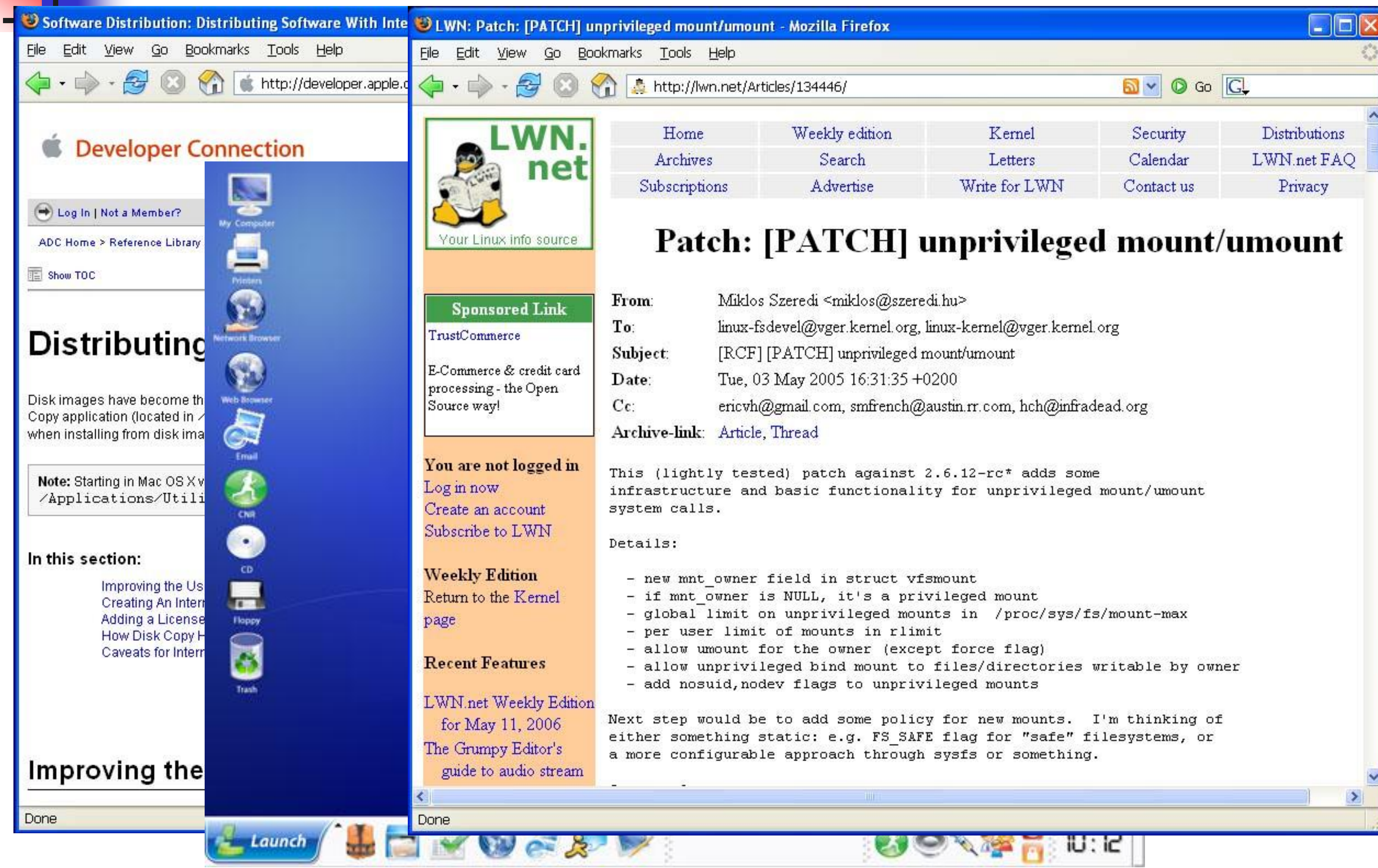
Automatically Generating Malicious Disks using Symbolic Execution

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!

Goal: automatically find many file system security holes

FS security holes are hard to test

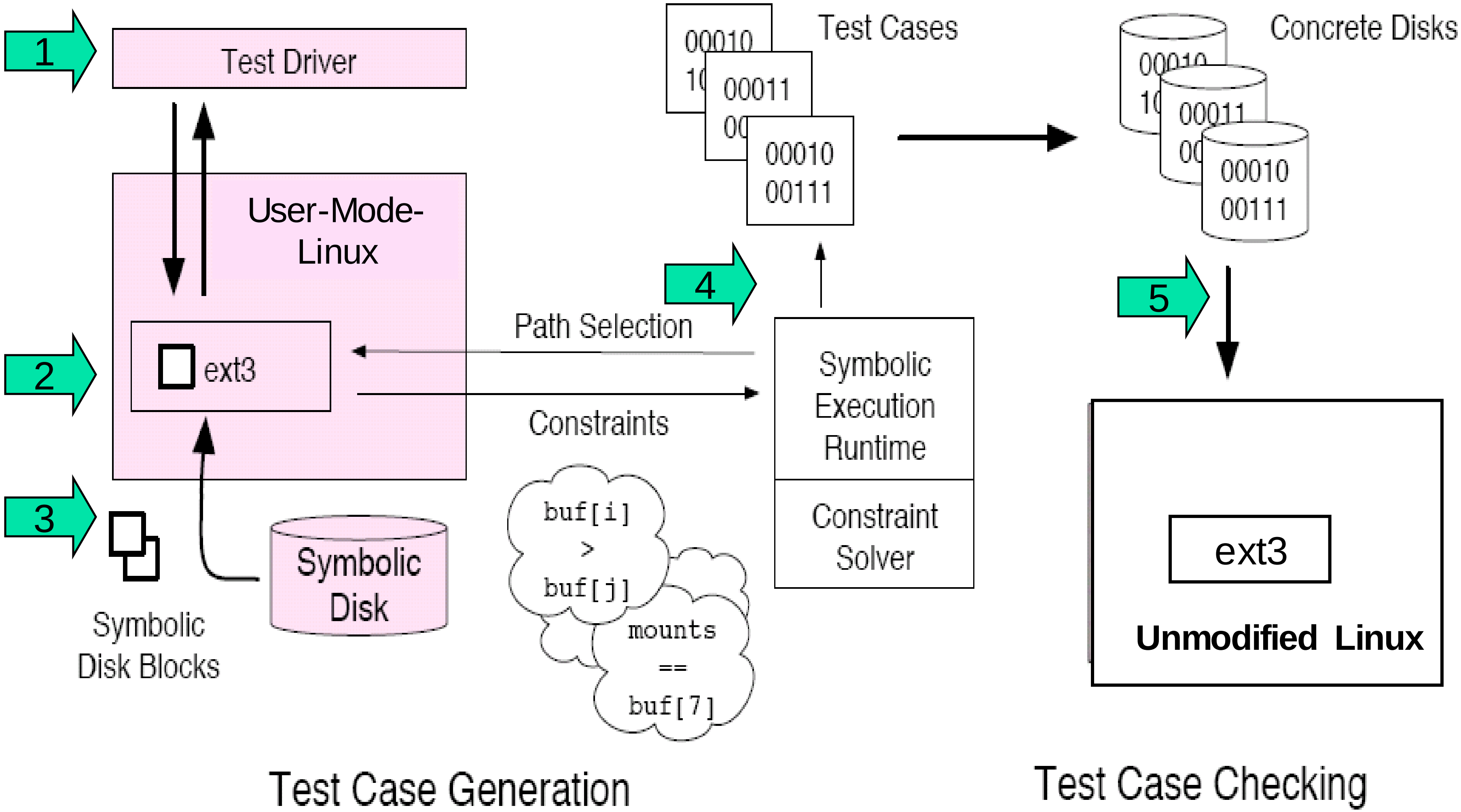
- Manual audit/test: labor, miss errors☹
- Random test: automatic☺. can't go far☹
  - Unlikely to hit narrow input range.
  - Blind to structures

```
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

A galactic view



EXE-cc instrumented

Outline

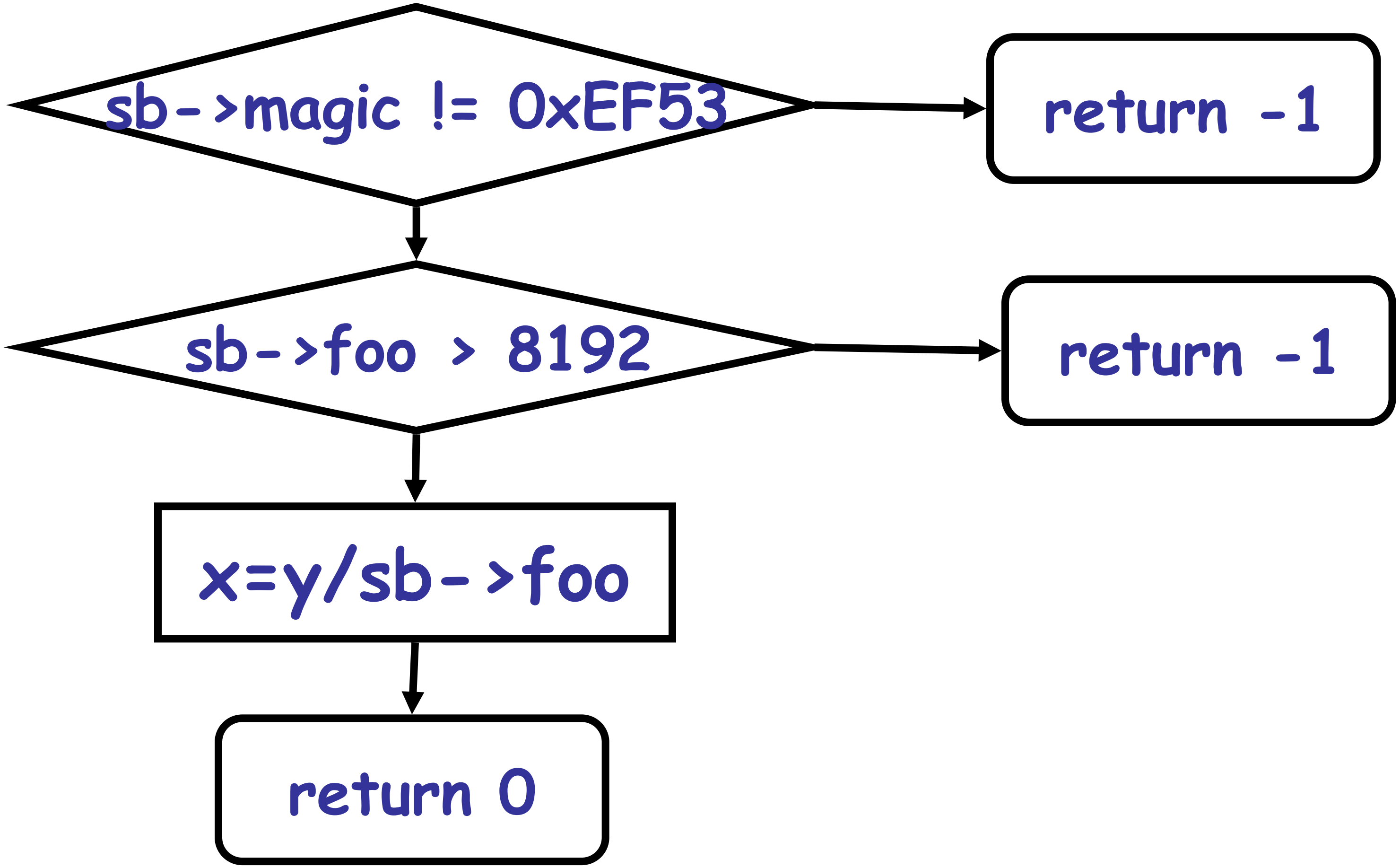
- How EXE works
- Apply EXE to Linux file systems
- Results

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;
}
```

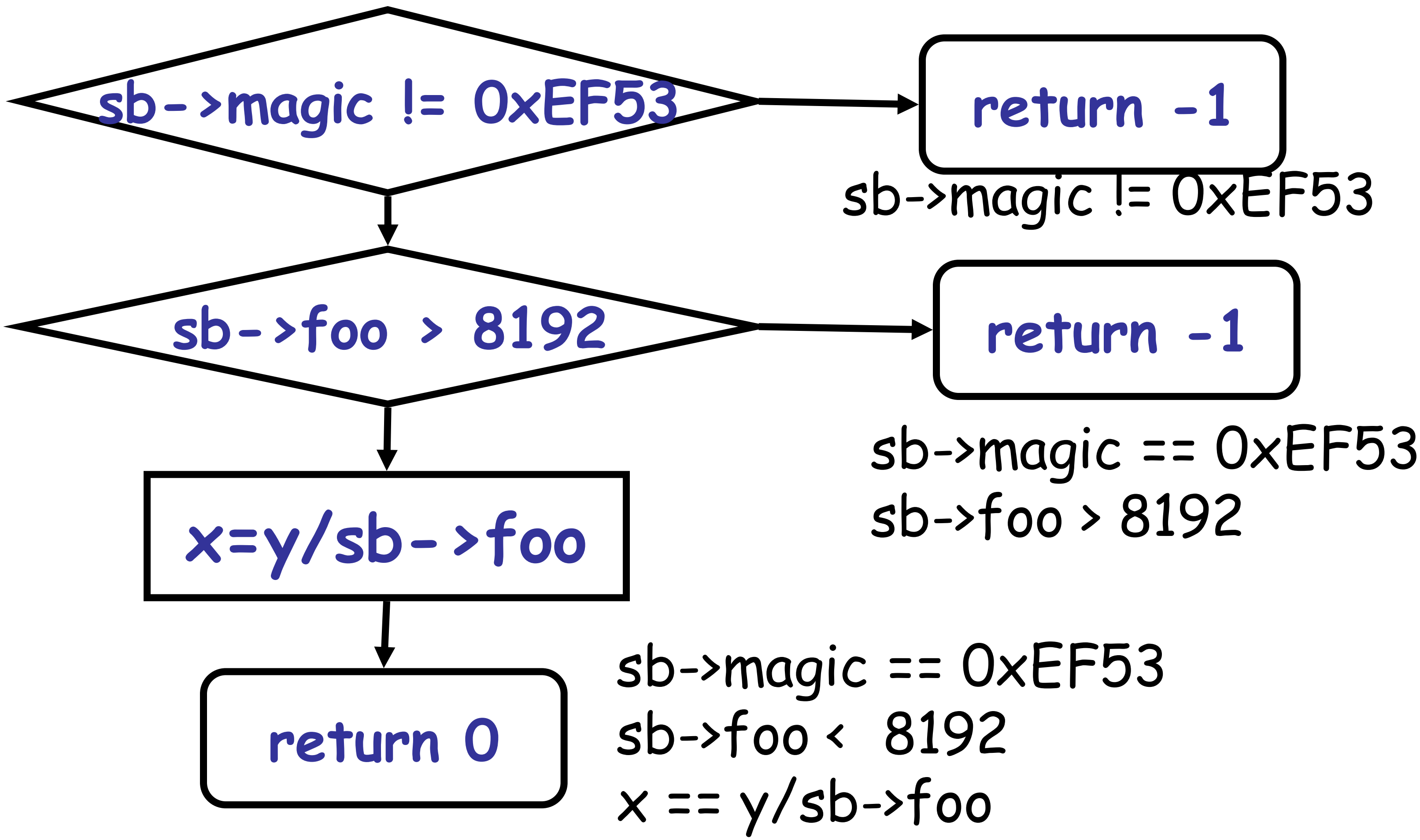
Concrete v.s. symbolic execution

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;
    return 0;
}
```

```
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;
    return 0;
}
```

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;
    return 0;
}
```

```
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;
    return 0;
}
```

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!")
        }
}
```

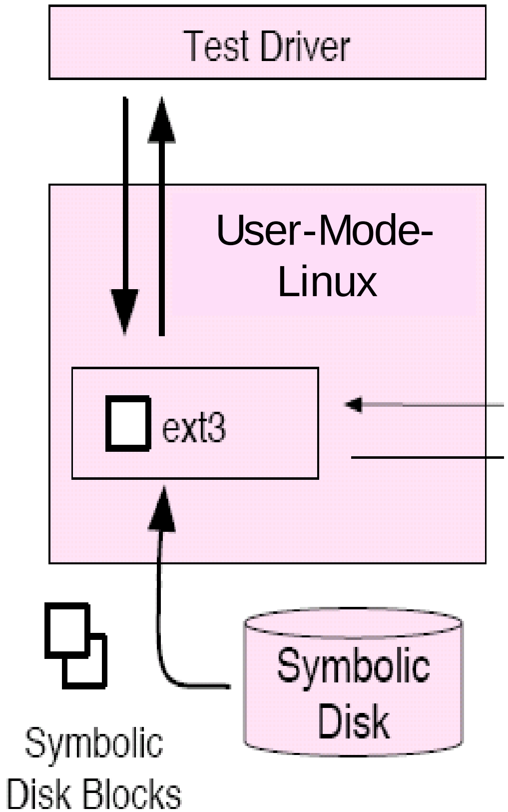
More on EXE (Stanford TR2006-1)

- Handling C constructs
  - Casts: untyped memory
  - Bitfield
  - Symbolic pointer, array index: disjunctions
- Limitations
  - Constraint solving NP
  - Uninstrumented functions
  - Symbolic div/mod: assert divisor = power of two
  - Symbolic double dereference: concretize
  - Symbolic loop: heuristic search

Outline

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

A galactic view



Test Case Generation

EXE-cc instrumented

Why User-Mode-Linux + disk driver

- Hard to cut Linux FS out of kernel. User-Mode-Linux=check in situ
- End-to-end check
- EXE needs to fork/wait for process
- Hard to debug OS on raw machine
- We already had the framework

Making Linux work with EXE

- Disable threading
- Replace ASM functions called by FS (strcmp, memcpy…) with C versions
- User-Mode-Linux loaded @ fixed (too small) location. Stripped down
- EXE-cc/CIL can't compile 8 files. Not called with symbolic args. Use gcc

Making EXE work with Linux

- Still research prototype → bugs
- EXE dies if too many constraints, too many symbolic var
  - Optimization: v = symbolic_exp, if symbolic_exp has unique value, don't make v symbolic. Slow down "tainting"
- No free of symbolic heap objects

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

Conclusion

- FS vulnerable to malicious disks
- Applied EXE to Linux file systems ext2, ext3, JFS mounts. Worked well. Found 5 unique security holes
- EXE offers a promising approach to finding security holes

Future work

- Automatic exploit generation
  - User interacts with kernel through syscalls
  - Compile Linux with EXE. Mark data(syscall arg) from user as symbolic
  - Find paths to bugs
  - Generate concrete input + C code to call kernel.
  - Mechanized way to produce exploits.

Future work (Cont.)

- Automatic "hardening"
  - EXE finds error with path constraints.
  - Can translate constraints to if-statements and reject concrete input that satisfies.
    - E.g. wrap up disk reads. If disk malicious, return "Cannot mount."
    - Similar to Shield, vulnerability signature checking
    - Nice feature: fully automatic, no manual filter, automatically detect exploit