Experiences using static analysis & model checking for bug finding

Dawson Engler and Madanlal Musuvathi
Based on work with Andy Chou, David (Lie, Park, Dill)
Stanford University

Quick, crude definitions
- "Static analysis" = our approach [DSL'97, OSDI'00]
  Flow-sensitive, inter-procedural, extensible analysis
  Goal: max bugs, min false pos
  May underestimate work factor: not sound, no annotation
  Works well: 1000s of bugs in Linux, BSD, company code
  Expect similar tradeoffs to PReFiX, SLAM(7), ESP(7)
- "Model checker" = explicit state space model checker
  Use Murphi for FLASH, then home-grown for rest.
  May underestimate work factor: All case studies use techniques to eliminate need to manually write model.

The Talk
- An introduction
- Case 1: FLASH cache coherence protocol code
  Checked statically [ASPLOS'00], then model checked [ESCA'01]
  Surprise: static found 4x more bugs.
- Case 2: AODV loop free, ad-hoc routing protocol
  Checked w/ model checking [OSDI'02], then statically.
  Surprise: when checked some property static won.
- Case 3: Linux TCP
  Model checked [NSDI'04].  Static checked it & rest of Linux [OSDI'00, SOSP'01, ...]
  Surprise: So hard to rip TCP out of Linux that it was easier to jam Linux into model checker!
- Lessons and religion.

Context: bug finding in implementation code
- Goal: Find as many bugs as possible.
  - Not verification, not checking high level design
- Two promising approaches
  - Static analysis
  - Software model checking.
  Basis: used static analysis extensively for four years;
  model checking for several projects over two years.
- General perception:
  - Static analysis: easy to apply, but shallow bugs
  - Model checking: harder, but strictly better once done.
  Reality is a bit more subtle.
  This talk is about that.

Some caveats
- Talk bias:
  - OS designer who does static analysis and has been involved in some some model checking
  - Some things that surprise me will be obvious to you.
- Of course, is just a bunch of personal case studies
tarted up with engineers induction
  - to look like general principles. (1,2,3=QED)
  - Coefficients may change, but general trends should hold
- Not a jeremiad against model checking!
  - We want it to succeed.  Will write more papers on it.
  - Life has just not always been exactly as expected.

Case Study: FLASH
- ccNUMA with cache coherence protocols in software.
  Protocols: 8-19K LOC, long paths (73-183LOC ave)
  Tension: must be "very" fast, but 1 bug deadlocks/livelocks entire machine
  Heavily tested for 5 years.  Manually verified.

Interconnection Network

Mem0  Mem1  Mem2
CPU0  CPU1  CPU2

CPU 0
CPU 1
CPU 2
Mem 0
Mem 1
Mem 2
NC 0
NC 1
NC 2

In ter c o n n e c t io n  N e t w o r k
Finding FLASH bugs with static analysis

- Gross code with many ad hoc correctness rules
  - Key feature: they have a clear mapping to source code.
  - Easy to check with compiler.
  - Example: "you must call \texttt{WAIT\_FOR\_DB\_FULL()} before \texttt{MISCBUS\_READ\_DB()}".
  - (Intuition: msg but must have all data before you read it)

Nice: scales, precise, statically found 34 bugs

```
Handler:
  \textbf{if}(.)
  \texttt{WAIT\_FOR\_DB\_FULL();}
  \texttt{MISCBUS\_READ\_DB();}

GNU C compiler
\begin{itemize}
  \item \texttt{buf race check}\end{itemize}
\texttt{read msg w/o synch!}
```

FLASH results [ASPLOS'00]

<table>
<thead>
<tr>
<th>Rule</th>
<th>LOC</th>
<th>Bugs</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait_for_db_full before read</td>
<td>12</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>has_length parameter for msg sends must match specified message length</td>
<td>29</td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>Message buffers must be allocated before use, deallocated after, not used after deallocated</td>
<td>94</td>
<td>9</td>
<td>25</td>
</tr>
<tr>
<td>Messages can only be sent on pre-specified lanes</td>
<td>220</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>355</td>
<td>33</td>
<td>28</td>
</tr>
</tbody>
</table>

- Want to vet deeper rules
  - Nodes never overflow their network queues
  - Sharing list empty for dirty lines
  - Nodes do not send messages to themselves

- Perfect for model checking
  - Self-contained system that generates its own events
  - Bugs depend on intricate series of low-probability events

- The (known) problem: writing model is hard
  - Someone did it for one FLASH protocol.
  - Several months effort. No bugs. Inert.
  - But there is a nice trick.

A modicum of detail

```
```

When applicable, works well.

- Don't have to understand code
  - Wildly ignorant of FLASH details and still found bugs.

- Lightweight
  - Don't need annotations.
  - Checkers small, simple.

- Not weak
  - FLASH not designed for verification.
  - Heavily tested.
  - Still found serious bugs.

- These generally hold in all areas we've checked.
  - Linux, BSD, FreeBSD, 15+ large commercial code bases.

- But: not easy to check some properties with static...

Model checking FLASH

A striking similarity

```
```

```

```
The extraction process from 50K meters

- Reduce manual effort: Check at all. Check more things
- Important: more automatic = more fidelity
  - Reversed extraction: mapped manual spec back to code
  - Four serious model errors.

Myth: model checking will find more bugs

- Not quite: 4x fewer (8 versus 33)
  - While found 2 missed by static, it missed 24.
  - And was after trying to pump up model checking bugs...
- The source of this tragedy: the environment problem.
- FLASH:
  - No cache line data, so didn't check data buffer handling, missing all alloc errors (9) and buffer races (4)
  - No I/O subsystem (hairy): missed all errors in I/O sends
  - No uncached reads/writes: uncommon paths, many bugs.
  - No lanes: so missed all deadlock bugs (2)
  - Create model at all takes time, so skipped “sci” (5 bugs)

Case Study: AODV Routing Protocol

- Ad hoc, loop-free routing protocol.
- Checked three implementations
  - Mad-hoc
  - Kernel AODV (NIIST implementation)
  - AODV-UU (Uppsala Univ. implementation)
  - Deployed, used, AODV-UU was "certified"
- Model checked using CMC [OSDI'00]
  - Checks C code directly (similar to Verisoft)
  - Two weeks to build mad-hoc, 1 week for others (expert)
- Static: used generic memory checkers
  - Few hours (by me, but non-expert could do it.)
  - Lots left to check.

Model checking results [ISCA'01]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynptr(*)</td>
<td>6</td>
<td>12K</td>
<td>1100</td>
<td>1000</td>
<td>99</td>
</tr>
<tr>
<td>Bitvector</td>
<td>2</td>
<td>8k</td>
<td>700</td>
<td>1000</td>
<td>100</td>
</tr>
<tr>
<td>RAC</td>
<td>0</td>
<td>10K</td>
<td>1500</td>
<td>1200</td>
<td>119</td>
</tr>
<tr>
<td>Coma</td>
<td>0</td>
<td>15K</td>
<td>2800</td>
<td>1400</td>
<td>159</td>
</tr>
</tbody>
</table>

- Extraction a big win: more properties, more code, less chance of mistakes.
- (*) Dynptr previously manually verified (but no bugs found)

The Talk

- An introduction
- Case I: FLASH
  - Static: exploit fact that rules map to source code constructs. Checks all code paths, in all code.
  - Model checking: exploit same fact to auto-extract model from code. Checks more properties but only in run code.
- Case II: AODV
- Case III: TCP
- Lessons & religion
- A summary

Checking AODV with CMC [OSDI'02]

- Properties checked
  - CMC: seg faults, memory leaks, uses of freed memory
  - Routing table does not have a loop
  - At most one route table entry per destination
  - Hop count is infinity or <= nodes in network
  - Hop count on sent packet is not infinity
- Effort:
  - Protocol Code Checks Environment Conn’ic
  - Mad-hoc 3336  301  100 - 400  165
  - Kernel-aodv 4908 301 266 - 400 179
  - Aodv-uu 5286 332 128 - 400 185

- Results: 42 bugs in total, 35 distinct, one spec bug.
  - ~1 bug per 300 lines of code.
### Classification of Bugs

<table>
<thead>
<tr>
<th></th>
<th>madhoc</th>
<th>Kernel</th>
<th>AODV-</th>
<th>AODV-UU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mishandling malloc failures</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Memory leaks</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Use after free</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Invalid route table entry</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Unexpected message</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Invalid packet generation</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Program assertion failures</td>
<td>1</td>
<td>1</td>
<td>(1)</td>
<td>1</td>
</tr>
<tr>
<td>Routing loops</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>(1)</td>
</tr>
<tr>
<td>Total bugs</td>
<td>18</td>
<td>16</td>
<td>(5)</td>
<td>8</td>
</tr>
<tr>
<td>LOC/bug</td>
<td>185</td>
<td>281</td>
<td>661</td>
<td></td>
</tr>
</tbody>
</table>

### Who missed what and why.

- **Static**: more code + more paths = more bugs (13)
  - Check same property: static won. Only missed 1 CMC bug
- **Why CMC missed SA bugs**: no run, no bug.
  - 6 were in code cut out of model (e.g., multicast)
  - 6 because environment had mistakes (send_datagram())
  - 1 in dead code
  - 1 null pointer bug in model
- **Why SA missed model checking bugs**: no check, no bug
  - Model checking: more rules = more bugs (21)
- Some of this is fundamental. Next three slides discuss.

### Significant model checking win #2.

- **End-to-end**: catch bug no matter how generated
  - Static detects ways to cause error, model checking checks for the error itself.
  - Many bugs easily found with SA, but they come up in so many ways that there is no percentage in writing checker
- **Perfect example**: The AODV spec bug:
  ```c
  cur_rt = getentry(recv_rt->dst_ip);
  // bug if: recv_rt->dst_seq < cur_rt->dst_seq!
  if(cur_rt && cur_rt->dst_seq < recv_rt->dst_seq) {
      cur_rt->dst_seq = recv_rt->dst_seq;
  }
  ```
  Not hard to check, but hard to recoup effort.

### Model checking vs static analysis (SA)

<table>
<thead>
<tr>
<th></th>
<th>CMC &amp; SA</th>
<th>CMC only</th>
<th>SA only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mishandling malloc failures</td>
<td>11</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Memory leaks</td>
<td>8</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Use after free</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invalid route table entry</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unexpected message</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invalid packet generation</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program assertion failures</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Routing loops</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total bugs</td>
<td>21</td>
<td>21</td>
<td>13</td>
</tr>
</tbody>
</table>

### Significant model checking win #3

- **Subtle errors**: run code, so can check its implications
  - Data invariants, feedback properties, global properties.
  - Static better at checking properties in code, model checking better at checking properties implied by code.
- **The CMC bug SA checked for and missed**:

```c
for(i=0; i<cnt;i++) {
    tp = malloc(sizeof *tp);
    if(!tp)
        break;
    tp->next = head; head = tp;
    ...
for(i=0, tp = head; i<cnt; i++, tp=tp->next) {
    rt_entry = getentry(tp->unr_dst_ip);
}
```

### Significant model checking win #4


```c
Verifies that code was correct on checked executions.
If coverage good and state reduction works, very hard for implementation to get into new, untested states.
As everyone knows: Most bugs show up with a small value of N (where N counts the noun of your choice)
```
The Talk

- An introduction
- Case I: FLASH
- Case II: AODV
  - Static: all code, all paths, hours, but fewer checks.
  - Model checking: more properties, smaller code, weeks.
  - Surprise: most bugs shallow.
- Case III: TCP
- Lessons & religion
- A summary

The approach that failed: kernel-lib.c

- The obvious approach: Rip TCP out, run on libLinux
- Where to cut?
  - Basic question: TCP calls foo().
  - Fake foo() or include?
  - Faking takes work. Including leads to transitive closure
- Conventional wisdom: cut on narrowest interface
  - Doesn’t really work. 150+ functions, many poorly doc’d
  - Make corner-case mistakes in faking them. Model checkers good at finding such mistakes.
  - Result: many false positives. Can cost “days” for one.
  - Wasted months on this, no clear fixed point.

Fundamental law: no run, no bug.

<table>
<thead>
<tr>
<th>Method</th>
<th>line coverage</th>
<th>protocol coverage</th>
<th>branching factor</th>
<th>additional bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard client &amp; server</td>
<td>47%</td>
<td>64.7%</td>
<td>2.9</td>
<td>2</td>
</tr>
<tr>
<td>+ simultaneous connect</td>
<td>51%</td>
<td>66.7%</td>
<td>3.67</td>
<td>0</td>
</tr>
<tr>
<td>+ partial close</td>
<td>53%</td>
<td>79.5%</td>
<td>3.89</td>
<td>2</td>
</tr>
<tr>
<td>+ corruption</td>
<td>51%</td>
<td>84.3%</td>
<td>7.01</td>
<td>0</td>
</tr>
<tr>
<td>Combined cov.</td>
<td>59.4%</td>
<td>92.1%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Big static win: check all paths, all compiled code.
- CMC coverage for rest of Linux: 0%. Static ~ 100%.

Case study: TCP [NSDI’04]

- “Gee, AODV worked so well, let’s check the hardest thing we can think of”
  - Linux version 2.4.19
  - About 50K lines of heavily audited, heavily tested code.
  - A lot of work.
  - 4 bugs, sort of.
- Statically checked:
  - TCP (0 bugs)
  - rest of linux (1000s of bugs, 100s of security holes)
- Serious problems because model check = run code
  - Cutting code out of kernel (environment)
  - Getting it to run (false positives)
  - Getting the parts that didn’t run to run (coverage)

Shocking alternative: jam Linux into CMC.

- Different heuristic: only cut along well-defined interfaces
  - Only two in Linux:
    - syscall boundary and
    - hardware abstraction layer
  - Result: run Linux in CMC.
  - Cost: State ~300K, transition ~5ms.

Nice: can reuse to model check other OS subsystems (currently checking file system recovery code)

The Talk

- An introduction
- Case I: FLASH
- Case II: AODV
- Case III: TCP
  - Model checking found 4 bugs static did not, static found 1000s model checking missed.
  - Environment is really hard. We’re not kidding.
  - Executing lots of code not easy, either.
  - Myth: model checking does not have false positives
- Some religion
- A summary
Open Q: how to get the bugs that matter?

- Myth: all bugs matter and all will be fixed
  
  *FALSE*

  Find 10 bugs, all get fixed. Find 10,000...

- Reality

  - All sites have many open bugs (observed by us & PREfix)
  - Myth lives because state-of-art is so bad at bug finding
  - What users really want: The 5-10 that *really* matter

- General belief: bugs follow 90/10 distribution

  Out of 1000, 100 account for most pain.

  Fixing 900 waste of resources & may make things worse

- How to find worst? No one has a good answer to this.

Some cursory static analysis experiences

- Bugs are everywhere

  Initially worried we’d resort to historical data...

  100 checks? You’ll find bugs (if not, bug in analysis)

- Finding errors often easy, saying why is hard

  Have to track and articulate all reasons.

- Ease-of-inspection “crucial”

  Extreme: Don’t report errors that are too hard.

- The advantage of checking human-level operations

  Easy for people? Easy for analysis. Hard for analysis?

  Hard for people.

- Soundness not needed for good results.

Myth: Soundness is a virtue.

- Soundness: Find all bugs of type X.

  Not a bad thing. More bugs good.

  BUT: can only do if you check weak properties.

- What soundness really wants to be when it grows up:

  Total correctness: Find all bugs.

  Most direct approximation: find as many bugs as possible.

- Opportunity cost:

  Diminishing returns: Initial analysis finds most bugs.

  Spend on what gets the next biggest set of bugs.

- Easy experiment: bag counts for sound vs unsound tools.

- End-to-end argument:

  “It generally does not make much sense to reduce the

  residual error rate of one system component (property)

  much below that of the others.”

Open Q: Do static tools really help?

- Bugs found

  Bugs that mattered

  The hope

- Bugs found

  Bugs that mattered

  The null hypothesis

  A Possibility

  Dangers: Opportunity cost. Deterministic bugs to non-

  deterministic.

Myth: more analysis is always better

- Does not always improve results, and can make worse

- The best error:

  Easy to diagnose

  True error

- More analysis used, the worse it is for both

  More analysis = the harder error is to reason about,

  since user has to manually emulate each analysis step.

  Number of steps increase, so does the chance that one

  went wrong. No analysis = no mistake.

- In practice:

  Demote errors based on how much analysis required

  Revert to weaker analysis to cherry pick easy bugs

  Give up on errors that are too hard to diagnose.

Related work

- Tool-based static analysis

  PREFix/PREfast

  SLAM

  ESP

- Generic model checking

  Murphi

  Spin

  SMV

- Automatic model generation model checking

  Pathfinder

  Bandera

  Verisoft

  SLAM (sort of)
static analysis vs model checking

<table>
<thead>
<tr>
<th>First question:</th>
<th>“How big is code?”</th>
<th>“What does it do?”</th>
</tr>
</thead>
<tbody>
<tr>
<td>To check?</td>
<td>Must compile</td>
<td>Must run.</td>
</tr>
<tr>
<td>Time:</td>
<td>Hours.</td>
<td>Weeks.</td>
</tr>
<tr>
<td>Don’t understand?</td>
<td>So what.</td>
<td>Problem.</td>
</tr>
<tr>
<td>Coverage?</td>
<td>All paths! All paths!</td>
<td>Executed paths.</td>
</tr>
<tr>
<td>FP/Bug time:</td>
<td>Seconds to min</td>
<td>Seconds to days.</td>
</tr>
<tr>
<td>Bug count:</td>
<td>100-1000s</td>
<td>0-10s</td>
</tr>
<tr>
<td>Big code:</td>
<td>10MLOC</td>
<td>10K</td>
</tr>
<tr>
<td>No results?</td>
<td>Surprised.</td>
<td>Less surprised.</td>
</tr>
<tr>
<td>(Relatively) better at?</td>
<td>Source visible rules</td>
<td>Code implications &amp; all ways to get errors</td>
</tr>
</tbody>
</table>

A formal methods opportunity

- "Systems" community undergoing a priority sea change
  - Performance was king for past 10-15 years. Moore's law has made it rather less interesting. Very keen on other games to play.
  - One "new" game: verification, defect detection
    - The most prestigious conferences (SOSP, OSDI) have had such papers in each of last few editions.
    - Warm audience: Widely read, often win "best paper;" program committees make deliberate effort to accept "to encourage work in the area."

  Perfect opportunity for formal methods community: Lots of low hanging fruit = systems people interested, but lack background in formal method’s secret weapons.

Summary

- First law of bug finding: no check, no bug
  - Static: don’t check property X? Don’t find bugs in it.
  - Model checking: don’t run code? Don’t find bugs in it.
- Second law of bug finding: more code = more bugs.
  - Easiest way to get 10x more bugs: check 10x more code.
  - Techniques with low incremental cost per LOC win.
- What surprised us:
  - How hard environment is.
  - How bad coverage is.
  - That static analysis found so many errors in comparison.
  - That bugs were so shallow.
- Availability:
  - Murphi from Stanford. CMC from Madan (now at MSR).
  - Static checkers from coverity.com

The fundamental law of defect detection: No check, no bug.

First order effects:

- Static: don’t check property X? Won’t find its bugs.
- Model checking: don’t check code? Won’t find bugs in it.