Constraint Solving Challenges in Dynamic Symbolic Execution

Cristian Cadar

Department of Computing

Imperial College London

Joint work with **Dawson Engler, Daniel Dunbar**
Peter Collingbourne, Paul Kelly, Vijay Ganesh, David Dill, Junfeng Yang
P. Pawlowski, J. Song, T. Ma, P. Pietzuch, P. Boonstoppel, P. Twohey, C. Sar

STANFORD
UNIVERSITY

Imperial College
London

1st International SAT/SMT Solver Summer School
June 12th 2011 • MIT, Cambridge, MA, USA

Writing Correct Software Is Hard

- Software complexity
  - Massive amounts of code
  - Tricky control flow
  - Complex dependencies
  - Abusive use of pointer operations
  - Intensive interaction w/ environment
    - E.g., data from OS, network, etc.

Systems code

- Current testing approaches are insufficient
  - Most projects still use only manual (*expensive*) and/or random testing (*often ineffective*)

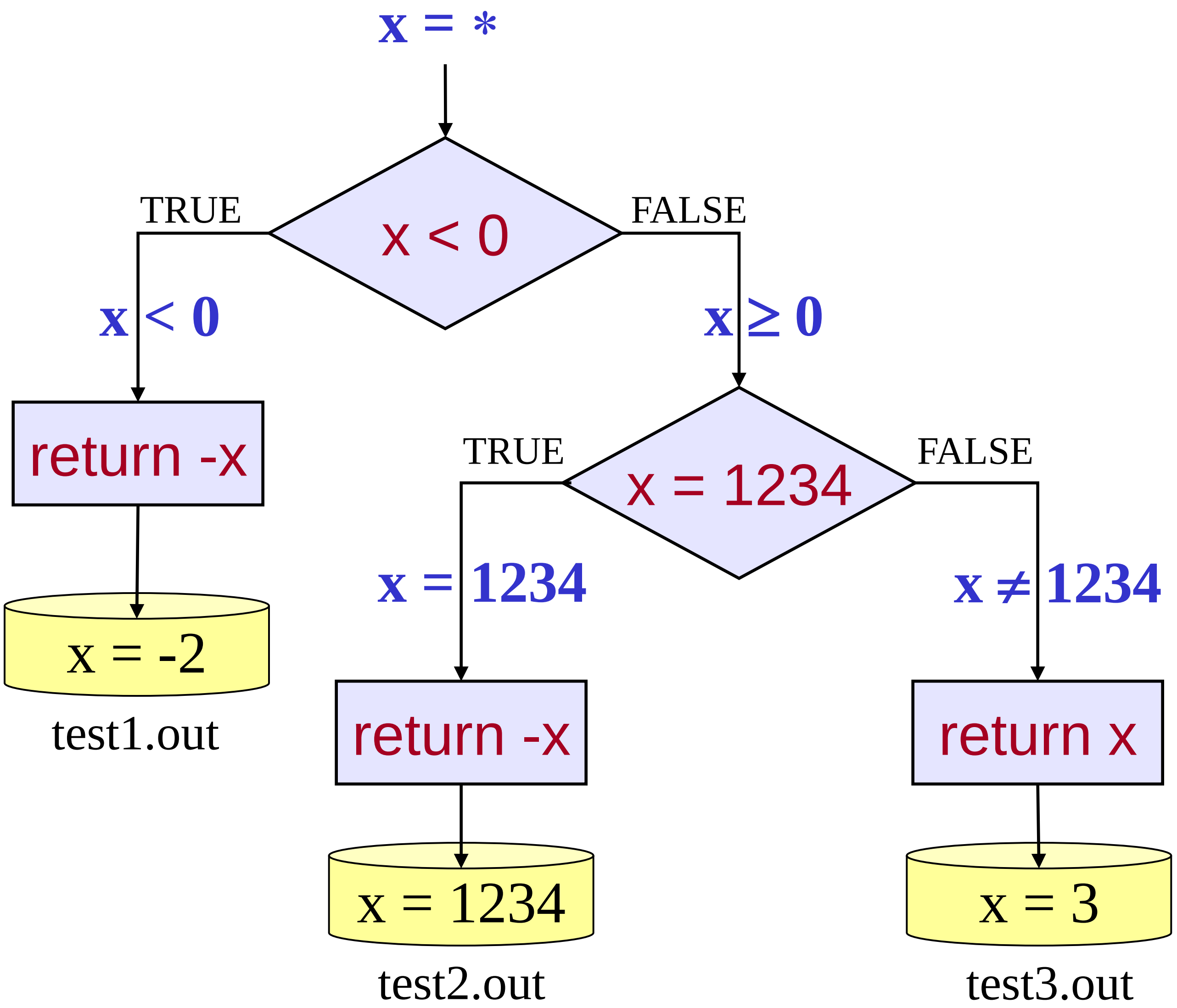
Dynamic Symbolic Execution

- **Let code to generate its own (complex) test cases!**

- Automatically generated high coverage test suites
  - Over 90% on average on ~160 user-level apps
- Found bugs and security vulnerabilities in complex software
  - Including file systems, device drivers, computer vision code, utilities, network servers, packet filters

Toy Example

```
int bad_abs(int x)
{
    if (x < 0)
        return -x;
    if (x == 1234)
        return -x;
    return x;
}
```



All-Value Checks

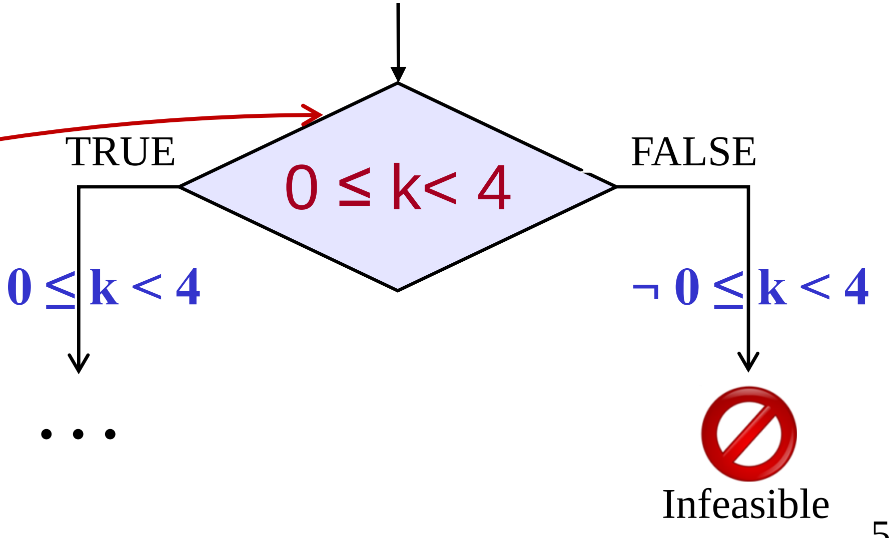
Implicit checks before each dangerous operation

- Null-pointer dereferences
- Buffer overflows
- Division/modulo by zero
- Assert violations

All-value checks!

- Errors are found if **any** buggy values exist on that path!

```
int foo(unsigned k) {
  int a[4] = {3, 1, 0, 4};
  k = k % 4;
  return a[a[k]];
}
```



All-Value Checks

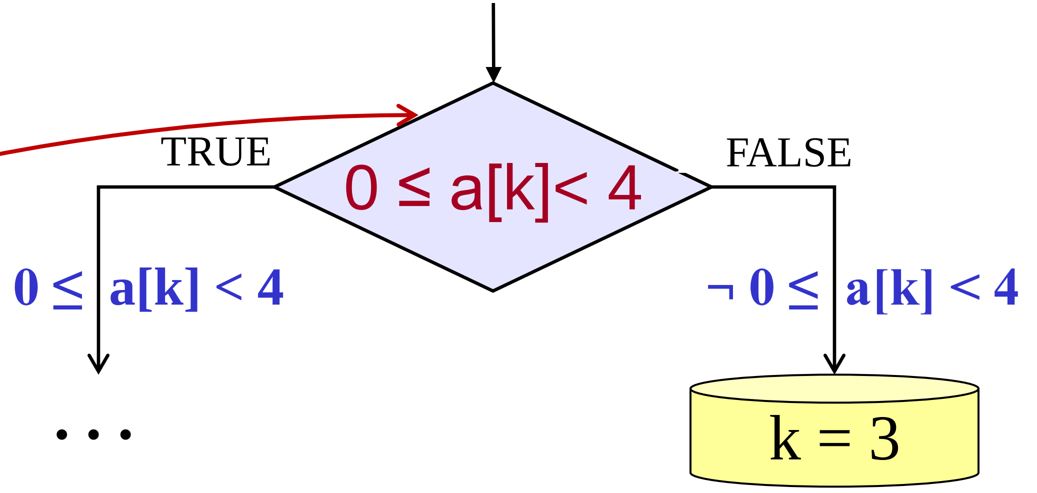
Implicit checks before each dangerous operation

- Null-pointer dereferences
- Buffer overflows
- Division/modulo by zero
- Asserts violations

All-value checks!

- Errors are found if **any** buggy values exist on that path!

```
int foo(unsigned k) {
  int a[4] = {3, 1, 0, 4};
  k = k % 4;
  return a[a[k]];
}
```



TRUE

FALSE

$0 \leq a[k] < 4$

$0 \leq a[k] < 4$

$\neg\, 0 \leq a[k] < 4$

. . .

k = 3

Buffer overflow!

Dynamic (vs. Static) SymEx

- Each path explored separately as in regular testing
  - EXE uses **`fork()`** system call to fork execution!
- Mixed concrete/symbolic execution
  - All operations that do not depend on the symbolic inputs are (essentially) executed as in the original code!
    - E.g., **`malloc(5)`** allocates object on the heap in EXE

Dynamic (vs. Static) SymEx

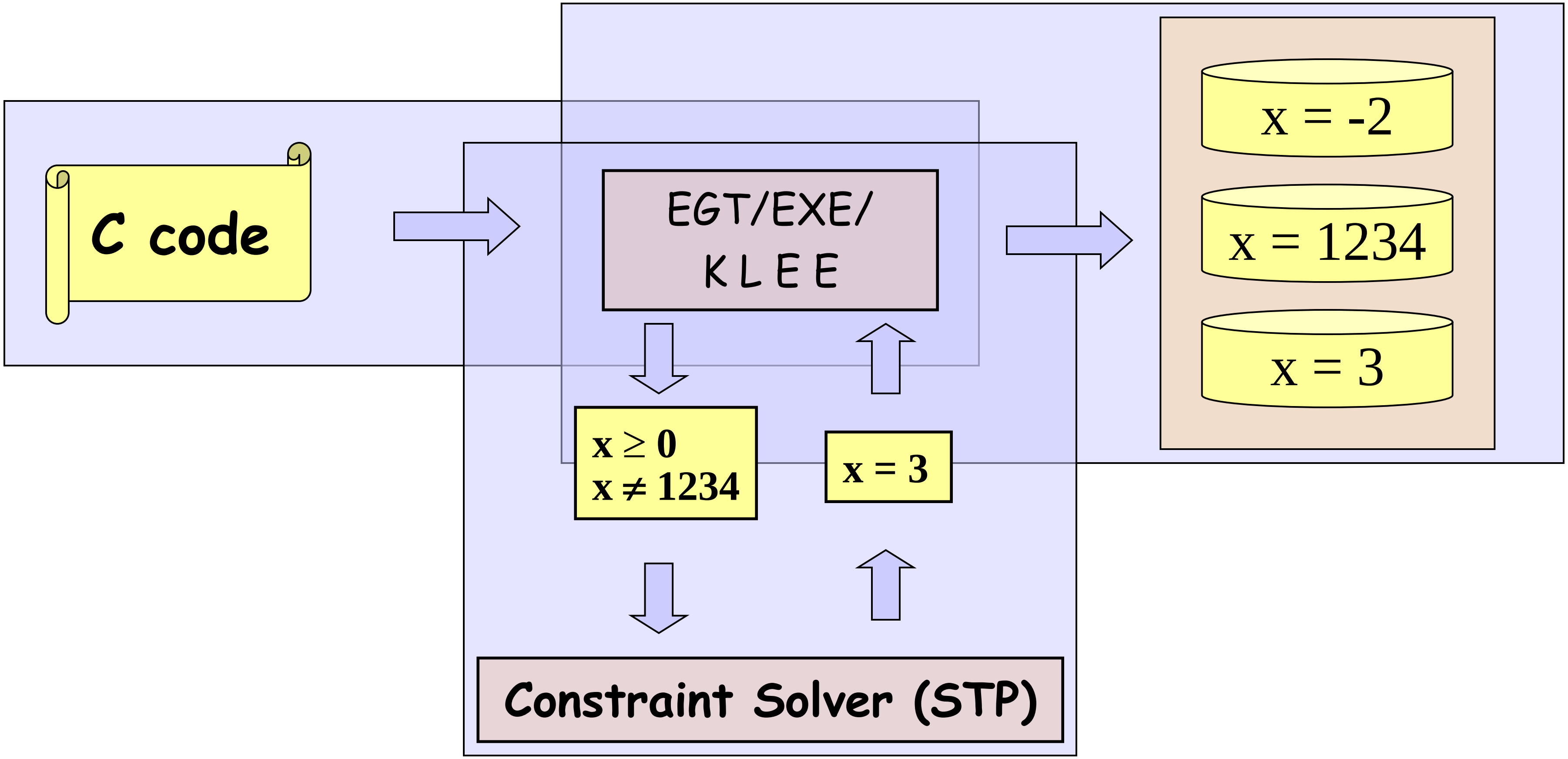
Advantages:

- Ability to interact with the outside environment
  - System calls, uninstrumented libraries
- Only relevant code executed symbolically
  - Without the need to extract it explicitly

…and disadvantages:

- Can only explore a finite number of paths!
  - Important to prioritize most “interesting” ones

Three tools: EGT, EXE, KLEE



Scalability Challenges

Path exploration challenges

Constraint solving challenges

Constraint Solving Challenges

1. **Accuracy:** need constraint solver that allows bit-level modeling of memory:
   - Systems code often observes the same bytes in different ways: e.g., using pointer casting to treat an array of chars as a network packet, inode, etc.
   - Bugs in systems code are often triggered by corner cases such as arithmetic overflows

2. **Performance:** real programs generate expensive constraints

STP

- Modern constraint solver, based on *eager* translation to SAT (uses MiniSAT)
- Developed at Stanford by Ganesh and Dill, initially targeted to (and driven by) EXE

- Two data types: **bitvectors** and **arrays of bitvectors**
- We model each memory block as an array of bitvectors
- We can translate all C expressions into STP constraints with bit-level accuracy
  - Main exception: floating-point

Constraint Solving: Performance

Constraint solving optimizations essential:

- STP optimizations
- Higher-level optimizations

Reasoning about Arrays in STP

- Many programs generate large constraints involving arrays with symbolic indexes

- STP handles this via **array-based refinement**

Reasoning about Arrays in STP

STP’s conversion of array terms to SAT is expensive

$$(a[i_1] = e_1) \wedge (a[i_2] = e_2) \wedge (a[i_3] = e_3) \wedge (i_1+i_2+i_3=6)$$

$$(v_1 = e_1) \wedge (v_2 = e_2) \wedge (v_3 = e_3) \wedge (i_1+i_2+i_3=6)$$

$$(i_1 = i_2 \Rightarrow v_1 = v_2) \wedge (i_1 = i_3 \Rightarrow v_1 = v_3) \wedge (i_2 = i_3 \Rightarrow v_2 = v_3)$$

Expands each formula by $n \cdot (n-1)/2$ terms, where n is the number of syntactically distinct indexes

Array-based Refinement in STP

STP's conversion of array terms to SAT is expensive

$$(a[i_1] = e_1) \wedge (a[i_2] = e_2) \wedge (a[i_3] = e_3) \wedge (i_1+i_2+i_3=6)$$

$$(v_1 = e_1) \wedge (v_2 = e_2) \wedge (v_3 = e_3) \wedge (i_1+i_2+i_3=6)$$

~~$(i_1 = i_2 \Longrightarrow v_1 = v_2) \wedge (i_1 = i_3 \Longrightarrow v_1 = v_3) \wedge (i_2 = i_3 \Longrightarrow v_2 = v_3)$~~

Under-approximation
UNSATISFIABLE

→

Original formula
UNSATISFIABLE

Array-based Refinement in STP

STP’s conversion of array terms to SAT is expensive

$$(a[i_1] = e_1) \wedge (a[i_2] = e_2) \wedge (a[i_3] = e_3) \wedge (i_1+i_2+i_3=6)$$

$$(v_1 = e_1) \wedge (v_2 = e_2) \wedge (v_3 = e_3) \wedge (i_1+i_2+i_3=6)$$

~~$(i_1 = i_2 \Rightarrow v_1 = v_2) \wedge (i_1 = i_3 \Rightarrow v_1 = v_3) \wedge (i_2 = i_3 \Rightarrow v_2 = v_3)$~~

$i_1 = 1$
$i_2 = 2$
$i_3 = 3$
$v_1 = e_1 = 1$
$v_2 = e_2 = 2$
$v_3 = e_3 = 3$

⇒

$(a[1] = 1) \wedge (a[2] = 2) \wedge$
$(a[3] = 3) \wedge (1+2+3 = 6)$

✓

Array-based Refinement in STP

STP’s conversion of array terms to SAT is expensive

$$(a[i_1] = e_1) \land (a[i_2] = e_2) \land (a[i_3] = e_3) \land (i_1+i_2+i_3=6)$$

$$(v_1 = e_1) \land (v_2 = e_2) \land (v_3 = e_3) \land (i_1+i_2+i_3=6)$$

~~$(i_1 = i_2 \Rightarrow v_1 = v_2) \land (i_1 = i_3 \Rightarrow v_1 = v_3) \land (i_2 = i_3 \Rightarrow v_2 = v_3)$~~

$i_1 = 2$
$i_2 = 2$
$i_3 = 2$
$v_1 = e_1 = 1$
$v_2 = e_2 = 2$
$v_3 = e_3 = 3$

$$(a[2] = 1) \land (a[2] = 2) \land (a[2] = 3) \land (2+2+2 = 6)$$

Array-based Refinement in STP

STP's conversion of array terms to SAT is expensive

$$(a[i_1] = e_1) \wedge (a[i_2] = e_2) \wedge (a[i_3] = e_3) \wedge (i_1+i_2+i_3=6)$$

$$(v_1 = e_1) \wedge (v_2 = e_2) \wedge (v_3 = e_3) \wedge (i_1+i_2+i_3=6)$$

$(i_1 = i_2 \Longrightarrow v_1 = v_2)$ ~~$\wedge (i_1 = i_3 \Longrightarrow v_1 = v_3) \wedge (i_2 = i_3 \Longrightarrow v_2 = v_3)$~~

$i_1 = 2$
$i_2 = 2$
$i_3 = 2$
$v_1 = e_1 = 1$
$v_2 = e_2 = 2$
$v_3 = e_3 = 3$

$$\boxed{(a[2] = 1) \wedge (a[2] = 2) \wedge (a[2] = 3) \wedge (2+2+2 = 6)}$$

Evaluation

Solver	Total time (min)	Timeouts
STP (baseline)	56	36
STP (array-based refinement)	10	1

8495 test cases from our symbolic execution benchmarks

- Timeout set at 60s (which are added as penalty), underestimates performance differences

Higher-Level Constraint Solving Optimizations

- Two simple and effective optimizations
  - Eliminating irrelevant constraints
  - Caching solutions
    - Dramatic speedup on our benchmarks

Eliminating Irrelevant Constraints

- In practice, each branch usually depends on a small number of variables

```
…
…
if (x < 10) {
    …
}
```

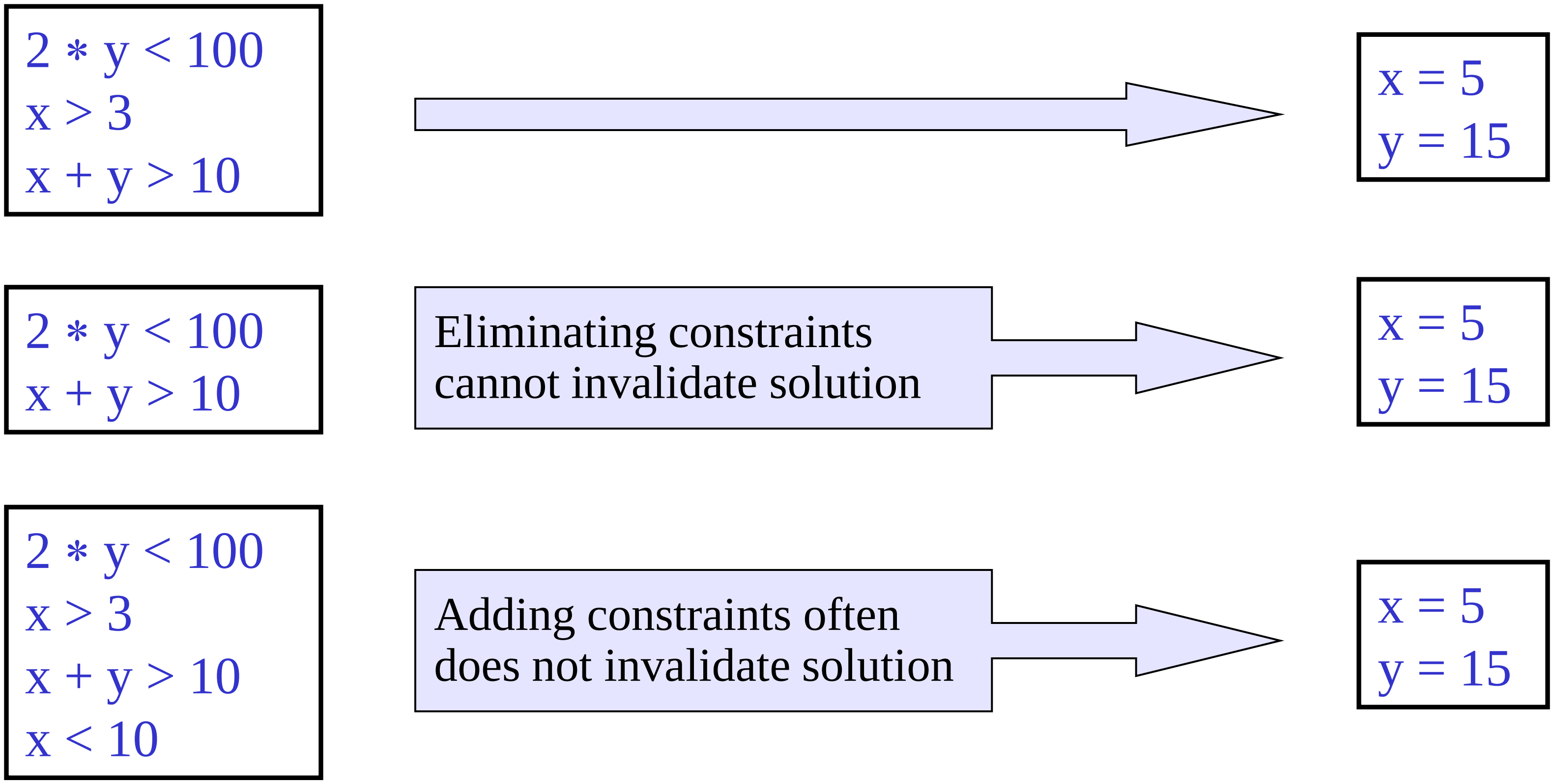
x + y > 10

~~z & -z = z~~

x < 10 ?

Caching Solutions

- Static set of branches: lots of similar constraint sets



$2 * y < 100$
$x > 3$
$x + y > 10$

$x = 5$
$y = 15$

$2 * y < 100$
$x + y > 10$

Eliminating constraints cannot invalidate solution

$x = 5$
$y = 15$

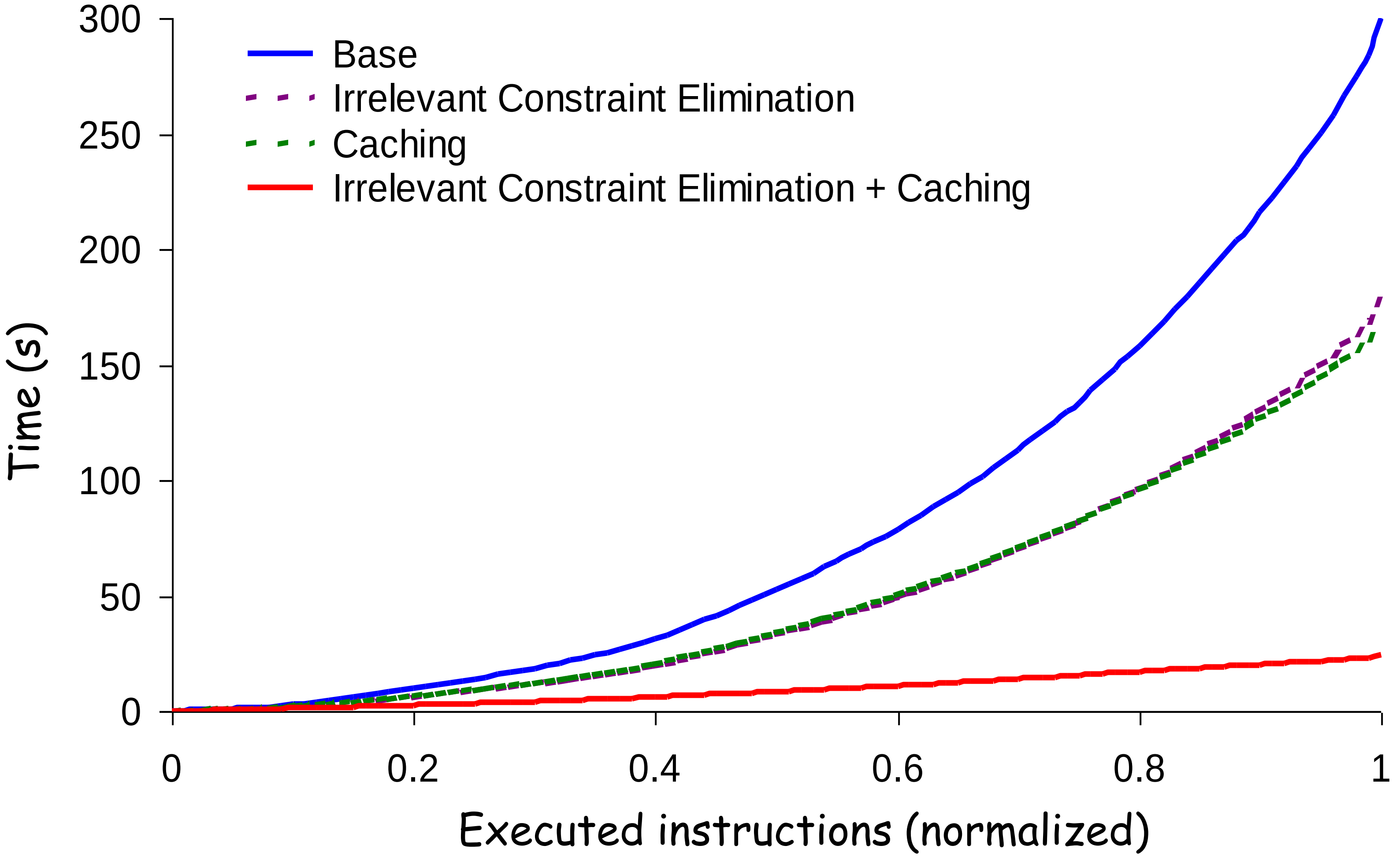
$2 * y < 100$
$x > 3$
$x + y > 10$
$x < 10$

Adding constraints often does not invalidate solution

$x = 5$
$y = 15$

Dramatic Speedup

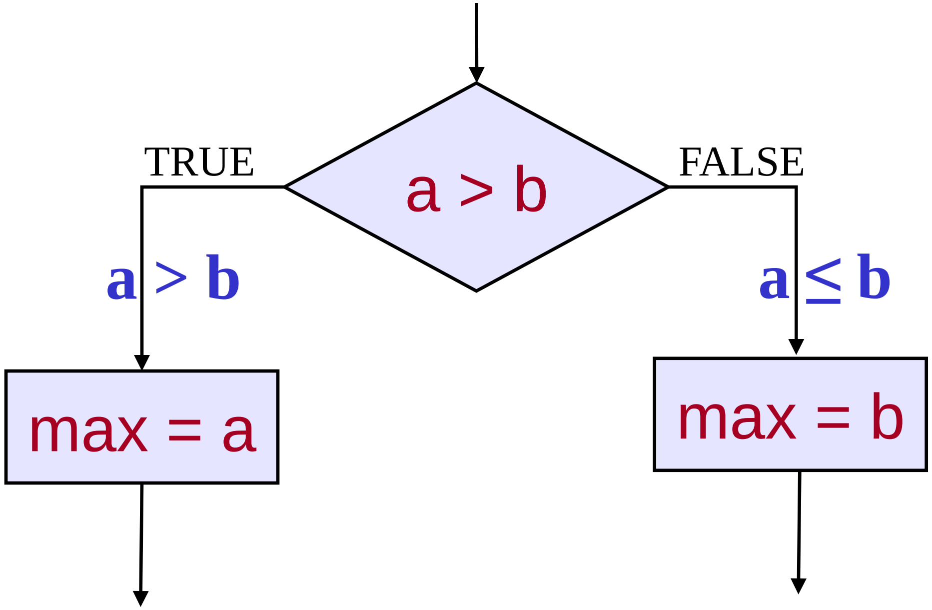
Aggregated data over 73 applications



Statically Merging Paths

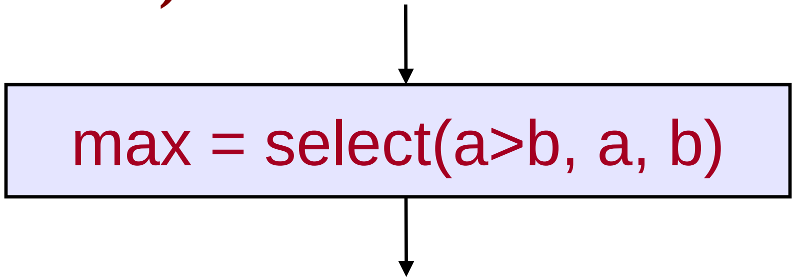
Default behaviour

```
if (a > b)
    max = a;
else max = b;
```



Phi-Node Folding (when no side effects)

```
if (a > b)
    max = a;
else max = b;
```



Statically Merging Paths

```
for (i=0; i < N; i++) {
    if (a[i] > b[i])
        max[i] = a[i];
    else max[i] = b[i];
}
```

- Default: 2^N paths
- Phi-node folding: 1 path

morph computer vision algorithm: $2^{256} \rightarrow 1$

Path merging ≡ Outsourcing problem to constraint solver

(especially problematic for solvers optimized for conjunctions of constraints)

Evaluation

- Motivation and Overview
- Example and Basic Architecture
- Constraint Solving Challenges
- Evaluation
  - Coverage results
  - Bug finding
  - Crosschecking
  - Attack generation

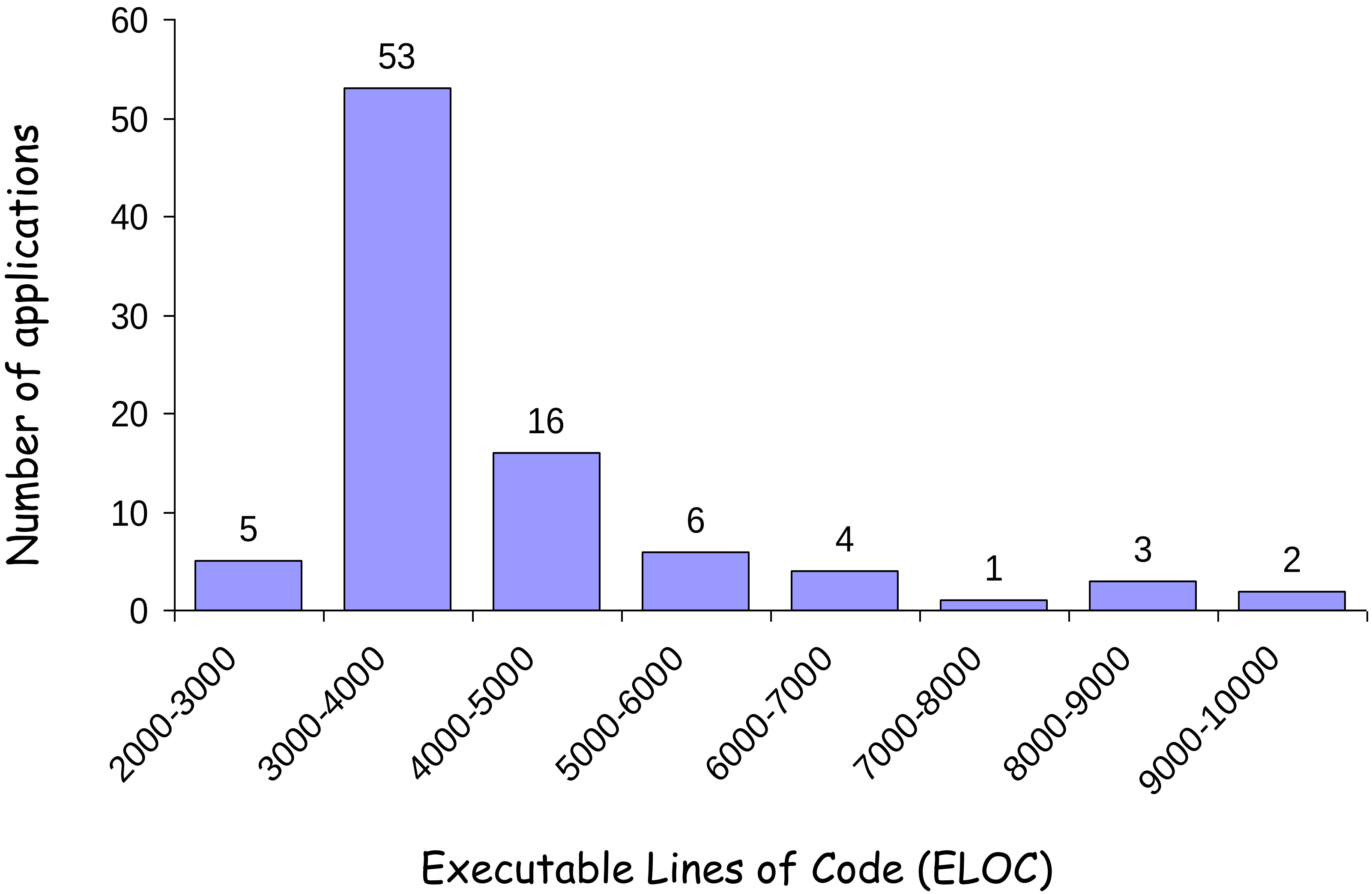
GNU Coreutils Suite

- Core user-level apps installed on many UNIX systems
- 89 stand-alone (i.e. excluding wrappers) apps (v6.10)
  - File system management: ls, mkdir, chmod, etc.
  - Management of system properties: hostname, printenv, etc.
  - Text file processing : sort, wc, od, etc.
  - …

Variety of functions, different authors, intensive interaction with environment

Heavily tested, mature code

Coreutils ELOC (incl. called lib)



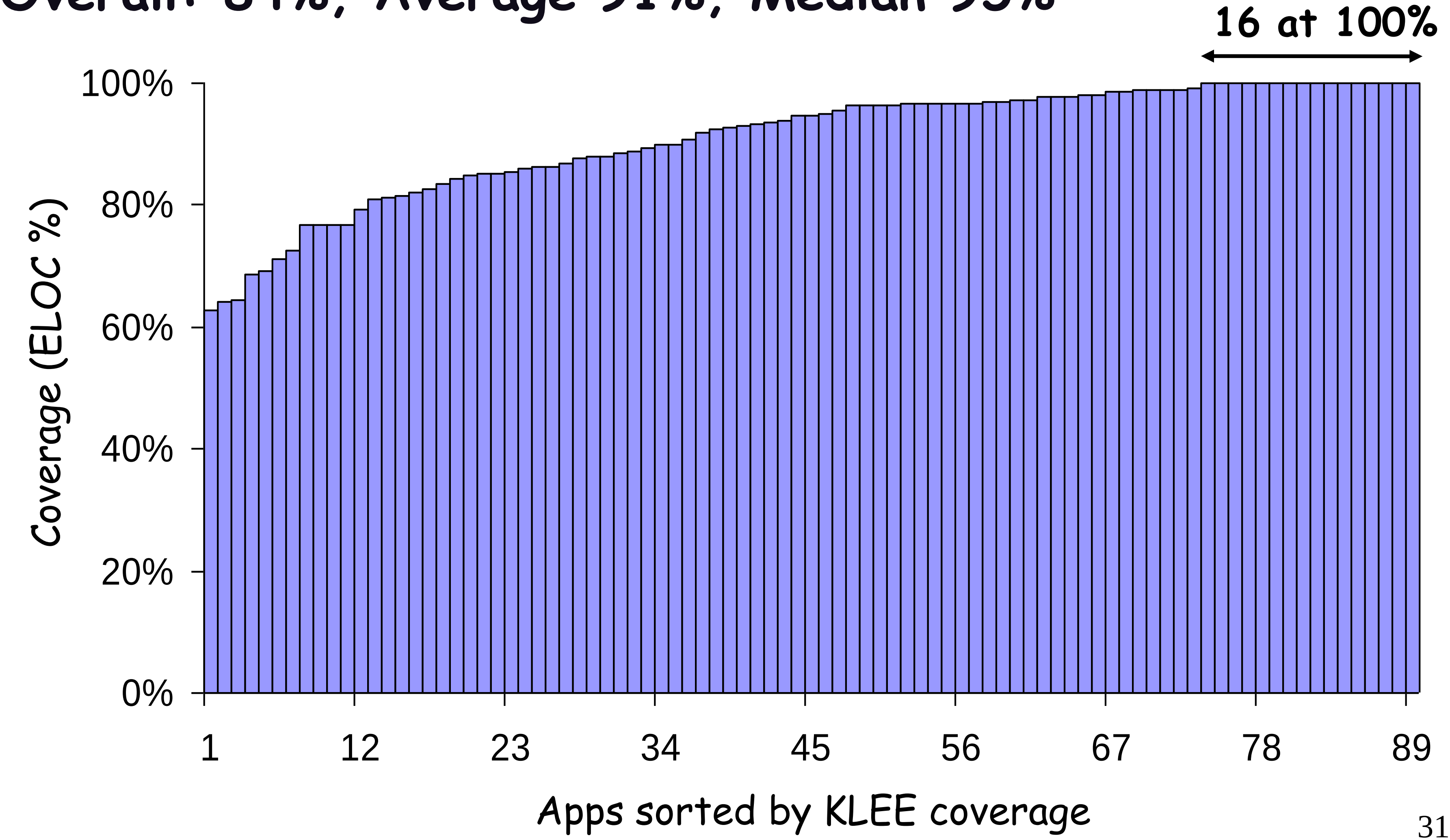
Methodology

- Fully automatic runs
- Run KLEE one hour per utility, generate test cases
- Run test cases on *uninstrumented* version of utility
- Measure line coverage using `gcov`
  - Coverage measurements not inflated by potential bugs in our tool

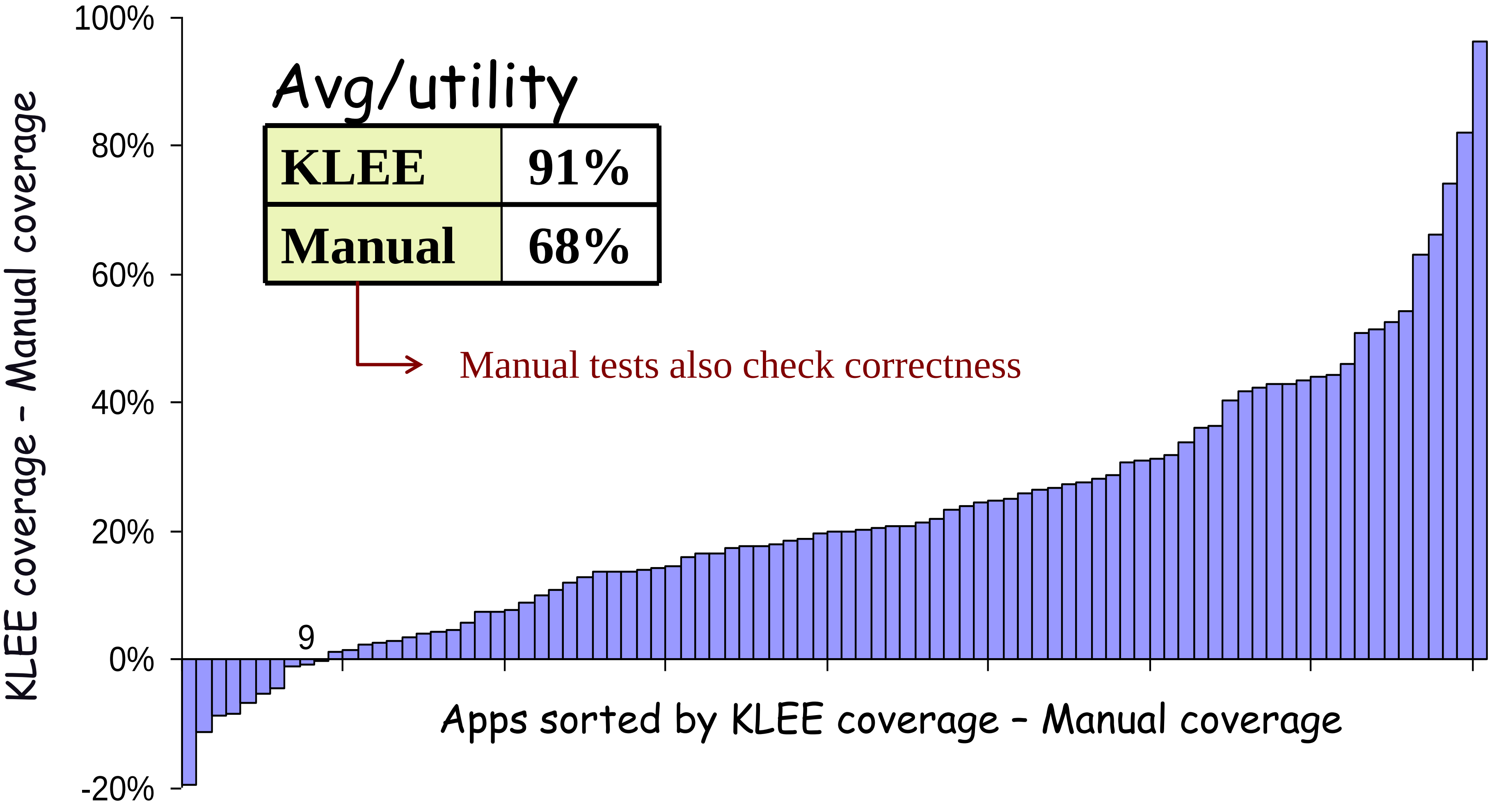
High Line Coverage

(Coreutils, non-lib, 1h/utility = 89 h)

Overall: 84%, Average 91%, Median 95%



Beats 15 Years of Manual Testing



Avg/utility

KLEE	**91%**
Manual	**68%**

Manual tests also check correctness

Evaluation

- Motivation and Overview
- Example and Basic Architecture
- Constraint Solving Challenges
- Evaluation
  - Coverage results
  - Bug finding
  - Crosschecking
  - Attack generation

Bug Finding Summary

	Applications
UNIX file systems	ext2, ext3, JFS
UNIX utilities	Coreutils, Busybox, Minix suites
MINIX device drivers	pci, lance, sb16
Library code	PCRE, uClibc, Pintos
Packet filters	FreeBSD BPF, Linux BPF
Networking servers	udhcpd, Bonjour, Avahi, telnetd, WsMp3
Operating Systems	HiStar kernel
Computer vision code	OpenCV

- Most bugs fixed promptly

GNU Coreutils Bugs

- Ten crash bugs
  - More crash bugs than approx previous three years combined
  - KLEE generates actual command lines exposing crashes

Ten command lines of death

```
md5sum -c t1.txt
mkdir -Z a b
mkfifo -Z a b
mknod -Z a b p
seq -f %0 1
```

```
pr -e t2.txt
tac -r t3.txt t3.txt
paste -d\\ abcdefghijklmnopqrstuvwxyz
ptx -F\\ abcdefghijklmnopqrstuvwxyz
ptx x t4.txt
```

t1.txt: `\t \tMD5(`

t2.txt: `\b\b\b\b\b\b\b\t`

t3.txt: `\n`

t4.txt: `A`

Experimental Evaluation

- Motivation and Overview
- Example and Basic Architecture
- Constraint Solving Challenges
- Results
    - Coverage results
    - Bug finding
    - Crosschecking
    - Attack generation

High-Level Semantic Bugs via Crosschecking

Assume f(x) and f’(x) implement the same interface

1. Make input x symbolic
2. Run tool on `assert(f(x) == f'(x))`
3. Find **mismatches!**

What to Crosscheck?

Lots of available opportunities

- Different implementations of the same functionality
  - e.g., libraries, servers, compilers
- Optimized versions of reference implementations
- Refactorings
- Reverse computation
  - e.g., compress/uncompress

Coreutils vs. Busybox

UNIX utilities should conform to *IEEE Std.1003.1*

- Crosschecked pairs of Coreutils and Busybox utilities
  - Busybox: implementation for embedded devices
- Found lots of mismatches

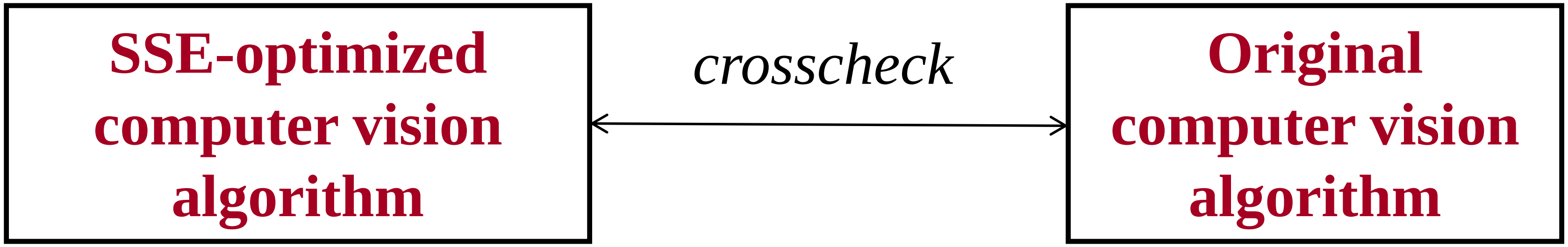
Mismatches Found

Input	Busybox	Coreutils
comm t1.txt t2.txt	[doesn’t show diff]	[shows diff]
tee -	[copies once to stdout]	[copies twice]
tee "" <t1.txt	[infinite loop]	[terminates]
cksum /	"4294967295 0 /"	"/: Is a directory"
split /	"/: Is a directory"	
tr	[duplicates input]	"missing operand"
[0 “<” 1]		"binary op. expected"
tail –2l	[rejects]	[accepts]
unexpand –f	[accepts]	[rejects]
split –	[rejects]	[accepts]

t1.txt: a   t2.txt: b   (no newlines!)

SSE Optimizations in Computer Vision Algorithms

- Computer vision algorithms often optimized to use SSE instructions
  - Operate on multiple data concurrently
  - Provide significant speedup
- Translation to SSE is usually done manually
  - Starting from a reference scalar implementation



SSE-optimized computer vision algorithm

crosscheck

Original computer vision algorithm

Computer Vision Algorithms and Floating Point Operations

- Computer vision algorithms make intensive use of **floating-point**
- No constraint solvers for floating-point available (IEEE 754 standard not pretty!)
  - Recent development: FP internal support in CMBC
  - Any other solvers that we can try?

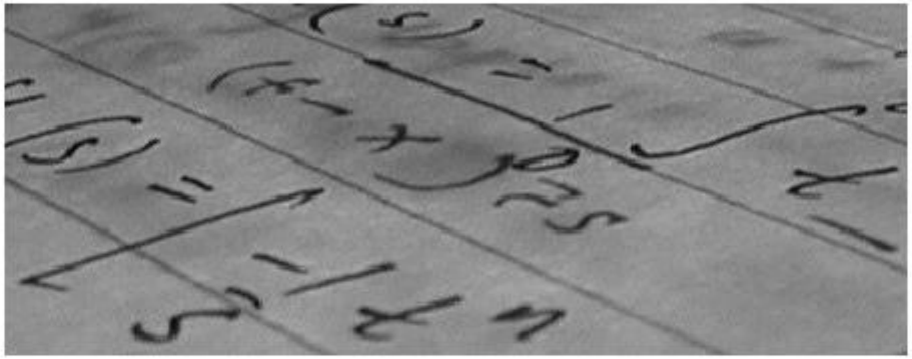
Computer Vision Algorithms and Floating Point Operations

- To ensure equality, the optimized SSE version needs to build FP values in roughly the same way
  - Observed developers try to mimic the scalar code using SSE
- Usually can cheaply prove/disprove equivalence via

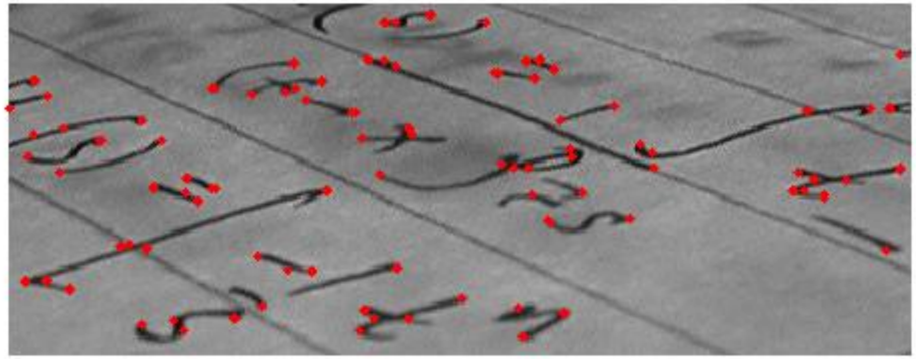
expression canonicalization + **syntactical expression matching**

SSE Optimizations in OpenCV

OpenCV: popular open-source computer vision library from Intel and Willow Garage



Corner detection algorithm



[from wikipedia.org]

OpenCV Results

- Crosschecked 51 SSE/scalar pairs
  - Proved the bounded equivalence of 41
  - Found mismatches in 10
- Most mismatches due to tricky FP-related issues:
  - Precision
  - Rounding
  - Associativity
  - Distributivity
  - NaN values

Example Source of Mismatches

min/max not commutative nor associative!

```
min(a,b) = a < b ? a : b

a < b (ordered) → always returns false if one
                  of the operands is NaN


min(NaN, 5) = 5
min(5, NaN) = NaN
```

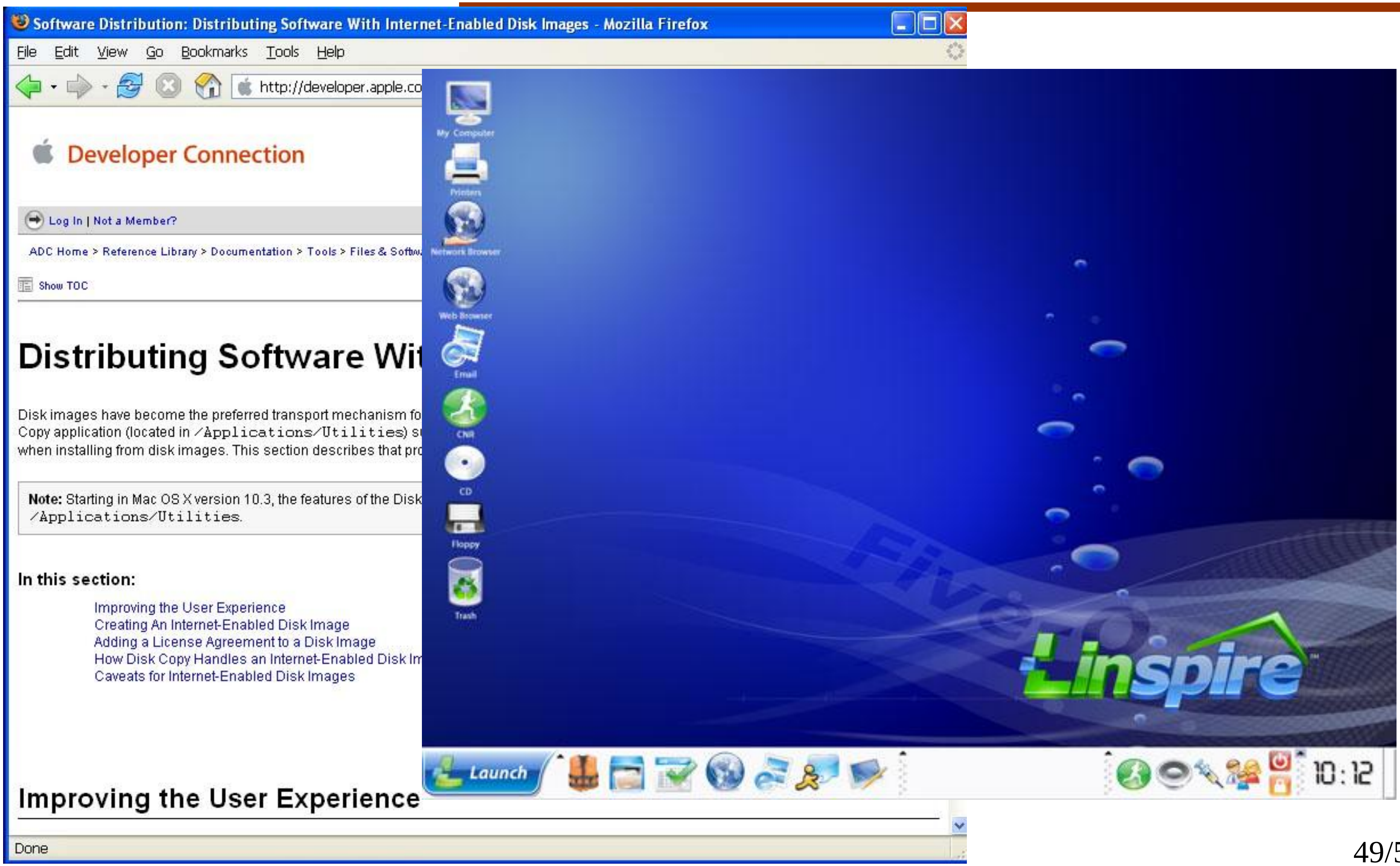
Could lead to arbitrarily large differences:

```
min(min(5, NaN),  100) = min(NaN, 100) = 100
min(5, min(NaN, 100))  = min(5, 100) = 5
```

Experimental Evaluation

- Motivation and Overview
- Example and Basic Architecture
- Constraint Solving Challenges
- Evaluation
  - Coverage results
  - Bug finding
  - Crosschecking
  - Attack generation

Attack Generation – File Systems



Distributing Software Wi

Disk images have become the preferred transport mechanism fo
Copy application (located in /Applications/Utilities) s
when installing from disk images. This section describes that pr

Note: Starting in Mac OS X version 10.3, the features of the Disk
/Applications/Utilities.

In this section:

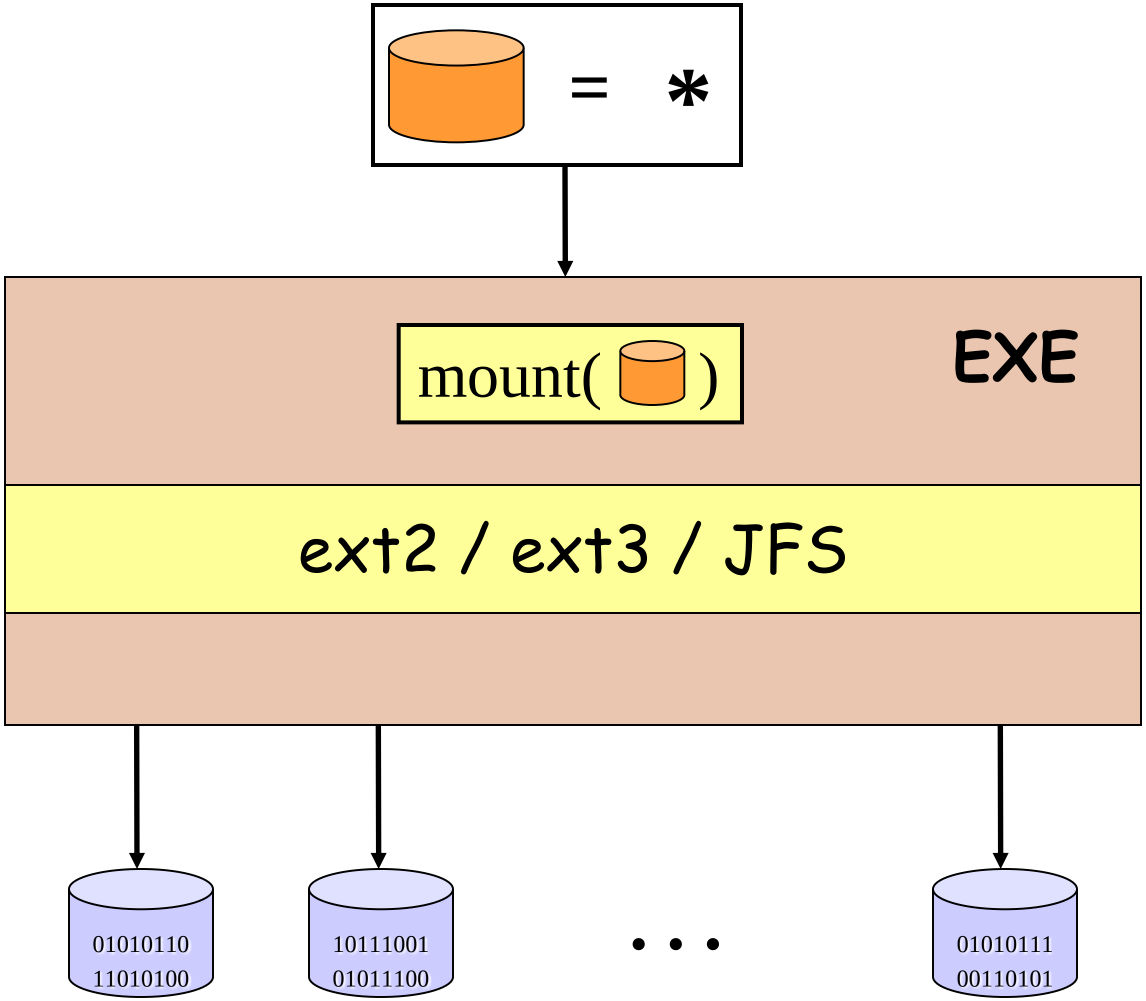
Improving the User Experience
Creating An Internet-Enabled Disk Image
Adding a License Agreement to a Disk Image
How Disk Copy Handles an Internet-Enabled Disk Im
Caveats for Internet-Enabled Disk Images

Improving the User Experience

Attack Generation – File Systems

- Mount code is executed by the kernel!
- Attackers may create malicious disk images to attack a system

Attack Generation – File Systems



Disk of death (JFS, Linux 2.6.10)

Offset	Hex Values							
00000	0000	0000	0000	0000	0000	0000	0000	0000
. . .				. . .				
08000	464A	3135	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								

64th sector of a 64K file. Mount.

And **PANIC** your kernel!

Dynamic Symbolic Execution: Effective Testing of Complex Software

Our techniques and tools can effectively:

– Generate high coverage test suites
  - Over 90% on average on Coreutils and Busybox utilities

– Generate inputs exposing bugs and security vulnerabilities in complex software
  - Including file systems, device drivers, library code, utility applications, network tools, packet filters

– Find semantic bugs via crosschecking
  – Crosschecked Coreutils and Busybox utilities, checked correctness of SSE optimizations

Symbolic Execution: Related Work

Symbolic execution for program testing introduced in the 1970s:

- James C. King. A new approach to program testing
  International Conference on Reliable Software, April 1975

Dynamic symbolic execution for automatic test case generation:

- EGT paper: [Cadar and Engler 2005]
- Independent work at Bell Labs on DART [Godefroid, Klarlund, Sen 2005]

Very active area of research, e.g:

- SAGE, Pex @ Microsoft Research
- JPF-SE, Symbolic JPF @ NASA Ames
- CREST @ UC Berkeley
- S2E, Cloud9, Oasis @ EPFL
- BitBlaze, WebBlaze @ UC Berkeley

KLEE: Available as Open-Source

http://klee.llvm.org

Already used and extended in many interesting ways by several research groups, in the areas of:

- wireless sensor networks
- schedule memoization in multithreaded code
- automated debugging
- exploit generation
- online gaming, etc.