Dynamic Symbolic Execution

Cristian Cadar

SOFTWARE RELIABILITY GROUP

Department of Computing

Imperial College London

Imperial College London

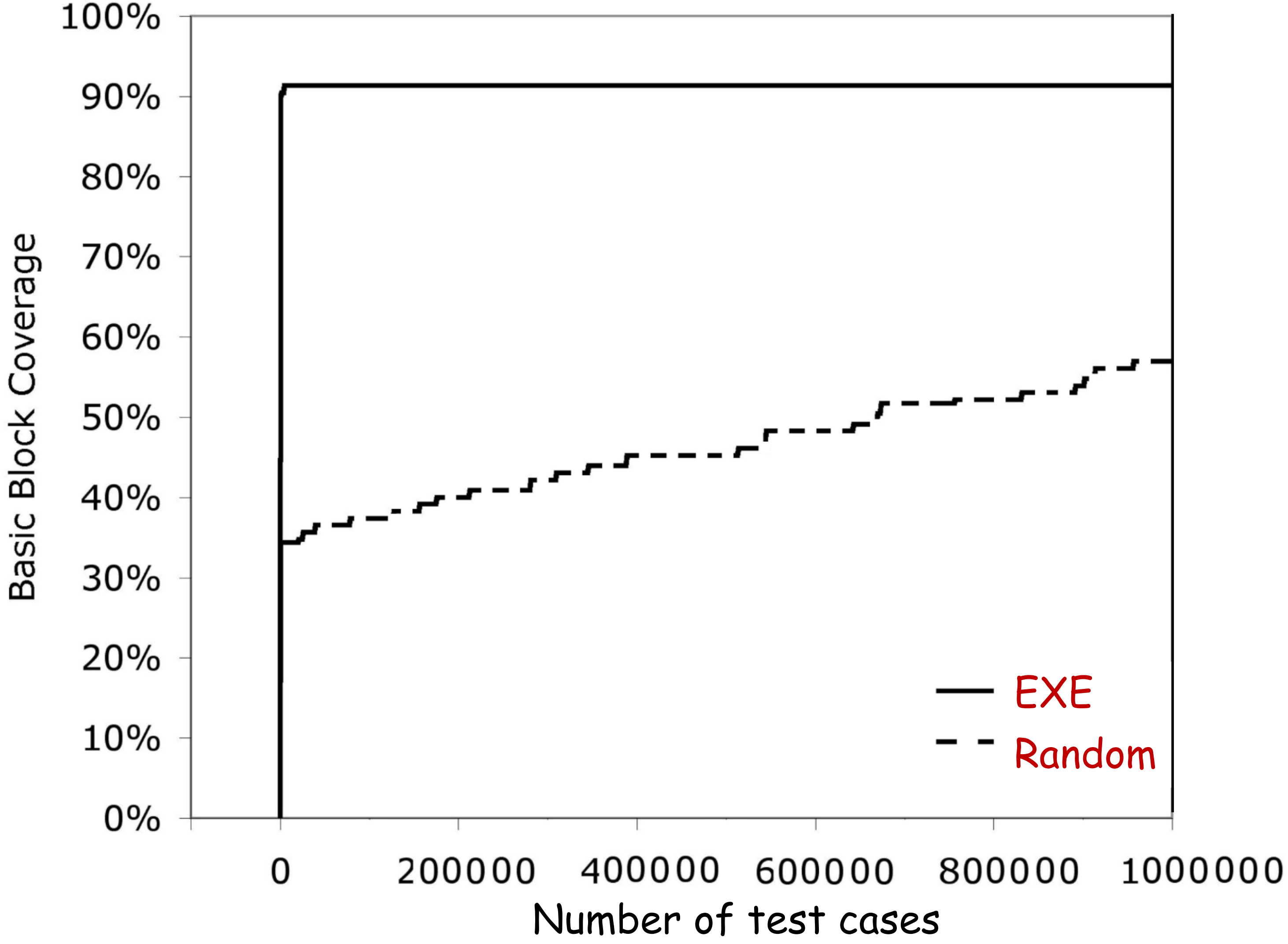
440:Software Reliability
Autumn 2016

Motivation

- Testing is hard
  - Manual testing is very expensive
  - Random (“fuzz”) testing is often ineffective
    - Hard to hit narrow input ranges
    - Hard to generate structured input

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

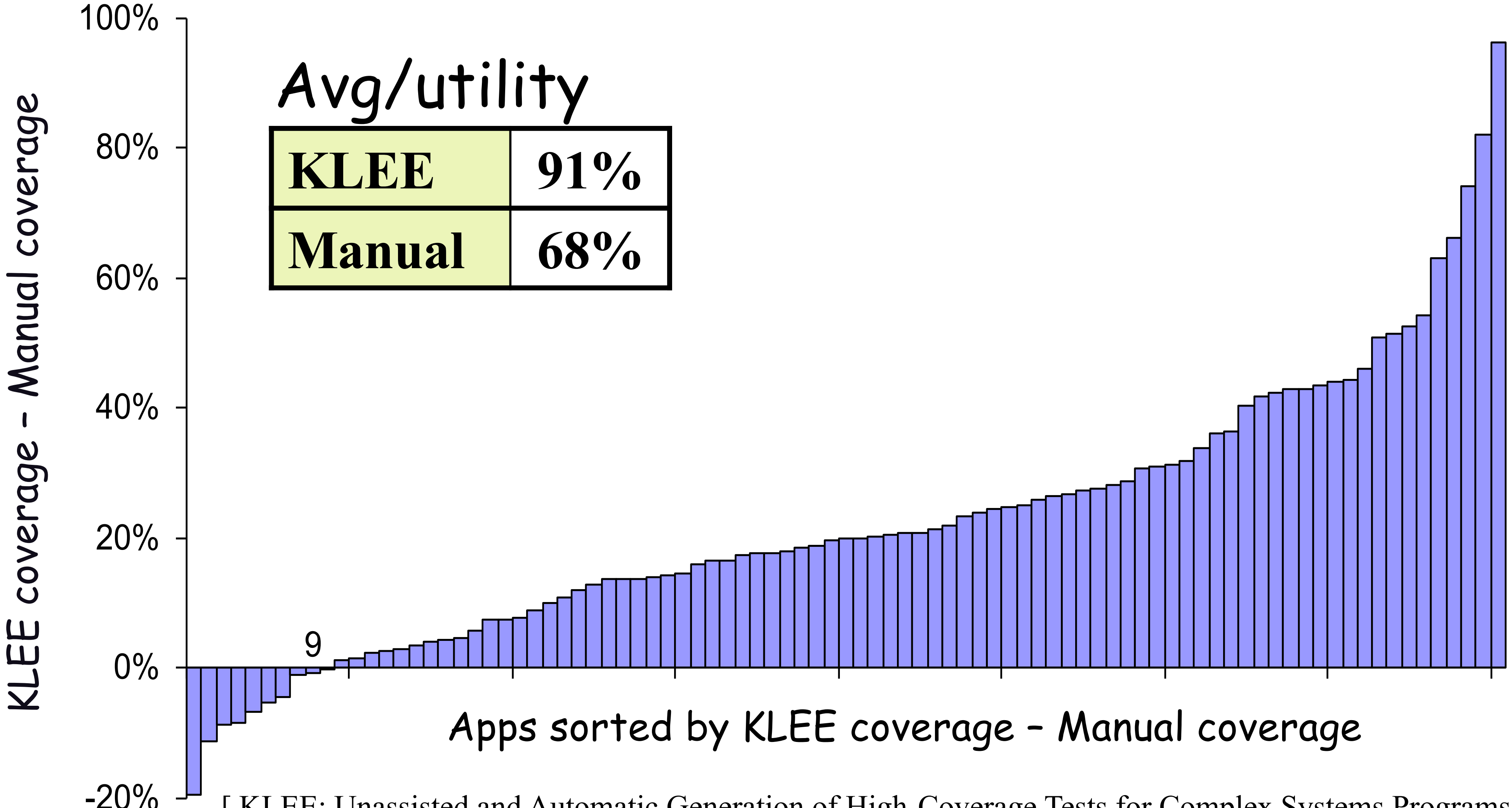
Sym Ex vs. Random Testing
(EXE on Berkeley Packet Filter)



[EXE: *Automatically generating inputs of death*

C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, D. Engler, CCS 2006]

Sym Ex vs. Manual Testing
(KLEE on Coreutils)



Avg/utility

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

[KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs
C. Cadar, D. Dunbar, D. Engler, OSDI 2006]

Sym ex vs dyn/static analysis

- Dynamic analysis requires test cases
  - It cannot reason about all possible values on a path
- Static analysis is imprecise
  - hard to find bugs dependent on specific values and/or memory layout; functional bugs (e.g. crosschecking)
  - false positives
  - does not generate test cases
  - + but it usually finds more bugs
  - + easier to apply (don’t need full program)
- Both are complementary
  - No reason not to run static; can use static info to improve symbolic execution
  - Can run tests cases generated by DSE on dynamic tools

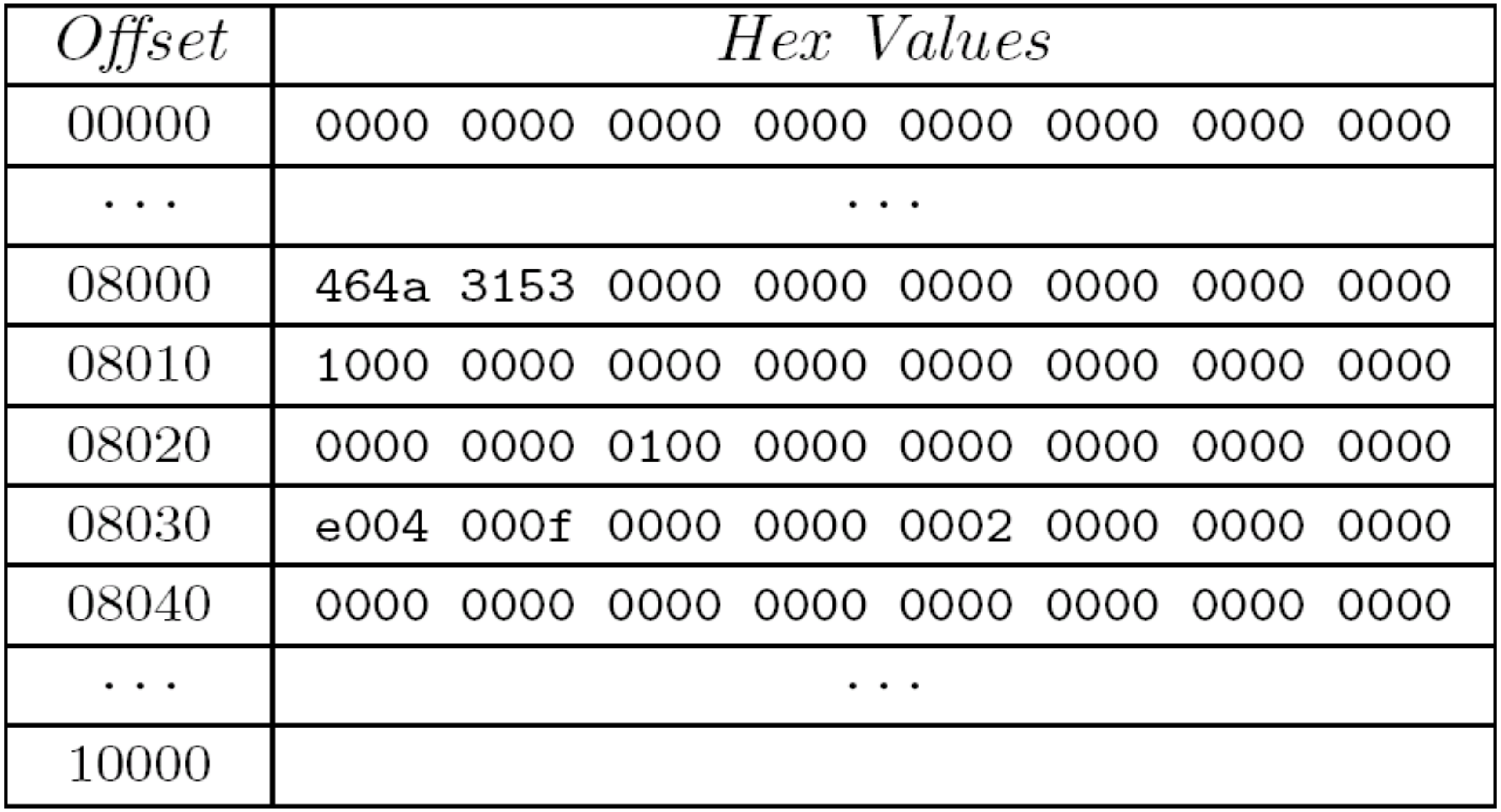
Dynamic Symbolic Execution (or Symbolic Execution or DSE)

- Automatic
  - does not require test cases
- Highly systematic
  - reaches deep code paths
  - achieves high statement/branch coverage
  - can reason about all possible values on a path
- Finds deep bugs
  - including those depending on specific values and/or memory layout
  - including functional bugs (see crosschecking study)
- Generates concrete test cases for explored paths
  - error reports for paths hitting a bug

PCRE – expressions of death

```
[^[\0^\0]\*-?]{\0
[\*-\`[\0^\0]\`-?]{\0
[\*-\`[\0^\0]\`-?]\0
(?#)\?[[[\0\0]\-]{\0
(?#)\?[[[\0\0]\[]\0
(?#)\?[[[\0\0]\-]\0
(?#)\?[[[\0\0][\0^\0]]\0
(?#)\?[[[\0\0][\0^\0]\]]\0
[\-\`[\0^\0]\`]{\0
[\*-\`[\0^\0]\`-?]\0
[\-\`[\0^\0]\`-]\0
(?#)\?[[[\0\0]\-]\0
(?#)\?[:[[\0\0]\-]\0
(?#)\?[[[\0\0]\]\0
(?#)\?[[[\0\0][\0^\0]-]\0
(?#)\?[=[[\0\0][\0^\0]\?]\0
```

Disk of death (JFS, Linux 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	

[*Automatically generating malicious disks using symbolic execution*
J. Yang, C. Sar, P. Twohey, C. Cadar, D. Engler , IEEE Security 2006]

Basic idea

- Run program on symbolic input, whose initial value is *anything*
- Program instructions become operations on symbolic expressions
- At conditionals that use symbolic inputs, fork execution and follow both paths:
  - On true branch, add constraint that condition is true
  - On false, that it is not
- When a path terminates, generate a test case by solving the constraints on that path

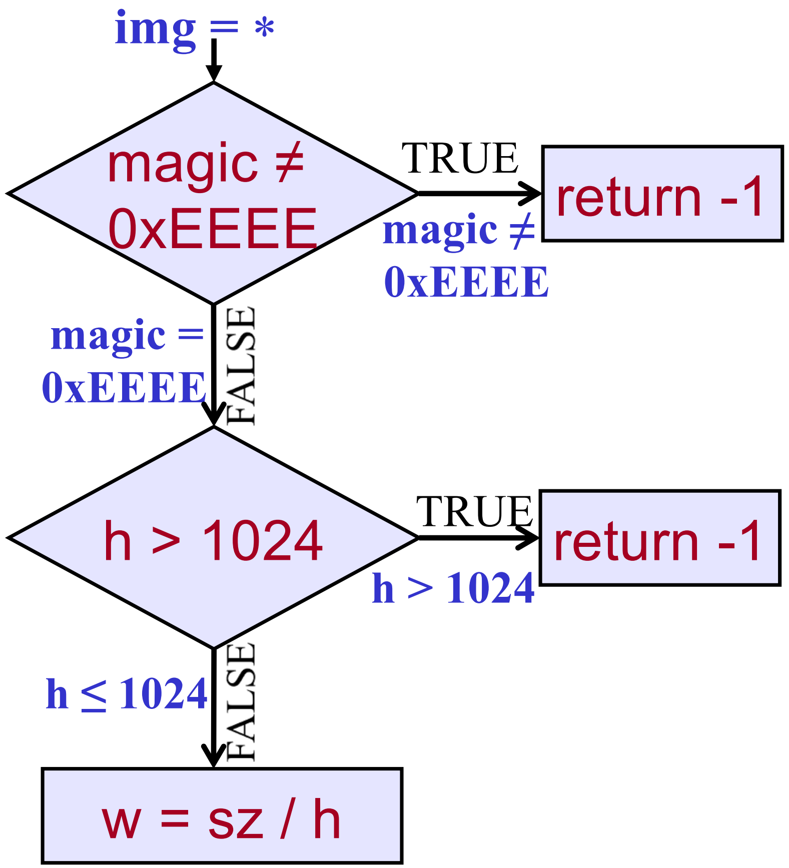
Dynamic SymEx in Practice

- Significant interest in the last few years
- Several dynamic symbolic execution/concolic tools available as open-source:
  - **KLEE, CREST, SYMBOLIC JPF**, etc.
- Started to be explored/adopted by industry:
  - Microsoft, Fujitsu, Hitachi, Intel, NASA, etc.

Toy Example

```
struct image_t {
    unsigned short magic;
    unsigned short h, sz;
    ...
```

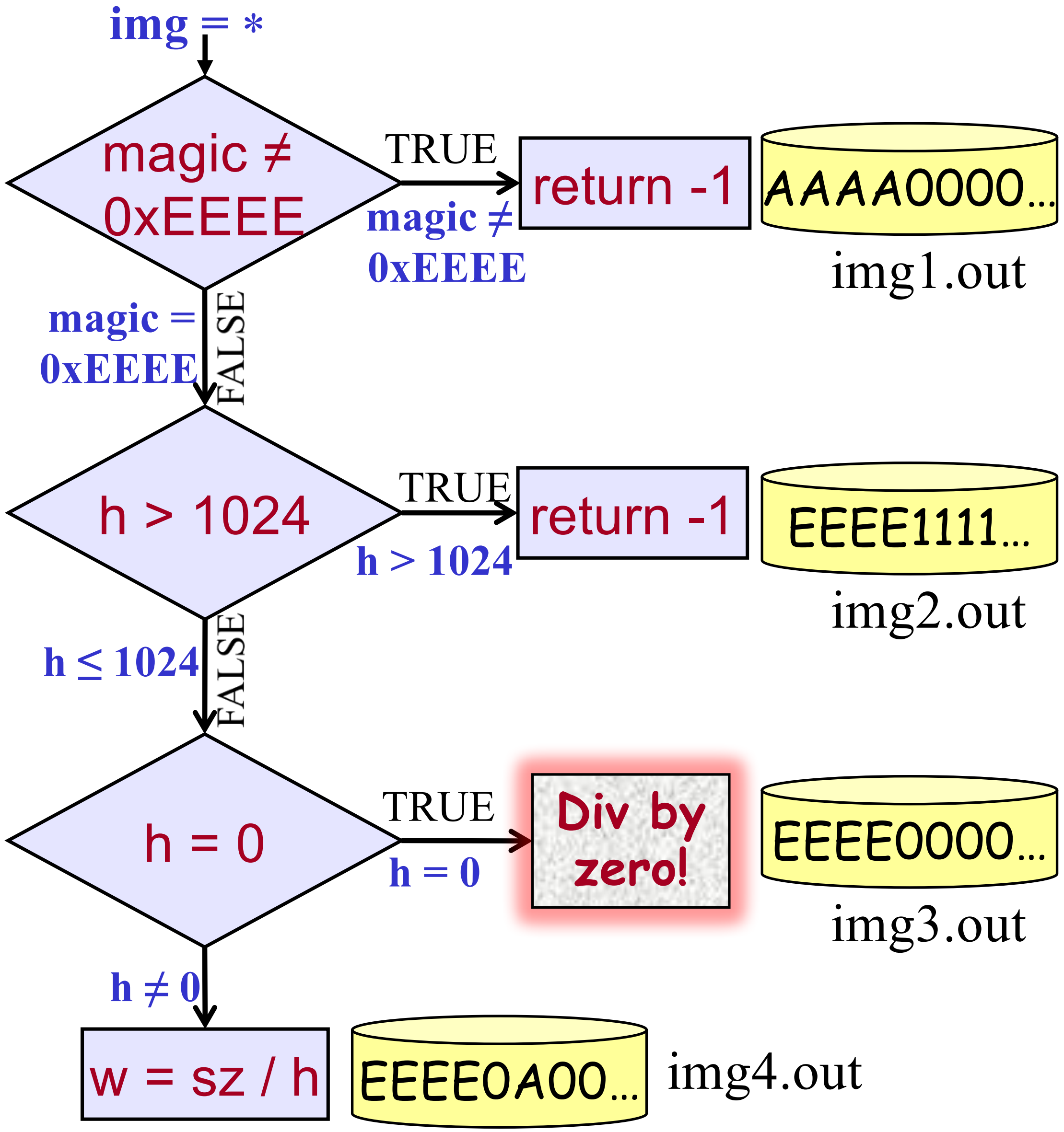
```
int main(int argc, char** argv) {
  ...
  image_t img = read_img(file);
  if (img.magic != 0xEEEE)
    return -1;
  if (img.h > 1024)
    return -1;
  w = img.sz / img.h;
  ...
}
```



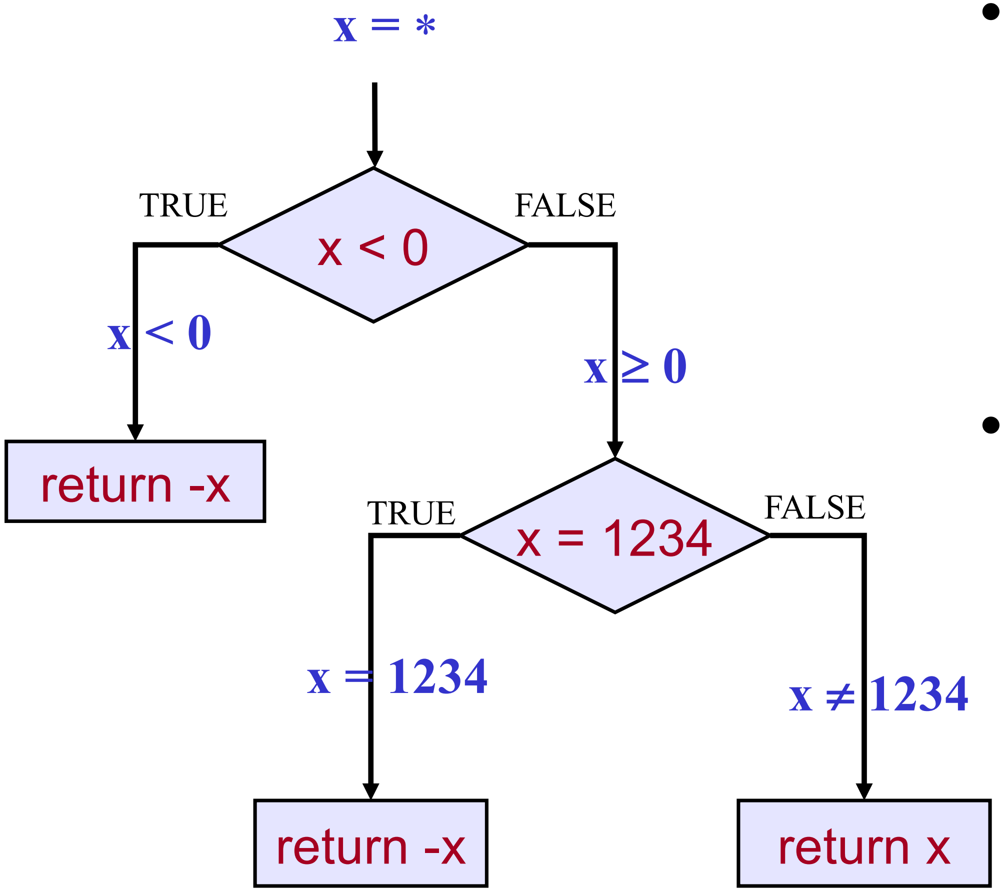
Toy Example

```
struct image_t {
    unsigned short magic;
    unsigned short h, sz;
    ...
```

```
int main(int argc, char** argv) {
  ...
  image_t img = read_img(file);
  if (img.magic != 0xEEEE)
    return -1;
  if (img.h > 1024)
    return -1;
  w = img.sz / img.h;
  ...
}
```

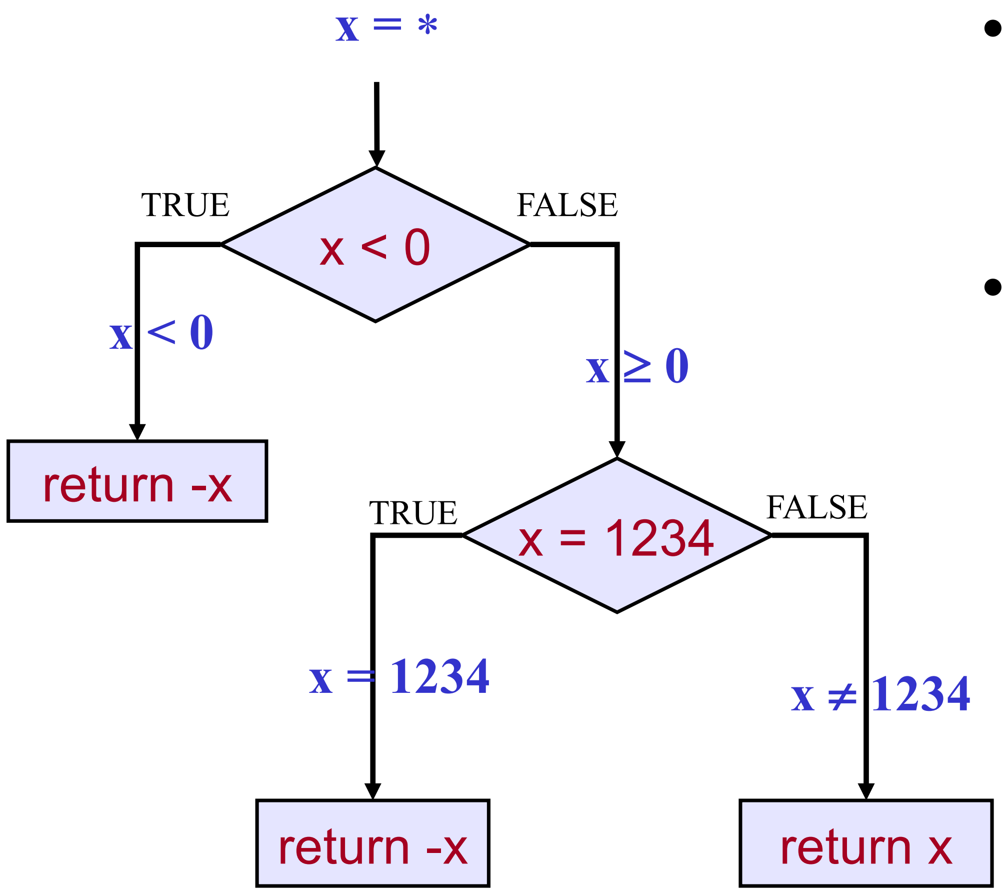


Some Concepts and Terminology



- Execution paths of a program can be seen as a binary **execution tree**
  - Internal nodes are decision points in the program
  - Leaves are program exit points
- Execution trees of real programs are essentially infinite
  - Symbolic execution incrementally explores parts of the execution tree
  - The leaves of the "current" execution tree form the set of **active states**

Some Concepts and Terminology



- Each path from the root to a leaf represents the execution of an **equivalent set of inputs**
- The conjunction of constraints gathered on an execution path is called the **path condition** or **path constraints (PC)**

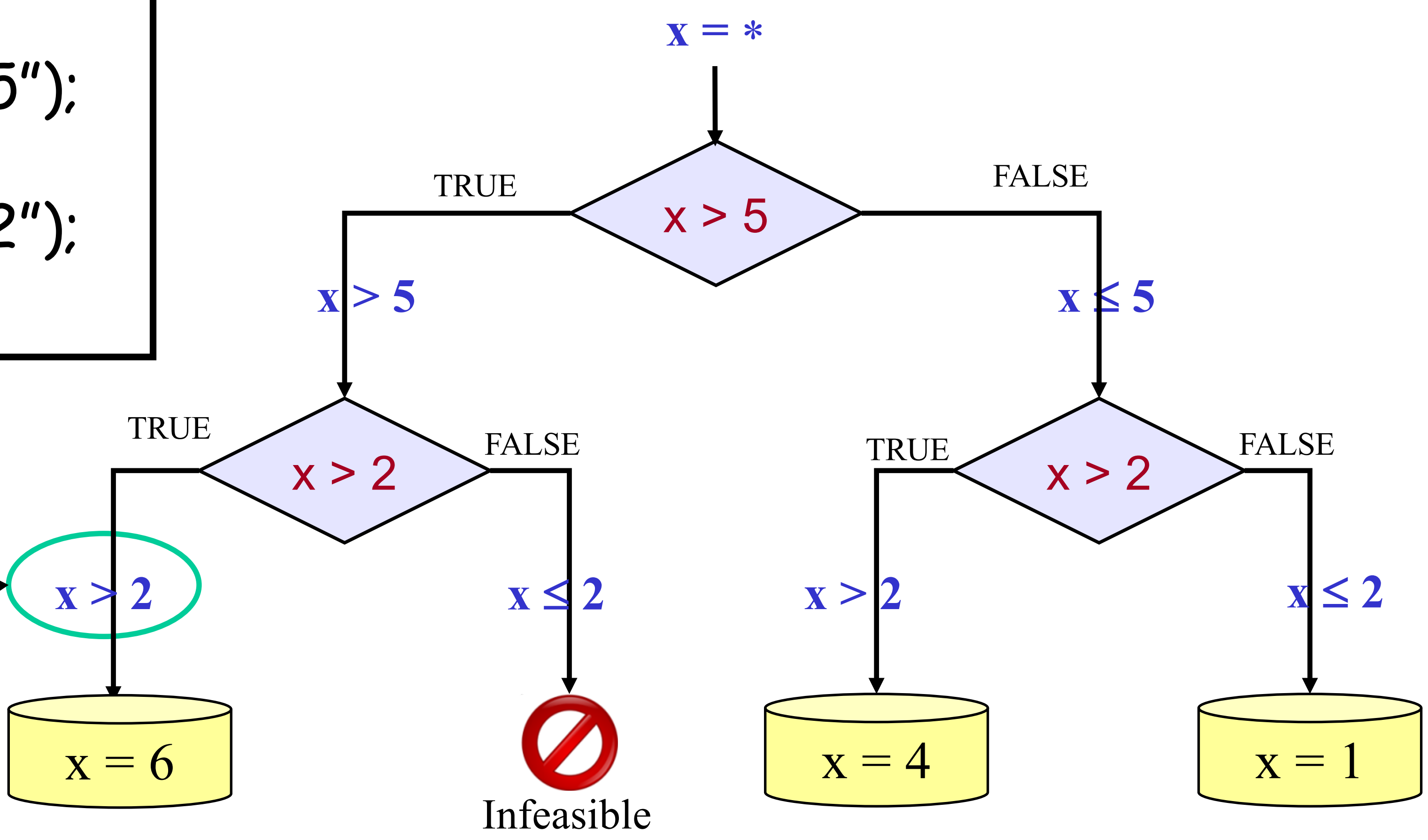
Feasible vs Infeasible Paths

How many paths?

```
int foo(int x) {
   if (x > 5)
      printf(">5");
   if (x > 2)
      printf(">2");
}
```

Symbolic execution explores only feasible paths!

No need to add implied constraints



All-Value Checks

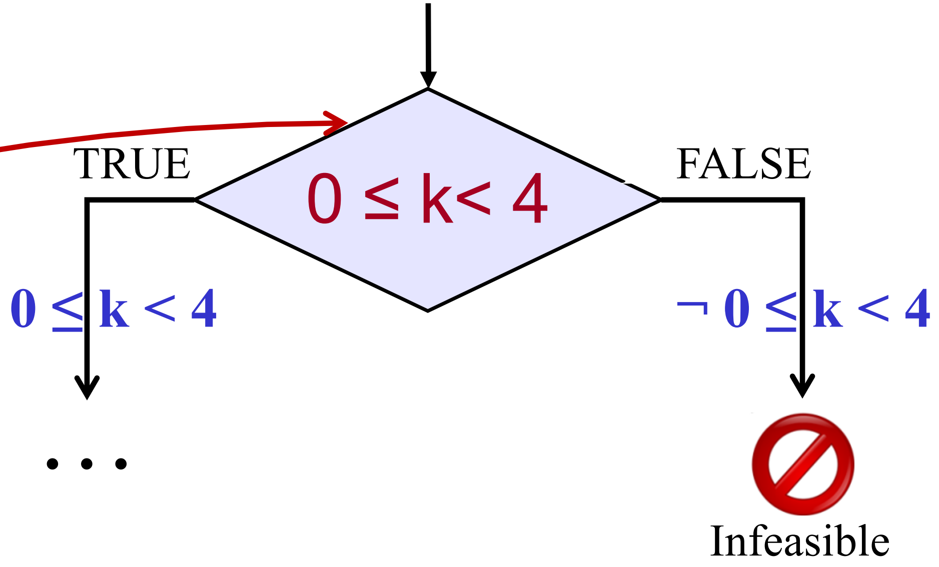
Implicit checks for general properties:

- Pointer dereferences
- Array indexing
- Division/modulo operations
- Assert statements

All-value checks!

- Errors are found if **any** buggy values exist on that path!
- Discussion: compare with regular testing, then Valgrind

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



$0 \le k < 4$

TRUE

FALSE

$0 \le k < 4$

$\neg\ 0 \le k < 4$

. . .

Infeasible

All-Value Checks

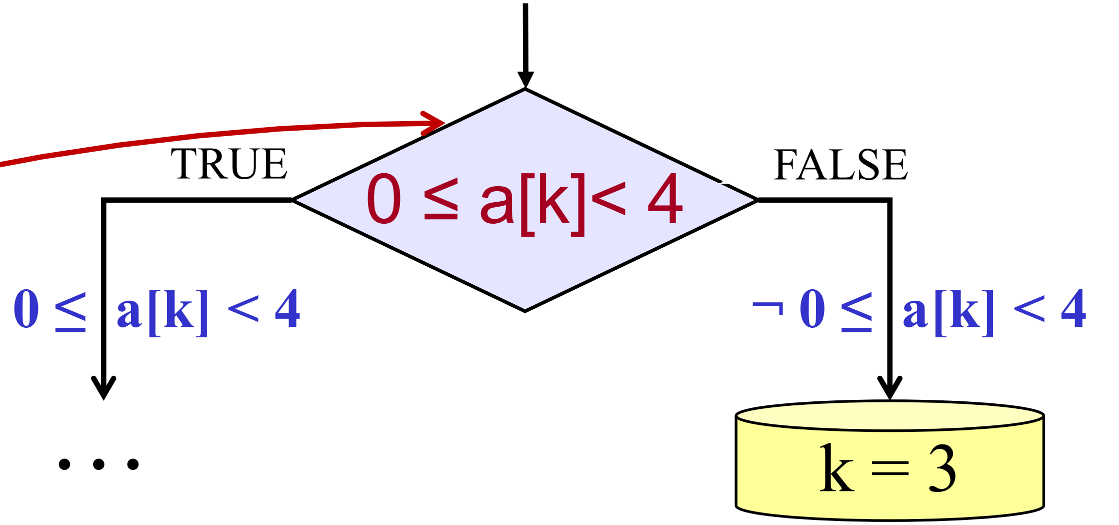
Implicit checks for general properties:

- Pointer dereferences
- Array indexing
- Division/modulo operations
- Assert statements

All-value checks!

- Errors are found if **any** buggy values exist on that path!
- Discussion: compare with regular testing, then Valgrind

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



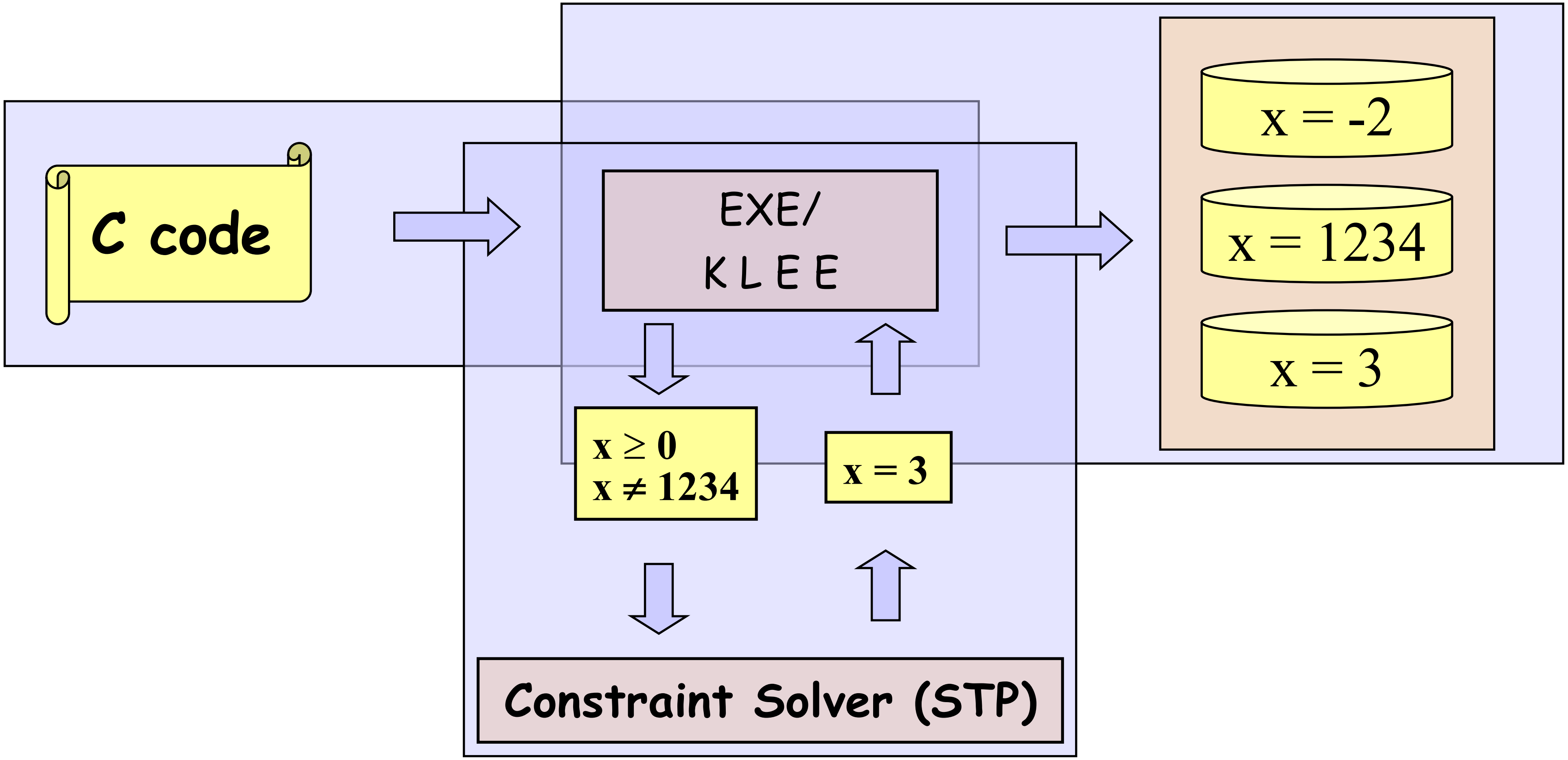
Buffer overflow!

Mixed Concrete/Symbolic Execution

- All operations that do not depend on the symbolic inputs are (essentially) executed as in the original code!
- Ability to interact with the outside environment
  - System calls, uninstrumented libraries
- Only relevant code executed symbolically
  - Without the need to extract it explicitly
  - For many real programs (and test drivers) most operations are concrete
  - The statements executed symbolically form **the symbolic slice**

[Discussion: scalability of symbolic execution?]

EXE and KLEE


C code
EXE/
KLEE
$x \geq 0$
$x \neq 1234$
$x = 3$
Constraint Solver (STP)
$x = -2$
$x = 1234$
$x = 3$

Implementing Dynamic Analyses

- Dynamic analysis: run program and observe execution
- Simplest form: run program, check output
- More sophisticated analyses require finer-grained observations

E.g., buffer overflow detection tool would likely need to instrument:

- Memory accesses
- Allocations and deallocations
- Pointer arithmetic

Instrumentation Choices

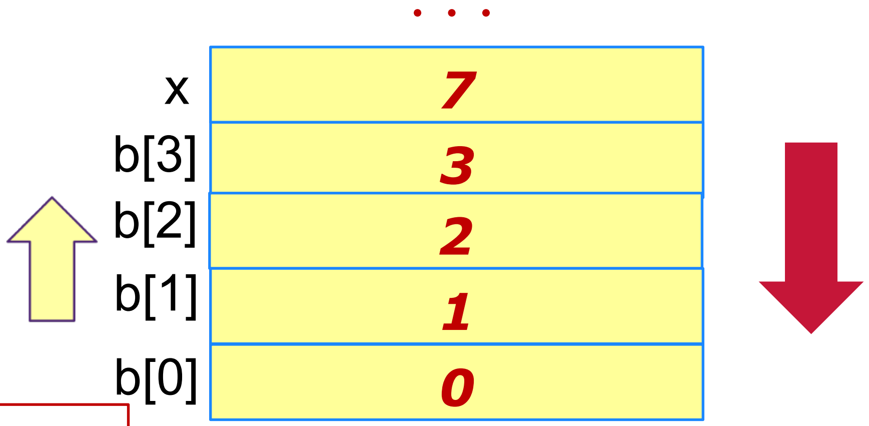
1) Instrumentation level
   - Source-level
   - Binary-level
   - Intermediate-language level
2) Instrumentation time
   - Static instrumentation
   - Dynamic/runtime instrumentation

Source vs binary-level

	SOURCE	BINARY
Source access	⬇	⬆
Recompiling	⬇	⬆
Ease of instrumentation	⬆	⬇
Information available	⬆ ⬇	⬆ ⬇

⬆ advantage, ⬇ disadvantage (not absolute)

```
foo() {
   int x = 7, b[4] = {0,1,2,3};
   …
   b[4] = 4;
   …
```



Intermediate-level is somewhere in-between, depending on the intermediate language

Static vs Dynamic Instrumentation

Static instrumentation: change code before it is run, generate new binary, run it

Dynamic instrumentation: instrument programs as they run, like an interpreter

	STATIC	DYNAMIC
Tracking dependencies	↓	↑
Relinking	↓	↑
Dynamically changing instrumentation	↓	↑
Self-modifying code	↓	↑
Performance	↑	↓
Ease of implementing	↑ (esp. source)	↓

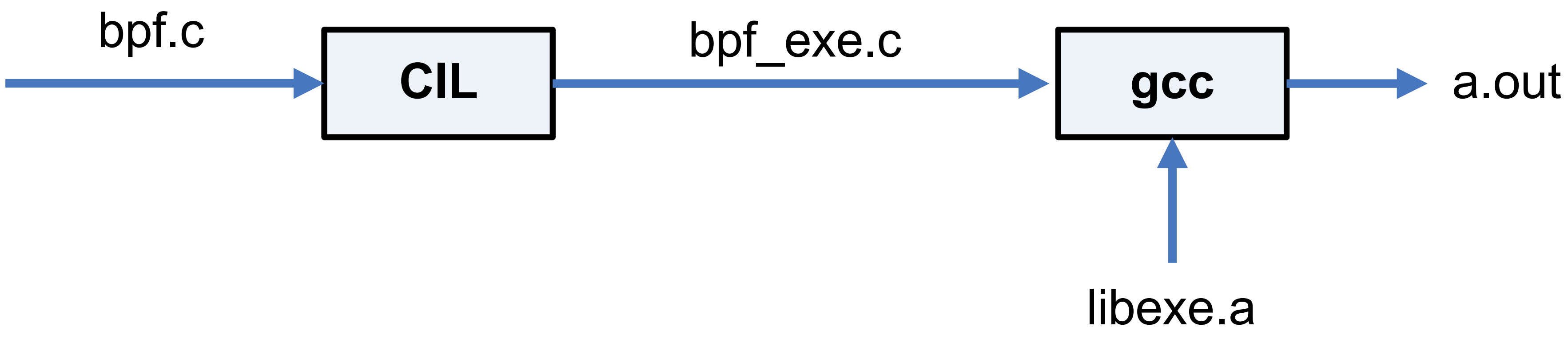
↑ **advantage,** ↓ **disadvantage (not absolute)**

Back to DSE: EXE vs KLEE

- EXE: Static instrumentation @ source-level
- KLEE: Dynamic instrumentation @ intermediate level (LLVM)

Running EXE

```
$ exe-cc bpf.c
$ ./a.out
```



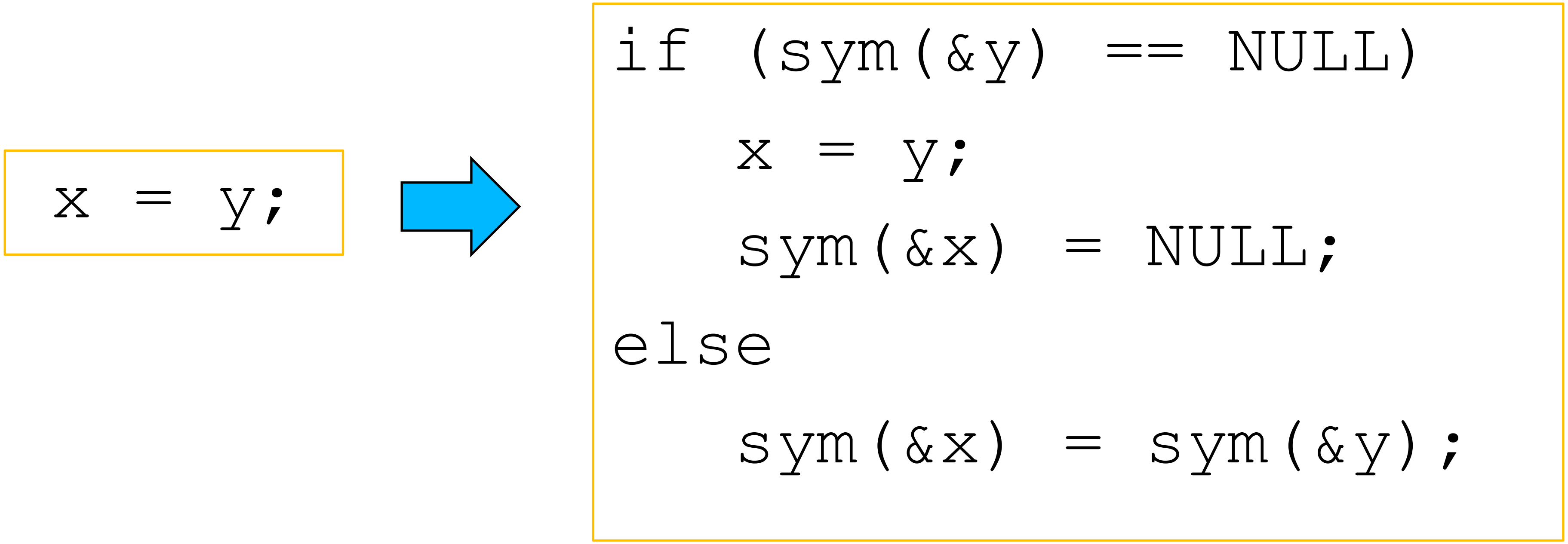
[CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs
Necula, McPeak, Rahul, and Weimer, CC 2002]

exe-cc: x = y

`sym(&x)` =

  Pointer to symbolic expression, *if x is symbolic*

  NULL, *if x is concrete*



```
x = y;
```

```
if (sym(&y) == NULL)
    x = y;
    sym(&x) = NULL;
else
    sym(&x) = sym(&y);
```

exe-cc: v = x OP y

```
sym_exp(OP, Sx, Sy) =
   create the symbolic expression Sx OP Sy
ct(x) = create constant expression with value c
```

```
v = x OP y
```

→

```
if (sym(&x) == NULL && sym(&y) == NULL)
   v = x OP y;
   sym(&v) = NULL;
else if (sym(&x) == NULL)
   sym(&v) = sym_exp(OP, ct(x), sym(&y));
else if (sym(&y) == NULL)
   sym(&v) = sym_exp(OP, sym(&x), ct(y));
else
   sym(&v) = sym_exp(OP, sym(&x), sym(&y));
```

exe-cc: if (x) s1; else s2

```
if (x)
    s1;
else s2;
```

```
if (sym(&x) == NULL)
   if (x)
        goto s1_label;
   else goto s2_label;
else
   if (fork() == 0)
      push_constr(sym_exp(NEQ,
                  sym(&x), ct(0)));
      goto s1_label;
   else
      push_constr(sym_exp(EQ,
                  sym(&x), ct(0)));
      goto s2_label;
```

```
void push_constr(c) {
   add_sym_constr(c, PC);
   if (unsat(PC))
      kill_path();
}
```

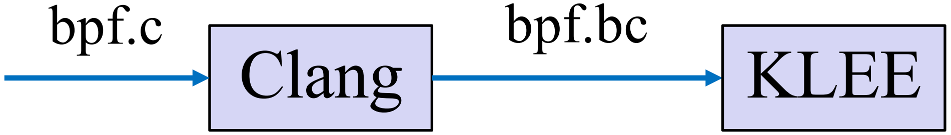
Rough sketch: some aspects (e.g, scheduling) and refinements omitted

exe-cc

- All other cases can be reduced to the cases above, or slight variations of them, via simple syntactic transformations, e.g., introducing temporary variables
  - CIL helps with most of this
- If there are any questions about any program constructs, let me know
- You can also refer to the extended journal version, available on my website

Running KLEE

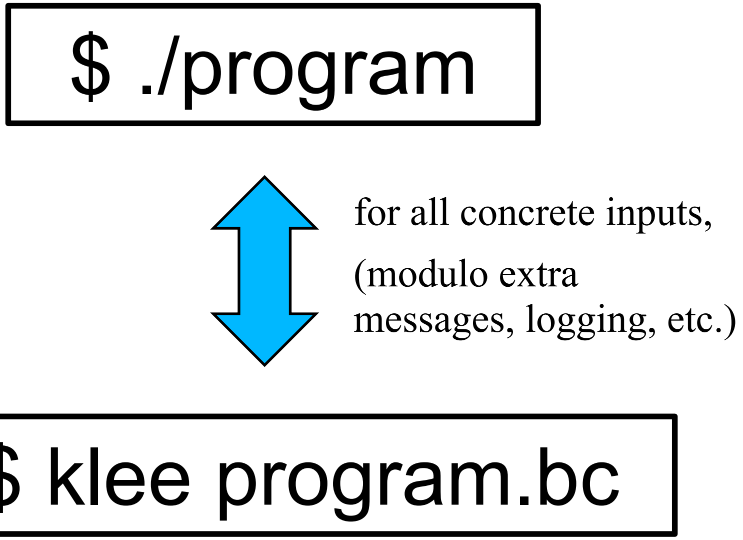
```
$ clang –c –emit-llvm bpf.c
$ klee bpf.bc
```



KLEE: LLVM Bitcode Interpreter

- Works as a mixed concrete/symbolic interpreter for LLVM bitcode

```
Instruction *i = ki->inst;
 switch (i->getOpcode()) {
     case Instruction::Ret:
     …
     case Instruction::Br:
        // if both sides feasible, fork
        …
```



```
$ ./program
```

for all concrete inputs,
(modulo extra
messages, logging, etc.)

```
$ klee program.bc
```

DSE Scalability Challenges

Path exploration

- Employing search heuristics [CCS'06, OSDI'08, ICSE'12, ESEC/FSE'13]
- Dynamically eliminating redundant paths [TACAS'08]
- Statically merging paths [EuroSys'11]
- Using existing test suites to prioritize execution [ICSE'12]
- Targeting patches [ESEC/FSE'13, ICSE'16]

Constraint solving

- Bit-level modeling of memory [CCS'06, IEEE S&P'06]
- Caching [CCS'06, OSDI'08]
- Exploiting subset/superset relations [OSDI'08]
- Using rewrite rules [EuroSys'11, HVC'11]
- Using a portfolio of solvers [CAV'13]
- etc.

Examples from our work; lots of great work from other groups.

Path Exploration Challenges

Naïve exploration can easily get "stuck"

- Employing search heuristics
- Dynamically eliminating redundant paths
- Statically merging paths
- Using existing regression test suites to prioritize execution
- etc.

Search Heuristics

- Depth-First Search
  - Advantage?
- Breadth-First Search
  - Advantage?
- Coverage-optimized search (best-first)
- Random state selection
- Random path search
- etc.

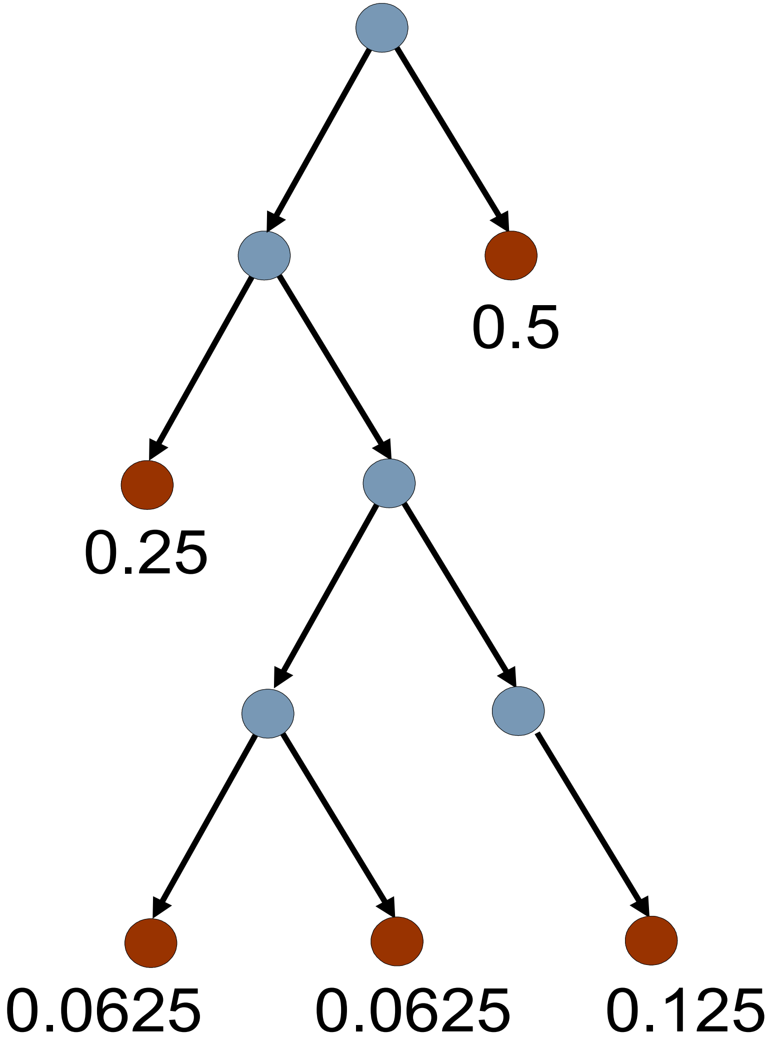
Search Heuristics

- EXE’s best-first heuristic to optimize coverage
  - Pick the process at the line of code run the fewest number of times
  - Run it in DFS mode for a while, then iterate
- KLEE’s Random Path Selection
  - See next slide

Random Path Selection

Key idea: subtrees have equal prob. of being selected, irresp. of their size

- NOT random state selection
- Favors paths high in the tree
  - fewer constraints
- Avoid starvation
  - e.g. symbolic loop



Which Search Heuristic?

One approach [KLEE]: use multiple heuristics in a round-robin fashion!

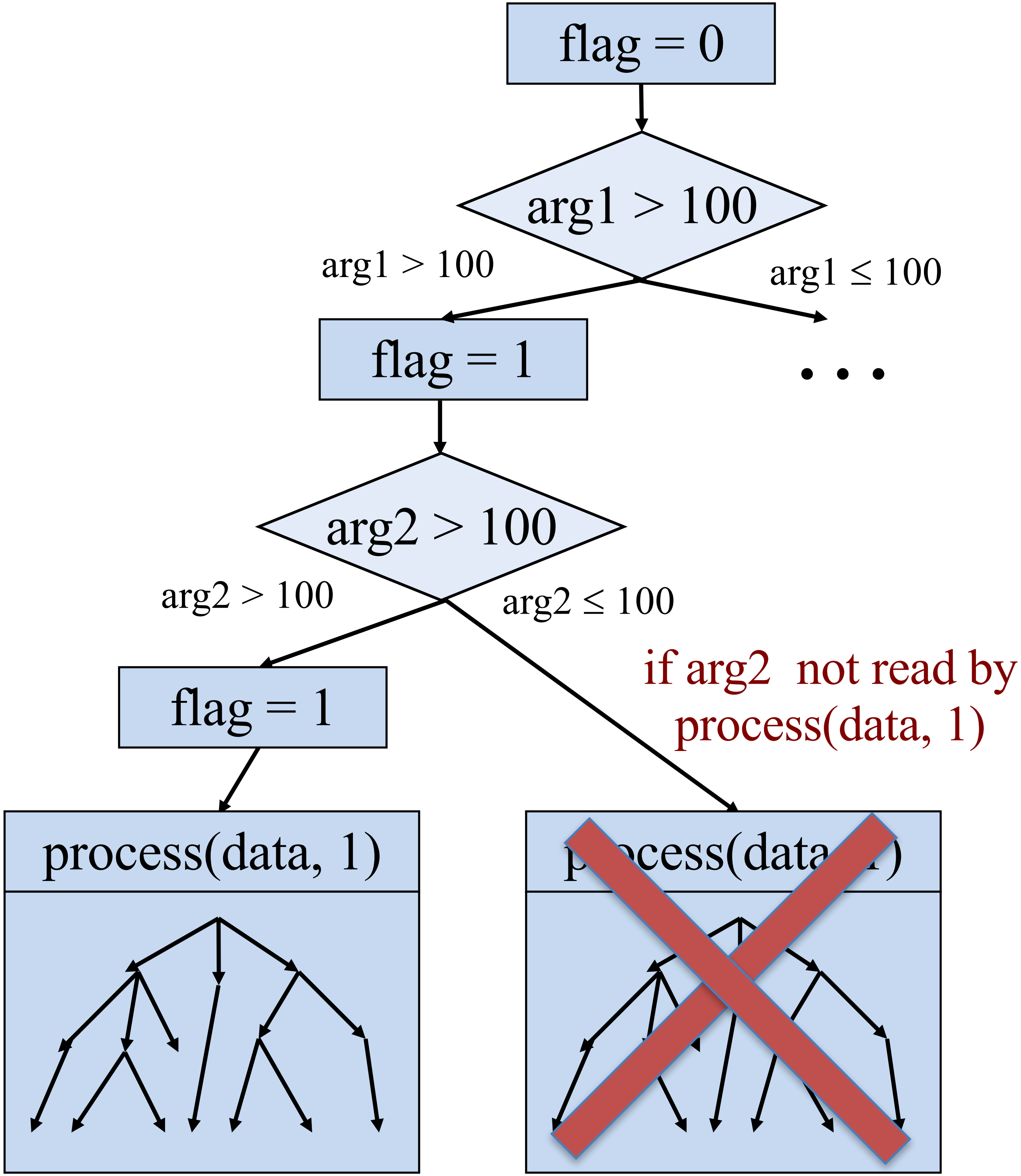
- Protects against individual heuristics getting stuck in a local maximum

Eliminating Redundant Paths

- If two paths reach the same program point with the same constraint sets, we can prune one of them
- We can discard from the constraint sets of each path those constraints involving memory which is never read again

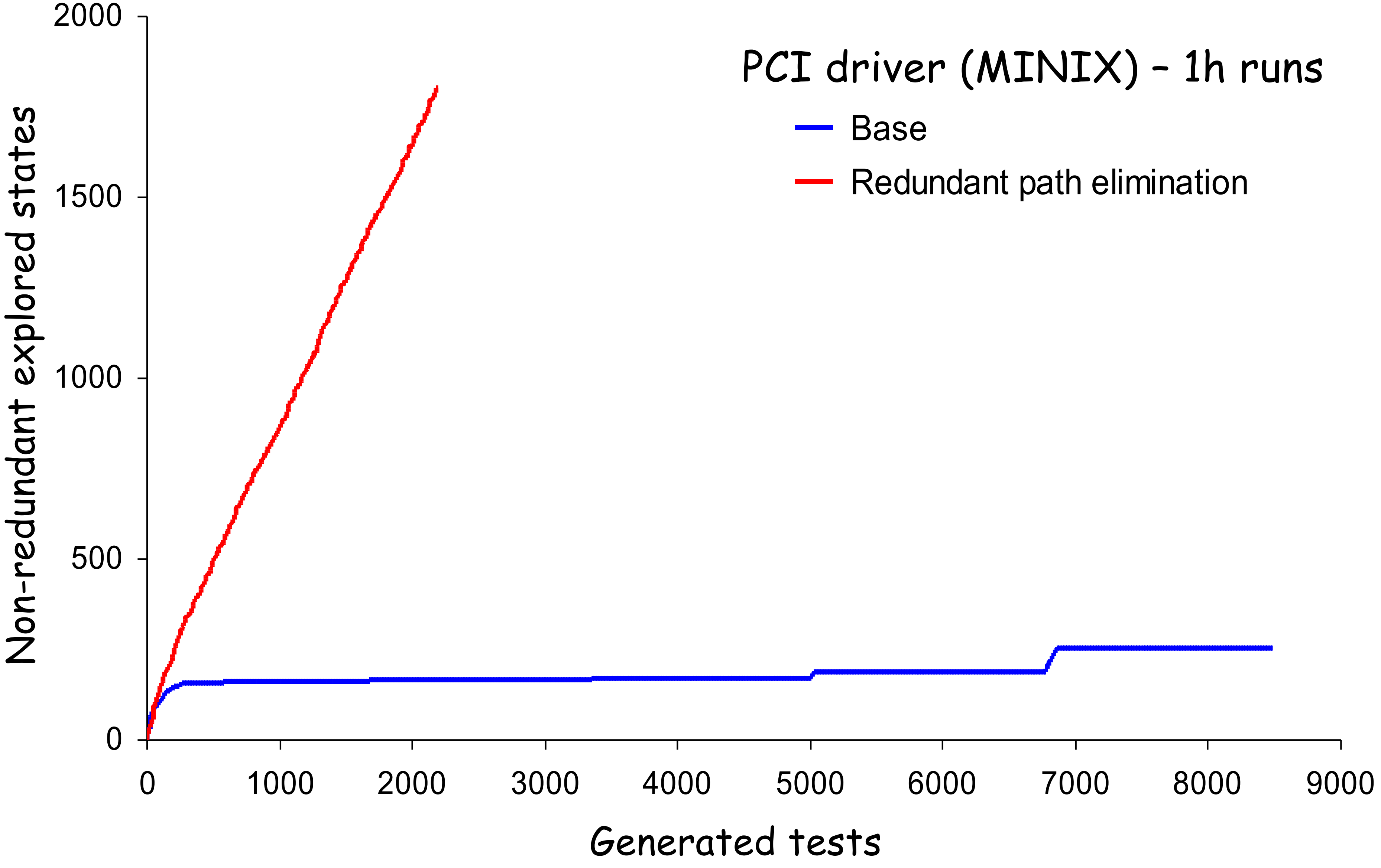
[RWset: Attacking Path Explosion in Constraint-Based Test Generation, Boonstoppel, Cadar, Engler, TACAS 2008]

```
data, arg1, arg2 = *

flag =  0;

if (arg1 > 100)
    flag = 1;

if (arg2 > 100)
    flag = 1;

process(data, flag);
```



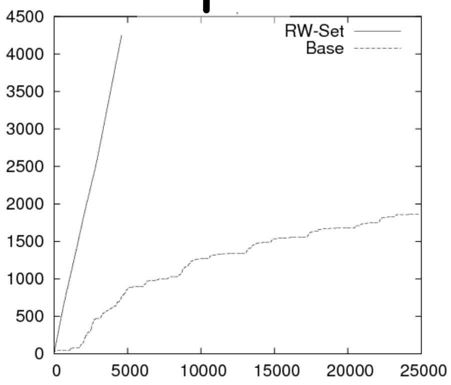
if arg2 not read by process(data, 1)

Many Redundant Paths

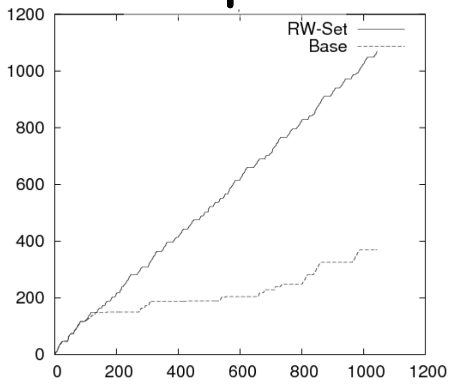


Lots of Redundant Paths

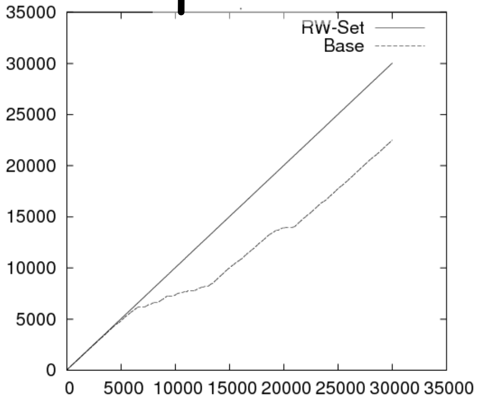
bpf



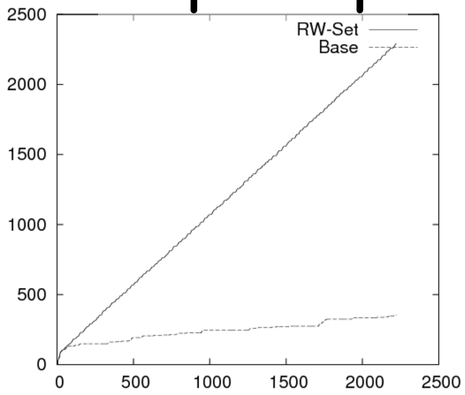
expat



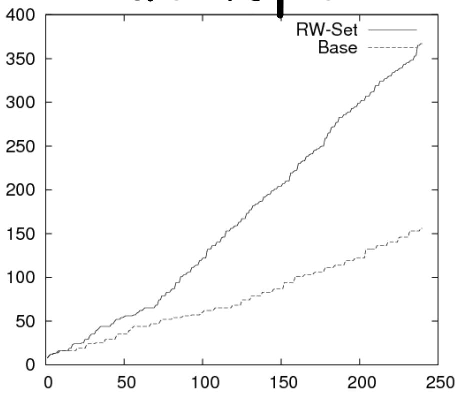
pcre



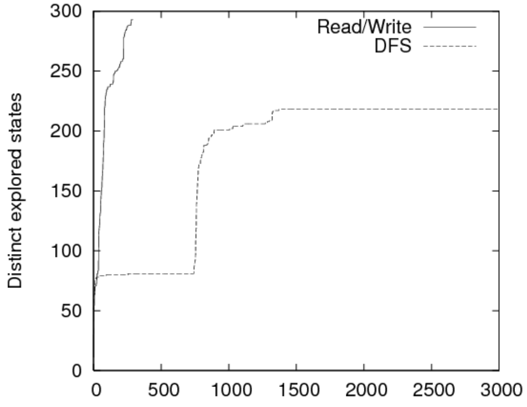
tcpdump



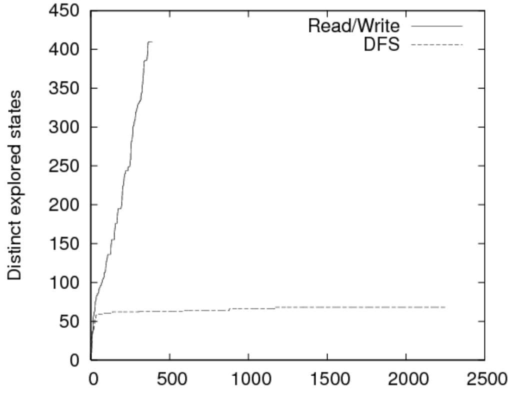
udhcpd



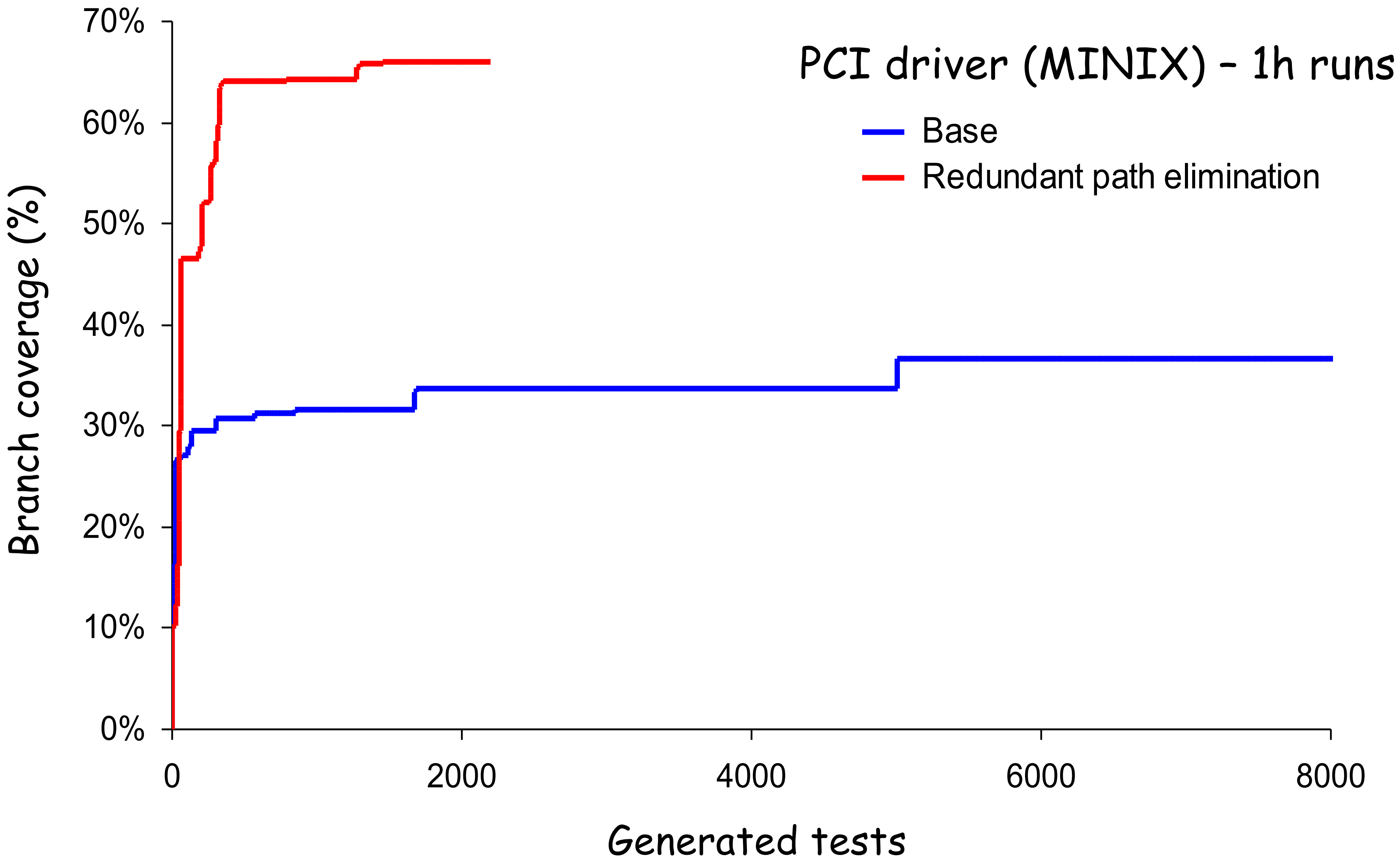
sb16



lance



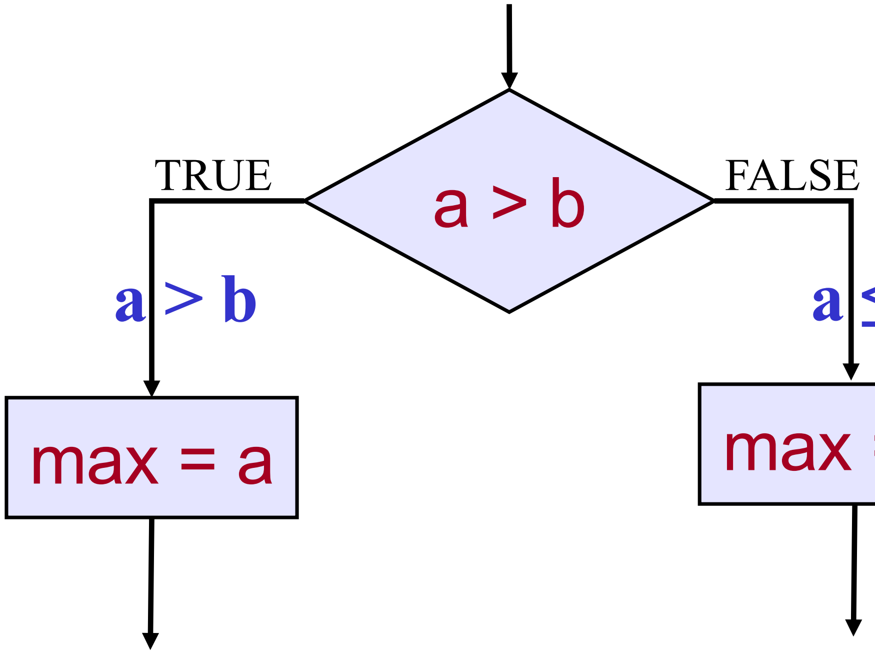
Redundant Path Elimination



Statically Merging Paths

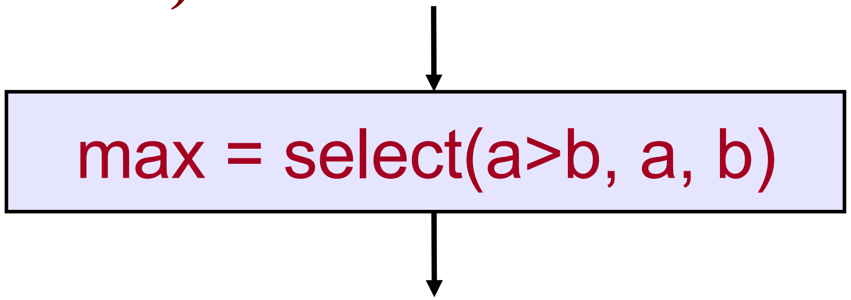
Default behaviour

```
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} → 1

Path merging ≡ Outsourcing problem to constraint solver

Using Existing Regression Suites

- Most applications come with a manually-written regression test suite

```
$ cd lighttpd-1.4.29
$ make check
...
./cachable.t .......... ok
./core-404-handler.t .. ok
./core-condition.t .... ok
./core-keepalive.t .... ok
./core-request.t ...... ok
./core-response.t ..... ok
./core-var-include.t .. ok
./core.t .............. ok
./lowercase.t ......... ok
./mod-access.t ........ ok
...
```

Regression Suites

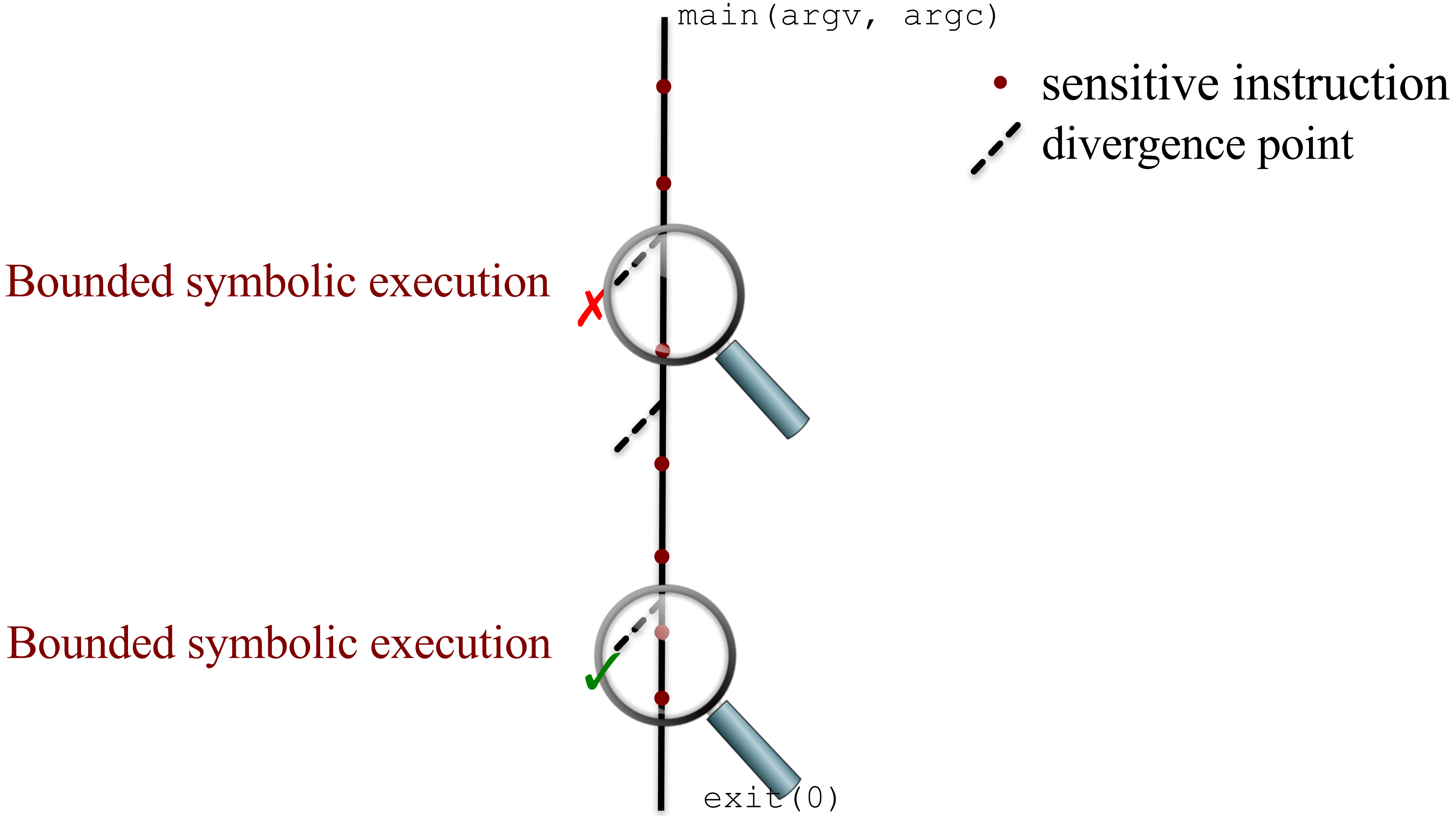
PROS	CONS
• Designed to execute interesting program paths	• Execute each path with a single set of inputs
• Often achieve good coverage of different program features	• Often exercise the general case of a program feature, missing corner cases

ZESTI: Using Existing Regression Suites

1. Use the paths executed by the regression suite to bootstrap the exploration process (to benefit from the coverage of the manual test suite and find additional errors on those paths)
2. Incrementally explore paths around the dangerous operations on these paths, in increasing distance from the dangerous operations (to test all possible corner cases of the program features exercised by the test suite)

[make test-zesti: A Symbolic Execution Solution for Improving Regression Testing , Marinescu, Cadar, ICSE 2012]

Multipath Analysis



main(argv, argc)

● sensitive instruction

divergence point

Bounded symbolic execution ✗

Bounded symbolic execution ✓

exit(0)

ZESTI and test drivers

- No need to construct a test driver
  - Existing tests are drivers!
  - Developers often do a good job choosing the right number and size of inputs

Scalability Challenges

Path exploration challenges

Constraint solving challenges

Constraint Solving Challenges

1. **Accuracy:** need 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 related to pointer/integer casting and arithmetic overflows

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

Our Constraint Solver: 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 (BVs)** and **arrays of BVs**
- We model each memory block as an array of 8-bit BVs
- We can translate all C expressions into STP constraints with bit-level accuracy
  - Main exception: floating-point

Constraint Solving: Performance

- Inherently expensive (NP-complete)
- Invoked at every branch

- Key insight: exploit the characteristics of constraints generated by symex

Some Constraint Solving Statistics
[after optimizations]

Application	Instrs/s	Queries/s	Solver %
[	695	7.9	97.8
base64	20,520	42.2	97.0
chmod	5,360	12.6	97.2
comm	222,113	305.0	88.4
csplit	19,132	63.5	98.3
dircolors	1,019,795	4,251.7	98.6
echo	52	4.5	98.8
env	13,246	26.3	97.2
factor	12,119	22.6	99.7
join	1,033,022	3,401.2	98.1
ln	2,986	24.5	97.0
mkdir	3,895	7.2	96.6
Avg:	**196,078**	**675.5**	**97.1**

1h runs using KLEE with DFS and no caching

UNIX utilities (and many other benchmarks)

- Large number of queries
- Most queries <0.1s
- Most time spent in the solver (before and after optimizations!)

[Multi-solver Support in Symbolic Execution, Palikareva and Cadar, CAV'13]

Constraint Solving Optimizations

Implemented at several different levels:

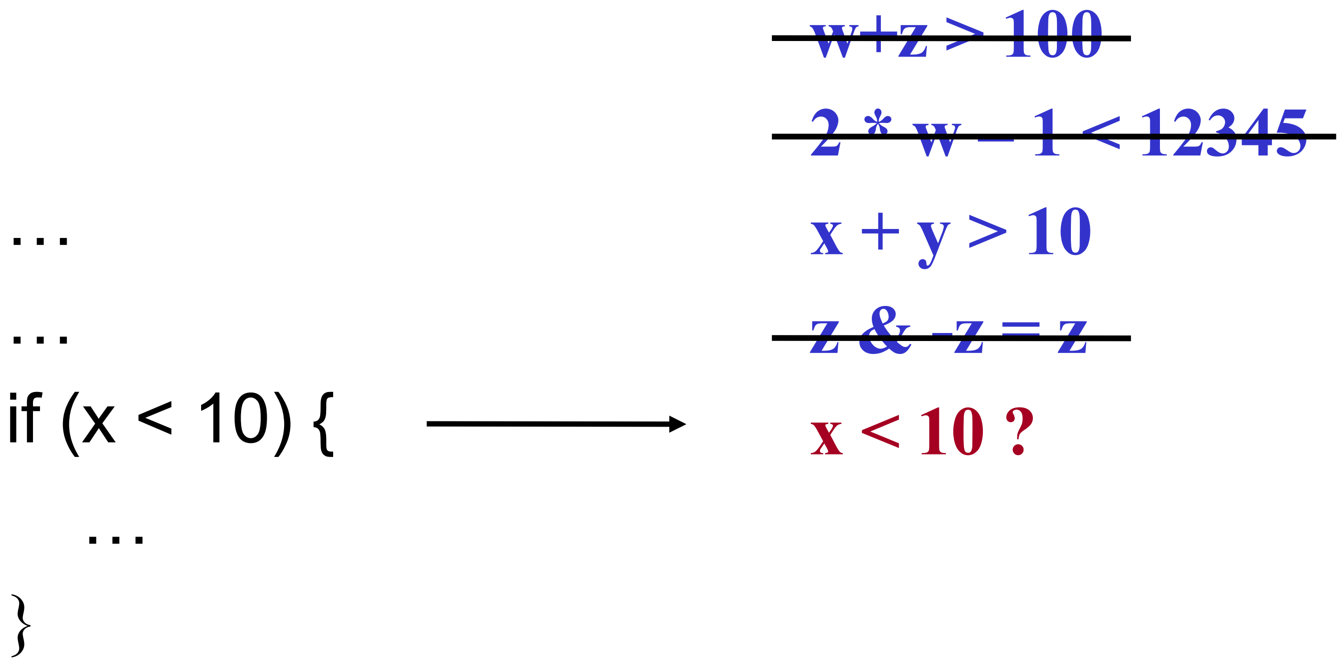
- SAT solvers
- SMT solvers
- Symbolic execution tools

Higher-Level Constraint Solving Optimizations

- Two simple and effective optimizations
  - Constraint independence optimization a.k.a. Eliminating irrelevant constraints
  - Caching constraints and solutions

Constraint Independence Optimization or Eliminating Irrelevant Constraints

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



```
…
…
if (x < 10) {
    …
}
```

~~w+z > 100~~

~~2 * w – 1 < 12345~~

x + y > 10

~~z & -z = z~~

x < 10 ?

Caching Constraints

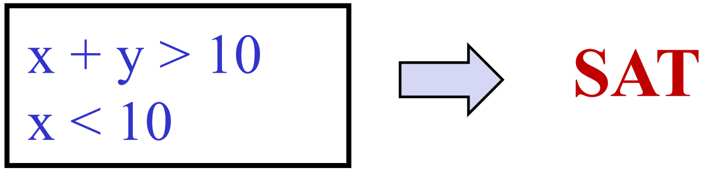
~~w+z > 100~~

~~2 * w – 1 < 12345~~

x + y > 10

~~z & -z = z~~

x < 10 ?



x + y > 10
x < 10

SAT

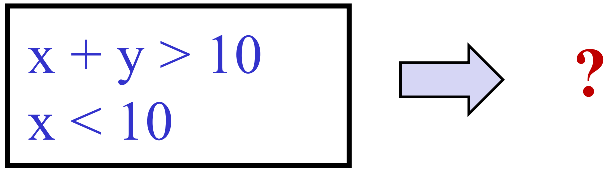
~~w+z > 100~~

~~2 * w – 1 < 12345~~

x + y > 10

~~z & -z != z~~

x < 10 ?

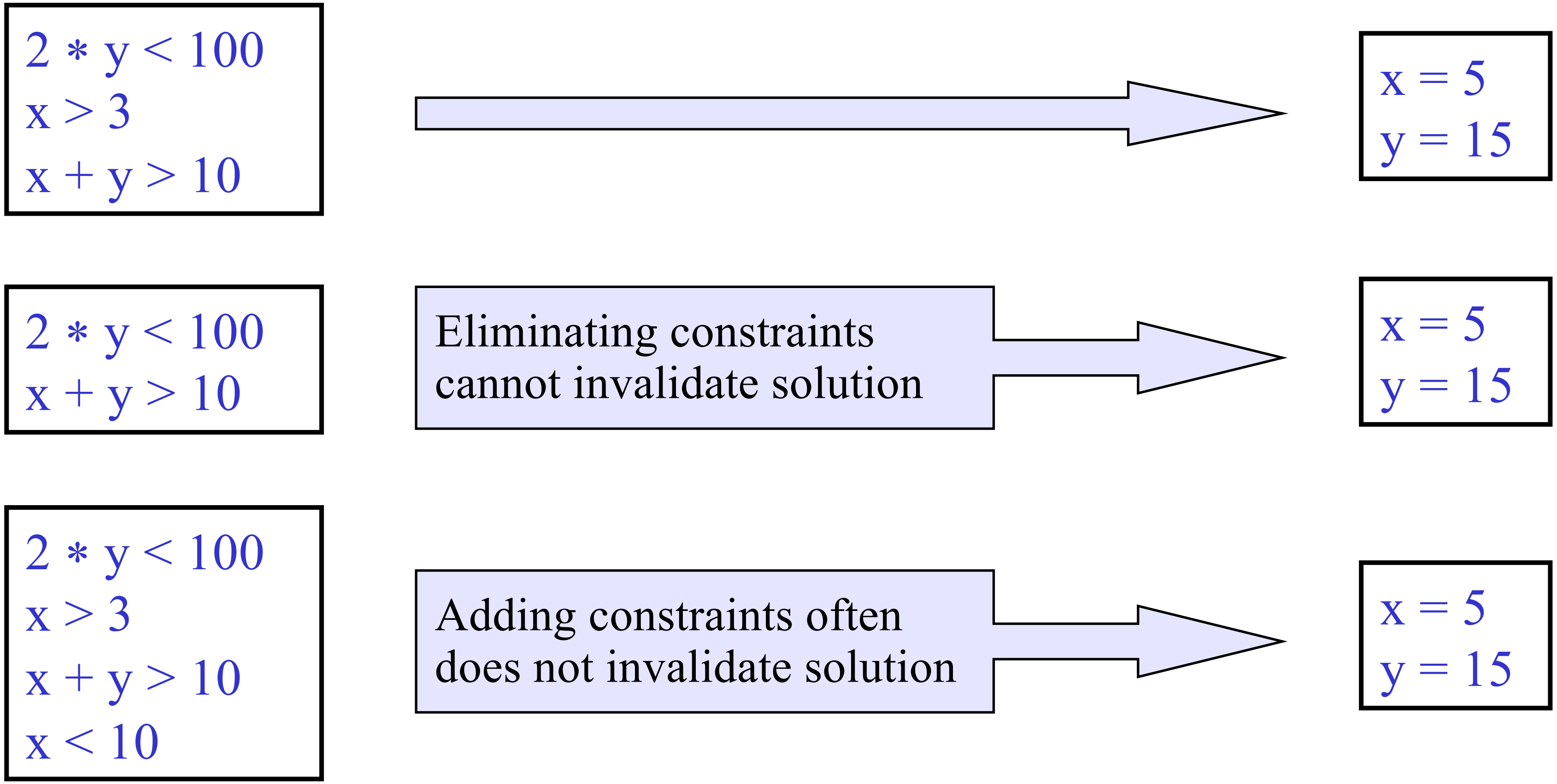


x + y > 10
x < 10

?

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

$x = 5$
$y = 15$

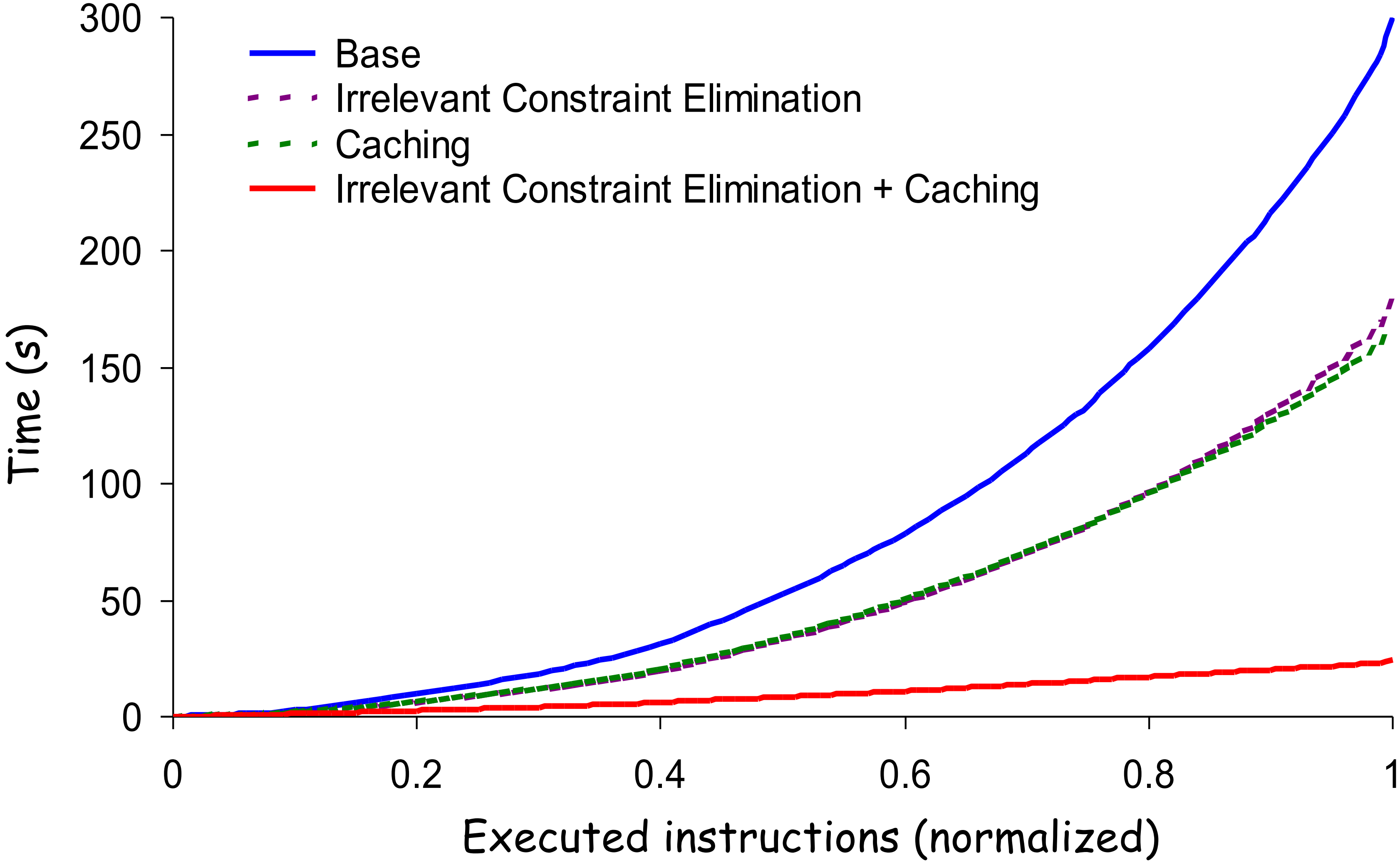
$2 * y < 100$
$x > 3$
$x + y > 10$
$x < 10$

Adding constraints often does not invalidate solution

$x = 5$
$y = 15$

Significant Speedup

Aggregated data over 73 applications



Usage Scenarios

Successfully used our tools to:

- Automatically generate high-coverage test suites
- Discover generic bugs and security vulnerabilities in complex software
- Enhance the quality of regression testing
- Perform comprehensive patch testing
- Flag potential semantic bugs via crosschecking
- Perform bounded verification of data-parallel optimizations

Bug Finding (EGT, EXE, KLEE, KATCH, etc): Focus on Systems and Security Critical Code

	Applications
Text, binary, shell and file processing tools	GNU Coreutils, findutils, binutils, diffutils, Busybox, MINIX (~500 apps)
Network servers	Bonjour, Avahi, udhcpd, lighttpd, etc.
Library code	libdwarf, libelf, PCRE, uClibc, etc.
File systems	ext2, ext3, JFS for Linux
Device drivers	pci, lance, sb16 for MINIX
Computer vision code	OpenCV (filter, remap, resize, etc.)
OpenCL code	Parboil, Bullet, OP2

- Most bugs fixed promptly

Coreutils Commands of Death

`md5sum -c t1.txt`	`pr -e t2.txt`
`mkdir -Z a b`	`tac -r t3.txt t3.txt`
`mkfifo -Z a b`	`paste -d\\ abcdefghijklmnopqrstuvwxyz`
`mknod -Z a b p`	`ptx -F\\ abcdefghijklmnopqrstuvwxyz`
`seq -f %0 1`	`ptx x t4.txt`
``printf %d ` ``	`cut -c3-5,8000000- --output-d=: file`

t1.txt:	`\t \tMD5(`	*t3.txt:*	`\n`
t2.txt:	`\b\b\b\b\b\b\b\t`	*t4.txt:*	**A**

[OSDI 2008, ICSE 2012]

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	. . .				

- **64th sector of a 64K disk image**
- **Mount it and PANIC your kernel**

[IEEE S&P 2008]

Packet of Death (Bonjour)

Offset	Hex Values							
0000	0000	0000	0000	0000	0000	0000	0000	0000
0010	003E	0000	4000	FF11	1BB2	7F00	0001	E000
0020	00FB	0000	14E9	002A	0000	0000	0000	0001
0030	0000	0000	0000	055F	6461	6170	045F	7463
0040	7005	6C6F	6361	6C00	000C	0001		

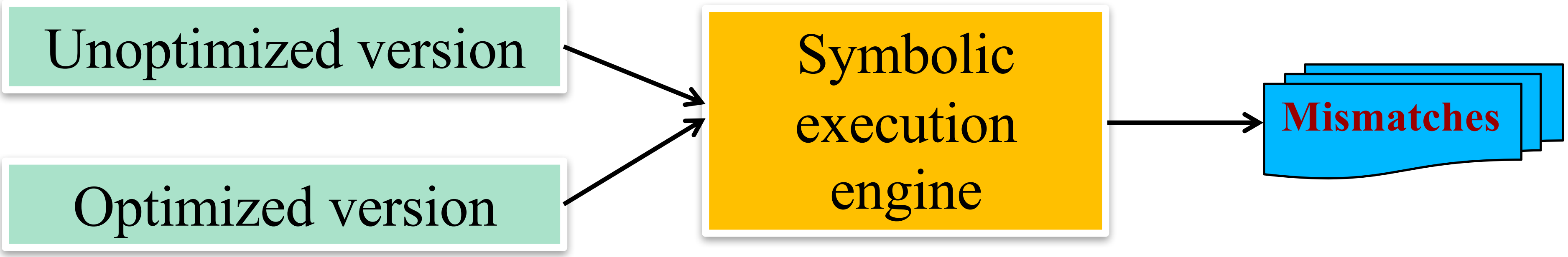
- **Causes Bonjour to abort, potential DoS attack**
- **Confirmed by Apple, security update released**

[IEEE TSE 2014]

Testing Semantics-Preserving Evolution via Crosschecking

Lots of available opportunities as code is:

Optimized frequently   Refactored frequently



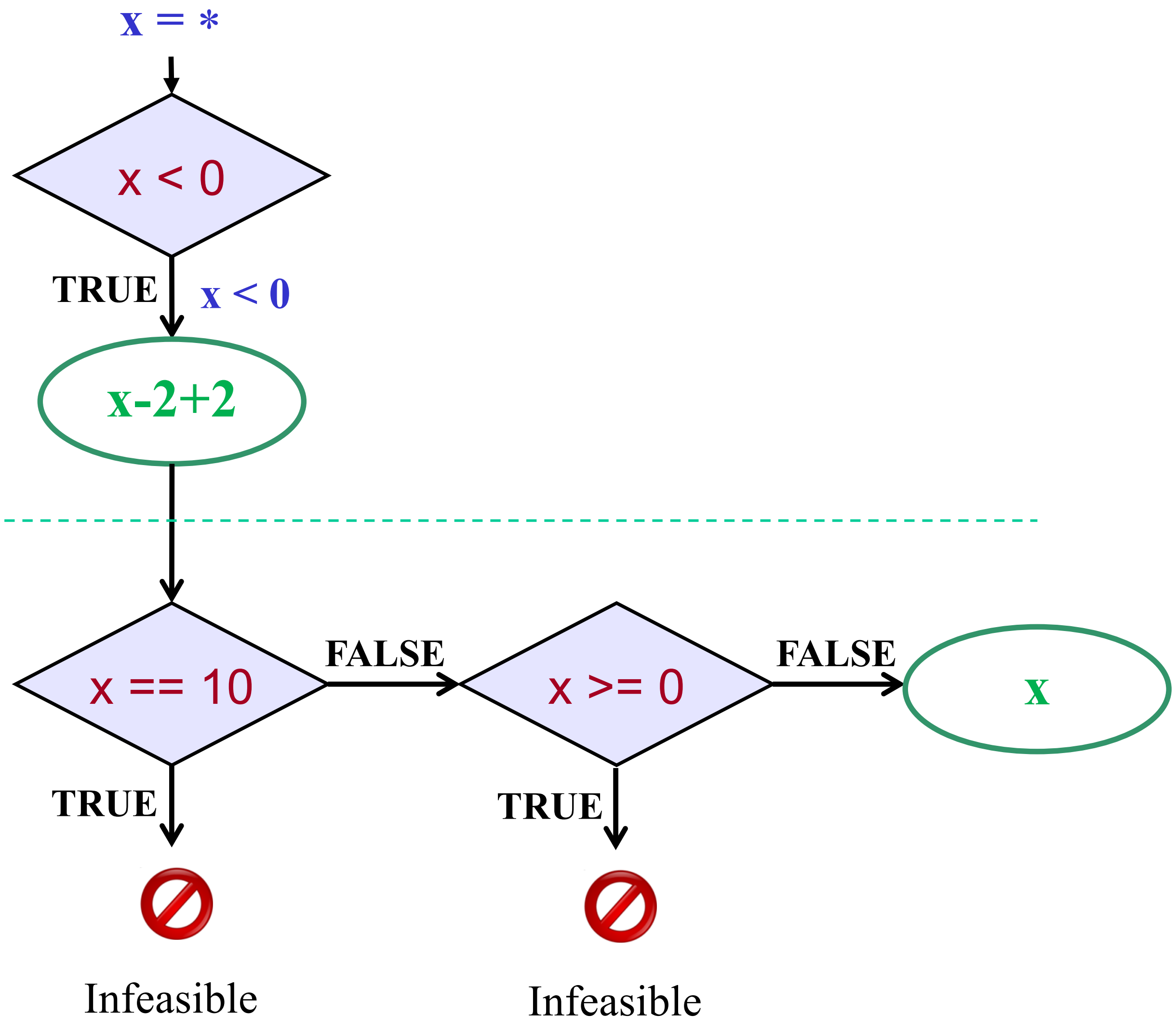
We can find any mismatches in their behavior by:

1. Using symbolic execution to explore pairs of paths
2. Comparing the (symbolic) output b/w versions

Crosschecking Two Software Versions

```
if (x < 0)
    x -= 2;
else
    if (x%2 != 0)
      x--;
return x+2;
```

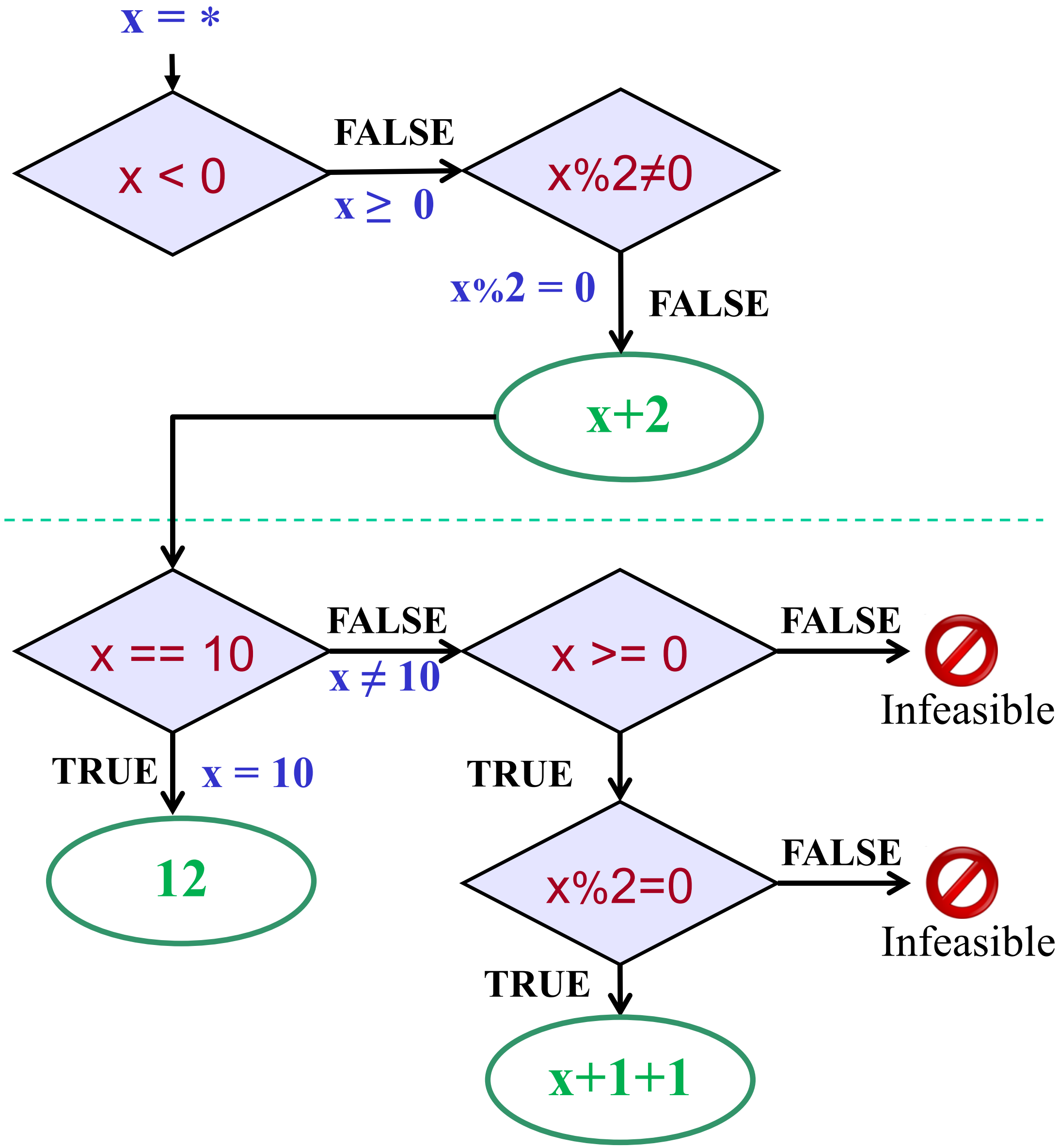
```
if (x == 10)
   return 12;

if (x >= 0) {
   if (x%2 == 0)
      x++;
   x++;
}
return x;
```



Crosschecking Two Software Versions

```
if (x < 0)
    x -= 2;
else
    if (x%2 != 0)
      x--;
return x+2;
```

```
if (x == 10)
   return 12;

if (x >= 0) {
   if (x%2 == 0)
      x++;
   x++;
}
return x;
```

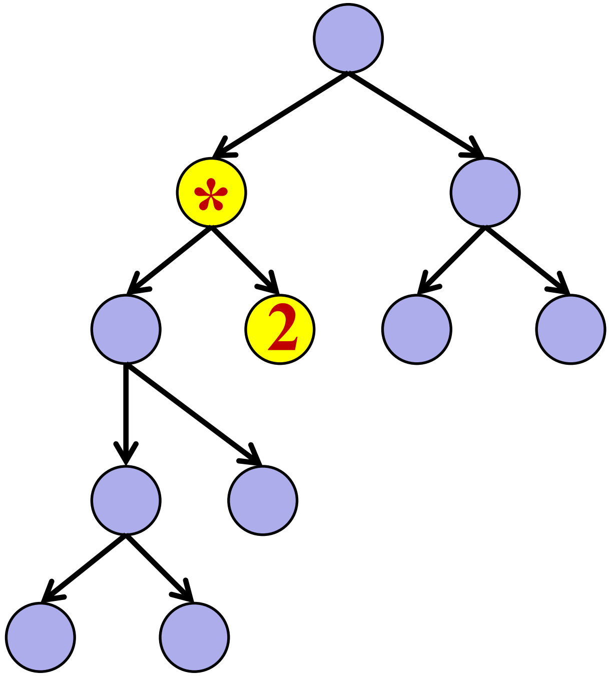


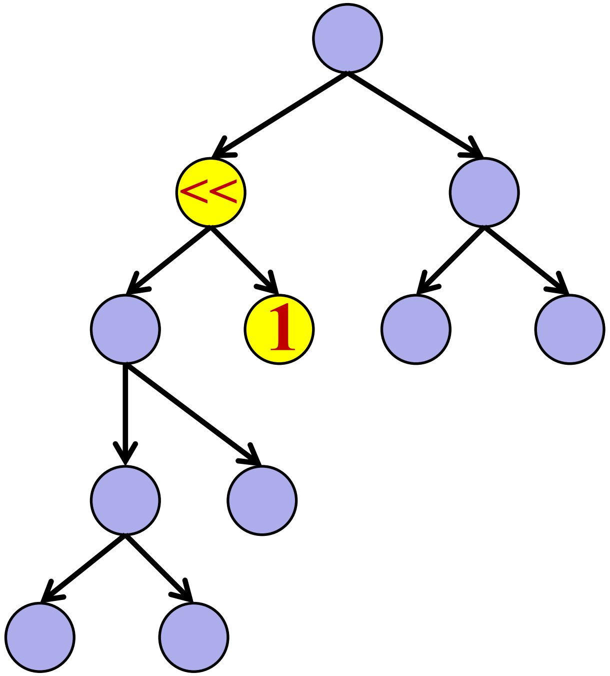
Crosschecking: Discussion

- Can find semantic errors
- No need to write (additional) specifications
- Crosschecking queries can be solved faster
- Can support constraint types not (efficiently) handled by the underlying solver, e.g., floating-point

Many crosschecking queries can be *syntactically* proved to be equivalent

Crosschecking: Advantages



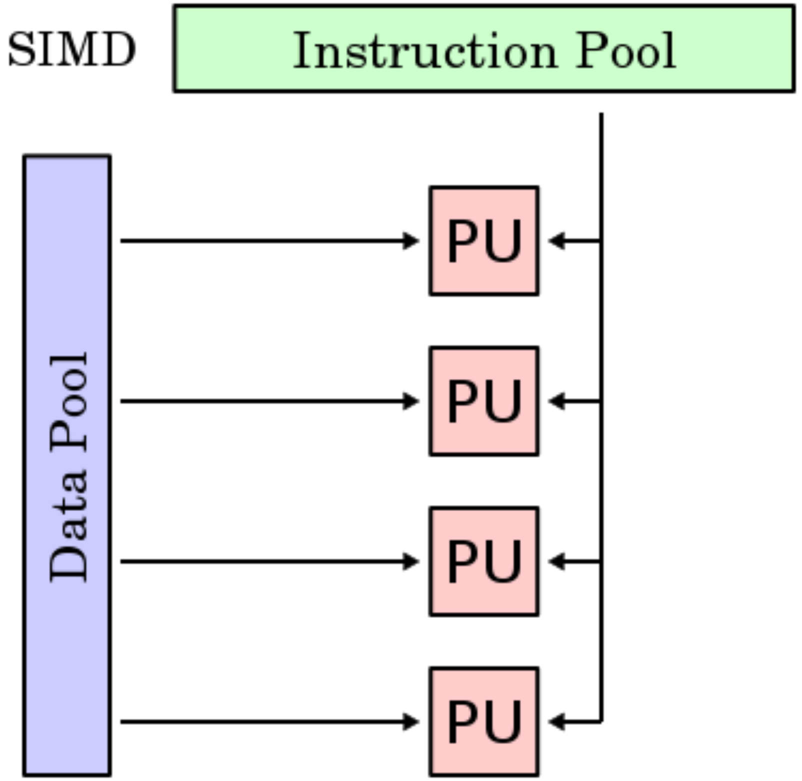


Many crosschecking queries can be *syntactically* proved to be equivalent via simple *rewrite rules*

SIMD Optimizations

Most processors offer support for SIMD instructions

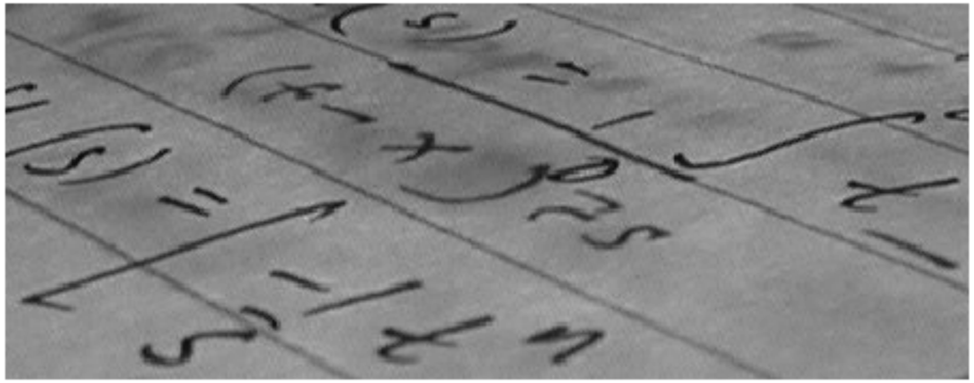
- Can operate on multiple data concurrently
- Many algorithms can make use of them (e.g., computer vision algorithms)



[EuroSys 2011]

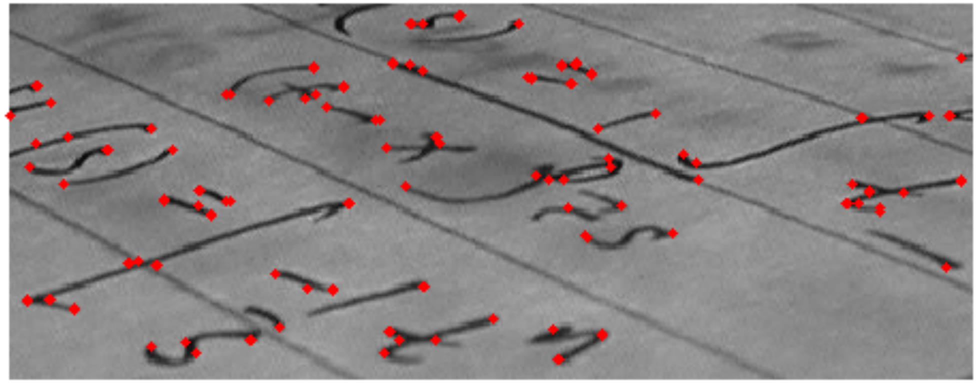
OpenCV

Popular computer vision library from Intel and Willow Garage



[Corner detection algorithm]

Computer vision algorithms were optimized to make use of SIMD



OpenCV Results

- Crosschecked 51 SIMD-optimized versions against their reference scalar implementations
  - Found mismatches in 10
- Most mismatches due to tricky FP-related issues:
  - Precision
  - Rounding
  - Associativity
  - Distributivity
  - NaN values

[EuroSys 2011]

OpenCV Results

- Crosschecked 51 SIMD-optimized versions against their reference scalar implementations
  - Verified the correctness of 41 of them up to a certain image size (*bounded verification*)
- Key idea:
  - Tame path explosion by statically merging paths

[EuroSys 2011]

OpenCV Results

Surprising find: 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

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

Symbolic Execution Summary

- Automatic, does not require test cases
- Highly systematic
  - reaches deep code paths
  - achieves high statement/branch coverage
  - can reason about all possible values on a path
- Finds deep bugs
  - including those depending on specific values and/or memory layout
  - including functional bugs (see crosschecking study)
  - generates inputs that hit the bugs found
- Scalability challenges
  - Path exploration
  - Constraint solving

KLEE: Freely Available as Open-Source
Demo later in the course

KLEE: http://klee.github.io

- Flexible symbolic execution tool based on the LLVM framework and the STP solver, primarily for C code
- Over 300 subscribers to the klee-dev mailing list
- Used/tried out by a large number of academic and industrial users
- Extended in many interesting ways by several research groups, in the areas of wireless sensor networks, automated debugging, schedule memoization in multithreaded code, exploit generation, online gaming, etc.

Want to get involved? Let me know!