DE11
DE14

Using model checking and execution generated tests to find bugs in real code

Dawson Engler
Cristian Cadar, Junfeng Yang, Ted Kremenek, Paul Twohey
Stanford

Slide 1

DE11 old formal verification technique that hasn't met much success with software. other is a very new one that optimistic abou

Dawson Engler, 8/4/2005

DE14 Dawson Engler, 8/4/2005

DE10

Background.

- Lineage
  - Did thesis work building a new OS (exokernel)
  - Spent most of last 7 years developing static techniques to find bugs in them
- The goal: find as many serious bugs as possible.
- This talk: two dynamic methods of finding bugs
  - Implementation-level model checking (since 2001)
  - Execution generated executions (this year)

  - Next: 1 minute overview of 7 years of static checking.

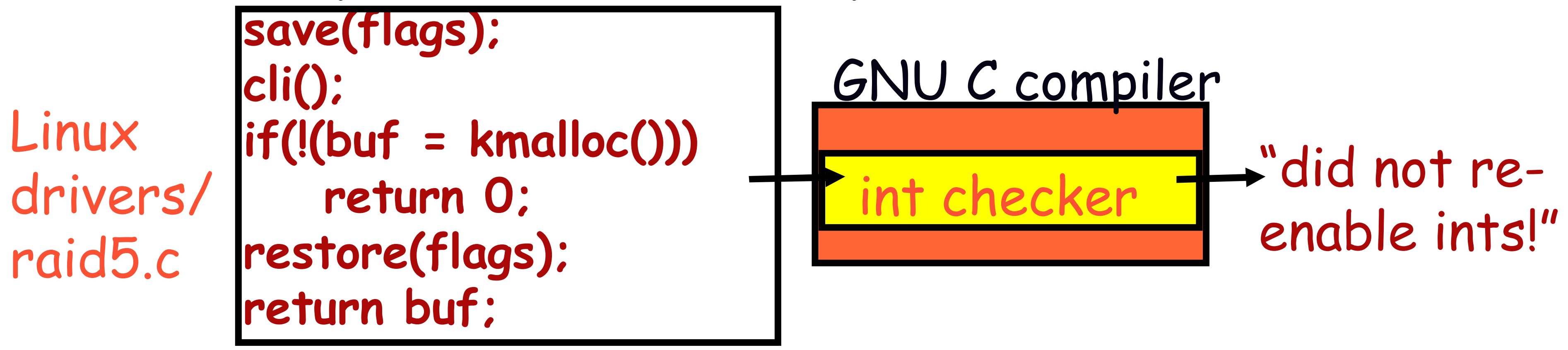
Slide 2

DE10 very slow process to generate bugs.
this was pretty hard, so spent the last 1/4th of
my life coming up with static techniques to find bugs.

our main religion is to find as many serious bugs as possible.
Dawson Engler, 8/4/2005

Context: finding bugs w/ static analysis

- Systems have many ad hoc correctness rules
  - "sanitize user input before using it"; "re-enable interrupts after disabling" "acquire lock l before using x"
  - One error = compromised system
- If we know rules, can check with extended compiler
  - Rules map to simple source constructs
  - Use compiler extensions to express them



```
save(flags);
cli();
if(!(buf = kmalloc()))
    return 0;
restore(flags);
return buf;
```

  - Nice: scales, precise, statically find 1000s of errors

High bit: Works well.

- A bunch of checkers:
  - System-specific static checking [OSDI'00] (Best paper)
  - Security checkers [Oakland'02] & annotations [CCS'03]
  - Race conditions and deadlocks [SOSP'03]
  - Path-sensitive memory overflows [FSE'03]
  - Others [ASPLOS'00,PLDI'02,PASTE'02,FSE'02(award)]
- Novel statistical-based program analysis
  - Infer correctness rules [SOSP'01]
  - Z-ranking [SAS'03]
  - Correlation ranking [FSE'03]
- Commercialized(ing): Coverity
  - Successful enough to have marketing dept: next 2 slides.

Coverity's commercial history

Breakthrough technology out of Stanford	Company incorporated	Achieved profitability	Product growth and proliferation
2000-2002	2002	2003	2004-05
• Meta-level compilation checker ("Stanford Checker") detects 2000+ bugs in Linux.	• Deluge of requests from companies wanting access to the new technology. • First customer signs: Sanera systems	• 7 early adopter customers, including VMWare, SUN, Handspring. • Coverity achieves profitability.	• Version 2.0 product released. • Company quadruples • 70+ customers including Juniper, Synopsys, Oracle, Veritas, nVidia, palmOne. • Self funded

A partial list of 70+ customers…

EDA
SYNOPSYS
Mentor Graphics
cadence

Storage
VERITAS
EMC² where information lives
panasas

Security
McAfee
PGP
Check Point SOFTWARE TECHNOLOGIES LTD.

Networking
Cisco Systems
Juniper NETWORKS
Marconi
AVICI SYSTEMS

Government
NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
Jet Propulsion Laboratory
California Institute of Technology

Embedded
QUALCOMM
symbol The Enterprise Mobility Company
KEITHLEY

Biz Applications
ORACLE
bmcsoftware

OS
WIND RIVER
palmOne
Sun microsystems

Open Source
MySQL
SLEEPYCAT SOFTWARE
mozilla
Apache

Talk: dynamic techniques a static guy likes.

- Static works well at checking surface visible rules
  - lock() paired with unlock(), don't deref tainted pointer…
  - Good: All paths + no run code = large bug counts.
  - Bad: weak at checking properties *implied* by code.
- Implementation-level model checking:
  - How to use formal verification techniques on real code.
  - Several years of mixed results, then one big win.
- Execution generated testing
  - Use code to automatically make up its inputs
  - Very new, but *very* optimistic.

Slide 7

DE12 seconds to minutes to inspect. quick. dynamic much harder typically.
Dawson Engler, 8/4/2005

Model checking

- Formal verification technique
  - Verification via "exhaustive testing": Do every possible action to every possible system state.
  - Lots of tricks to make exponential space "tractable"
  - Works well for hardware.
  - Software = not so clear. Successes: Wagner's MOPS, Mitchell's protocol work. But many failures.
- We've used for several years for bug hunting.
  - Do every possible action to a state before going to next
  - Main lever: makes low-probability events as common as high-probability = quickly find corner-case errors.
  - FLASH cache coherence protocol code [ISCA'01]
  - AODV ad-hoc routing protocol [OSDI'02]
  - Linux TCP [NSDI'04]
  - Storage systems [OSDI'04,BUGS'05]. (this talk)

Model checking file system code.

- Why:
  - File system bugs are some of most serious possible.
  - Extremely difficult to test: FS must be able to crash at *any* point and *always* recover to a valid state.
- How: CMC
  - Implementation-level model checking (similar to Verisoft)
  - Main trick: run entire Linux kernel in model checker.
- Results:
  - Checked 3 heavily-tested, widely-used Linux file systems: ext3, JFS, ReiserFS.
  - Found 32 bugs in total
  - Best result: 10 cases where crash = lost metadata or entire directories, including the root "/" directory.
  - Bugs considered serious: most patched within a day.

A four-slide crash course in model checking

- How to explore every FS state?
  - Start from empty, formatted disk.
  - Foreach possible operation OP, clone disk, apply OP

External actions

mkdir()
rmdir()
creat()
link()
unlink()
mount()
unmount()
...
CRASH

X

Internal actions

fail: malloc()
fail: diskread()
fail: cachelookup();
...
run: journal thread
...
write: block_i + crash

  - Check generated FS: if wrong, emit error, discard.
  - Cannonicalize + hash FS: If seen skip, otherwise explore

The core model checking algorithm

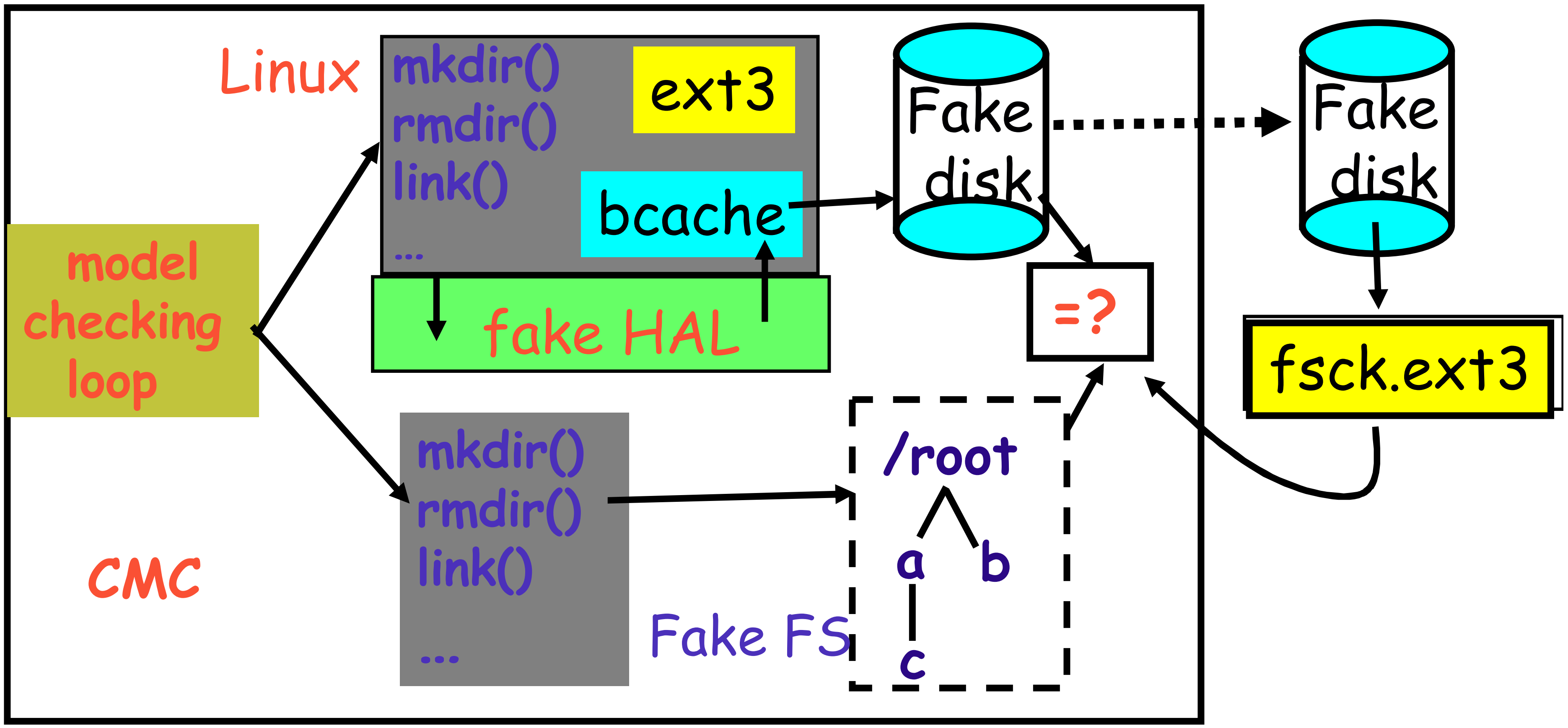
```
create initial state: S_0;
wl = {S0};      // initialize worklist

while(S = dequeue(wl)) {
  foreach op in S.ops
          // run "op" on copy of current state "S"
         S' = checkpoint(do_op(op, restore(clone(S))))
        if(check_state(S') != valid)
              error "Error!";
        else if(hash_lookup(hash(S')) == false)
             enqueue(wl, S')
```

- Do all possible actions to state S before going to next
- Viewing state as first class entity changes mindset.

- Main issue: do_op(), clone(), checkpoint(), restore().

Galactic view of system: FiSC



Main trick: Run entire linux kernel inside model checker.

clone() ~ fork(). checkpoint()/restore() ~ context switch

An OS hackers guide to building CMC

- What maps to what
  - clone() ~ fork()
  - checkpoint()/restore() ~ context switch. Copy heap, stack, globals to/from temporary memory.
  - run_op(): call into linux system call (mkdir(), rmdir(), …)

- How to make initial state?
  - Call init(), hack it so that when finishes returns to CMC
  - Checkpoint state, put on worklist.
  - Start checking.

How to check crashes?

- When FS adds new dirty blocks to buffer cache
  - Write out “every” possible order of all dirty blocks
  - Copy disk to scratch memory, mount as FS, run fsck().
  - Check to see if recovered FS is what we expect.
- During recovery too!
  - Cliched error: don’t handle crash during recovery
  - After “every” write fsck does to disk, crash & restart.
  - Should produce same disk: fsck(d) = fsck(crash(fsck(d)))
- Speed: fsck is a deterministic function
  - Simple: cache result of fsck(d)→d’ so don’t have to run.
  - Advanced: after an fsck write, don’t crash and rerun if would not change any value read by re-run fsck.

Plugging a file system into CMC

- Since CMC runs Linux, checking a Linux FS “simple”

- Need:
  - Fs utilities: mkfs, fsck
  - Specify which functions mark buffers as dirty.
  - Minimum disk and memory sizes (2MB, 16 pages for ext3)
  - Function to compute what FS must recover to after crash

- Takes roughly 1-2 weeks for Junfeng to do.

The two goals of model checking

- Expose all choice
  - Model checker needs to see these so it can try all behaviors.
- Do not explore "the same" state (choice)
  - State space is exponential
  - Exposing choice makes it more.

  - Many superficially different states ~ semantically same.
  - SO: Collapse, skip.

Checking all actions

- External actions (mkdir, write(), …)
  - Add as option in main model checking loop
  - Model abstract semantics so can apply to abstract FS.
- Environmental and internal actions:
  - In general: a operation will do a set of specific actions that, legally, could have been different ("choice points")

  - cachelookup() returns a pointer, could have returned null
  - diskread() returns block, could have returned error
  - timer does not expire, could have expired.
  - buffer cache entry is not written, could have been.

  - The first order win of model checking: in each state, do *each* of these choice point possibilities.

Mechanics of choice points

- If code hit N choice points, rerun 2^N times, doing each one differently.
  - Run on S, let kmalloc and cachelookup succeed
  - Run on S, only fail kmalloc
  - Run on S, only fail cachelookup
  - Run on S, fail both
  - (If hit new choice points, also do)

```
sys_mkdir(...) {
    ...
    if(!(b = cachelookup(...))
        readblock(b);
    ...
    if(!(p = kmalloc(...)))
        return -ENOMEM;
    ...
```

- Optimize: only do one "failure" choice per transition
  - Boring Error: "Bug if diskread fails & then kmalloc fails"

Adding a choice point to file system code

- Easy: insert call to "cmc_choose(N)"

  cmc_choose will return to callsite N times with return values 0, 1, …, n-1

```
struct block* read_block(int i) {
    struct block *b;
    if ((b = cache_lookup(i)))
        if(cmc_choose(2) == 0)
                return b;
   return disk_read (i);
}
```

Returns two times,
1st = return 0
2nd = return 1

- Insert whenever specification lets code do one of several actions.

Slightly subtle: make abstract concrete

- Spec:“foo() returns NULL when allocation fails”
  - **Where to put choice point?**

```
T *foo (…) {
   …
        if(…)
            if(…)
                 p = malloc();
```

Slightly subtle: make abstract concrete

- Spec:"foo() returns NULL when allocation fails"
- Wrong: put choice buried down where implementation does

```
T *foo (…) {
    …
        if(…)
            if(…)
                if(cmc_choose(2) == 0)
                        return NULL;
               else
                      p = malloc();
```

Will only make choice when implementation does, rather than when *could* make (always).

Slightly subtle: make abstract concrete

- Right: Put error choice as first action so always happens no matter what path implementation takes

```
T *foo (…) {
    if(cmc_choose(2) == 0)
        return NULL;
    …
        if(…)
            if(…)
                p = malloc();
```

- More extreme: insert error choice even if *implementation* of foo() doesn't allocate memory!

After expose choice, cut down exponentials.

- Downscale: less stuff = less states to explore.
  - **Small disks. 2MB for ext3**
  - **Small memory. 16 pages for ext3**
  - **Tiny FS topology. 2-4 node**
- Canonicalization: remove superficial differences
  - **General rule: setting things to constants: e.g. inode generation #, mount count**
  - **Filenames. "x", "y", "z" == "1", "2", "3"**
- Exponential space, but not uniformly interesting
  - **Guide search towards "more interesting" states**
  - **Bias towards states with more dirty buffers, new lines of code executed, new FS topologies, …**

Slide 23

DE13 set everything to constants: inode gen #, mount counts, time fields, zeroing freed memory, unused disk blocks
Dawson Engler, 8/4/2005

DE6

Results

Error Type	VFS	ext2	ext3	JFS	Reiser	total
Data loss	N/A	N/A	1	8	1	10
False clean	N/A	N/A	1	1		2
Security		2	2	1		3 + 2
Crashes	1			10	1	12
Other	1		1	1		3
Total	2	2	5	21	2	32

32 in total, 21 fixed, 9 of the remaining 11 confirmed

Slide 24

DE6 security: happened on mkdir/creat: did lookup of name, lookup rfails if (1) name does not exist or
(2) memory allocation fails. code did not differentiate. so under low memory conditions attacker could create files with same name in directory to hijack.

reiser just really big and slow. jfs guys responsive, so hit on. also have a really bad dynamic w.r.t. journal abort.
Dawson Engler, 8/4/2005

DE7

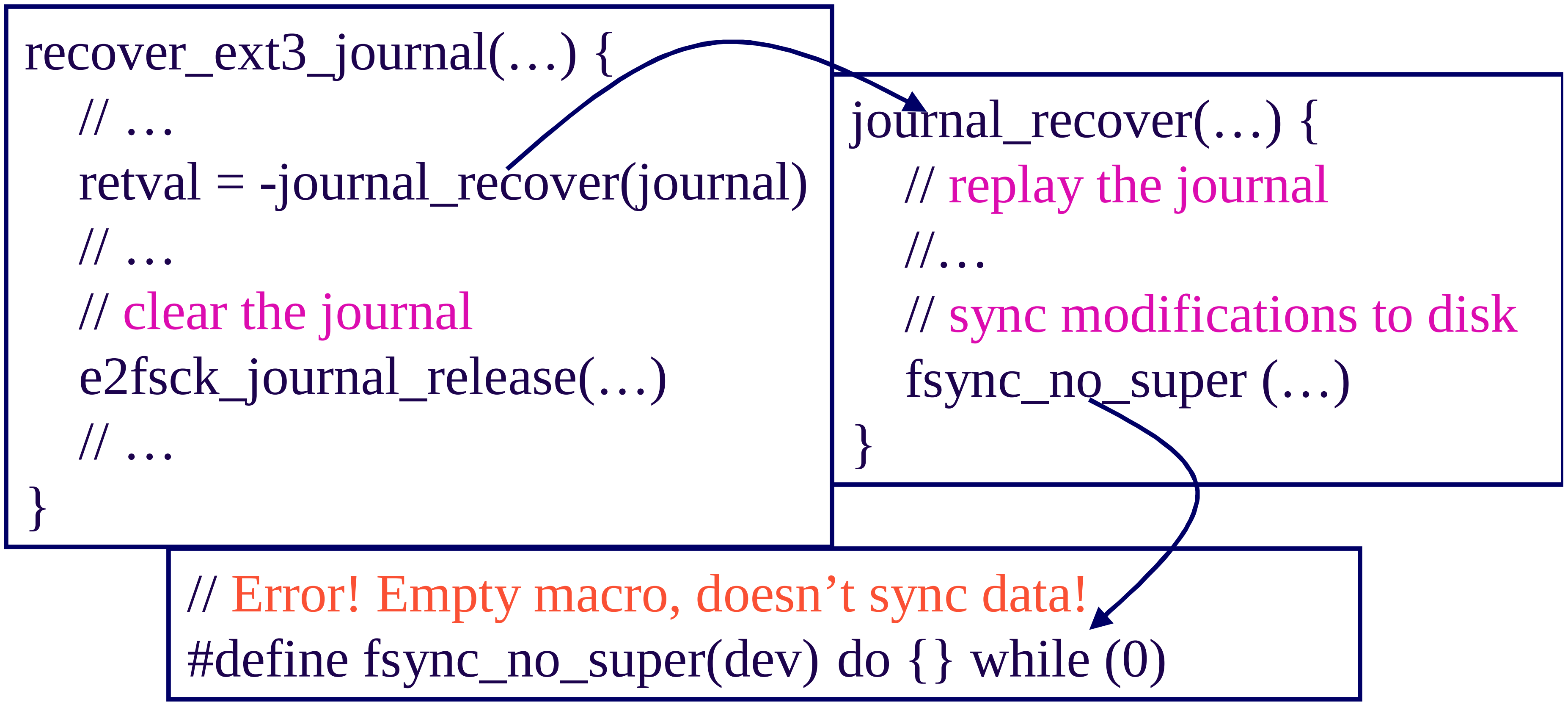
Root cause of worst errors: journalling bugs

- To do an operation:
  - Record effects of operation in log ("intent")
  - Apply operation to in-memory copy of FS data
  - Flush log (so know how to fix on-disk data). wait()
  - Flush data.
  - All FSes we check get this right.
- To recover after crash
  - Replay log to fix FS.
  - Flush FS changes to disk. wait()
  - Clear log. Flush to disk.
  - All FSes we check get this wrong.

Slide 25

DE7 like all reliability based on duplication: duplicate intent and then action. make intent persistent, then flush out screwed up datastructures.
Dawson Engler, 8/4/2005

ext3 Recovery Bug



```
recover_ext3_journal(…) {
    // …
    retval = -journal_recover(journal)
    // …
    // clear the journal
    e2fsck_journal_release(…)
    // …
}
```

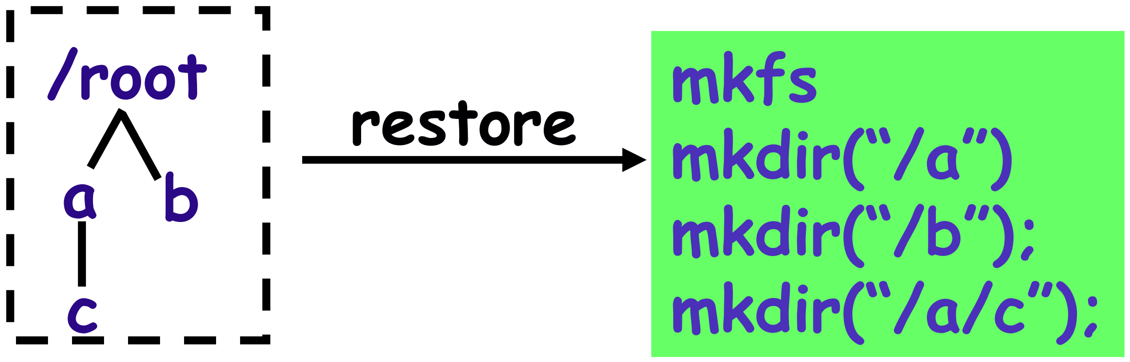
```
journal_recover(…) {
    // replay the journal
    //…
    // sync modifications to disk
    fsync_no_super (…)
}
```

```
// Error! Empty macro, doesn't sync data!
#define fsync_no_super(dev) do {} while (0)
```

- Code was directly adapted from the kernel
- But, fsync_no_super was defined as NOP

Recent work: making 10x lighter-weight

- Running Linux in model checker incredibly invasive.
  - **Hard to check different OS version, OS, or applications**
- Only needed so could checkpoint and restore! Instead
  - **Checkpoint state by recording choices from initial disk**
  - **Restore state by: mounting copy of initial disk and applying these choices to copy of initial disk**



```
mkfs
mkdir("/a")
mkdir("/b");
mkdir("/a/c");
```

- Result: reduces to single ramdisk driver plus model checking process

Results from version 2.0 (EXPLODE)

- Version control bugs:
  - CVS: "cvs commit" does not
  - Bitkeeper: "bk pull" + crash = completely wasted repo
  - Subversion: crash = corruption.
- Linux RAID:
  - does not reconstruct bad sectors: marks disk as faulty, removes from RAID, returns error.
  - Two sectors go bad on two disks, almost all reconstruction fails.
- NSF: write file, then read through hardlink = different result.

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.	Big 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).
Better at?	Source visible rules	Code implications & all ways to get errors

Model checking summary

- Main trick:
  - Do every possible action to a state before going to next
  - Makes low-probability events as probable as high.
  - Mechanics: (1) expose all choice points, (2) collapse semantically isomorphic states.
- Works well when:
  - Bugs are very serious (to recoup pain)
  - Huge number of possible interleavings (to beat testing)
- Checking storage systems a good case
  - Main source of errors: must always be able to recover to valid file system state no matter where crash occurs.
  - Very difficult to reason about.
  - Result: found serious errors in every system we checked.

Execution generated testing
How to make code blow itself up

Dawson Engler & Cristian Cadar
Stanford University

Goal: find many bugs in systems code

- Generic features:
  - Baroque interfaces, tricky input, rats nest of conditionals.
  - Enormous undertaking to hit with manual testing.
- Random "fuzz" testing
  - Charm: no manual work
  - Blind generation makes hard to hit errors for narrow input range
  - Also hard to hit errors that require structure

```
int bad_abs(int x) {
   if(x < 0)
        return -x;
   if(x == 12345678)
        return -x;
   return x;
}
```

- The rest of this talk: a simple trick to finesse.

EGT: Execution generated testing[SPIN'05]

- Basic idea: use the code itself to construct its input!
- Basic algorithm:
  - Symbolic execution + constraint solving.
  - Run code on symbolic input, initial value = "anything"
  - As code observes input, it tells us values input can be.
  - At conditionals that use symbolic input, fork
    - On true branch, add constraint that input satisfies check
    - On false that it does not.
  - Then generate inputs based on these constraints and re-run code using them.

The toy example

```
int bad_abs(int x) {
    if(x < 0)
        return -x;
    if(x == 12345678)
        return -x;
    return x;
}
```

- Initialize x to be "any int"
- Code will return 3 times.
- Solve constraints at each to get our 3 test cases.

```
int bad_abs_egt(int x) {
    if(fork() == child)
        set(x < 0 && ret = -x);
        return ret;
    else
        set(x >= 0);
    if(fork() == child)
        set(x = 12345678);
        set(ret = -x);
        return ret;
    else
        set(x != 12345678);
    set(ret = x);
    return ret;
```

The big picture

- Implementation prototype
  - Do source-to-source transformation using Cil
  - Use CVCL decision procedure solver to solve constraints, then re-run code on concrete values.
  - Robustness: use mixed symbolic and concrete execution.

- Three ways to look at what's going on
  - Grammar extraction.
  - Turn code inside out from input consumer to generator
  - Sort-of Heisenberg effect: observations perturb symbolic inputs into increasingly concrete ones. More definitive observation = more definitive perturbation.

- Next: one transformation, some results.

Mixed execution

- Basic idea: given an operation:
  - If all of its operands are concrete, just do it.
  - If any are symbolic, add constraint.

  - If current constraints are impossible, stop.
  - If current path causes something to blow up, solve+emit.
  - If current path calls unmodelled function, solve and call.
  - If program exits, solve+emit.

- How to track?
  - Use variable address to determine if symbolic or concrete
  - Note: symbolic assignment not destructive. Creates new symbol.

Example transformation: "+"

```
T plus_rule(T x, T y) {
   if(x and y are concrete)
       return (concrete=x.concrete+y.concrete,<invalid>);
   s = new symbolic var T;
  if(x is concrete)
       add_constraint(s = x.concrete + y.symbolic);
  else if(y is concrete)
       add_constraint(s = x.symbolic + y.concrete);
  else
       add_constraint(s = x.symbolic + y.symbolic);
  return (concrete=<invalid>, symbolic = s);
```

- Each var v has v.concrete and v.symbolic fields
  - If v is concrete, symbolic=<invalid> and vice versa.

Micro-case study: Mutt's UTF8 routine

- Versions <= 1.4 have buffer overflow.
  - Used as main working example in recent OSDI paper, which used carefully hand-crafted input to exploit.
  - Took routine, ran through EGT, immediately hit error.
  - Assumingly OSDI paper suggested increasing malloc from n*2 bytes to n*7/3. We did this and reran. Blew up.
  - For input size 4 it took 34minutes to generate 458 tests that give 96% statement coverage.

Case study: printf

- Typical of systems code:
  - Highly complex, tricky interface.
  - Format string = exceptionally ugly, startling language
- Nice for EGT:
  - complex conditionals = hard to test
  - mostly flat comparisons = easy to solve
  - multiple implementations = easy to check correctness.
- Checked:
  - PintOS printf (developer reads standards for fun)
  - Reduced-functionality printf for embedded devices.
  - GCCfast printf

Printf results

- Made format string symbolic. Test cases for size 4:
  - **Pintos: 40 minutes, 3234 cases, 95% coverage**
  - **Gccfast: 87 minutes, 2105 cases, 98% coverage**
  - **Embedded printf: 21 minutes, 337 cases, 95% coverage**

DE8

- Representative subset:

```
"%lle"   " %#0f"  "%g%."  " % +l"  "%#he"  "  %00."
" % #c"   "%----"  " %lG"   "%c%j"   " %9s%"   " %0g"
"  %.E"  " %+-u"   " %9d"  " %00"  " %#+-"   " %0  u"
```

Slide 40

DE8 code expects these. but i don't know what most of these mean. charm is you don't have to: automatically extracted

Dawson Engler, 8/4/2005

Two Bugs

- Incorrect grouping of integers.

```
printf("%'d", -155209728);
```

  - prints "-15,5209,728" instead of "-155,209,728"
  - "Dammit. I thought I fixed that." -- Ben Pfaff

- Incorrect handling of plus flags
  - "%" followed by a space means "a blank should be left before a positive number (or empty string) produced by a signed conversion".
  - Pintos incorrectly leaves a blank before an unsigned flag.

  - "This case is so obscure I never would have thought of that" --- Ben.

Server case study: Wsmp3

- Mp3 server
  - 2000 lines
  - Version 0.0.5 contains one known security hole.
  - We found this, plus three other overflows + one inf loop
- Simple:
  - Make recv input symbolic:

```
ssize_t recv_model(int s, char *buf, size_t len, int flags) {
      egt_make_bytes_symbolic(buf, msg_len);
      return msg_len;
}
```

  - ./configure && make && run
  - Re-feed packets back into wsmp3 while running valgrind.

DE5

Two EGT-found WsMp3 bugs

- Network controlled infinite loop:

```
// cp points to 6th character in msg.
while(cp[0] == '.' || cp[0] == '/')
      for(i =1; cp[i] != 0; i++) {
              ...
      }
}
```

- A buffer overflow

```
op = malloc(10);
…
// buf points to network msg
for(i = 0; buf[i] != ' '; i++)
    op[i] = buf[i];
```

Slide 43

DE5

6th msg character is "." or "/" followed by {.|/}*
then a 0 will inf loop

buf points to message: if no spaces in first 10 characters = buffer overflow
Dawson Engler, 8/4/2005

Related work

- Static test generation
  - Weak. Run into intractable problems promptly.
- Dynamic test generation
  - Much: use manually-written specification.
  - Others: try to hit a given program point.
  - Nothing on getting comprehensive coverage for real code
- Concurrent work: DART [pldi05].
  - Very similar. Does not handle pointers, commits to values early. Better at extracting code from environment.

Conclusion

- EGT [SPIN'05]:
  - Automatic input generation for comprehensive execution.
  - Make input symbolic. Run code. If operation concrete, do it. If symbolic, track constraints. Generate concrete solution at end (or on way), feed back to code.
  - Finds bugs in (small) real code.
    Zero false positives.
- Combine with
  - model checking "choice" to control environment decisions
  - static inference of rules (all paths = lots of data)

Current work

- Purify on steroids: partially symbolic dynamic checking
  - **Rather than "was there a memory overflow" to "could there have been a memory overflow?"**
- Deep checking of network facing code.
  - **Random tests = hard to hit packet of death.**
  - **EGT: jam symbolic packet through, automatically discover ones that drive code down different paths.**
- Automatic cross checking
  - **Take complex code and automatically discover how to drive; then cross check across multiple implementations**
  - **Standard libraries (like printf), V2.22 of code vs V2.23**
- GUIs: standard wisdom = impossible to test
  - **Generate "symbolic event" that can be anything, follow.**

Conclusion

- Static checking
  - Works well with surface visible rules
  - Big advantages: All paths. No run code.
  - Weak at: checking code implications. Dynamic opposite.
- Model checking
  - Do all actions to a state.
  - Expose choice. Ignore superficial differences.
  - Works if bugs serious, lots of potential actions.
- Execution generated testing
  - Automatic input generation for exhaustive testing.

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

Shocked: when they checked the same static won. Most bugs shallow: only missed 1! Found with model checking. Means bugs were relatively shallow, which was surprising. Also means that model checking missed them, which I found astounding. In the end, model checking beat it, but it's not entirely clear that is how it has to be.

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.**

–Find bugs no easily visible to inspection.E.g., tree is balanced, single cache line copy exists, routing table does not have loops

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);
```

Finds errors without having to anticipate all the ways that these errors could arise. In contrast, static analysis cannot do such end to end checks but must instead look for specific ways of causing an error.

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.

```
ERROR: syntaxerror
OFFENDING COMMAND: --nostringval--

STACK:

true
true
```