(f->name, "sys_", 4) == 0)
        f->multithreaded_p = 1;
```
The problem with rendezvous semaphores

- Two conflated semaphore uses
  - Sometimes as locks (dep)
    - `down(a);`  
    - `lock(b);`  
    - `up(a);`  
    - "a→b"
  - Sometimes for signaling (no dependency)
    - `down(a); // wait`  
    - `lock(b);`  
    - `up(a); // signal`  
    - "a→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-up pairings for lock]

down(a); up(b);  down(a); up(b);  down(a); up(b);  down(a); up(b);  down(a); up(b);

- Scheduling: (1) few down-up pairs, (2) many spurious ups

![Down-up pairs for scheduling]
down(s); up(s)  down(s);  down(s);  up(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\( (s) \)'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>PQFCH BA.complete</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>mm_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