We found bugs with static analysis and model checking and this is what we learned.

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

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**

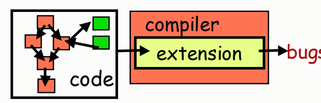
  Reality is a bit more subtle.
  This talk is about that.

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 PREfix, 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

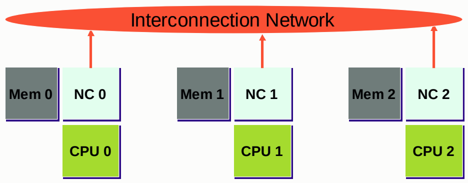
- Main bias:
  - **Static analysis guy that happens to do model checking.**
  - **Some things that surprise me will be obvious to you.**
- The talk is not a jeremiad against model checking!
  - **We want model checking to succeed.**
  - **We're going to write a bunch more papers on it.**
  - **Life has just not always been exactly as expected.**
- Of course
  - **This is just a bunch of personal case studies**
    - **tarted 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

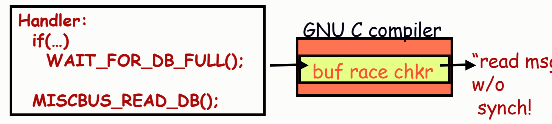
Case Study: FLASH

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



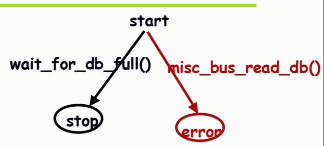
Finding FLASH bugs with static analysis

- 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



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 dealloc	94	9	25
Messages can only be sent on pre-specified lanes	220	2	0
Total	355	33	28

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

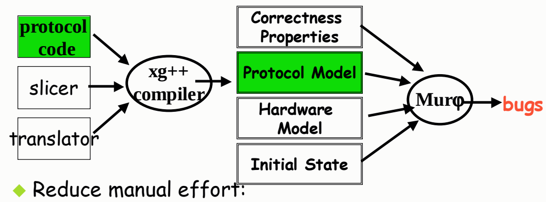
```
Rule "PI Local Get (Put)"
 1:Cache.State = Invalid
     & ! Cache.Wait
 2: & ! DH.Pending
 3: & ! DH.Dirty ==>
  Begin
 4: Assert !DH.Local;
 5: DH.Local := true;
 6: CC_Put(Home, Memory);
  EndRule;
```

FLASH

```
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
  - 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

```
sm len slicer {
 decl any_expr type, data, keep, swp, wait, nl;

 all:
      // match all uses of length field
      { nh.len }
      // match all uses of directory entries
   || { hl.Local } || { hl.Dirty } || { hl.List }
      // match all network and processor sends
   || { NI_SEND(type, data, keep, swp, wait, nl); }
   || { PI_SEND(type, data, keep, swp, wait, nl); }
        ➔ { mgk_tag(mc_stmt); }
   ;
}
```

Model checking results [ISCA'01]

Protocol	Errors	Protocol (LOC)	Extracted (LOC)	Manual (LOC)	Metal (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 win.
Two deep errors.
Dynptr 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.
  - 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 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
  - At most one route table entry per destination
  - Hop count is infinity or <= nodes in network
  - Hop count on sent packet cannot be 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.

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

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)

	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

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 SA 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;
```

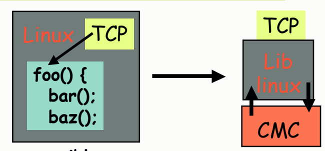
The Talk

- 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

- "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)

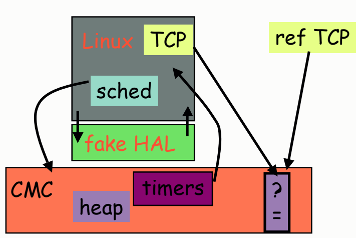
The approach that failed: kernel-lib.c



- The obvious approach:
  - Rip TCP out
- Where to cut?
  - Conventional wisdom: as small as possible.
  - Basic question: calls foo(). Fake foo() or include?
  - Faking takes work. Including leads to transitive 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.

Instead: jam Linux into CMC.

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



- Cost: State ~300K, each transition ~5ms

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%		

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

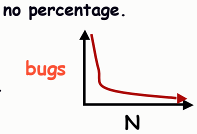
- An introduction
- Case I: FLASH
- Case II: AODV
- Case III: TCP
  - Myth: model checking does not have false positives
  - Environment is really hard. We're not kidding.
  - Executing lots of code not easy, either.
- A more refined view
- Some religion
- A summary

Where static wins.

	Static analysis	Model checking
	Compile → Check	Run → Check
Don't understand?	So what.	Problem.
Can't run?	So what.	Can't play.
Coverage?	All paths! All paths!	Executed paths.
First question:	"How big is code?"	"What does it do?"
Time:	Hours.	Weeks.
Bug counts	100-1000s	0-10s
Big code:	10MLOC	10K
No results?	Surprised.	Less surprised.

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.



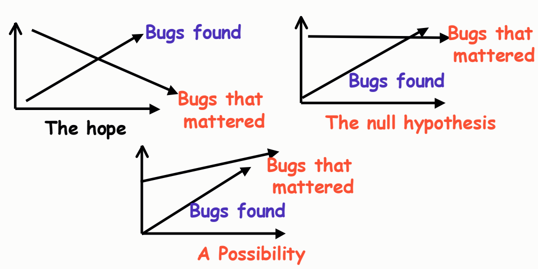
The Talk

- An introduction
- Case I: FLASH
- Case II: AODV
- Case III: TCP
- A more refined view
- Some questions & some dogma
- 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 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?



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
  - **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 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 same 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