What's this all about

- A general goal of humanity: automatically find bugs
  Success: lots of bugs, lots of code checked.

- Two promising approaches
  Static analysis
  Model checking
  We used static analysis heavily for a few 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.

What's the data

- Case 1: FLASH cache coherence protocol code
  Checked w/ static analysis [ASPLOS'00]
  Then w/ model checking [ISCA'01]
  Surprise: static analysis found 4x more bugs.

- Case 2: AODV loop free, ad-hoc routing protocol
  Checked w/ model checking [OSDI'02]
  Took 3+ weeks; found ~ 1 bug / 300 lines of code
  Checked w/ static (2 hours): more bugs when overlap

- Case 3: Linux TCP
  Model checking: 6 months, 4 “ok” bugs.
  Surprise: So hard to rip TCP out of Linux that it was easier to jam Linux into model checker

Crude definitions.

- “Static analysis” - our approach [DSL'97,OSDI'00]

  - Flow-sensitive, inter-procedural, extensible analysis
  - Goal: max bugs, min false pos

  - Not sound. No annotations.
  - Works well: 1000s of bugs in Linux, BSD, company code

  - Expect similar tradeoffs to Precise, SLAM(?), ESP(?)

- “Model checker” = explicit state space model checker

  - Use Murphi for FLASH, then home-grown for rest.
  - Probably underestimate work factor
  - Limited domain: applying model checking to implementation code.

Some caveats

- Main bias:
  - Static analysis guy that happens to do model checking.
  - Some things that surprise me will be obvious to you.

- The talk is not a jeremiad against model checking!
  - We want model checking to succeed.
  - We're going to write a bunch more papers on it.
  - Life has just not always been exactly as expected.

- Of course
  - This is just a bunch of personal case studies
  - Tore up with engineers induction:
    - to look like general principles. (1,2,3-QED)
  - While coefficients may change, general trends should hold.

The Talk

- An introduction

- Case I: FLASH

- Case II: AODV

- Case III: TCP

- Lessons & religion

- A summary
Case Study: FLASH

- ccNUMA with cache coherence protocols in software.
- Has to be extremely fast
  BUT: 1 bug deadlocks/live locks entire machine
- Heavily tested for 5 years.
  Low-level with long code paths (73-183LOC ave)

Finding FLASH bugs with static analysis

- Gross code with many ad hoc correctness rules
- But: they have a clear mapping to source code.
- Easy to check with compiler.
- Example:
  WAIT_FOR_DB_FULL must precede MISCBUS_READ_DB

 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</td>
<td>29</td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>message length</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message buffers must be allocated</td>
<td>94</td>
<td>9</td>
<td>25</td>
</tr>
<tr>
<td>deallocated before use,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>not used after dealloc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Messages can only be sent</td>
<td>220</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>on pre-specified lanes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>355</td>
<td>33</td>
<td>28</td>
</tr>
</tbody>
</table>

A modicum of detail

```
wait_for_db_full()

start

{ WAIT_FOR_DB_FULL(addr); } =>> stop
| { MISCBUS_READ_DB(addr); } =>> stop

misc_bus_read_db()

err("Buffer read not synchronized");
```

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</td>
<td>29</td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>message length</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message buffers must be allocated</td>
<td>94</td>
<td>9</td>
<td>25</td>
</tr>
<tr>
<td>deallocated before use,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>not used after dealloc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Messages can only be sent</td>
<td>220</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>on pre-specified lanes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>355</td>
<td>33</td>
<td>28</td>
</tr>
</tbody>
</table>

Some experiences

- Good:
  - Don’t have to understand FLASH to find bugs this way
  - Checkers small, simple
  - Doesn’t need much help: FLASH not designed for verification, still found bugs
  - Not weak: code tested for 5 years, still found bugs.

- Bad:
  - Bug finding is symmetric
  - We miss many deeper properties.

Finding FLASH bugs with model checking

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

- Perfect application for model checking
  - Hard to test: bugs depend on intricate series of low-probability events
  - Self-contained system that generates its own 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 striking similarity

Murphi model

```c
void PI_Local_Get(void) {
    // Boilerplate setup
    if (!hl.Pending) {
        if (!hl.Dirty) {
            // ASSERT(hl.Local);
            ...
        }
    }
    PI_SEND(F_DATA, F_FREE, F_SWAP, F_NOWAIT, F_DEC, 1);
    hl.Local = 1;
}
```

Use correspondence to auto-extract model from code

Use extension to mark features you care about

System does a backwards slice & translates to Murphi

---

The extraction process

- 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.

---

A simple user-written marker

```c
am len slicer { 
    decl any_expr type, data, keep, swap, wait, nl;
    all: 
        // match all uses of length field
        [ hl.len ] 
        // match all uses of directory entries
        || (hl.Local) || (hl.Dirty) || (hl.List) 
        // match all network and processor sends
        || (hl_SEND(type, data, keep, swap, wait, nl)) 
        || (hl.Local = 1) 
        [ mk_tag(mc_stmt); ]
    ;
}
```

---

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>Dyrptr(*)</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>2</td>
<td>15K</td>
<td>2800</td>
<td>1400</td>
<td>159</td>
</tr>
</tbody>
</table>

Extraction a win.
- Two deep errors.
- Dyrptr checked manually.
- But: 6 bugs found with static analysis...

---

Myth: model checking will find more bugs

- Not quite: 4x fewer.
  - And was after trying to pump up model checking bugs...
  - Two laws: No check, no bug. No run, no bug.
- Our 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 (hairly): 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)

---

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 less code.
- Case II: AODV
- Case III: TCP
- Lessons & religion
- A summary
Case Study: AODV Routing Protocol

- AODV: Ad-hoc On-demand Distance Vector
- Routing protocol for ad-hoc networks
draft-ietf-manet-aodv-12.txt
- Guarantees loop freeness
- Checked three implementations
  - Mad-hoc
    - Kernel AODV (NIST implementation)
    - AODV-UU (Uppsala Univ. implementation)
      - First used model checking, then static analysis.
  - Model checked using CMC
    - Checks C code directly
      - No need to slice, or translate to weak language.

Checking AODV with CMC [OSDI’02]

- Properties checked
  - CMC: seg faults, memory leaks, uses of freed memory
  - Routing table does not have a loop
    - Hop count is infinity or <= nodes in network
    - Hop count on sent packet cannot be infinity
- Effort:
  - Protocol Code Checks Environment Commic
    - Mad-hoc 3336 301 100+400 165
    - Kernel-aodv 4508 301 266+400 179
    - Aodv-uu 5286 332 128+400 185
- Results: 42 bugs in total, 35 distinct, one spec bug.

Classification of Bugs

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

Static analysis vs model checking

- Model checking:
  - Two weeks to build mad-hoc model
  - Then 1 week each for kernel-aodv and aodv-uu
  - Done by Madan, who wrote CMC.
- Static analysis:
  - Two hours to run several generic memory checkers.
    - Done by me, but non-expert could probably do easily.
    - Lots left to check...
- High bit
  - Model checking checked more properties
  - Static checked more code.
  - When checked same property, static won.

Model checking vs static analysis (SA)

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>Memory leaks</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>Use after free</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Invalid route table entry</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Unexpected message</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Invalid packet generation</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Program assertion failures</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Routing loops</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Total bugs</td>
<td>21</td>
<td>21</td>
</tr>
</tbody>
</table>

Fundamental law: No check, no bug.

- Static: checked more code = 13 bugs.
  - Check same property: static won. Only missed 1 CMC bug
- Why CMC missed 5A bugs:
  - 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
- Model checking: more properties = 21 bugs
  - Some fundamentally hard to get with static
  - Others checkable, but many ways to violate.
Two emblematic bugs

- The bug SA checked for & missed:
  for(i=0; i < cnt;i++)
  if(!((tp = malloc(...)))
    break;
    tp->next = head; head = tp;
    ...for(i=0; i < cnt;i++)
    tmp = head;
    head = head->next;
    free(tmp);

- The spec bug: time goes backwards if msg reordered.
  cur_rt = getentry(recv_rt->dst_ip);
  if(cur_rt & &)
    cur_rt->dst_seq = recv_rt->dst_seq;

Case study: TCP

- "Gee, AODV worked so well, let’s check the hardest thing we can think of."
  Linux version 2.4.19
  About 50K lines of code.
  A lot of work.
  4 bugs, sort of.
  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)

Instead: jam Linux into CMC.

- Main lesson: must cut along well-defined boundaries.
  Linux: syscall boundary and hardware abstraction layer

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
  Conventional wisdom: as small as possible.
  Basic question: calls foo(). Fake foo() or include?
  Faking takes work. Including leads to transient closure
  Building fake stubs
  Hard + Messy + Bad docs = easy to get slightly wrong.
  Model checker good at finding slightly wrong things.
  Result: most bugs were false. Take days to diagnose. Myth: model checking has no false positives.

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</td>
<td>51%</td>
<td>66.7%</td>
<td>3.67</td>
<td>0</td>
</tr>
<tr>
<td>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>3.67</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>

Nasty: unchecked code is silent. Can detect with static, but diagnostic rather than constructive.
Big static win: Check all paths, finding errors on any
The Talk

- An introduction
- Case I: FLASH
- Case II: AODV
- Case III: TCP
  Myth: model checking does not have false positives
  Environment is really hard. We're not kidding.
  Executing lots of code not easy, either.
- A more refined view
- Some religion
- A summary

Where static wins.

<table>
<thead>
<tr>
<th>Static analysis</th>
<th>Model checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile ➔ Check</td>
<td>Run ➔ Check</td>
</tr>
<tr>
<td>Don't understand?</td>
<td>So what.</td>
</tr>
<tr>
<td>Can't run?</td>
<td>So what.</td>
</tr>
<tr>
<td>Coverage?</td>
<td>All paths! All paths!</td>
</tr>
<tr>
<td>First question:</td>
<td>“How big is code?”</td>
</tr>
<tr>
<td>Time:</td>
<td>Hours.</td>
</tr>
<tr>
<td>Bug counts</td>
<td>100-1000s</td>
</tr>
<tr>
<td>Big code:</td>
<td>10MLOC</td>
</tr>
<tr>
<td>No results?</td>
<td>Surprised.</td>
</tr>
<tr>
<td></td>
<td>Less surprised.</td>
</tr>
</tbody>
</table>

Where model checking wins.

- 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.
- 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.
- Stronger guarantees:
  Most bugs show up with a small value of N.

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 1,000...
- 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.

Open Q: Do static tools really help?

- Bugs found: The hope
- Bugs that mattered: Bugs found
- The null hypothesis: Bugs that mattered
- A Possibility: Bugs found

Dangers: Opportunity cost. Deterministic bugs to non-deterministic.
Future? Combine more aggressively.

- Simplest: Find false negatives in both.
  - Run static, see why missed bugs. Run model checking, see why missed bugs.
  - Find a bug type with model checking, write static checker
- Use SA to give model checking visibility into code.
  - Smear invariant checks throughout code: memory corruption, race detection, assertions.
  - State space tricks: analyze if-statements and use to drive into different states. Capture the paths explored, favor states on new paths.
- Use model checking to deepen static analysis.
  - Simulation + state space tricks.

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
- Soundness not needed for good results.

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.

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 resources on what gets the next chunk of bugs
  - Easy experiment: bug counts for sound vs unsound tools.
- What users really care about:
  - Find just the important bugs. Very different.

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)

Summary

- Static analysis: exploit that rules map to source code
  - Push button, check all code, all paths. Hours.
  - Don’t understand? Can’t run? So what.
- Model checking: more properties, but less code.
  - Check code implications, check all ways to cause error.
  - Didn’t think of all ways to cause segfault? So what.
- What surprised us:
  - How hard environment is.
  - How bad coverage is.
  - That static analysis found so many errors in comparison.
  - The cost of simplifications.
  - That bugs were so shallow.
Main CMC Results

- 3 different implementations of AODV
  (AODV is an ad-hoc routing protocol)
  - 35 bugs in the implementations
  - 1 bug in the AODV specification
- Linux TCP (version 2.4.19)
  - CMC scales to such large systems (~50K lines)
  - 4 bugs in the implementation
- FreeBSD TCP module in OSKit
  - 4 bugs in OSKit
- DHCP (version 2.0 from ISC)
  - 1 bug

Case study: TCP

- “Gee, AODV worked so well, let’s check the hardest thing we can think of”
  - Linux version 2.4.19
  - About 50K lines of code.
  - A lot of work.
  - 4 bugs, sort of.
- Biggest problem: cutting it out of kernel.
  - Myth: model checking does not have false positives
  - Majority of errors found during development will be false
  - Mostly from environment and harness code mistakes
  - Easy to get environment slightly wrong. Model checker really good at finding slightly wrong things

TCP’s lessons for checking big code

- Touch nothing
  - Code is its best model
  - Any translation, approximation, modification = potential mistake.
- Manual labor is no fun
  - It’s really bad if your approach requires effort proportional to code size
- Only cut along well-defined interfaces.
  - Otherwise you’ll get FP’s from subtle misunderstandings.
- Best heuristic for bugs: hit as much code as possible
- Ideal: only check code designed for unit testing...

What this is all about.

- A goal of humanity: automatically find bugs in code
  - Success: lots of bugs, lots of code checked.
- We’ve used static analysis to do this for a few years.
  - Found bugs, generally happy.
  - Lots of properties we couldn’t check.
- Last couple of years started getting into model checking
- The 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.

Summary

- Static
  - Orders of magnitude easier: push a button and check all code, all paths
  - Find bugs when completely ignorant about code
  - Finds more bugs when checking some properties.
- Model checking:
  - Misses many errors because misses code
  - Environment: big source of false positives and negatives
  - Finds all ways to get a error
  - Checks implications of code
- Surprises
  - Model checking finds less bugs
  - Many bugs actually shallow