Experiences using static analysis & model checking for bug finding

Dawson Engler and Madanlan Musuvathi
Based on work with
Andy Chou, David {Lie, Park, Dill}
Stanford University

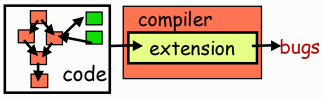
Context: bug finding in implementation code

- 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.

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(?), ESP(?)
- "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.

Some caveats

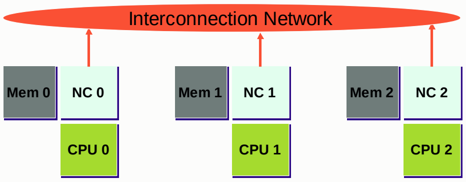
- Talk bias:
  - OS designer who does static analysis and has been involved in some some model checking
  - Some things that surprise me will be obvious to you.
- Of course, is just a bunch of personal case studies
  - tarted up with engineers induction to look like general principles. (1,2,3=QED)
  - 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.

The Talk

- An introduction
- Case 1: FLASH cache coherence protocol code
  - Checked statically [ASPLOS'00], then model checked [ISCA'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 same property static won.
- Case 3: Linux TCP
  - Model checked [NSDI'04]. Statically 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.

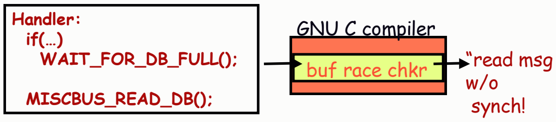
Case Study: FLASH

- ccNUMA with cache coherence protocols in software.
  - Protocols: 8-15K 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.



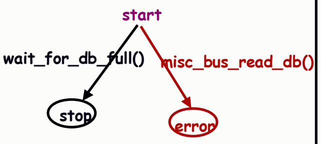
Finding FLASH bugs with static analysis

- 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 WAIT_FOR_DB_FULL() before MISCBUS_READ_DB()".**
  - **(Intuition: msg buf must have all data before you read it)**



Nice: scales, precise, statically found 34 bugs

A modicum of detail



```
sm wait_for_db {
 decl any_expr addr;

 start:
    { WAIT_FOR_DB_FULL(addr); } ==> stop
  | { MISCBUS_READ_DB(addr); } ==>
         { err("Buffer read not synchronized"); }
   ;
}
```

FLASH results [ASPLOS'00]

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

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

- 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 striking similarity

Hand-written
Murphi model

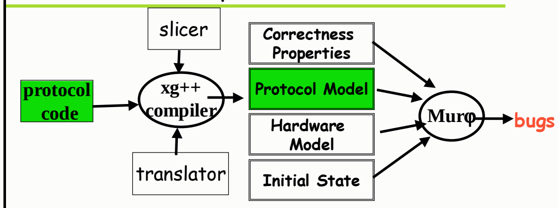
```
Rule "PI Local Get (Put)"
 1:Cache.State = Invalid
    & ! Cache.Wait
 2: & ! DH.Pending
 3: & ! DH.Dirty ==>
  Begin
 4: Assert !DH.Local;
 5: DH.Local := true;
 6: CC_Put(Home, Memory);
  EndRule;
```

FLASH code

```
void PILocalGet(void) {
     // ... Boilerplate setup
2   if (!hl.Pending) {
3     if (!hl.Dirty) {
4!      // ASSERT(hl.Local);
         ...
6       PI_SEND(F_DATA, F_FREE, F_SWAP,
                 F_NOWAIT, F_DEC, 1);
5       hl.Local = 1;
```

- Use correspondence to auto-extract model from code
  - **User writes static extension to mark features**
  - **System does a backwards slice & translates to Murphi**

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.

Model checking results [ISCA'01]

Protocol	Errors	Protocol (LOC)	Extracted (LOC)	Manual (LOC)	Extens. (LOC)
Dynptr(*)	6	12K	1100	1000	99
Bitvector	2	8k	700	1000	100
RAC	0	10K	1500	1200	119
Coma	0	15K	2800	1400	159

Extraction a big win: more properties, more code, less chance of mistakes.

(*) Dynptr previously manually verified (but no bugs found)

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.
  - Hard. Messy. Tedious. So omit parts. And omit bugs.
- 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)

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

Case Study: AODV Routing Protocol

- Ad hoc, loop-free routing protocol.
- Checked three implementations
  - Mad-hoc
  - Kernel AODV (NIST 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.

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	Cann'ic
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.
  - ~1 bug per 300 lines of code.

Classification of Bugs

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

Model checking vs static analysis (SA)

	CMC & SA	CMC only	SA only
Mishandling malloc failures	11	1	8
Memory leaks	8		5
Use after free	2		
Invalid route table entry		1	
Unexpected message		2	
Invalid packet generation		7	
Program assertion failures		3	
Routing loops		7	
Total bugs	21	21	13

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 #1

- 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:

```
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 #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:
  - Time goes backwards if old message shows up:

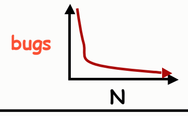
```
cur_rt = getentry(recv_rt->dst_ip);
// bug if: recv_rt->dst_seq < cur_rt->dst_seq!
if(cur_rt && …) {
    cur_rt->dst_seq = recv_rt->dst_seq;
```

Not hard to check, but hard to recoup effort.

Significant model checking win #3

- Gives guarantees much closer to total correctness:
  - Check code statically, run, it crashes. Surprised? No.
  - Crashes after model checking? Much more surprised.

  - 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)



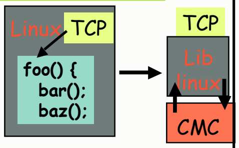
The Talk

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

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)

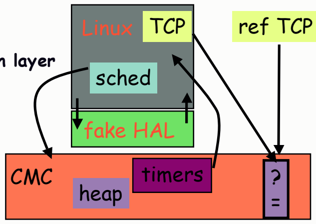
The approach that failed: kernel-lib.c



- 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.

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)

Fundamental law: no run, no bug.

Method	line coverage	protocol coverage	branching factor	additional bugs
Standard client&server	47%	64.7%	2.9	2
+ simultaneous connect	51%	66.7%	3.67	0
+ partial close	53%	79.5%	3.89	2
+ corruption	51%	84.3%	7.01	0
Combined cov.	55.4%	92.1%		

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

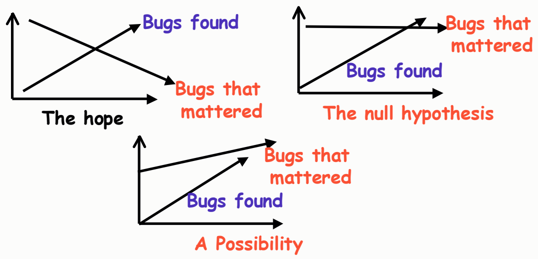
The Talk

- 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.

Open Q: Do static tools really help?



Dangers: Opportunity cost. Deterministic bugs to non-deterministic.

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: 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 on what gets the next biggest set of bugs
  - Easy experiment: bug 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.

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

First question:	"How big is code?"	"What does it do?"
To check?	Must compile	Must run.
Time:	Hours.	Weeks.
Don't understand?	So what.	Problem.
Coverage?	All paths! All paths!	Executed paths.
FP/Bug time:	Seconds to min	Seconds to days.
Bug counts	100-1000s	0-10s
Big code:	10MLOC	10K
No results?	Surprised.	Less surprised.
Crash after check?	Not surprised.	More surprised (much).
(Relatively) better at?	Source visible rules	Code implications & all ways to get errors

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**

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 makes 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.

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.