A crash course on some recent bug finding tricks.

Junfeng Yang, Can Sar, Cristian Cadar, Paul Twohey

Dawson Engler

Stanford

Background

- Lineage
  - Thesis work at MIT building a new OS (exokernel)
  - Spent last 7 years developing methods to find bugs in them (and anything else big and interesting)
- Goal: find as many serious bugs as possible.
  - Agnostic on technique: system-specific static analysis, implementation-level model checking, symbolic execution.
  - Our only religion: results. Works? Good. No work? Bad.
- This talk
  - eXplode: model-checking to find storage system bugs.
  - EXE: symbolic execution to generate inputs of death
  - Maybe: weird things that happen(ed) when academics try to commercialize static checking.

EXPLODE: a Lightweight, General System for Finding Serious Storage System Errors

Junfeng Yang, Can Sar, Dawson Engler
Stanford University

The problem

- Many storage systems, one main contract
  - You give it data. It does not lose or corrupt data.
  - File systems, RAID, databases, version control, ...
  - Simple interface, difficult implementation: failure
- Wonderful tension for bug finding
  - Some of the most serious errors possible.
  - Very difficult to test: system must *always* recover to a valid state after any crash
  - Typical: inspection (erratic), bug reports (users mad), pull power plug (advanced, not systematic)

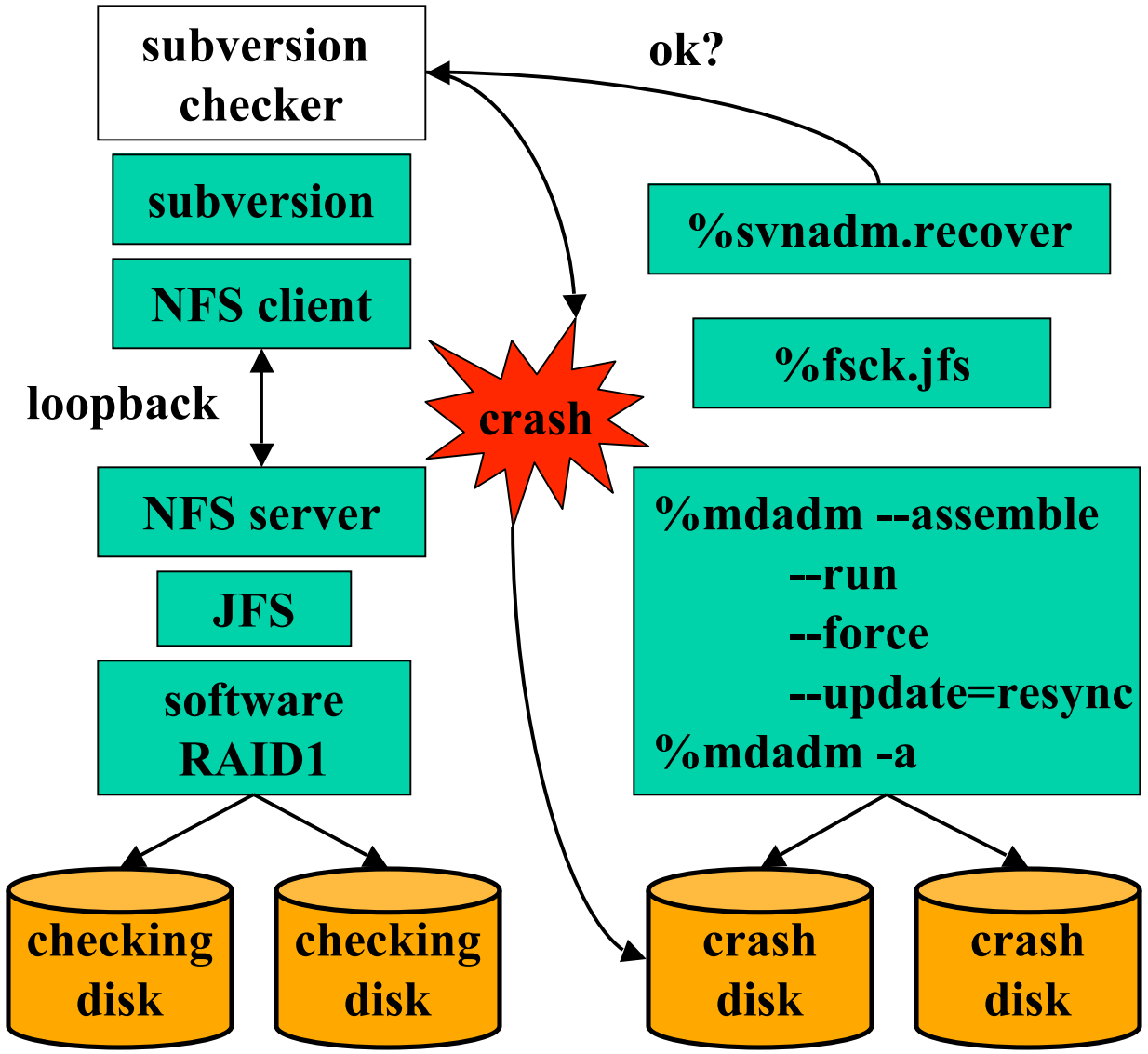
Goal: ***comprehensively*** check ***many*** storage systems with ***little*** work

EXPLODE summary

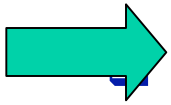
- Comprehensive: uses ideas from model checking
- Fast, easy
  - Check new storage system: 200 lines of C++ code
  - Port to new OS: 1 device driver + optional instrumentation
- General, real: check live systems.
  - Can run (on Linux, BSD), can check, even w/o source code
- Effective
  - checked 10 Linux FS, 3 version control software, Berkeley DB, Linux RAID, NFS, VMware GSX 3.2/Linux
  - Bugs in all, 36 in total, mostly data loss
- This work [OSDI'06] subsumes our old work FiSC [OSDI'04]

Checking complicated stacks

- All real
- Stack of storage systems
  - subversion: an open-source version control software
- User-written checker on top
- Recovery tools run after EXPLODE-simulated crashes



Outline



- Core idea
- Checking interface
- Implementation
- Results
- Related work, conclusion and future work

The two core eXplode principles

- Expose all choice:

When execution reaches a point in program that can do one of N different actions, fork execution and in first child do first action, in second do second, etc.

- Exhaust states:

Do every possible action to a state before exploring another.

- Result of systematic state exhaustion:
  - Makes low-probability events as common as high-probability ones. Quickly hit tricky corner cases

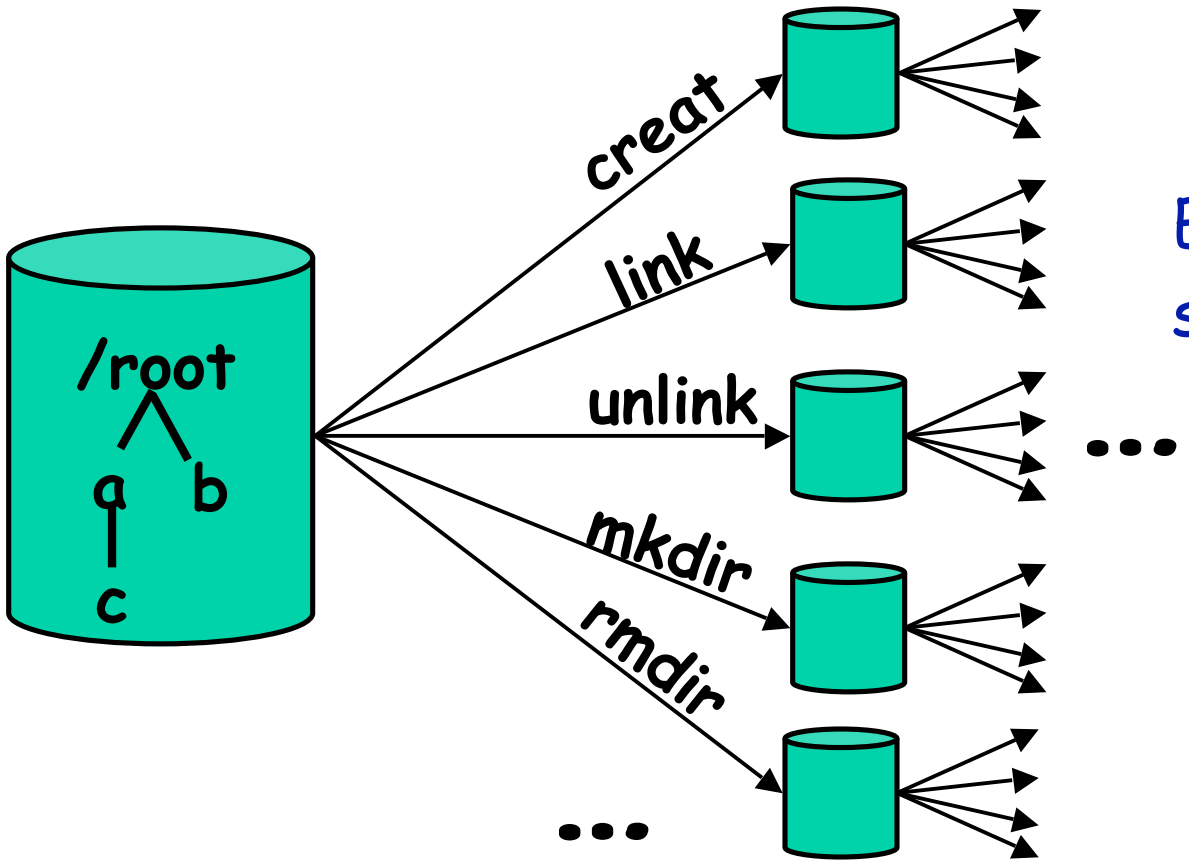
Core idea: explore all choices

- Bugs are often triggered by corner cases
- How to find: drive execution down to these tricky corner cases

When execution reaches a point in program that can do one of N different actions, fork execution and in first child do first action, in second do second, etc.

External choices

- Fork and do every possible operation

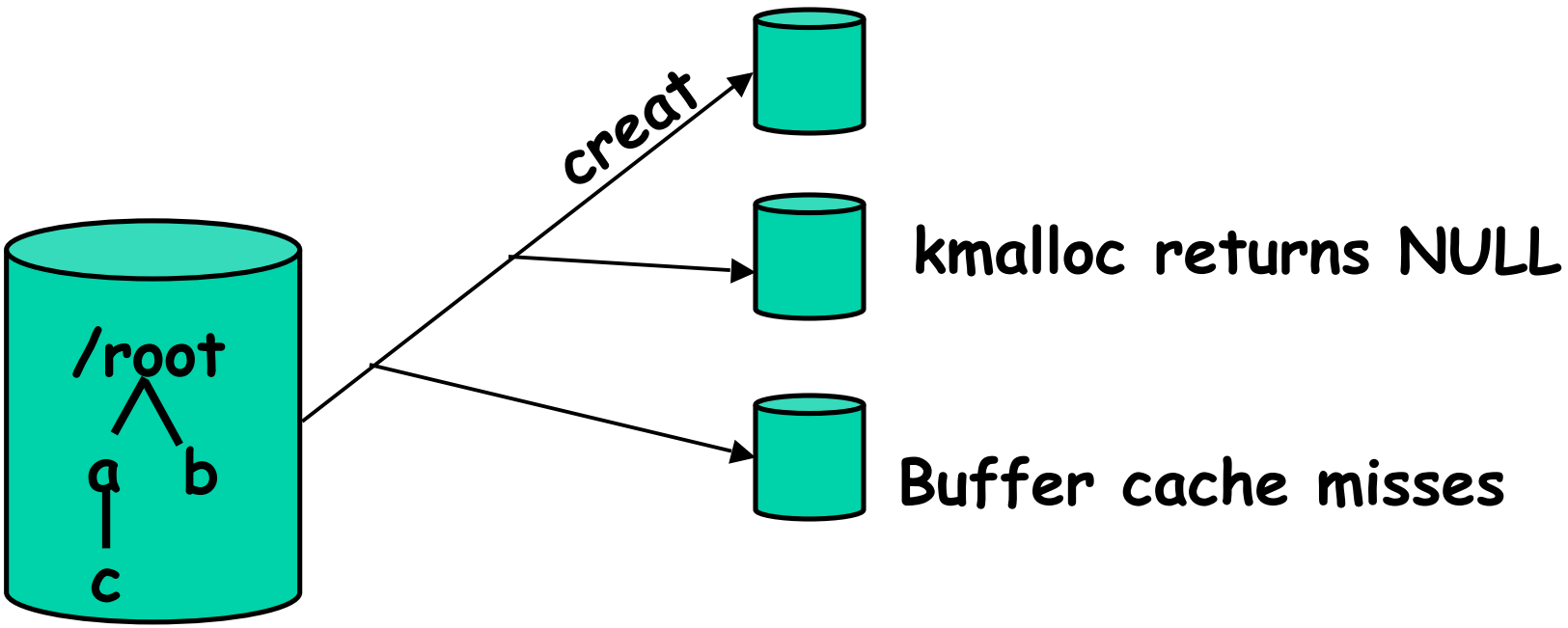


Explore generated states as well

Speed hack: hash states, discard if seen, prioritize interesting ones.

Internal choices

- Fork and explore all internal choices



How to expose choices

- To explore N-choice point, users instrument code using choose(N)

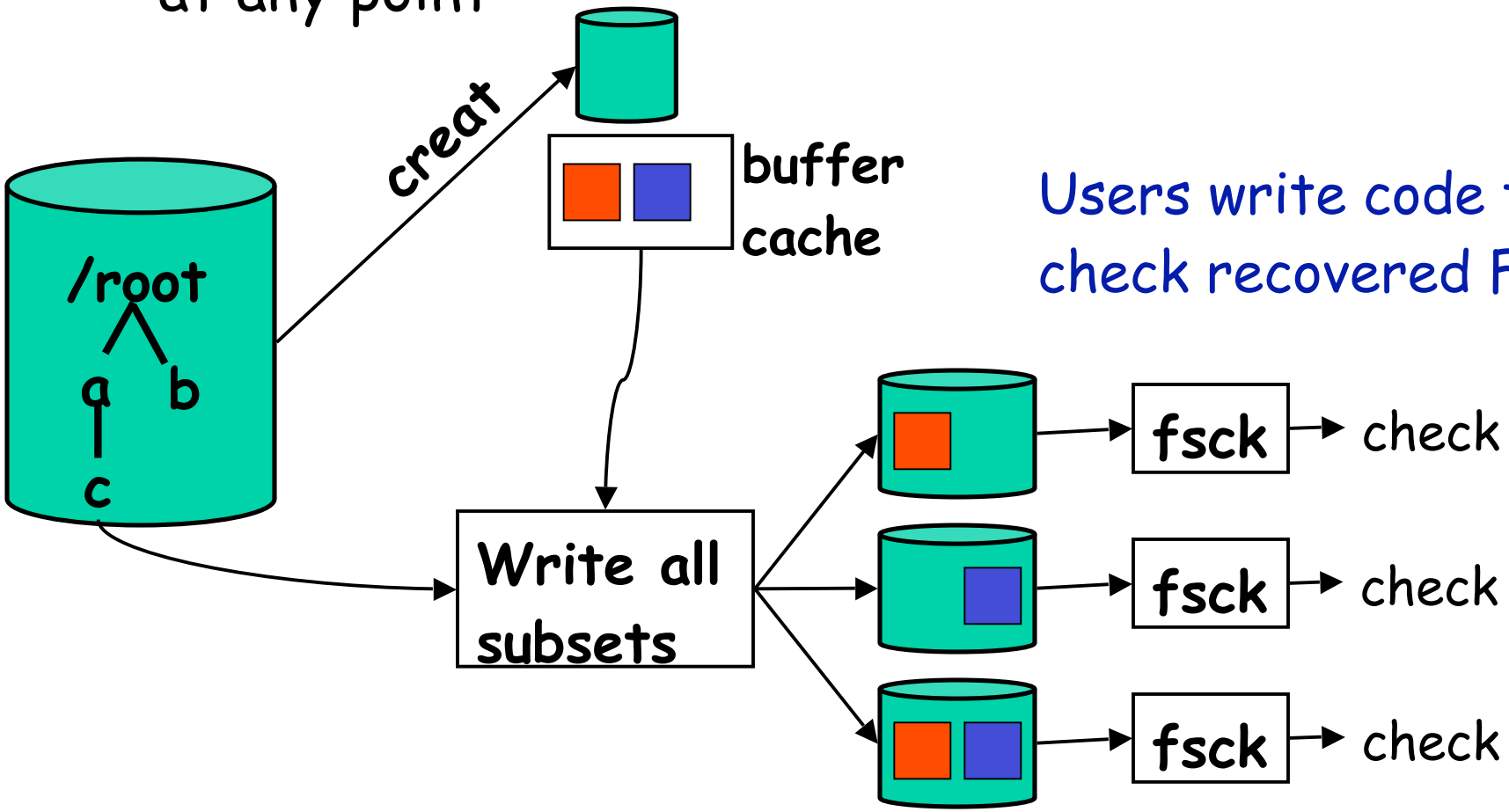
- choose(N): N-way fork, return K in K'th kid

```
void* kmalloc(size s) {
  if(choose(2) == 0)
    return NULL;
  … // normal memory allocation
}
```

- We instrumented 7 kernel functions in Linux

Crashes

- Dirty blocks can be written in any order, crash at any point



Outline

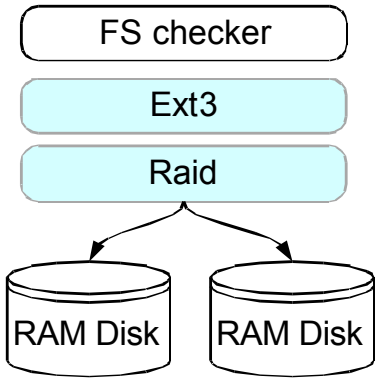
- Core idea: exhaustively do all verbs to a state.
  - external choices X internal choices X crashes.
  - This is the main thing we'd take from model checking
  - Surprised when don't find errors.
- Checking interface
  - What EXPLODE provides
  - What users do to check their storage system
- Implementation
- Results
- Related work, conclusion and future work

What EXPLODE provides

- choose(N): conceptual N-way fork, return K in K'th child execution
- check_crash_now(): check all crashes that can happen at the current moment
  - Paper talks about more ways for checking crashes
  - Users embed non-crash checks in their code. EXPLODE amplifies them
- error(): record trace for deterministic replay

What users do

- Example: ext3 on RAID

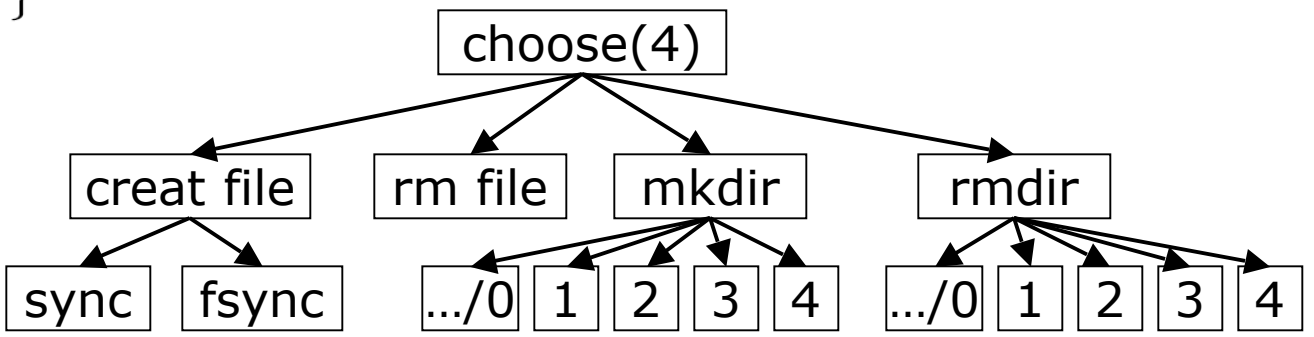


- checker: drive ext3 to do something: mutate(), then verify what ext3 did was correct: check()

- storage component: set up, repair and tear down ext3, RAID. Write once per system

- **FS Checker**
  - **mutate**
- ext3 Component
- Stack

```
const char *dir = "/mnt/sbd0/";
const char *file = "/mnt/sbd0/test-file";
void FsChecker::mutate(void) {
  switch(choose(4)) {
  case 0: systemf("echo \"test\" > %s", file);
    if (choose(2) == 0)
      sync();
    else {
      do_fsync(file);
      // fsync parent to commit the new directory entry
      do_fsync("/mnt/sbd0");
    }
    check_crash_now(); // invokes check() for each crash
    break;
  case 1: systemf("rm %s", file); break;
  case 2: systemf("mkdir %s%d", dir, choose(5)); break;
  case 3: systemf("rmdir %s%d", dir, choose(5)); break;
  }
}
```



- **FS Checker**
  - check
- ext3 Component
- Stack

```
void FsChecker::check(void) {
  ifstream in(file);
  if(!in)
    error("fs", "file gone!");
  char buf[1024];
  in.read(buf, sizeof buf);
  in.close();
  if(strncmp(buf, "test", 4) != 0)
    error("fs", "wrong file contents!");
}
```

Check file exists

Check file contents match

Even trivial checkers work:finds JFS fsync bug which causes lost file.

Checkers can be simple (50 lines) or very complex(5,000 lines)

Whatever you can express in C++, you can check

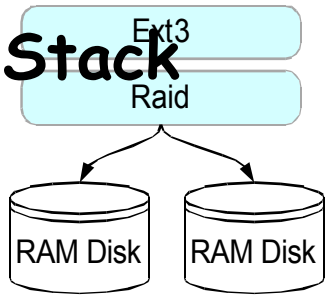
- FS Checker
- **ext3 Component**
- Stack

- storage component: initialize, repair, set up, and tear down your system
  - Mostly wrappers to existing utilities. "mkfs", "fsck", "mount", "umount"
  - threads(): returns list of kernel thread IDs for deterministic error replay
- Write once per system, reuse to form stacks
- Real code on next slide

- FS Checker
- **ext3 Component**
- Stack

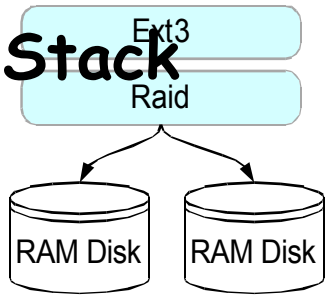
```
void Ext3::init(void) {
   // create an empty ext3 FS with
   // user-specified block size
   systemf("mkfs.ext3 -F -j -b %d %s",
     get_option(blk_size), children[0]->path());
}
void Ext3::recover() {
  systemf("fsck.ext3 -y %s", children[0]->path())
}
void Ext3::mount(void) {
   int ret = systemf("sudo mount -t ext3 %s %s",
      children[0]->path(), path());
   if(ret < 0) error("Corrupt FS: Can't mount!");
}
void Ext3::umount(void) {
  systemf("sudo umount %s", path());
}
void Ext3::threads(threads_t &thids) {
   int thid;
   if((thid=get_pid("kjournald")) != -1)
      thids.push_back(thid);
   else
      explode_panic("can't get kjournald pid!");
}
```

- FS Checker
- ext3 Component
- Stack



- assemble a checking stack
- Let EXPLODE know how subsystems are connected together, so it can initialize, set up, tear down, and repair the entire stack
- Real code on next slide

- FS Checker
- ext3 Component
- Stack




```
// Assemble FS + RAID storage stack step by step.
void assemble(Component *&top, TestDriver *&driver) {
  // 1. load two RAM disks with size specified by user
  ekm_load_rdd(2, get_option(rdd, sectors));
  Disk *d1 = new Disk("/dev/rdd0");
  Disk *d2 = new Disk("/dev/rdd1");

  // 2. plug a mirrored RAID array onto the two RAM disks.
  Raid *raid = new Raid("/dev/md0", "raid1");
  raid->plug_child(d1);
  raid->plug_child(d2);

  // 3. plug an ext3 system onto RAID
  Ext3 *ext3 = new Ext3("/mnt/sbd0");
  ext3->plug_child(raid);
  top = ext3; // let eXplode know the top of storage stack

  // 4. attach a file system test driver onto ext3 layer
  driver = new FsChecker(ext3);
}
```

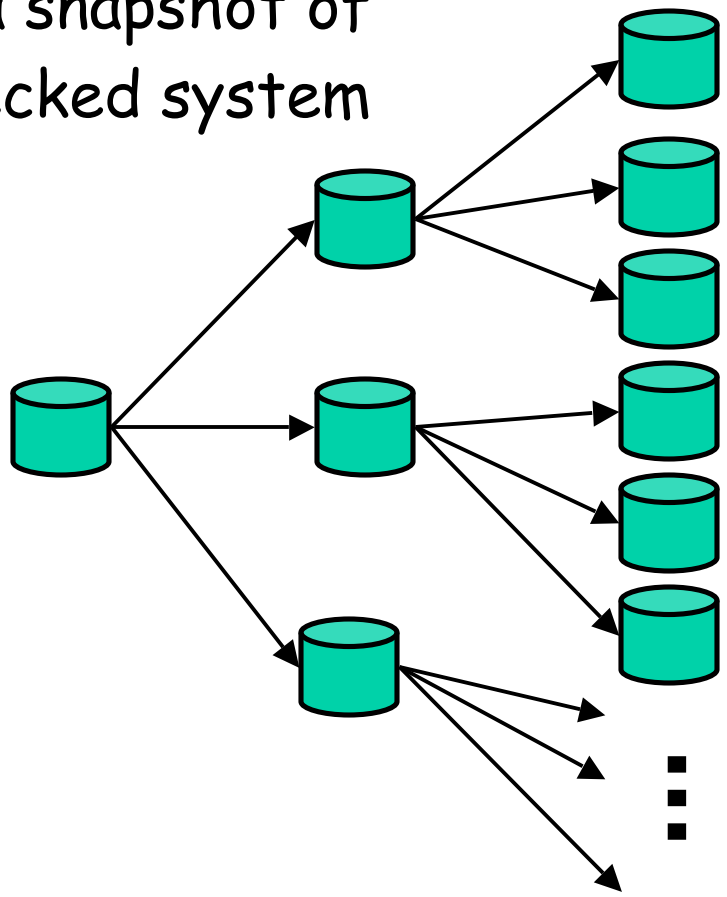
Outline

- Core idea: explore all choices
- Checking interface: 200 lines of C++ to check a system
- Implementation
  - Checkpoint and restore states
  - Deterministic replay
  - Checking process
  - Checking crashes
  - Checking "soft" application crashes
- Results

Recall: core idea

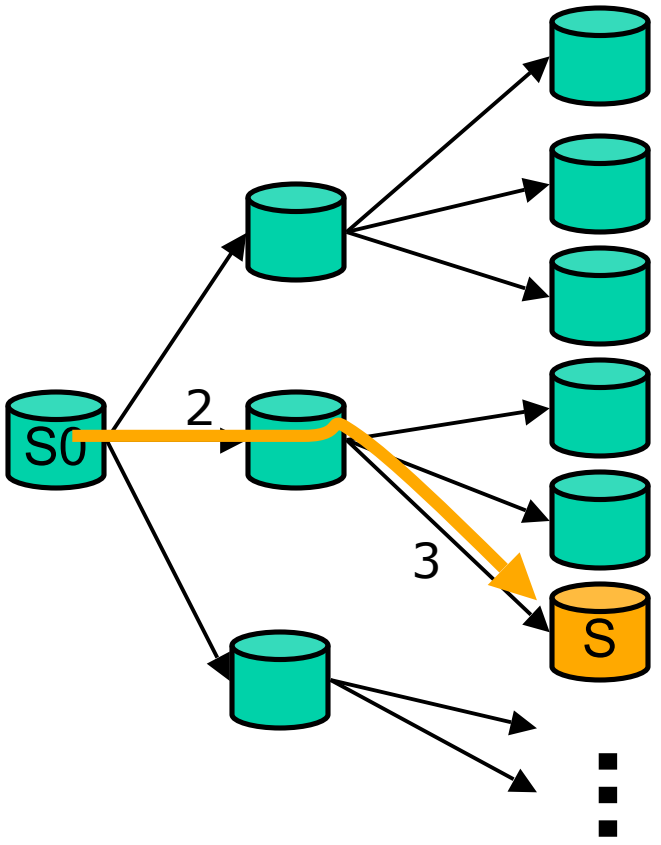
- "Fork" at decision point to explore all choices

state: a snapshot of the checked system



How to checkpoint live system?

- Hard to checkpoint live kernel memory
  - VM checkpoint heavy-weight
- checkpoint: record all choose() returns from S0
- restore: umount, restore S0, re-run code, make K'th choose() return K'th recorded values



S = S0 + redo choices (2, 3)

Deterministic replay

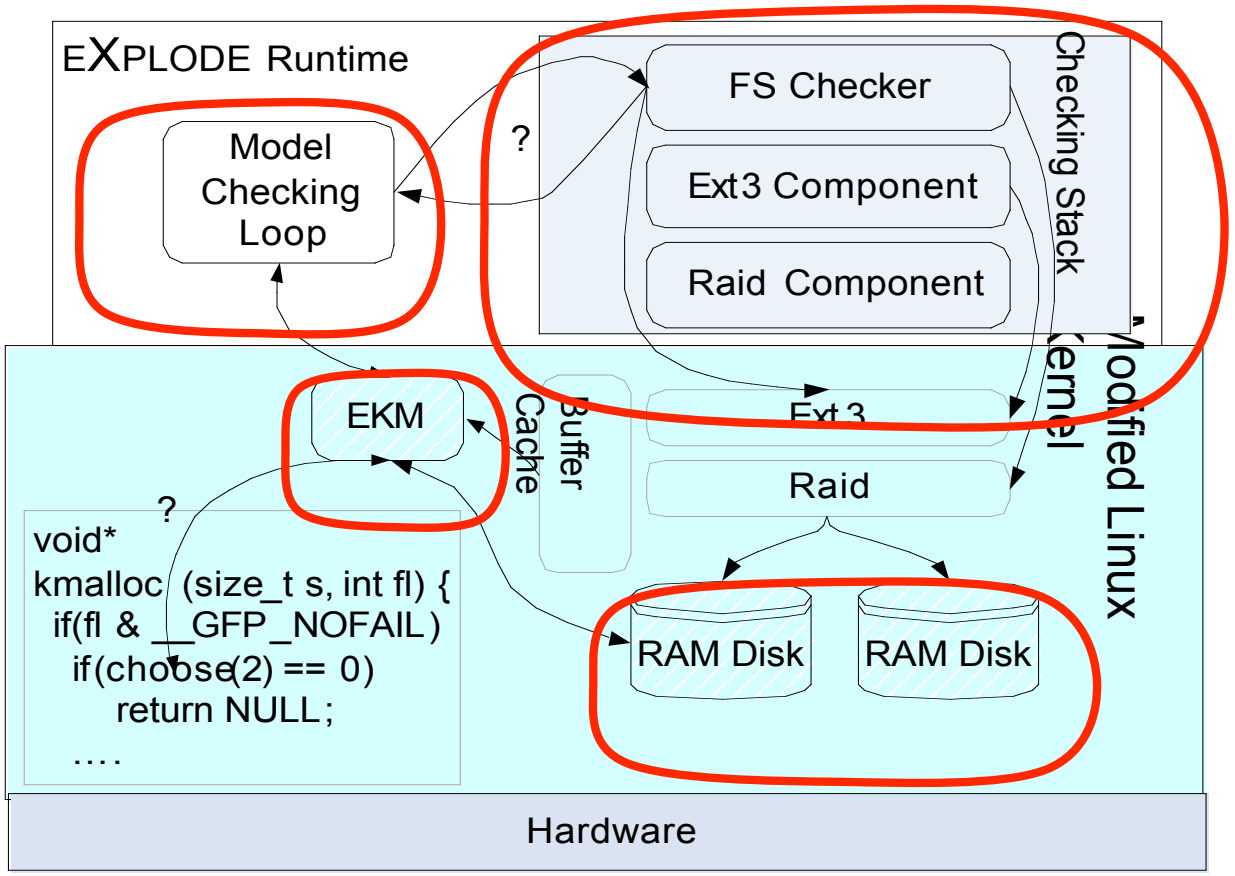
- Need it to recreate states, diagnose bugs

Sources of non-determinism

- Kernel choose() can be called by other code
  - Fix: filter by thread IDs. No choose() in interrupt
- Kernel scheduler can schedule any thread
  - Opportunistic hack: setting priorities. Worked well
  - Can't use lock: deadlock. A holds lock, then yield to B
- Other requirements in paper

- Worst case: non-repeatable error. Automatic

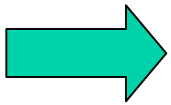
EXPLODE: put it all together



EXPLODE User code EKM = EXPLODE device driver

Outline

- Core idea: explore all choices
- Checking interface: 200 lines of C++ to check a system
- Implementation



- Results
  - Lines of code
  - Errors found

EXPLODE core lines of code

		Lines of code
Kernel patch	**Linux**	1,915 (+ 2,194 generated)
	FreeBSD	1,210
User-level code		6,323

3 kernels: Linux 2.6.11, 2.6.15, FreeBSD 6.0.

FreeBSD patch doesn't have all functionality yet

Checkers lines of code, errors found

Storage System Checked		Component	Checker	Bugs
10 file systems		744/10	5,477	18
Storage applications	**CVS**	27	68	1
	Subversion	31	69	1
	"EXPENSIVE"	30	124	3
	Berkeley DB	82	202	6
Transparent subsystems	**RAID**	144	FS + 137	2
	NFS	34	FS	4
	VMware GSX/Linux	54	FS	1
Total		1,115	6,008	36

Outline

- Core idea: explore all choices
- Checking interface: 200 lines of C++ to check new storage system
- Implementation
- Results
  - Lines of code
  - Errors found

FS Sync checking results

FS	sync	mount sync	fsync	O_SYNC
ext2		✘	✘	✘
ext3				✘
ReiserFS		✘		✘
Reiser4				✘
JFS		✘	✘	✘
XFS		✘		✘
MSDOS	✘	✘		✘
VFAT	✘	✘		✘
HFS	✘	✘	✘	✘
HFS+	✘	✘	✘	✘

✘ indicates a failed check

App rely on sync operations, yet they are broken

ext2 fsync bug

Events to trigger bug

truncate A

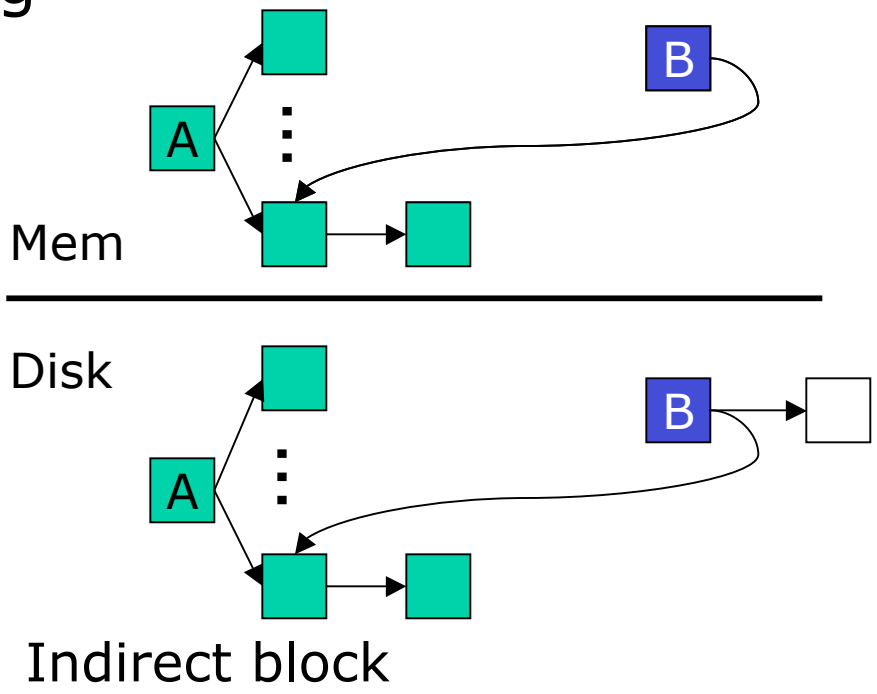
creat B

write B

fsync B

crash!

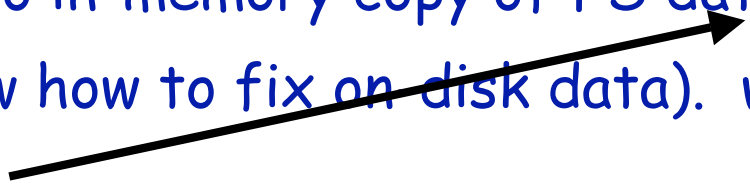
fsck.ext2



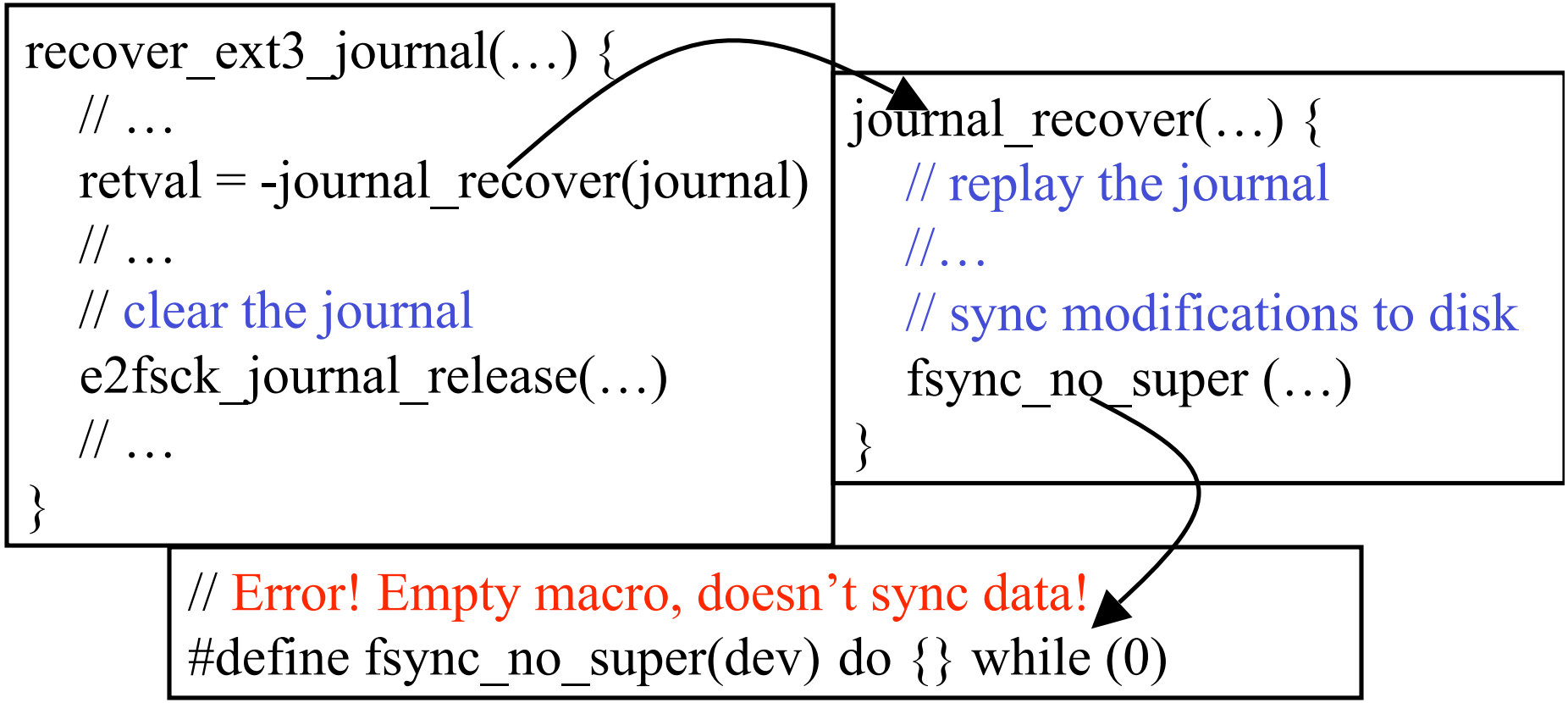
Bug is fundamental due to ext2 asynchrony

Classic: mishandle crash during recovery

- ext3, JFS, reiserfs: All had this bug
  - Result: can lose directories (e.g., "/")
  - Root cause: the same journalling mistake.
- To do a file system 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 get this right.
- To recover after crash
  - Replay log to fix FS. Flush FS changes to disk.
  - wait()



ext3 Recovery Bug



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

Easy checking of "transparent" subsystems

- Many subsystems intend to invisibly augment storage
  - Easy checking: checker run with and without = equivalent.
  - Sync-checker on NFS, RAID or VMM should be same as not
  - Ran it. All are broken.
- Linux RAID:
  - Does not reconstruct bad sectors: marks disk as faulty, removes from RAID, returns error.
  - Two bad sectors, two disks: almost all reconstruct fail
- NFS:
  - write file, then read through hardlink = different result.
- GSX/Linux:

Even simple test drivers find bugs

- Version control: cvs, subversion, "ExPENsive"
  - Test: create repository with single file, checkout, modify, commit, use eXplode to crash.
  - All do careful atomic rename, but don't do fsync!
  - Result: all lose commited data. Bonus: crash during "exPENsive" merge = completely wasted repo
- BerkeleyDB:
  - Test: loop does transaction, choose() to abort or commit.
  - After crash: all (and only) commited transactions in DB.
  - Result: commited get lost on ext2, crash on ext3 can leave DB in unrecoverable state, uncommited can appear after

Classic app mistake: "atomic" rename

- All three version control app. made this mistake

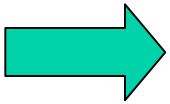
- Atomically update file *A* to avoid corruption

```
fd = creat(A_tmp, …);
write(fd, …);
fsync(fd); // missing!
close(fd);
rename(A_tmp, A);
```

- Problem: rename guarantees nothing abt. Data

Outline

- Core idea: explore all choices
- Checking interface: 200 lines of C++ to check a system
- Implementation
- Results: checked many systems, found many bugs



- Related work, conclusion and future work

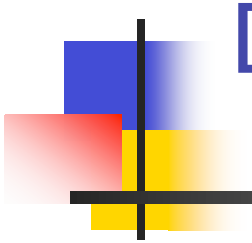
Related work

- FS testing
  - IRON

- Static analysis
  - Traditional software model checking
  - Theorem proving
  - Other techniques

Conclusion and future work

- EXPLODE
  - Easy: need 1 device driver. simple user interface
  - General: can run, can check, without source
  - Effective: checked many systems, 36 bugs
- Current work:
  - Making eXplode open source
  - Junfeng on academic job market.
- Future work:
  - Work closely with storage system implementers to check more systems and more properties
  - Smart search
  - Automatic diagnosis
  - Automatically inferring "choice points"
  - Approach is general, applicable to distributed systems, secure systems

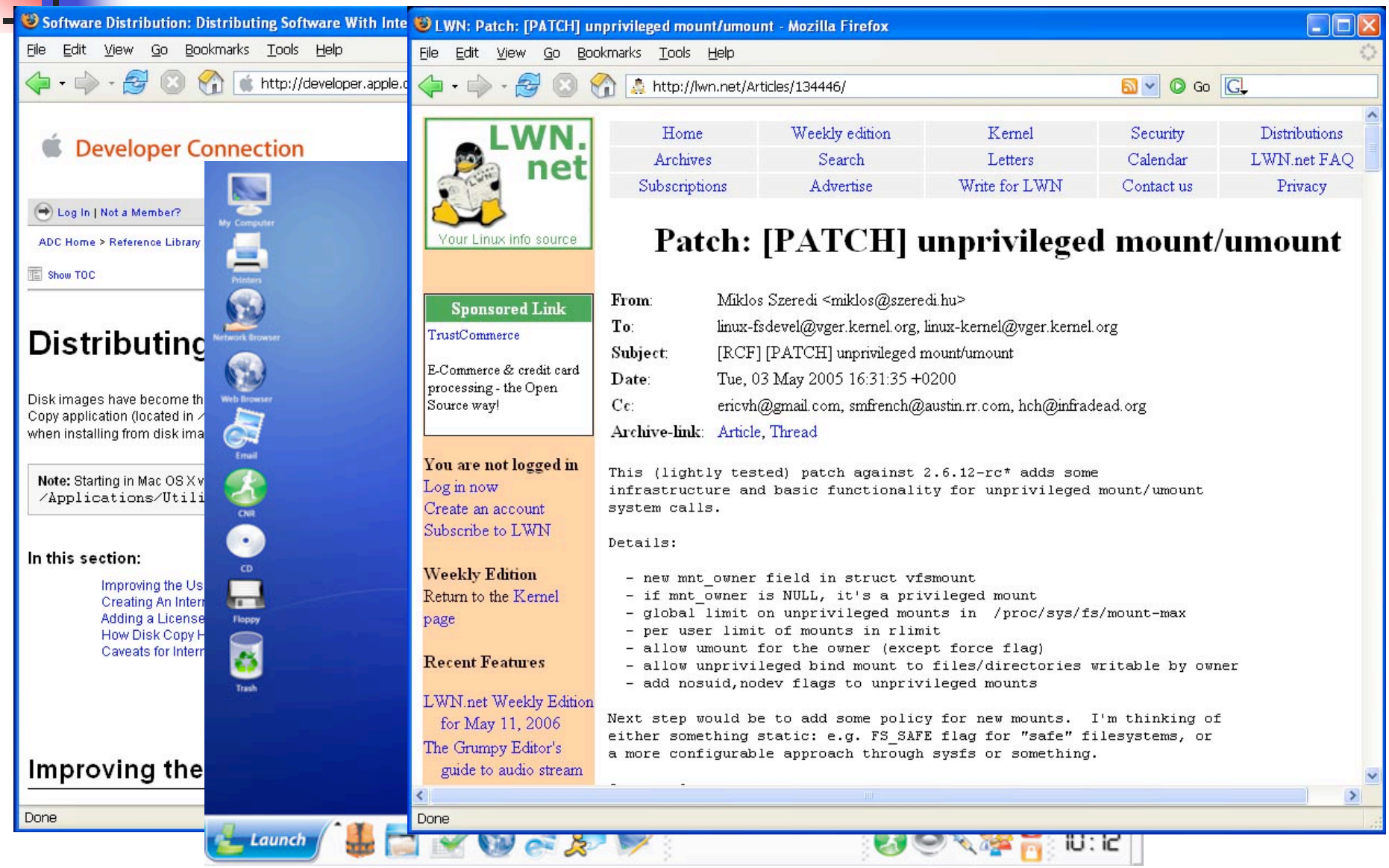


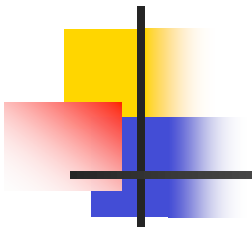
Automatically Generating Malicious Disks using Symbolic Execution

Junfeng Yang, Can Sar, Paul Twohey,
Cristian Cadar and Dawson Engler

Stanford University

Trend: mount untrusted disks





File systems vulnerable to malicious disks

- Privileged, run in kernel
- Not designed to handle malicious disks. FS folks not paranoid (v.s. networking)
- Complex structures (40 if statements in ext2 mount) ➔ many corner cases. Hard to sanitize, test
- Result: easy exploits

Generated disk of death (JFS, Linux 2.4.19, 2.4.27, 2.6.10)

Offset	*Hex Values*
00000	0000 0000 0000 0000 0000 0000 0000 0000
...	...
08000	464a 3153 0000 0000 0000 0000 0000 0000
08010	1000 0000 0000 0000 0000 0000 0000 0000
08020	0000 0000 0100 0000 0000 0000 0000 0000
08030	e004 000f 0000 0000 0002 0000 0000 0000
08040	0000 0000 0000 0000 0000 0000 0000 0000
...	...
10000	

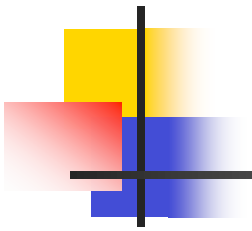
Create 64K file, set 64th sector to above. Mount.

And **PANIC** your kernel!

FS security holes are hard to test

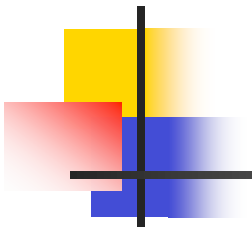
- Manual audit/test: labor, miss errors☹
- Random test: automatic☺. can't go far☹
  - Unlikely to hit narrow input range.
  - Blind to structures

```
int fake_mount(char* disk) {
    struct super_block *sb = disk;
    if(sb->magic != 0xEF53) //hard to pass using random
        return -1;
    // sb->foo is unsigned, therefore >= 0
    if(sb->foo > 8192)
        return -1;
    x = y/sb->foo; //potential division-by-zero
    return 0;
}
```



Soln: let FS generate its own disks

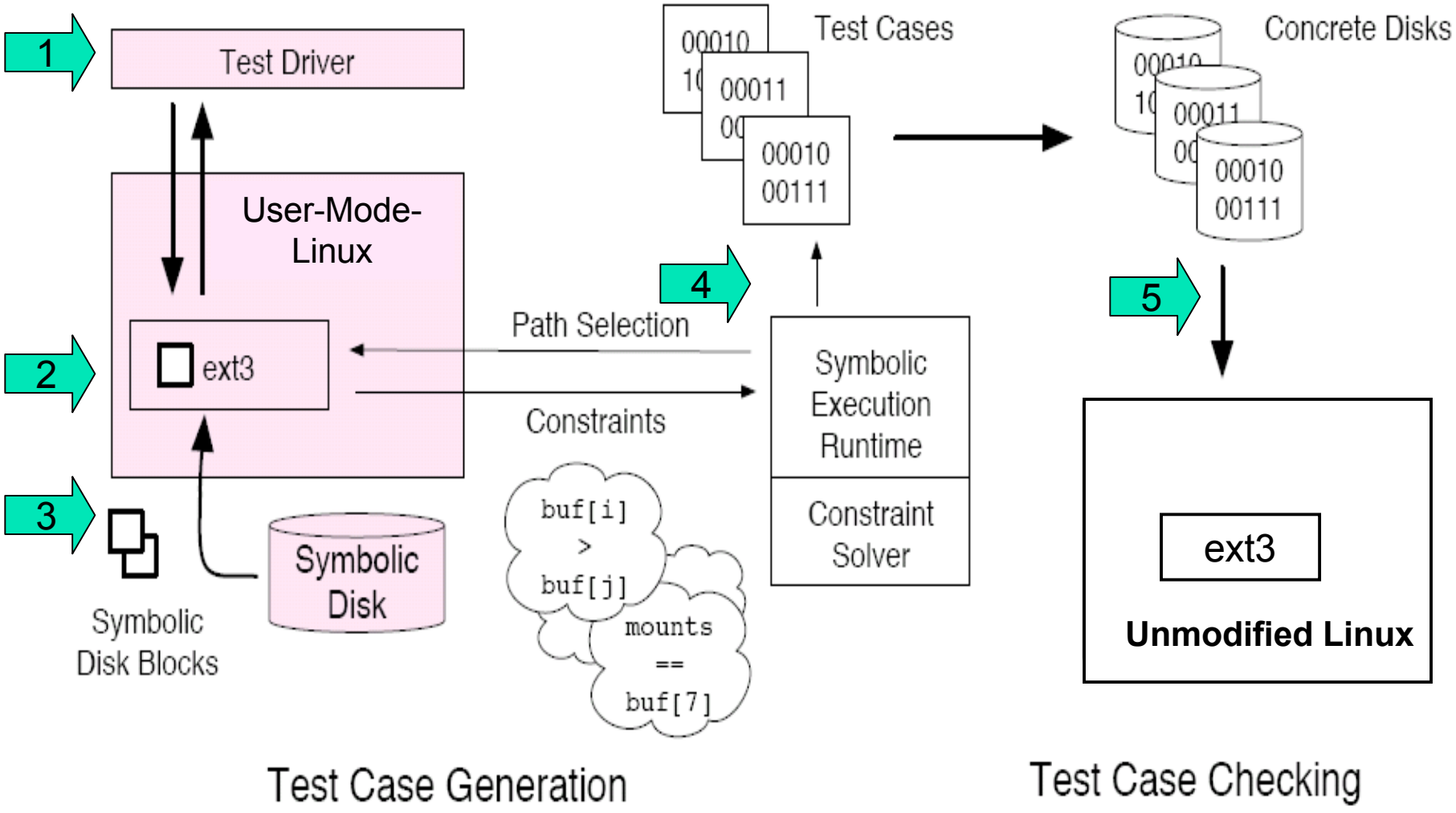
- EXE: Execution generated Executions [Cadar and Engler, SPIN'05] [Cadar et al Stanford TR2006-1]
  - Run code on symbolic input, initial value = "anything"
  - As code observes input, it tells us values input can be
  - At conditional branch that uses symbolic input, explore both
    - On true branch, add constraint input satisfies check
    - On false that it does not
  - exit() or error: solve constraints for input.
- To find FS security holes, set disk symbolic



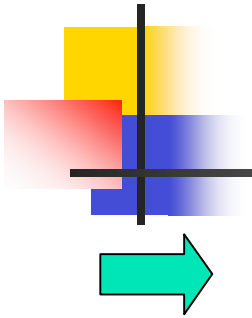
Key enabler: STP constraint solver

- Handles: All of C (except floating point)
  - Memory, arrays, pointers, updates, bit-operations.
  - Full bit-level accurate precision. No approximations.
  - One caveat: **p, where p is symbolic.
- Written by David Dill and Vijay Ganesh.
  - Destroy's previous CVCL system
  - 10-1000+x faster, 6x smaller.
  - Much simpler, more robust

A galactic view



EXE-cc instrumented



Outline

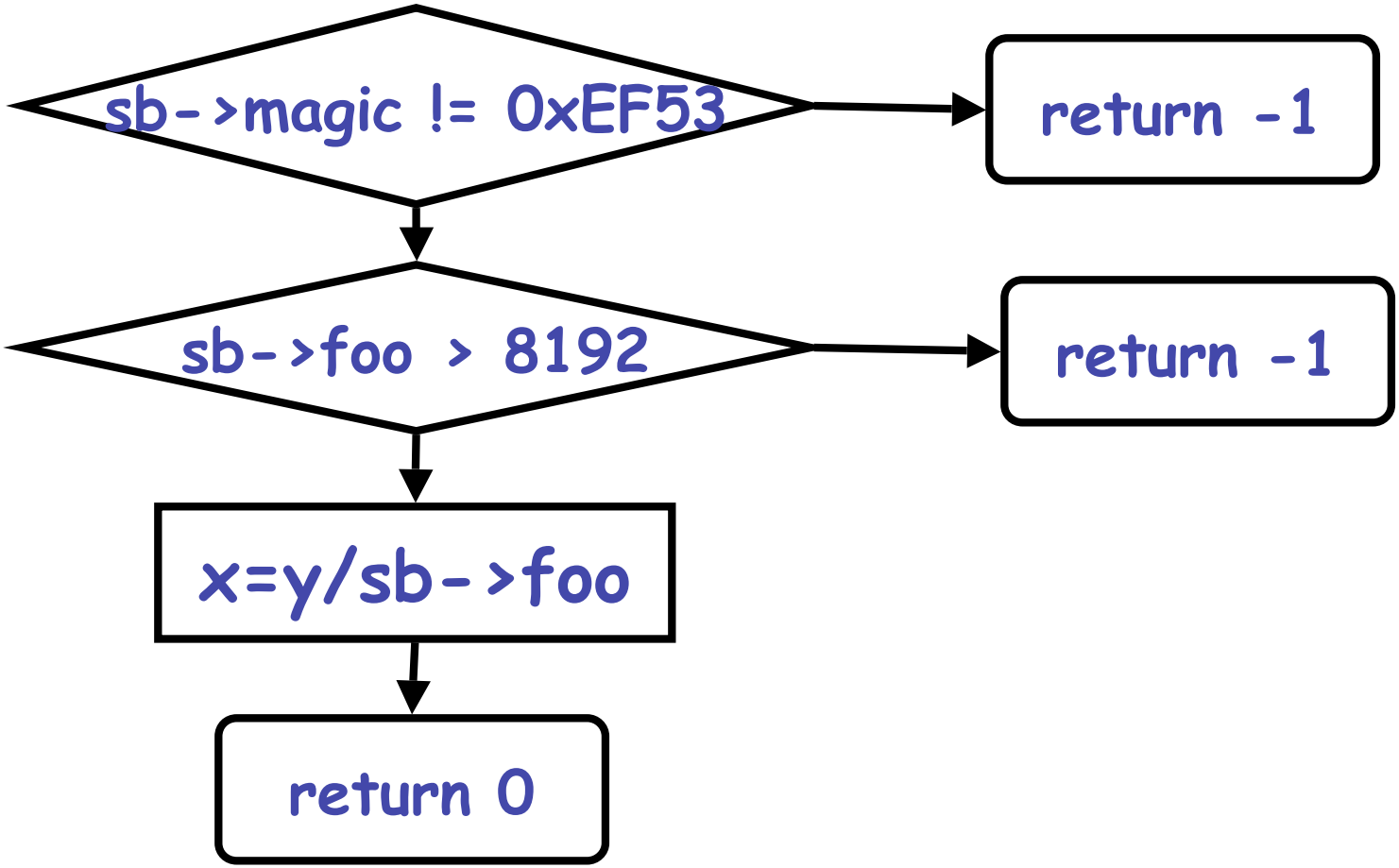
- How EXE works
- Apply EXE to Linux file systems
- Results

The toy example

```
int fake_mount(char* disk) {
    struct super_block *sb = disk;
    if(sb->magic != 0xEF53) //hard to pass using random
        return -1;
    // sb->foo is unsigned, therefore >= 0
    if(sb->foo > 8192)
        return -1;
    x = y/sb->foo; //potential division-by-zero
    return 0;
}
```

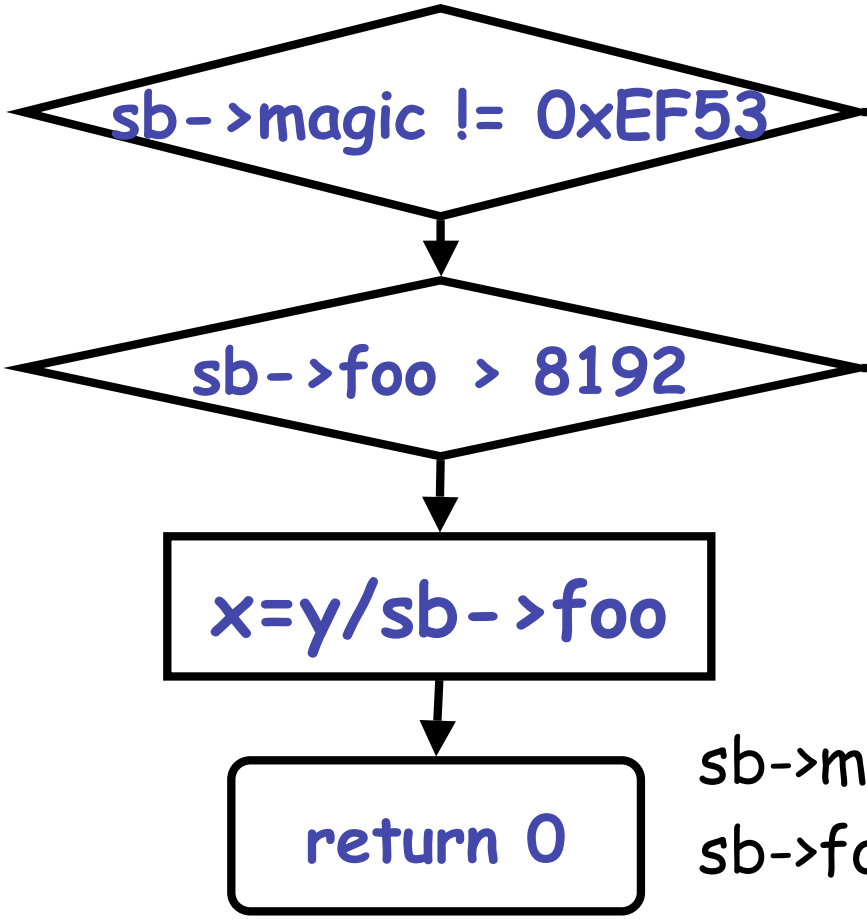
Concrete v.s. symbolic execution

Concrete: sb->magic = 0xEF53, sb->foo = 9000



Concrete v.s. symbolic execution

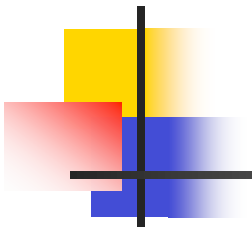
Symbolic: sb->magic and sb->foo unconstrained



The toy example: instrumentation

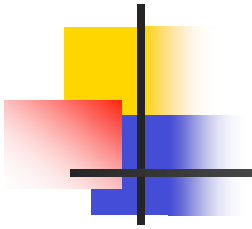
```
int fake_mount(char* disk) {
    struct super_block *sb = disk;


    if(sb->magic != 0xEF53)
        return -1;



    if(sb->foo > 8192)
        return -1;



    x = y/sb->foo;
```

```
int fake_mount_exe(char* disk) {
    struct super_block *sb = disk;
    if(fork() == child) {
        constraint(sb->magic != 0xEF53);
        return -1;
    } else
        constraint(sb->magic == 0xEF53);

    if(fork() == child) {
        constraint(sb->foo > 8192);
        return -1;
    } else
        constraint(sb->foo <= 8192);

    check_symbolic_div_by_zero(sb->foo);
    x=y/sb->foo;
```



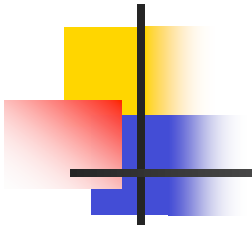
How to use EXE

- Mark disk blocks as symbolic
  - void make_symbolic(void* disk_block, unsigned size)
- Compile with EXE-cc (based on CIL)
  - Insert checks around every expression: if operands all concrete, run as normal. Otherwise, add as constraint
  - Insert fork when symbolic could cause multiple acts
- Run: forks at each decision point.
  - When path terminates, solve constraints and generate disk images
  - Terminates when: (1) exit, (2) crash, (3) error
- Rerun concrete through uninstrumented Linux



Why generate disks and rerun?

- Ease of diagnosis. No false positive
- One disk, check many versions
- Increases path coverage, helps correctness testing



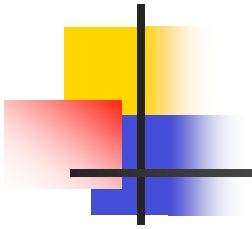
Mixed execution

- Too many symbolic var, too many constraints ➔ constraint solver dies
- Mixed execution: don't run everything symbolically
  - Example: x = y+z;
  - if y, z both concrete, run as in uninstrumented
  - Otherwise set "x == y + z", record x = symbolic.
- Small set of symbolic values
  - disk blocks (make_symbolic) and derived
- Result: most code runs concretely, small slice deals w/ symbolics, small # of constraints
  - Perhaps why worked on Linux mounts, sym on demand

Symbolic checks

```
int fake_mount(char* disk) {
    struct super_block *sb = disk;


    if(sb->magic != 0xEF53)
        return -1;



    if(sb->foo > 8192)
        return -1;



    x = y/sb->foo;
```

```
int fake_mount_exe(char* disk) {
    struct super_block *sb = disk;
    if(fork() == child) {
        constraint(sb->magic != 0xEF53);
        return -1;
    } else
        constraint(sb->magic == 0xEF53);

    if(fork() == child) {
       constraint(sb->foo > 8192);
        return -1;
    } else
        constraint(sb->foo <= 8192);
check_symbolic_div_by_zero(sb->foo);



    x=y/sb->foo;
```



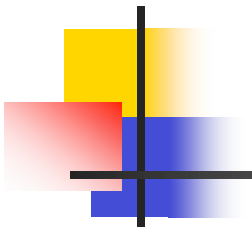
Symbolic checks

- Key: Symbolic reasons about many possible values simultaneously. Concrete about just current ones (e.g. Purify).
- Symbolic checks:
  - When reach dangerous op, EXE checks if any input exists that could cause blow up.
  - Builtin: x/0, x%0, NULL deref, mem overflow, arithmetic overflow, symbolic assertion

Check symbolic div-by-0: x/y, y symbolic

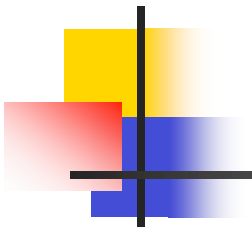
- Found 2 bugs in ext2, copied to ext3

```
void check_sym_div_by_zero (y) {
    if(query(y==0) == satisfiable)
        if(fork() == child) {
                constraint(y != 0);
                return;
        } else {
                constraint(y == 0);
                solve_and_generate_disk();
                error("divided by 0!")
        }
```



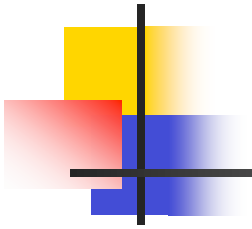
More on EXE ([CCS'06])

- Handling C constructs
  - Casts: untyped memory
  - Bitfield
  - Symbolic pointer, array index: disjunctions
- Limitations
  - Constraint solving NP
  - Uninstrumented functions
  - Symbolic double dereference: concretize
  - Symbolic loop: heuristic search



Outline

- How EXE works
- Apply EXE to Linux file systems
- Results



Results

- Checked ext2, ext3, and JFS mounts
- Ext2: four bugs.
  - One buffer overflow ➔ read and write arbitrary kernel memory (next slide)
  - Two div/mod by 0
  - One kernel crash
- Ext3: four bugs (copied from ext2)
- JFS: one NULL pointer dereference
- Extremely easy-to-diagnose: just mount!

Simplified: ext2 r/w kernel memory

block is symbolic

block + count can overflow and becomes negative!

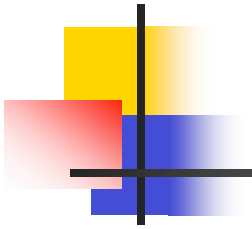
Pass block to bar

block_group is symbolic

block can be large!
Symbolic read off bound

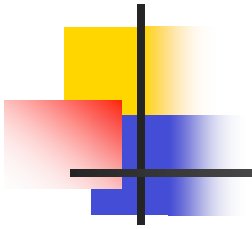
Symbolic write off bound

```
int ext2_overflow(int block, unsigned count) {
  if(block < lower_bound
     || (block+count) > higher_bound)
        return -1;
  while(count--)
        bar(block++);
}
void bar(int block) {
  // B = power of 2
  int block_group = (block-A)/B;
  …
  //array length is 8
  … = array[block_group]
  …
  array[block_group] = …
  …
```



Related Work

- FS testing
  - Mostly stress test for functionality bugs
  - Linux ISO9660 FS handling flaw, Mar 2005 (http://lwn.net/Articles/128365/)
- Static analysis
- Model checking
  - Symbolic model checking
- Input generation
  - Using symbolic execution to generate testcases



BPF, Linux packet filters

- "We'll never find bugs in that"
  - heavily audited, well written open source
- Mark filter & packet as symbolic.
  - Symbolic = turn check into generator
  - Safe filter check: generates all valid filters of length N.
  - BPF Interpreter: will produce all valid filter programs that pass check of length N.
  - Filter on message: generates all packets that accept, reject.

Results: BPF, trivial exploit.

```
// Check that memory operations only uses valid addresses.
// => Check forgets LDX,STX!
if( (BPF_CLASS(p->code) == BPF_ST || (BPF_CLASS(p->code) == BPF_LD &&
     (p->code & 0xe0) == BPF_MEM)) && p->k >= BPF_MEMWORDS )
  return 0;
```

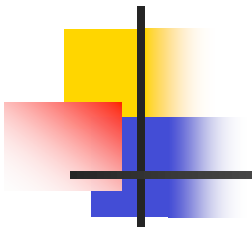
```
case BPF_LDX|BPF_MEM:
     X = mem[pc->k]; continue;
...
case BPF_STX:
     mem[pc->k] = X; continue;
```

Linux Filter

- Generated filter:

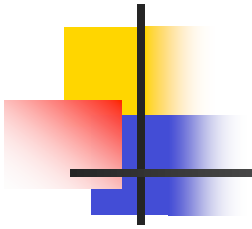
```
// other filters that cause this error...
//    => BPF_LD|BPF_B|BPF_IND
//    => BPF_LD|BPF_H|BPF_IND
s[0].code  = BPF_LD|BPF_B|BPF_ABS;
s[0].k     = 0x7fffffffUL;
s[1].code  = BPF_RET;
s[1].k     = 0xffffffff0UL;
```

```
nline void * skb_header_pointer(struct sk_buff *skb, int offset, int len,
  int hlen = skb_headlen(skb);
  if (offset + len <= hlen)
          return skb->data + offset;
```



Conclusion [Oakland'06, CCS'06]

- Automatic all-path execution, all-value checking
  - 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 real code.
- Zero false positives.



Exponential forking?

- Only fork on symbolic branch
- Mixed execution: to reduce # of symbolic var, don't run everything symbolically. Mix concrete execution and symbolic execution
  - Example: x = y+z;
  - if y, z both concrete, run as in uninstrumented
  - Otherwise set "x == y + z", record x = symbolic.
- Small set of symbolic values
  - disk blocks (make_symbolic) and derived
- Result: most code runs concretely, small slice deals w/ symbolics, small # of constraints
  - Perhaps why worked on Linux mounts, sym on demand