Improving Software Reliability and Security via Symbolic Execution

Cristian Cadar

Department of Computing

Imperial College London

Joint work with Paul Marinescu, Peter Collingbourne, Paul Kelly, JaeSeung Song, Peter Pietzuch, Hristina Palikareva (Imperial)

Dawson Engler, Daniel Dunbar, Junfeng Yang, Peter Pawlowski, Can Sar, Paul Twohey, Vijay Ganesh, David Dill, Peter Boonstoppel (Stanford)

Imperial College London

UPMARC Summer School
Upsalla, 10-11 June 2013

Writing Correct Software is Hard!

- Software complexity
  - Massive amounts of code
  - Tricky control flow
  - Complex dependencies
  - Intensive interaction w/ environment
- Code has to anticipate all possible interactions
  - Including malicious ones!

Dynamic Symbolic Execution

Provides an automated way to reason about program behavior and the interaction with users and environment

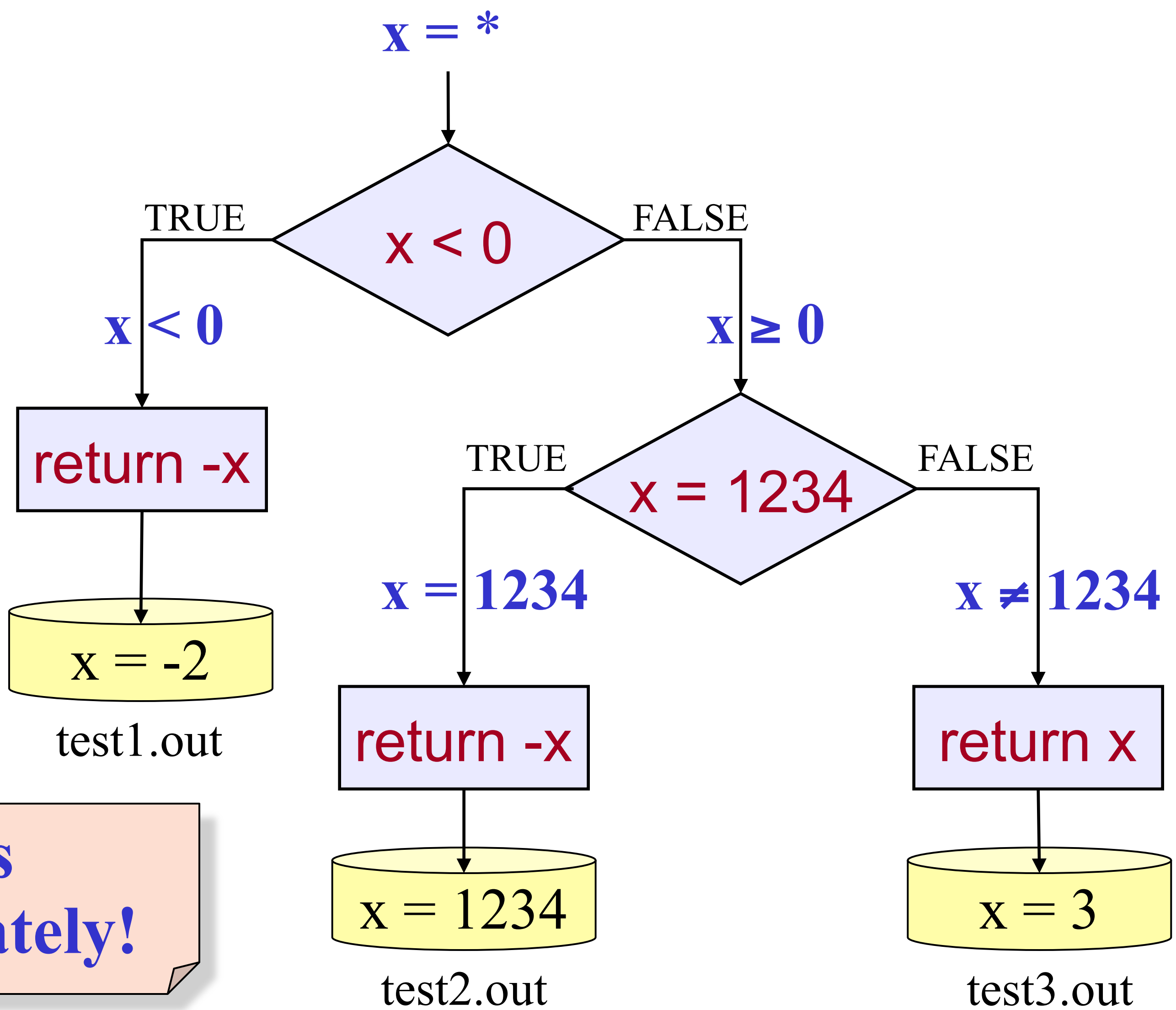
- Dynamic symbolic execution can ***automatically explore multiple paths*** through a program
  - Using a constraint solver to determine the feasibility of each explored path
- Before each dangerous operation, can check if there are ***any*** values that can cause an error
- For each path, can usually generate a ***concrete input triggering the path***

Dynamic Symbolic Execution

- **Toy example**
- **Scalability challenges**
  - Path explosion, constraint solving challenges
- **Generic bug finding**
  - User-level utilities, kernel code, drivers, computer vision code, etc.
  - Attack generation against file systems and network servers
- **Symbolic race detection for GPGPU code**
- **Semantic errors via crosschecking**
  - Server interoperability, SIMD optimizations, GPU optimizations
- **Patch testing**
  - Testing six years of patches

Toy Example

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



Each path is explored separately!

All-Value Checks

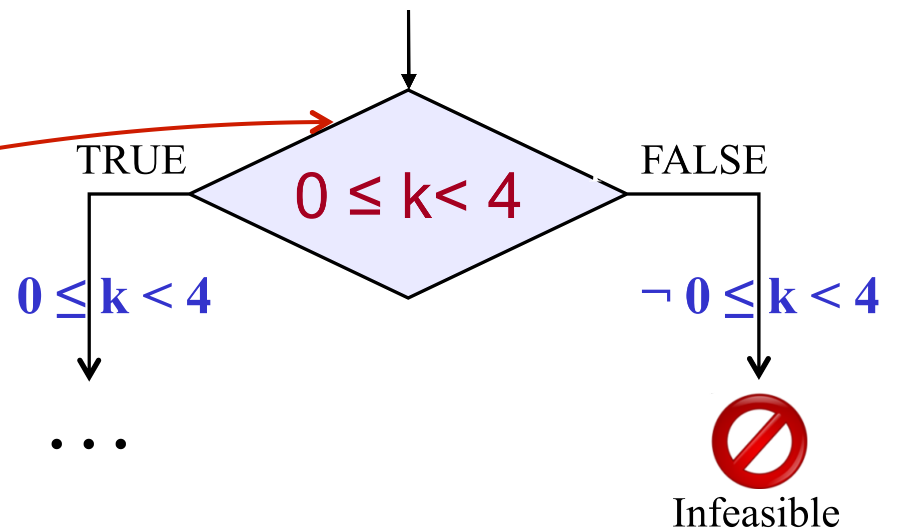
Implicit checks before each dangerous operation

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

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

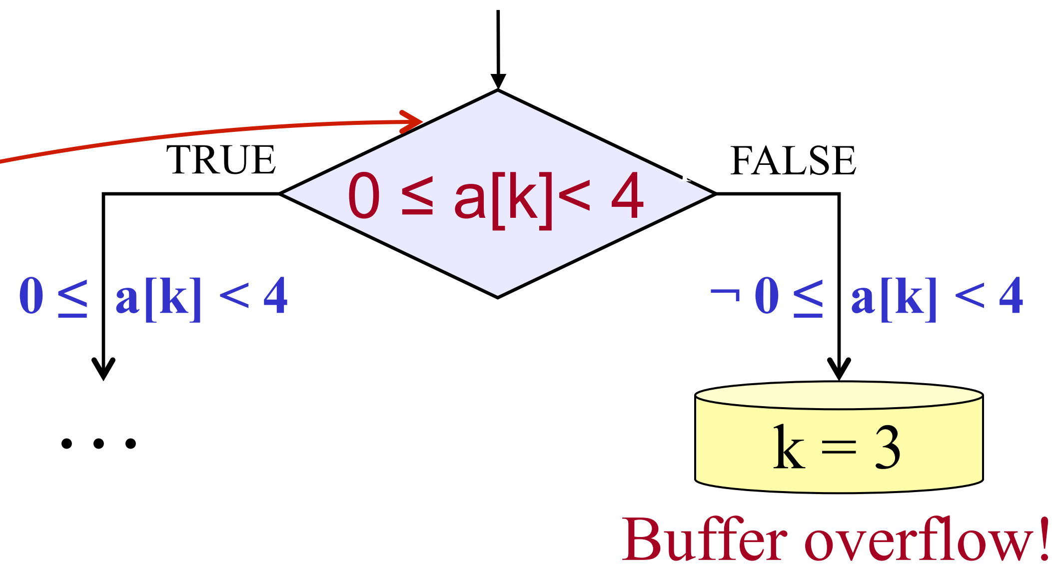
Implicit checks before each dangerous operation

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

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]];
}
```



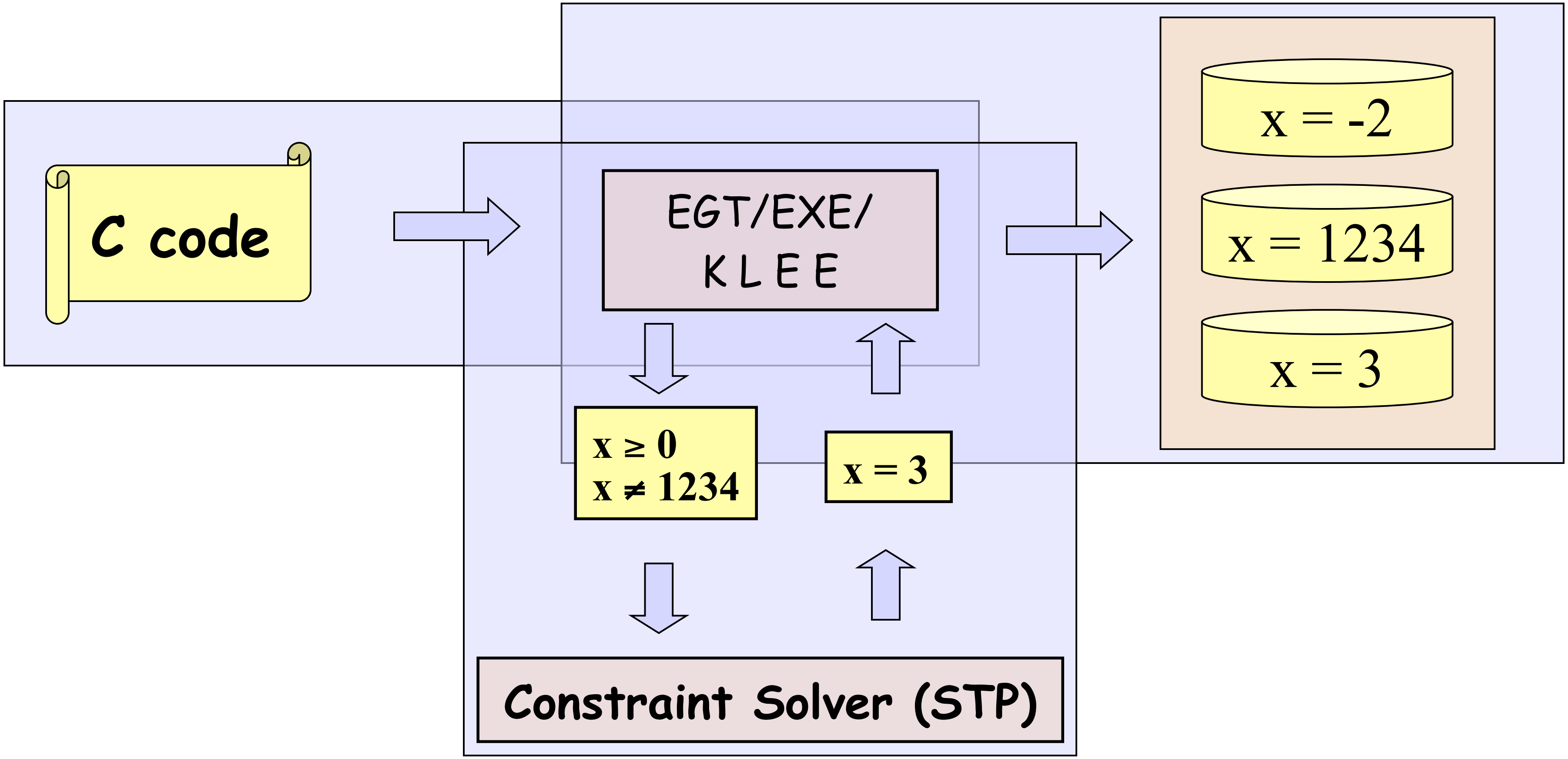
Mixed Concrete/Symbolic Execution

All operations that do not depend on the symbolic inputs are (essentially) executed as in the original code

Advantages:

- Ability to interact with the outside environment
  - E.g., system calls, uninstrumented libraries
- Can partly deal with limitations of constraint solvers
  - E.g., unsupported theories
- Only relevant code executed symbolically
  - Without the need to extract it explicitly

Three tools: EGT, EXE, KLEE



Scalability Challenges

Path exploration challenges

Constraint solving challenges

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

- Coverage-optimized search
  - Select path closest to an uncovered instruction
  - Favor paths that recently hit new code
- Best-first search
- Random path search
- etc.

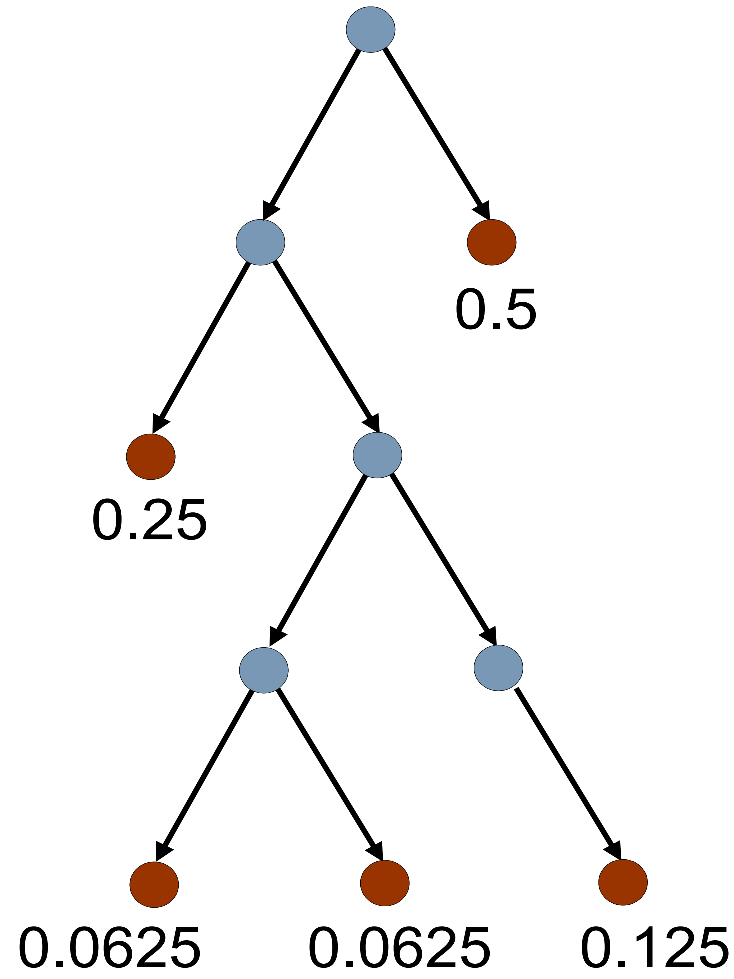
[CCS'06, OSDI'08, ICSE'11, etc.]

Random Path Selection

- Maintain a binary tree of active paths
- Subtrees have equal prob. of being selected, irresp. of size

---

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



Which Search Heuristic?

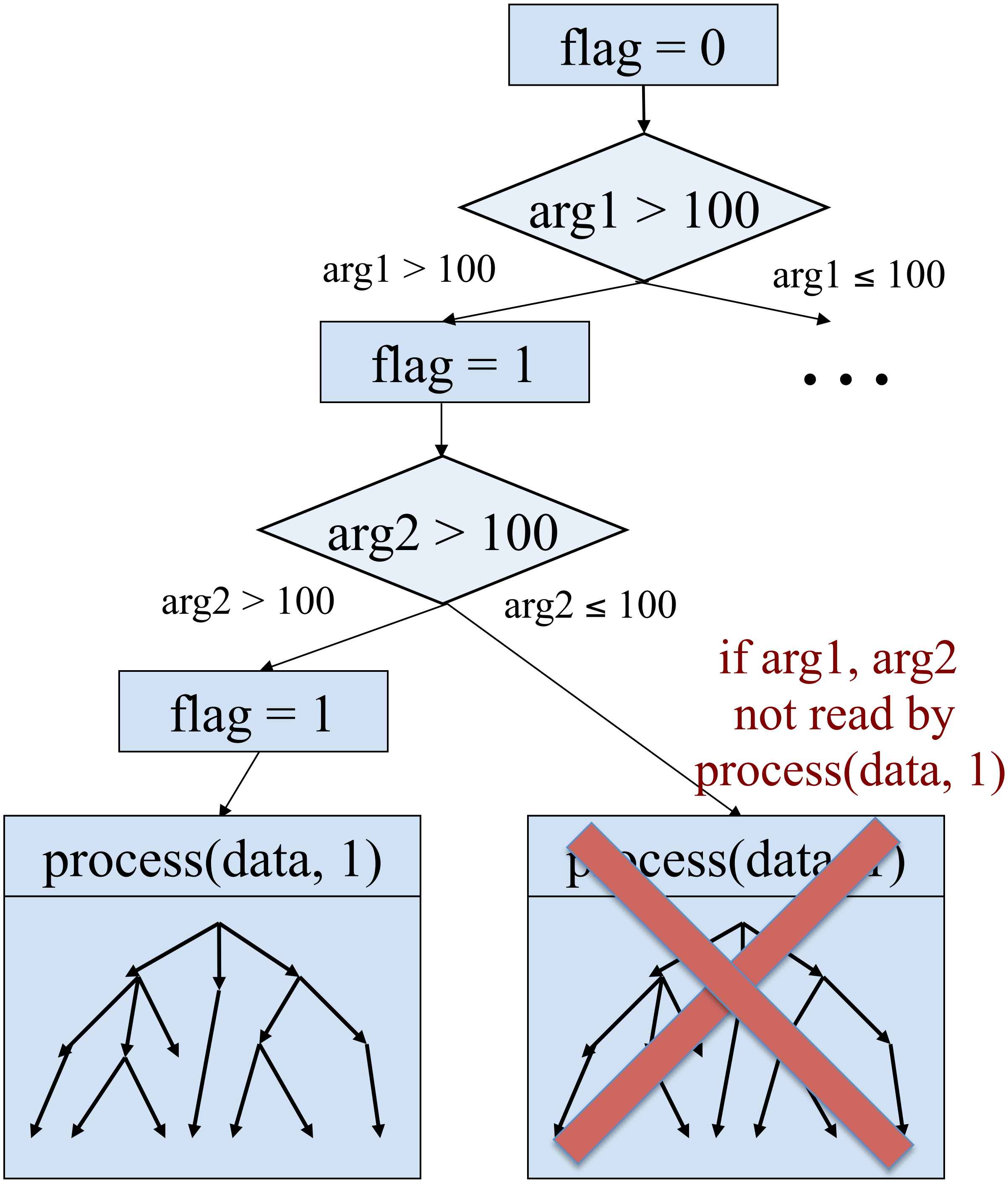
Our latest tool KLEE uses multiple heuristics in a round-robin fashion, to protect against individual heuristics getting stuck in a local maximum.

Eliminating Redundant Paths

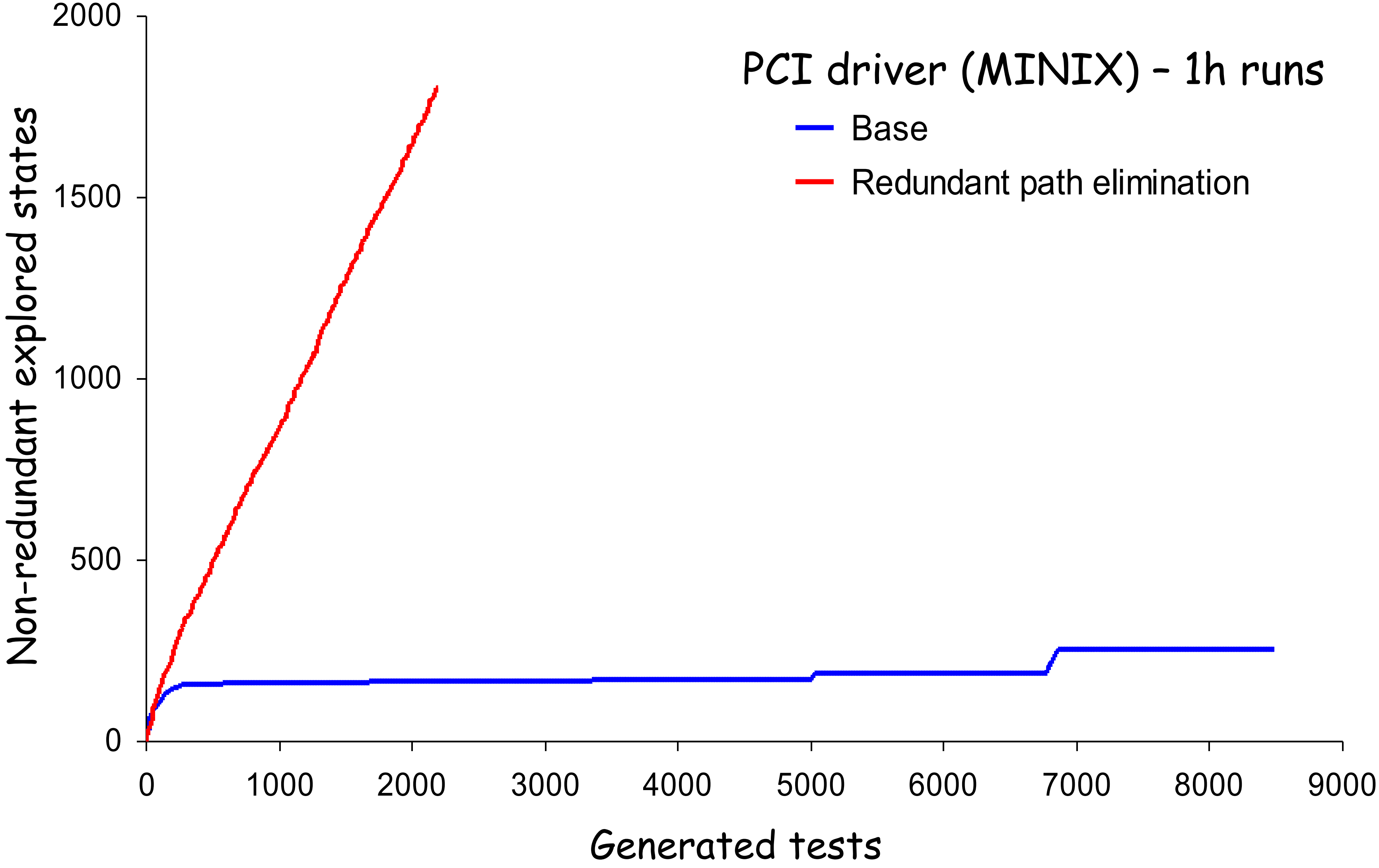
- If two paths reach the same program point with the same constraint sets, we can prune one of them
- We can discard from the constraint sets of each path those constraints involving memory which is never read again

[TACAS'08]

```
data, arg1, arg2 = *

flag =  0;

if (arg1 > 100)
    flag = 1;

if (arg2 > 100)
    flag = 1;

process(data, flag);
```

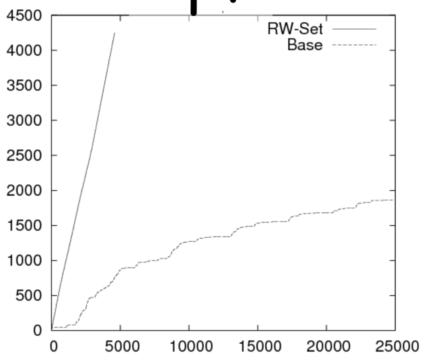


Many Redundant Paths

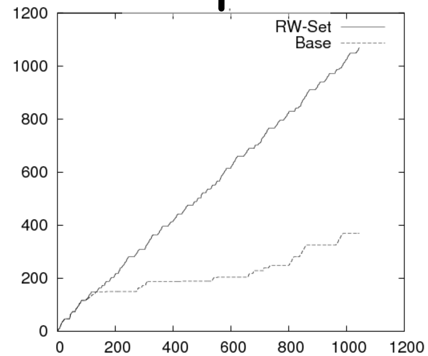


Lots of Redundant Paths

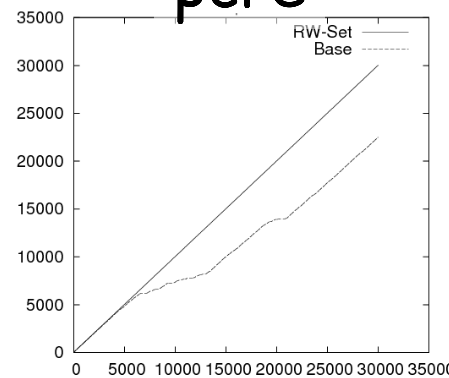
bpf



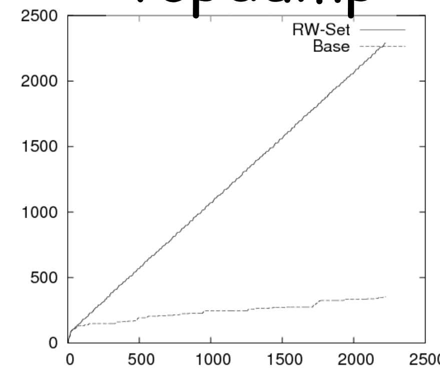
expat



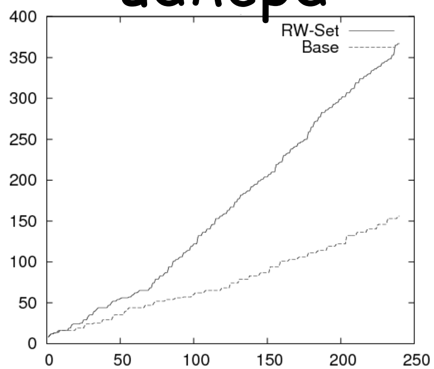
pcre



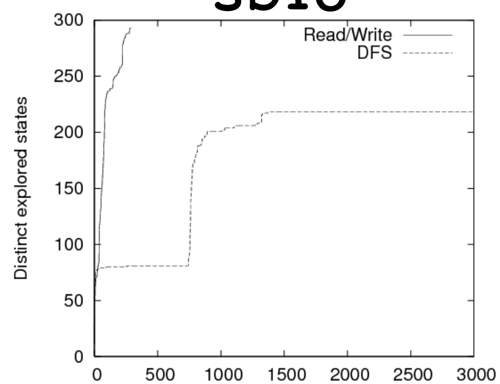
tcpdump



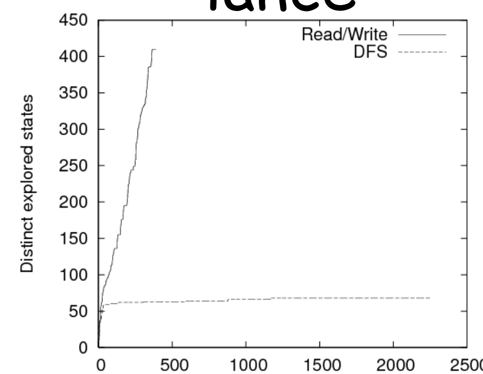
udhcpd



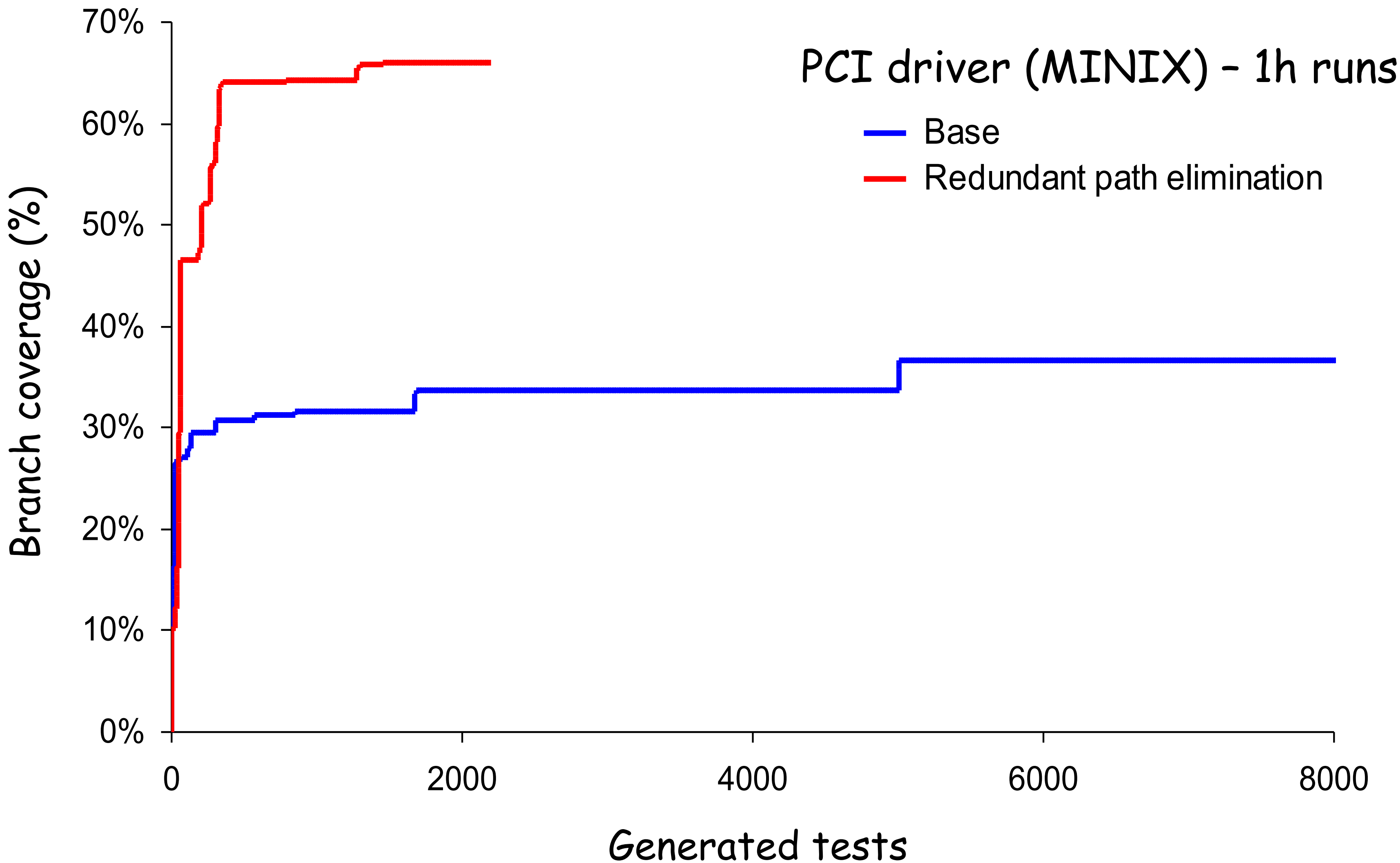
sb16



lance



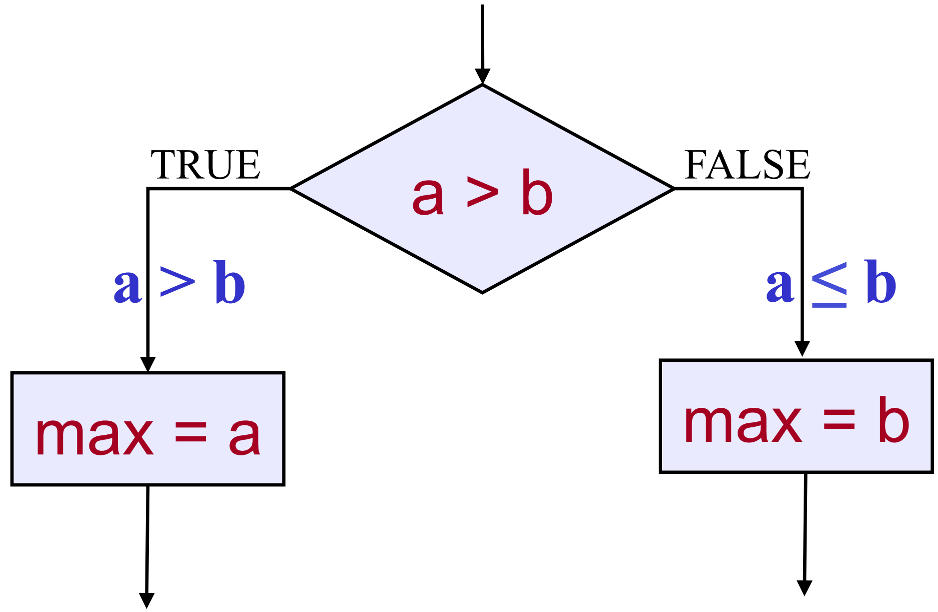
Redundant Path Elimination



Statically Merging Paths

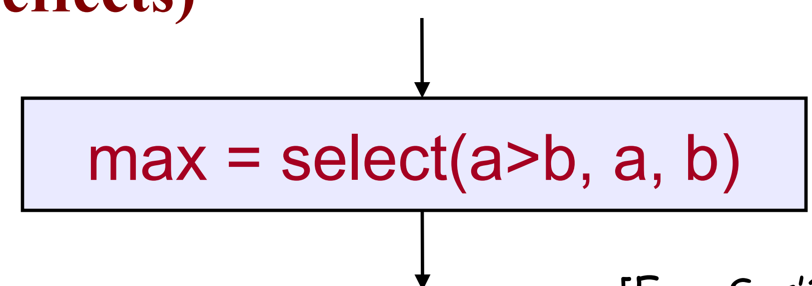
Default behaviour

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



Phi-Node Folding (when no side effects)

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



[EuroSys'11]

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 $\equiv$ Outsourcing problem to constraint solver

(which are often optimized for conjunctions of constraints)

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
...
```

[ICSE'11]

Regression Suites

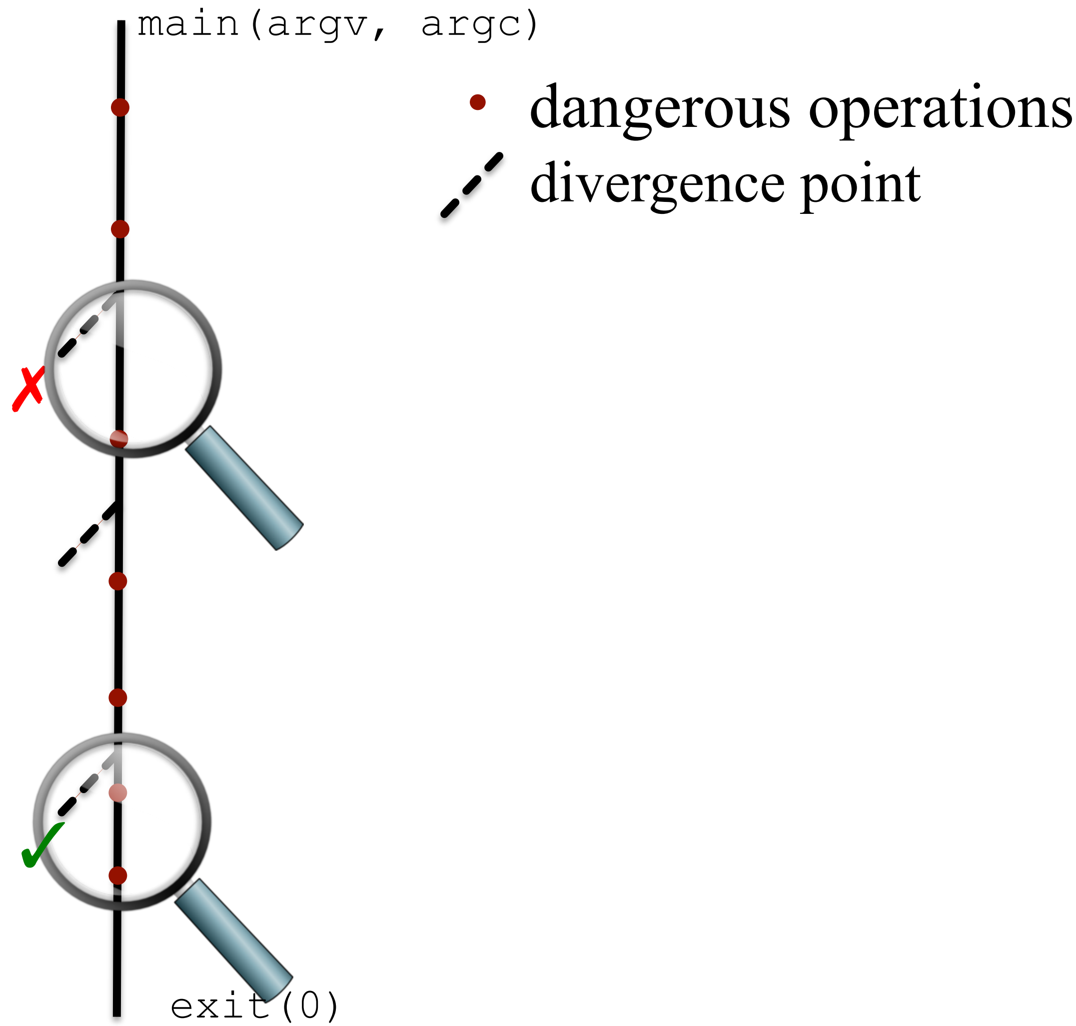
PROS	CONS
• Designed to execute interesting program paths • Often achieve good coverage of different program features	• Execute each path with a single set of inputs • Often exercise the general case of a program feature, missing corner cases

ZESTI: SymEx+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)

[ICSE'11]

Multipath Analysis



Bounded symbolic execution

Bounded symbolic execution

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

STP Constraint Solver [Ganesh, Dill]

- 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: Accuracy

- Mirror the (lack of) type system in C
  - Model each memory block as an array of 8-bit BVs
  - Bind types to expressions, not bits

```
char buf[N]; // symbolic
struct pkt1 { char x, y, v, w; int z; } *pa = (struct pkt1*) buf;
struct pkt2 { unsigned i, j; } *pb = (struct pkt2*) buf;
if (pa[2].v < 0) { assert(pb[2].i >= 1<<23); }
```

```
buf: ARRAY BITVECTOR(32)OF BITVECTOR(8)
```

```
SBVLT(buf[18], 0x00)
```

```
BVGE(buf[19]@buf[18]@buf[17]@buf[16], 0x00800000)
```

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 utilites (and many other benchmarks)

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

[CAV'13]

Constraint Solving Optimizations

Implemented at several different levels:

- SAT solvers
- SMT solvers
- Symbolic execution tools

Reasoning about Arrays in STP

- Many programs generate large constraints involving arrays with symbolic indexes
- STP handles this via **array-based refinement**

[CCS'06]

Reasoning about Arrays in STP

STP's conversion of array terms to SAT is expensive

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

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

Expands each formula by n·(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

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

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

$$\cancel{(i_1 = i_2 \Longrightarrow v_1 = v_2) \wedge (i_1 = i_3 \Longrightarrow v_1 = v_3) \wedge (i_2 = i_3 \Longrightarrow v_2 = v_3)}$$

Simplified formula UNSATISFIABLE $\Rightarrow$ Original formula UNSATISFIABLE

Array-based Refinement in STP

STP's conversion of array terms to SAT is expensive

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

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

$$\cancel{(i_1 = i_2 \Longrightarrow v_1 = v_2) \land (i_1 = i_3 \Longrightarrow v_1 = v_3) \land (i_2 = i_3 \Longrightarrow v_2 = v_3)}$$

$$\boxed{\begin{array}{c} i_1 = 1 \\ i_2 = 2 \\ i_3 = 3 \\ v_1 = e_1 = 1 \\ v_2 = e_2 = 2 \\ v_3 = e_3 = 3 \end{array}}$$

$$\boxed{\begin{array}{l} (a[1] = 1) \land (a[2] = 2) \land \\ (a[3] = 3) \land (1+2+3 = 6) \end{array}}$$

Array-based Refinement in STP

STP's conversion of array terms to SAT is expensive

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

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

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

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

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

Array-based Refinement in STP

STP's conversion of array terms to SAT is expensive

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

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

$$\boxed{\begin{array}{c} i_1 = 2 \\ i_2 = 2 \\ i_3 = 2 \\ v_1 = e_1 = 1 \\ v_2 = e_2 = 2 \\ v_3 = e_3 = 3 \end{array}} \Longrightarrow \boxed{\begin{array}{c} (a[2] = 1) \land (a[2] = 2) \land \\ (a[2] = 3) \land (2+2+2 = 6) \end{array}}$$

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

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 ?

[CCS'06]

Caching Solutions

- Static set of branches: lots of similar constraint sets

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

$\longrightarrow$

$x = 5$
$y = 15$

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

Eliminating constraints cannot invalidate solution $\longrightarrow$

$x = 5$
$y = 15$

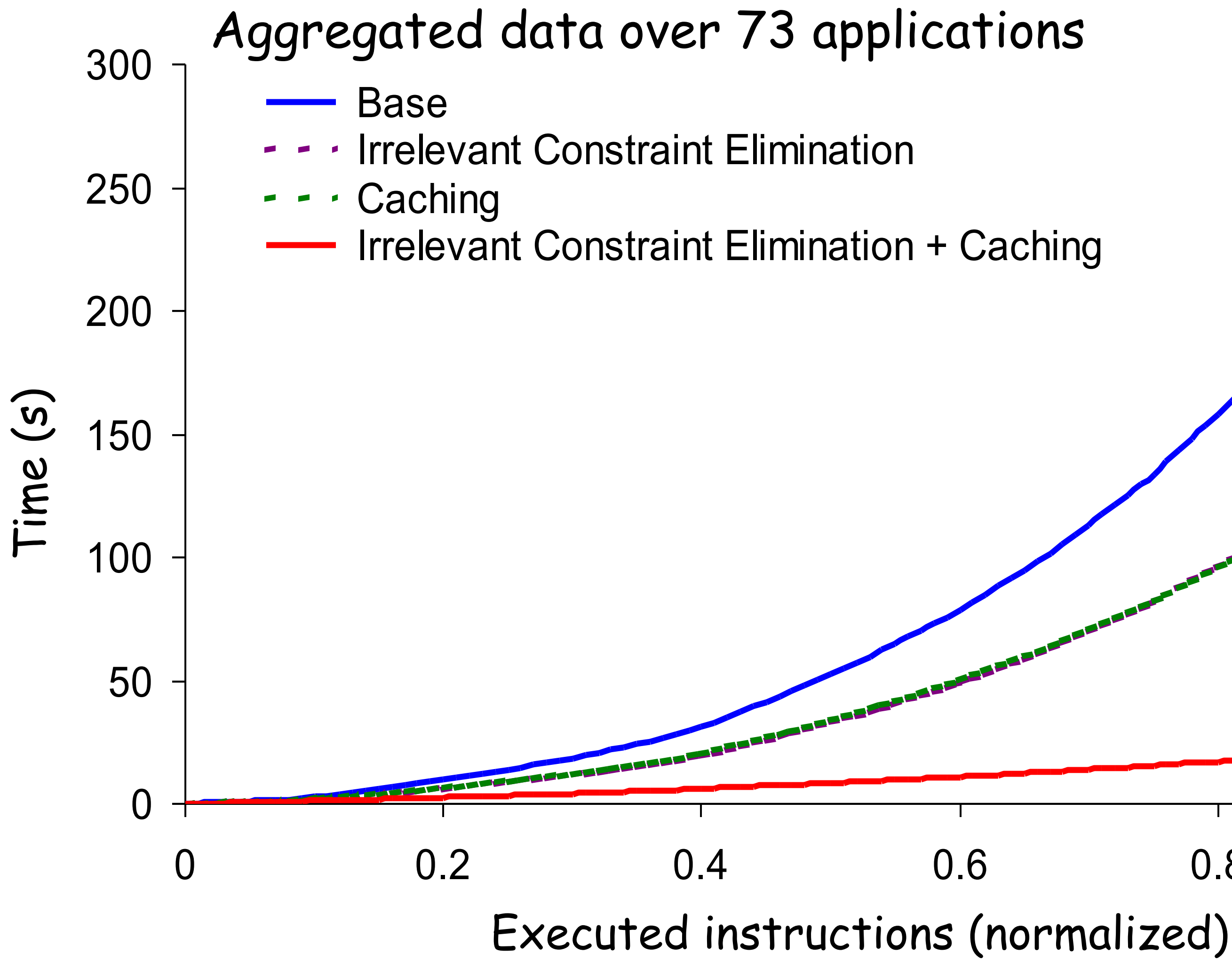
$2 * y < 100$
$x > 3$
$x + y > 10$
$x < 10$

Adding constraints often does not invalidate solution $\longrightarrow$

$x = 5$
$y = 15$

[OSDI'08]

Speedup



Aggregated data over 73 applications
300
250
200
150
100
50
0
Time (s)
Base
Irrelevant Constraint Elimination
Caching
Irrelevant Constraint Elimination + Caching
0
0.2
0.4
0.6
0.8
1
Executed instructions (normalized)

More on Caching: Instrs/Sec

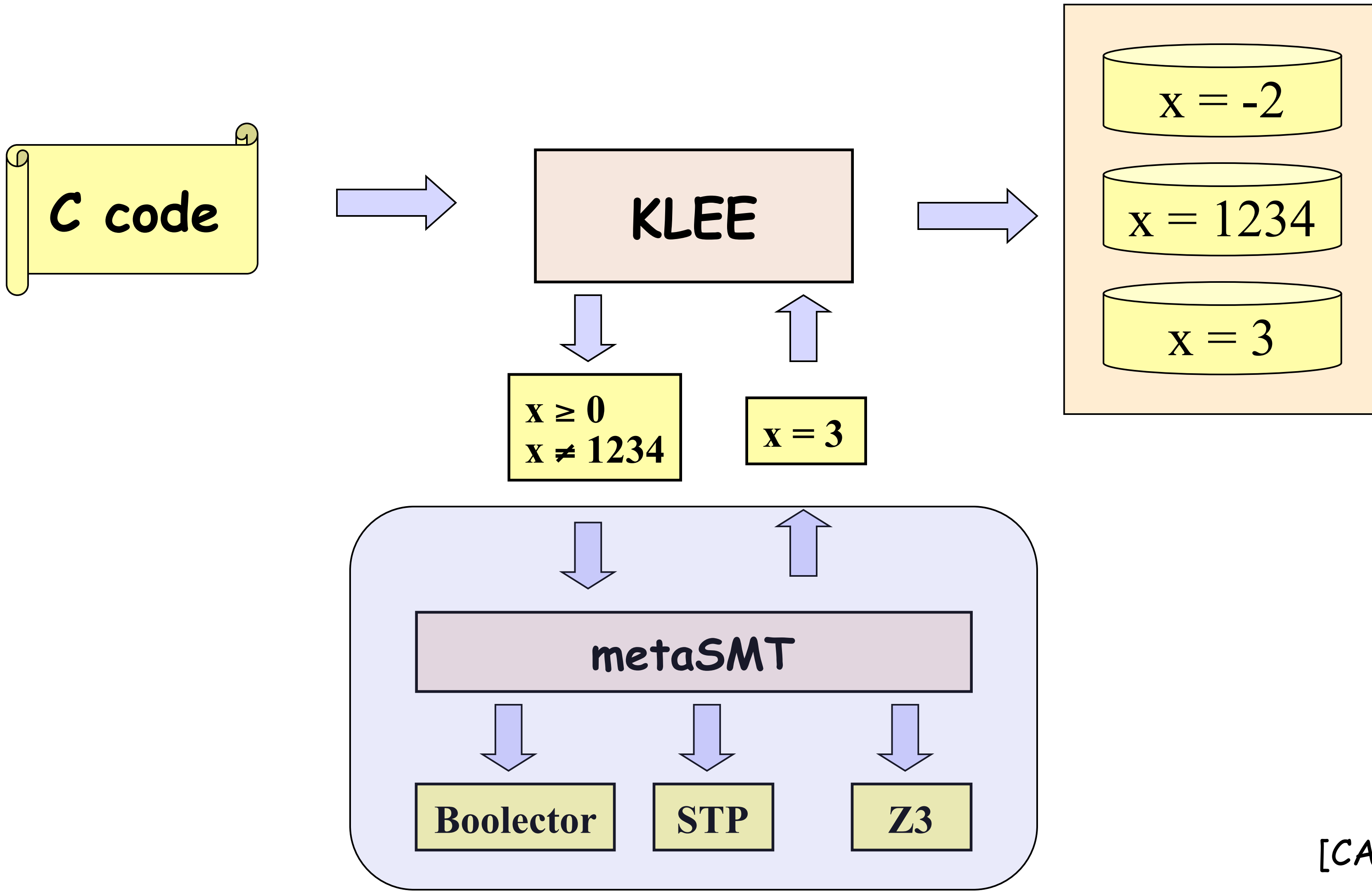
Application	No caching	Caching	Speedup
[	3,914	695	0.17
base64	18,840	20,520	1.08
chmod	12,060	5,360	0.44
comm	73,064	222,113	3.03
csplit	10,682	19,132	1.79
dircolors	8,090	1,019,795	126.05
echo	227	52	0.22
env	21,995	13,246	0.60
factor	1,897	12,119	6.38
join	12,649	1,033,022	81.66
ln	13,420	2,986	0.22
mkdir	25,331	3,895	0.15
Avg:	**16,847**	**196,078**	**11.63x**

- Instrs/sec on ~1h runs, using DFS, w/ and w/o caching

Need for better, more adaptive caching algorithms!

[CAV'13]

Portfolio of SMT Solvers



[CAV'13]

EGT, EXE, KLEE

Successfully used our tools to:

- Automatically generate high-coverage test suites
- Find bugs and security vulnerabilities in complex software
- Perform bounded verification

Bug Finding with EGT, EXE, KLEE:
Focus on Systems and Security Critical Code

	Applications
UNIX utilities	Coreutils, Busybox, Minix (over 450 apps)
UNIX file systems	ext2, ext3, JFS
Network servers	Bonjour, Avahi, udhcpd, lighttpd
Library code	libdwarf, libelf, PCRE, uClibc, Pintos
Packet filters	FreeBSD BPF, Linux BPF
MINIX device drivers	pci, lance, sb16
Kernel code	HiStar kernel
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]

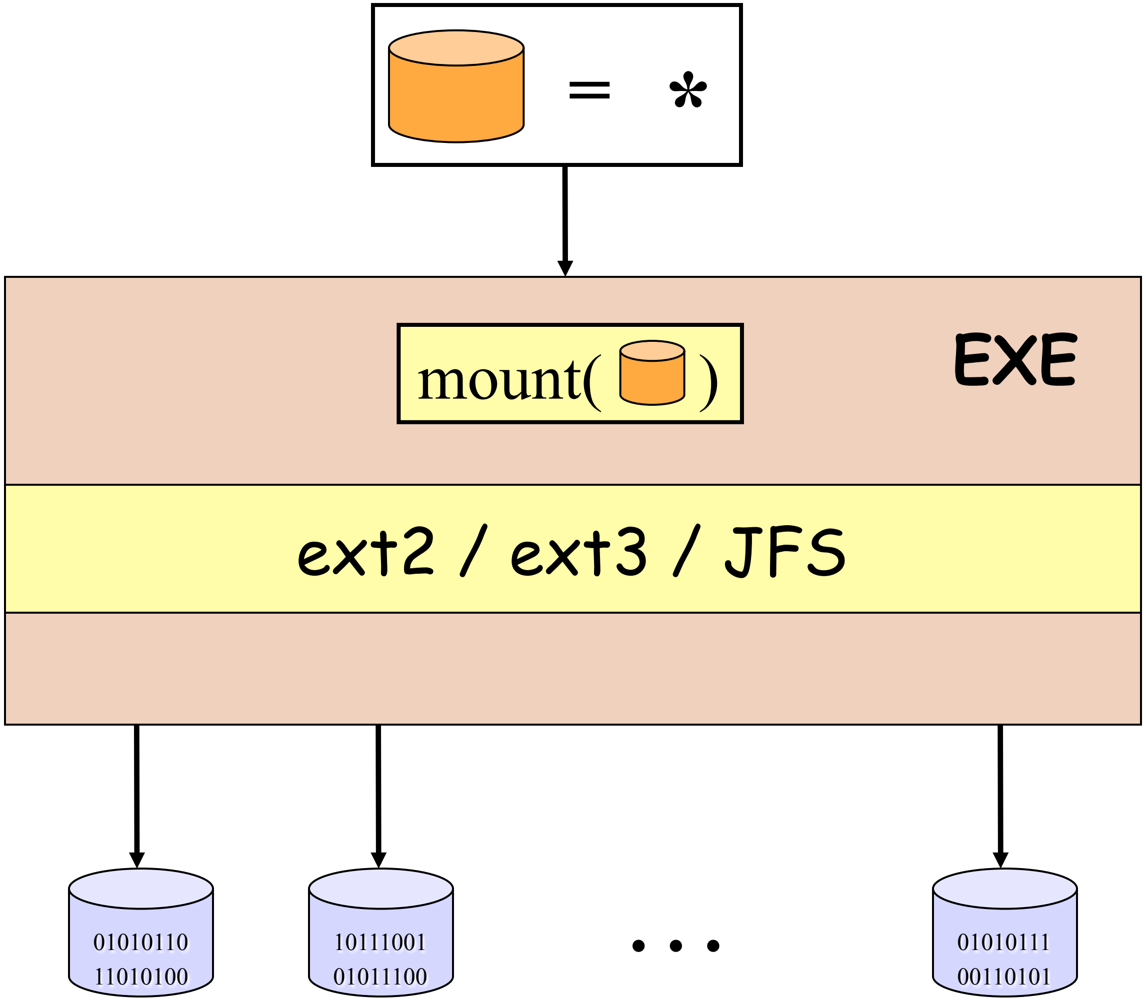
Attack Generation: File Systems

Some modern operating systems allow untrusted users to mount regular files as disk images!

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



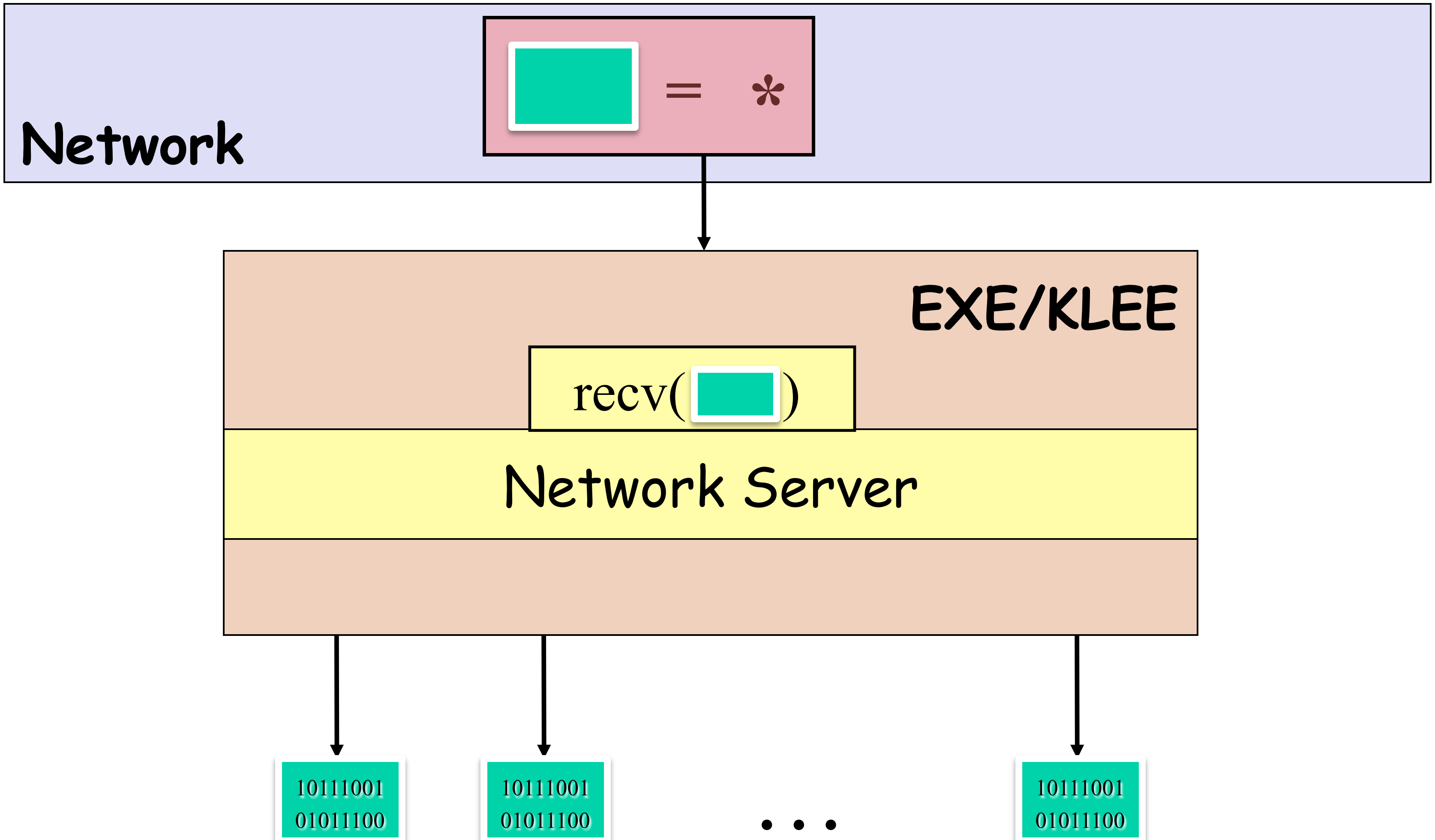
[Oakland 2006]

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	. . .				

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

Attack Generation: Network Servers



[CCS 2006, ICCCN 2011]

Bonjour: Packet of Death

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		

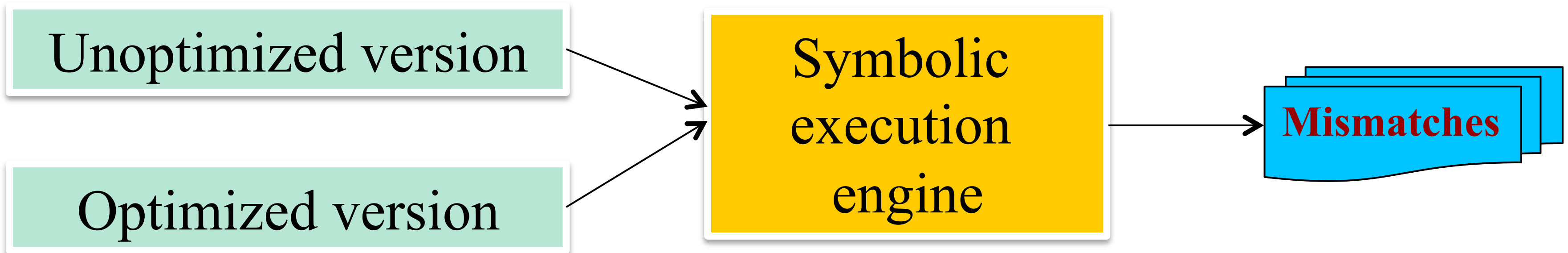
- **Causes Bonjour to abort, potential DoS attack**
- **Confirmed by Apple, security update released**

Semantic Errors via Crosschecking (Equivalence Checking)

Lots of available opportunities as code is:

Optimized frequently   Refactored frequently

Different implementations of the same interface



We can find any mismatches in their behavior by:

1. Using symbolic execution to explore multiple paths
2. Comparing the path constraints or input/output pairs across implementations

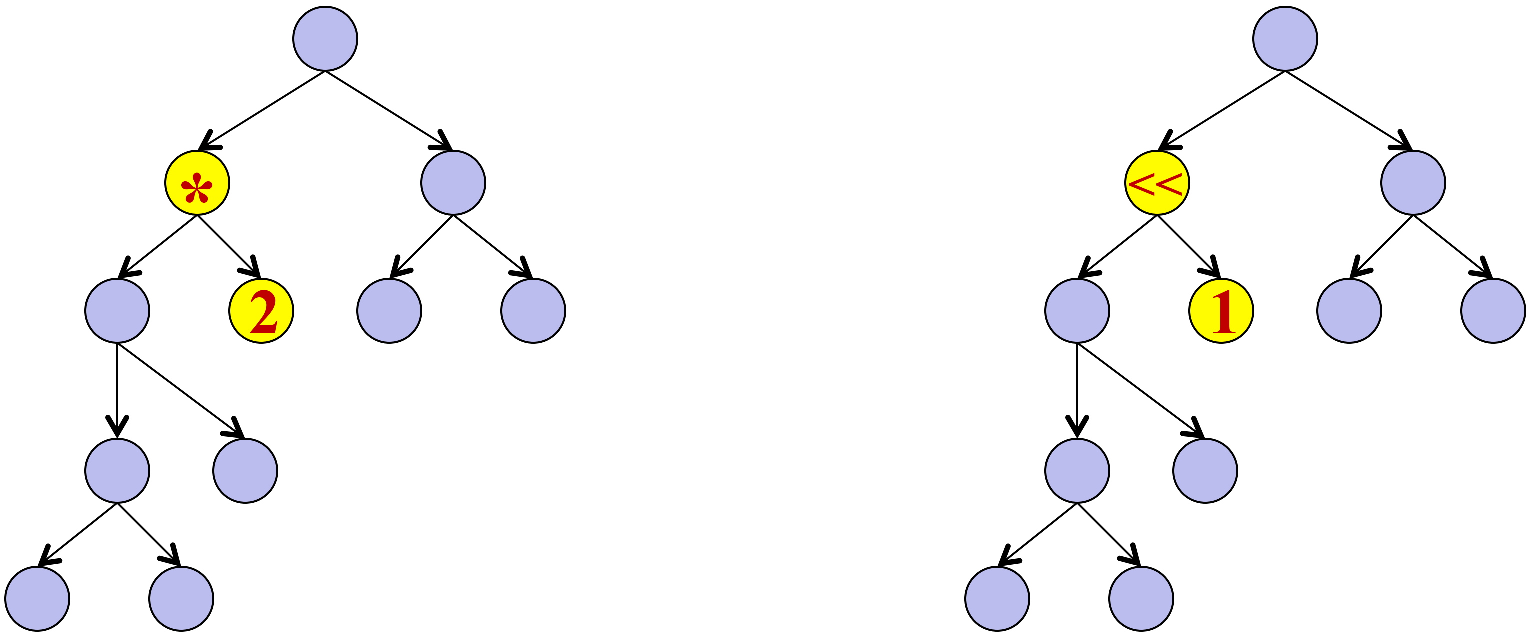
Crosschecking: Advantages

- Can find semantic errors without the need for specifications!

- Constraint solving 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* proven to be equivalent

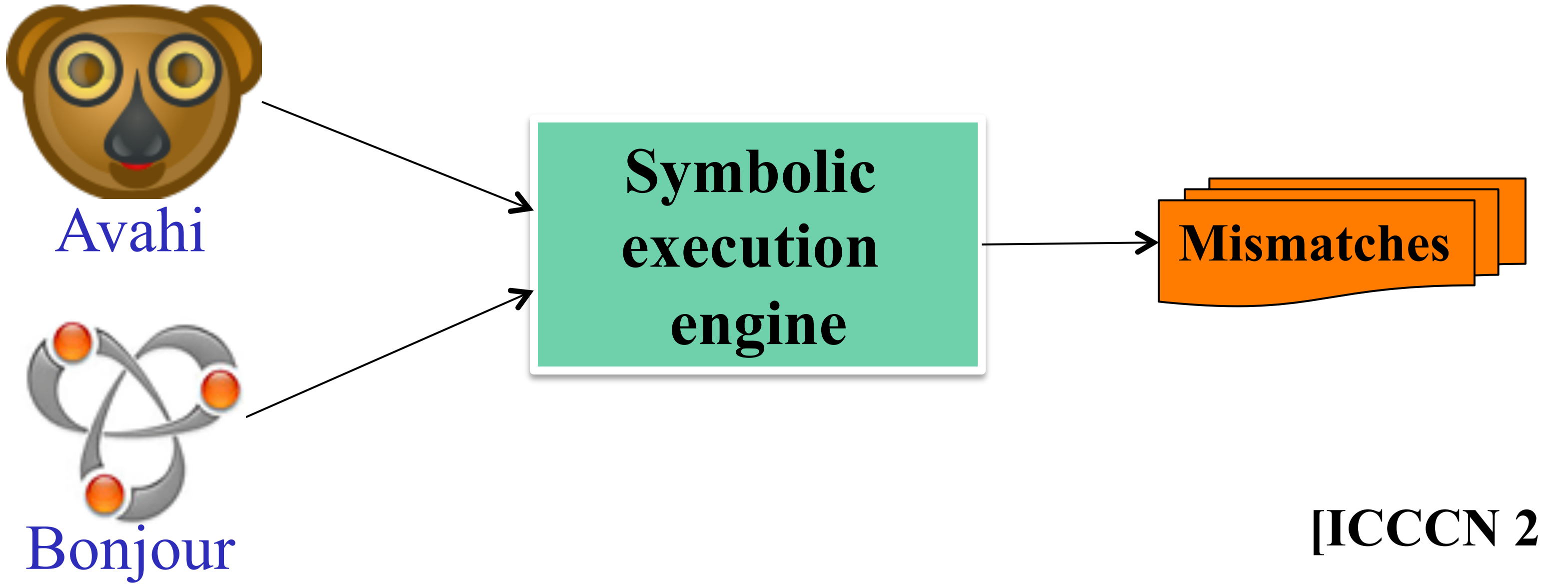
Crosschecking: Advantages



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

ZeroConf Protocol

- Enables devices to automatically configure themselves and their services and be discovered without manual intervention
- Two popular implementations: **Avahi** (open-source), and **Bonjour** (open-sourced by Apple)



[ICCCN 2011]

Server Interoperability Bonjour vs. Avahi

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	$\boxed{0002}$	0001
0030	0000	0000	0000	055F	6461	6170	045F	7463
0040	7005	6C6F	6361	6C00	000C	0001		

- **mDNS specification (§18.11):**
  "Multicast DNS messages received with non-zero Response Codes MUST be silently ignored."
- **Avahi ignores this packet, Bonjour does NOT**

New Platforms, New Code

- Recent years have seen the emergence of new computing platforms which provide many opportunities for optimizations

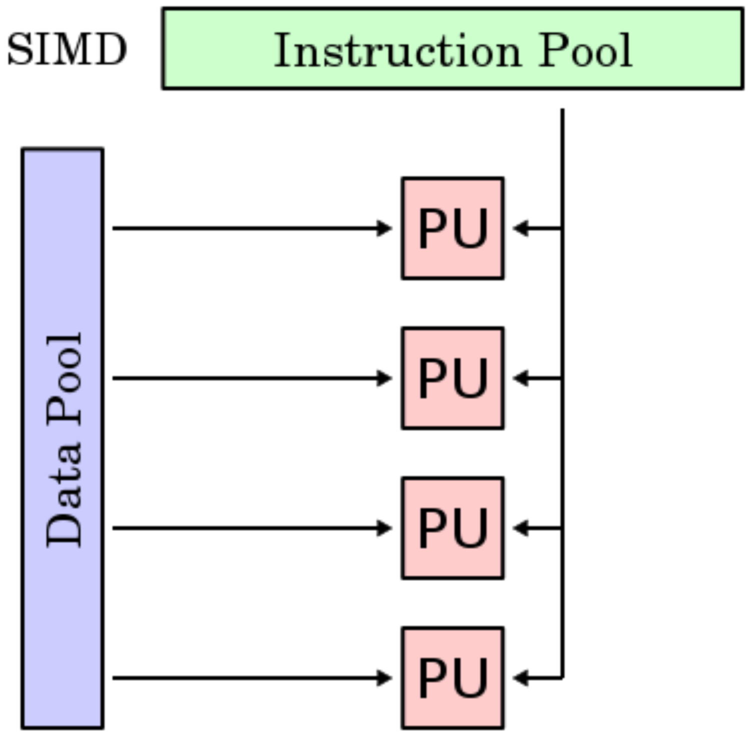
- Code is often adapted manually to benefit from these platforms

***Error-prone*, as any manual process**

SIMD Optimizations

Most processors offer support for SIMD instructions

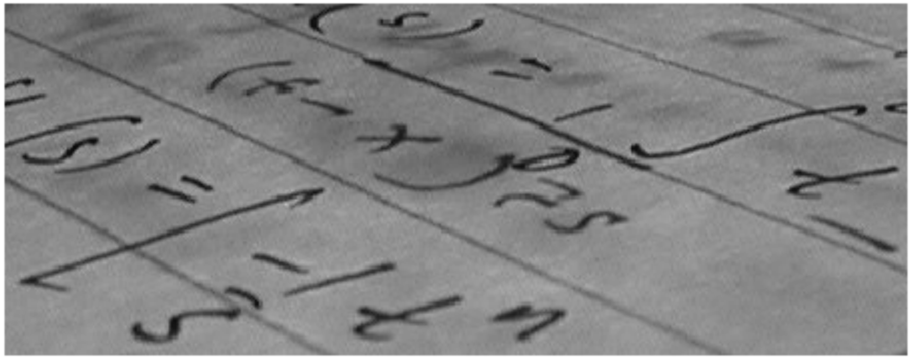
- Can operate on multiple data concurrently
- Many algorithms can make use of them (e.g., computer vision algorithms)



[EuroSys 2011]

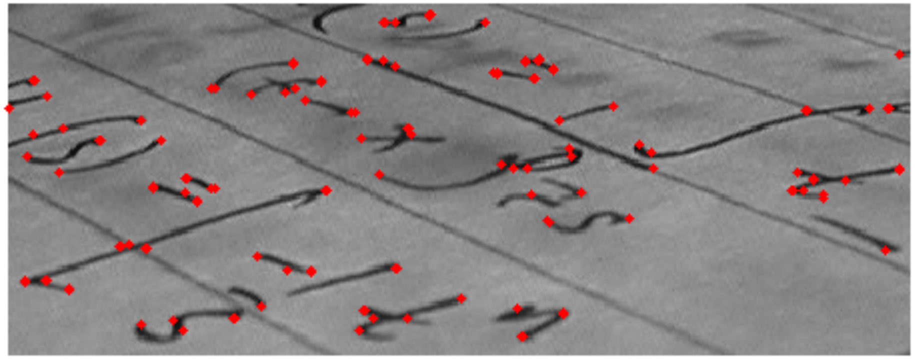
OpenCV

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

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
```

GPU Optimizations

Scalar vs. GPGPU code

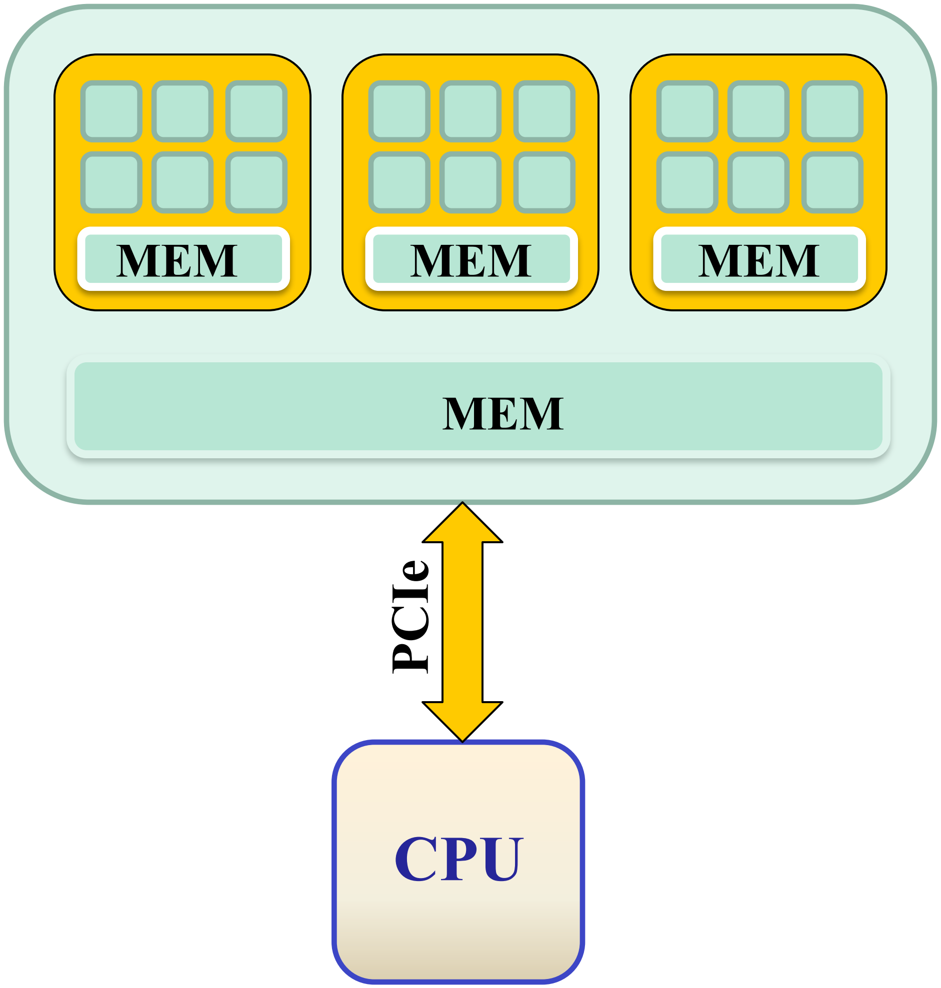
[HVC 2011]

General Purpose GPU Computing

(2006) NVIDIA CUDA

(2008) OpenCL

General Purpose GPUs (GPGPUs)



General-Purpose Graphics Processing Units (GPGPUs) are a programmable platform for highly parallel computation

New programming model:

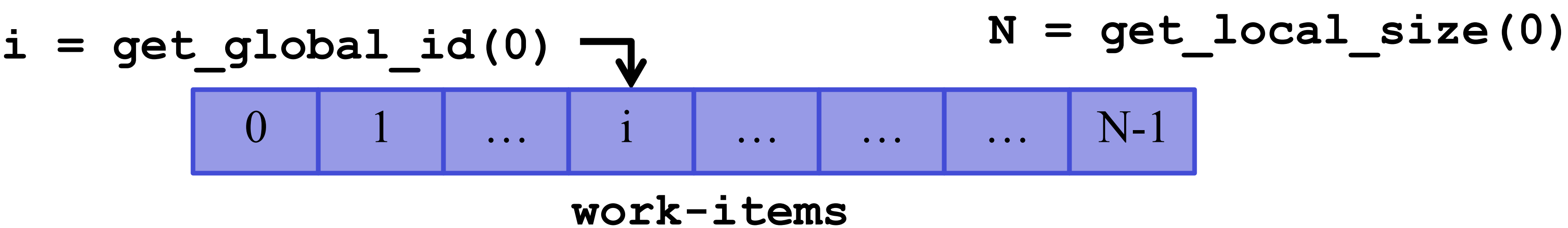
- Large number of threads
- Hierarchical execution and memory model

OpenCL

- Open Computing Language (OpenCL): an open standard for parallel computation
  - Targets both CPUs and GPGPUs
- OpenCL C language is a dialect of C99
- An OpenCL C program consists of one or more OpenCL C kernels which are executed on a device (such as a GPU)
- OpenCL C kernels may be invoked in parallel by the host using multiple work-items

OpenCL Example

```
__kernel void arr_sqrt(global float *a) {
     size_t i = get_global_id(0),
            N = get_local_size(0);
     float r0 = i > 0 ? a[i-1]:0;
     float r1 = a[i];
     float r2 = i < N ? a[i+1]:0;
     a[i] = (r0+r1+r2)/3;
}
```



Race Conditions

```
__kernel void arr_sqrt(global float *a) {
     size_t i = get_global_id(0),
            N = get_local_size(0);
     float r0 = i > 0 ? a[i-1]:0;
     float r1 = a[i];
     float r2 = i < N ? a[i+1]:0;
     a[i] = (r0+r1+r2)/3;
}
```

```
R/W race: work-item i writes a[i],
          work-item i+1 reads a[i]
```

Barriers

```
__kernel void arr_sqrt(global float *a) {
     size_t i = get_global_id(0),
            N = get_local_size(0);
     float r0 = i > 0 ? a[i-1]:0;
     float r1 = a[i];
     float r2 = i < N ? a[i+1]:0;
     barrier(CLK_GLOBAL_MEM_FENCE);
     a[i] = (r0+r1+r2)/3;
}
```

barrier() blocks until all work-items (in the same work-group) reached the call

Race Checking for GPGPUs

```
__kernel void arr_sqrt(global float *a) {
      size_t i = get_global_id(0),
             N = get_local_size(0);
      float r0 = i > 0 ? a[i-1]:0;
      float r1 = a[i];
      float r2 = i < N ? a[i+1]:0;
      a[i] = (r0+r1+r2)/3;
}
```

MAR(a[0]): work item 0

Wid	R	W
-	-	-
-	-	-
-	-	-
-	-	-
0	X	-
0	X	-
0	X	X

- Wid: Work-item that accessed byte
- R: whether byte was read
- W: whether byte was written

Race Checking for GPGPUs

```
__kernel void arr_sqrt(global float *a) {
     size_t i = get_global_id(0),
            N = get_local_size(0);
     float r0 = i > 0 ? a[i-1]:0;
     float r1 = a[i];
     float r2 = i < N ? a[i+1]:0;
     a[i] = (r0+r1+r2)/3;
}
```

MAR(a[0]): work item 1

Wid	R	W	
0	X	X	
0	X	X	
0	X	X	
0	X	X	R/W

- Wid: Work-item that accessed byte
- R: whether byte was read
- W: whether byte was written

Race Checking for GPGPUs

MAR(a[0]): work item 0

Wid	R	W
-	-	-
-	-	-
-	-	-
-	-	-
0	X	-
0	X	-

```
__kernel void arr_sqrt(global float *a) {
    size_t i = get_global_id(0),
           N = get_local_size(0);
    float r0 = i > 0 ? a[i-1]:0;
    float r1 = a[i];
    float r2 = i < N ? a[i+1]:0;
    barrier(CLK_GLOBAL_MEM_FENCE);
    a[i] = (r0+r1+r2)/3;
}
```

We model barrier() by resetting the MAR before continuing execution any of the work-items past the barrier

Race Checking for GPGPUs

MAR(a[0]): work item 1

```
__kernel void arr_sqrt(global float *a) {
     size_t i = get_global_id(0),
            N = get_local_size(0);
     float r0 = i > 0 ? a[i-1]:0;
     float r1 = a[i];
     float r2 = i < N ? a[i+1]:0;
     barrier(CLK_GLOBAL_MEM_FENCE);
     a[i] = (r0+r1+r2)/3;
}
```

Wid	R	W
0	X	-
0	X	-
0	X	-
0	X	-
0	X	-
0	X	-

We model barrier() by resetting the MAR before continuing execution any of the work-items past the barrier

Symbolic Races

Work-item i

`a[i] = ...`

Work-item j

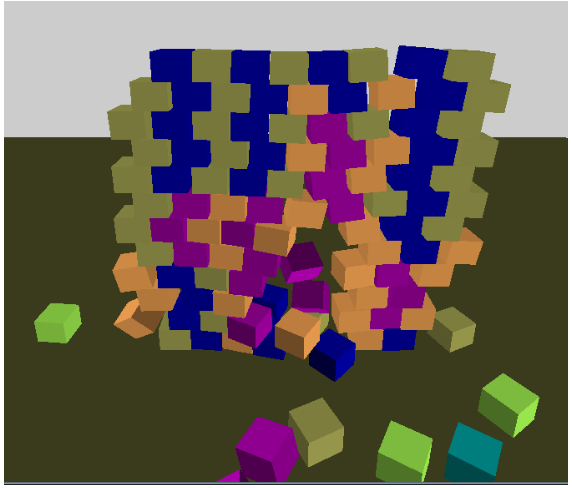
`a[j] = ...`

Write-after-write race
if **`i=j`** satisfiable

GPGPU (OpenCL) Optimizations

- Parboil:
  - GPU benchmark suite, originally written in CUDA
- OP2
  - Library for applications on unstructured grids
- Bullet open-source physics library
  - Popular library used movie studios and professional game developers
  - Analyzed soft body engine

Bullet library



OpenCL Benchmarks: Bugs and Mismatches

Several bugs and mismatches:

- 2 mismatches between C and OpenCL code
  - Incorrect FP associativity and distributivity assumptions (CP in Parboil)
- 3 memory errors
  - Buffer overflows (MRI-Q&MRI-FHD in Parboil)
  - Use-after-free: incorrect synchronization between host and kernel code (MRI-Q in Parboil)
  - Uninitialized memory (MRI-FHD in Parboil)
- 1 race condition
  - Missing synchronization barrier (OP2)
- 1 compiler bug
  - NVidia compiler bug (incorrect optimization)

Integrating Crosschecking into Development Process

Semantic mismatches not always errors

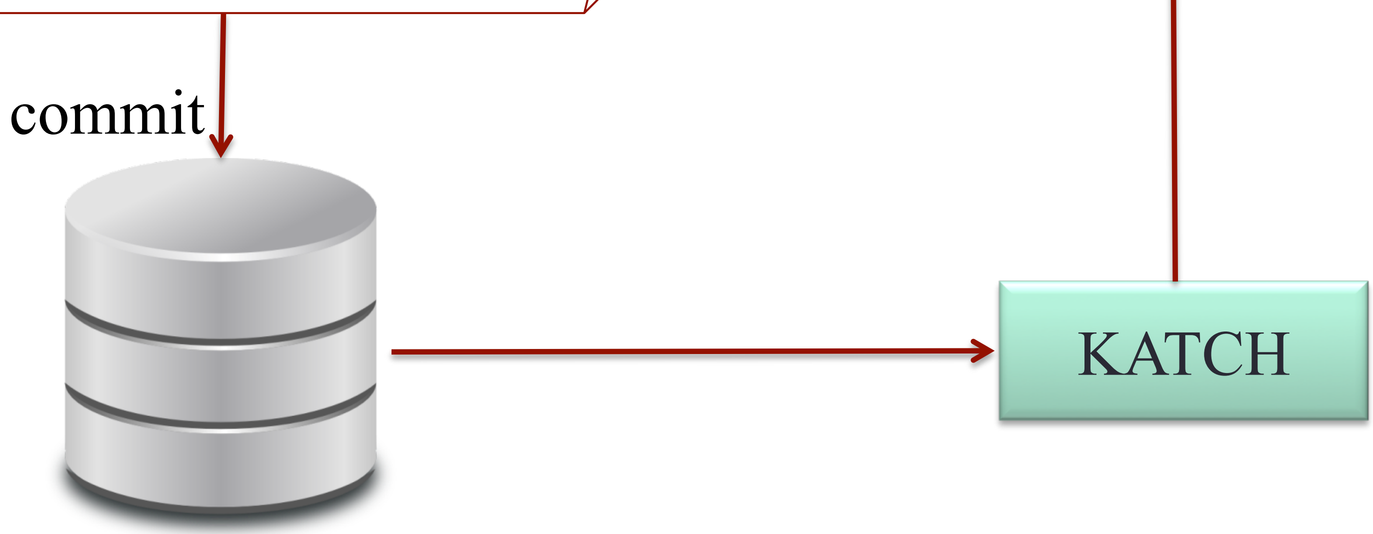
- Underspecified behavior

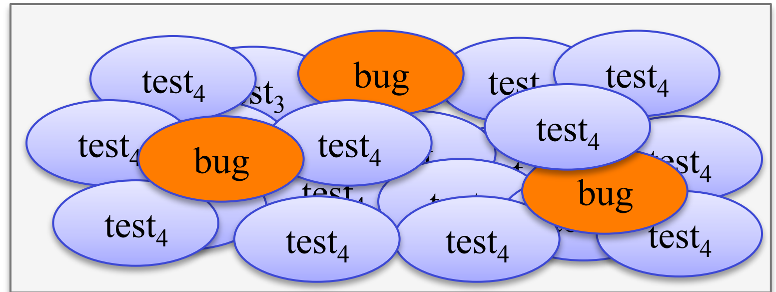
Two (anecdotal) insights:

1. Provide developers the **ability to add "assumptions"** eg:
   - Floating-point associativity holds:
     - A+(B+C) = (A+B)+C
   - Disregard the difference between 0_- and 0_+:
     - A+0 = A
2. All things being equal, developers **prefer to keep the behavior of the reference implementation**
   - Particularly if we can provide some guarantees
     - bounded equivalence

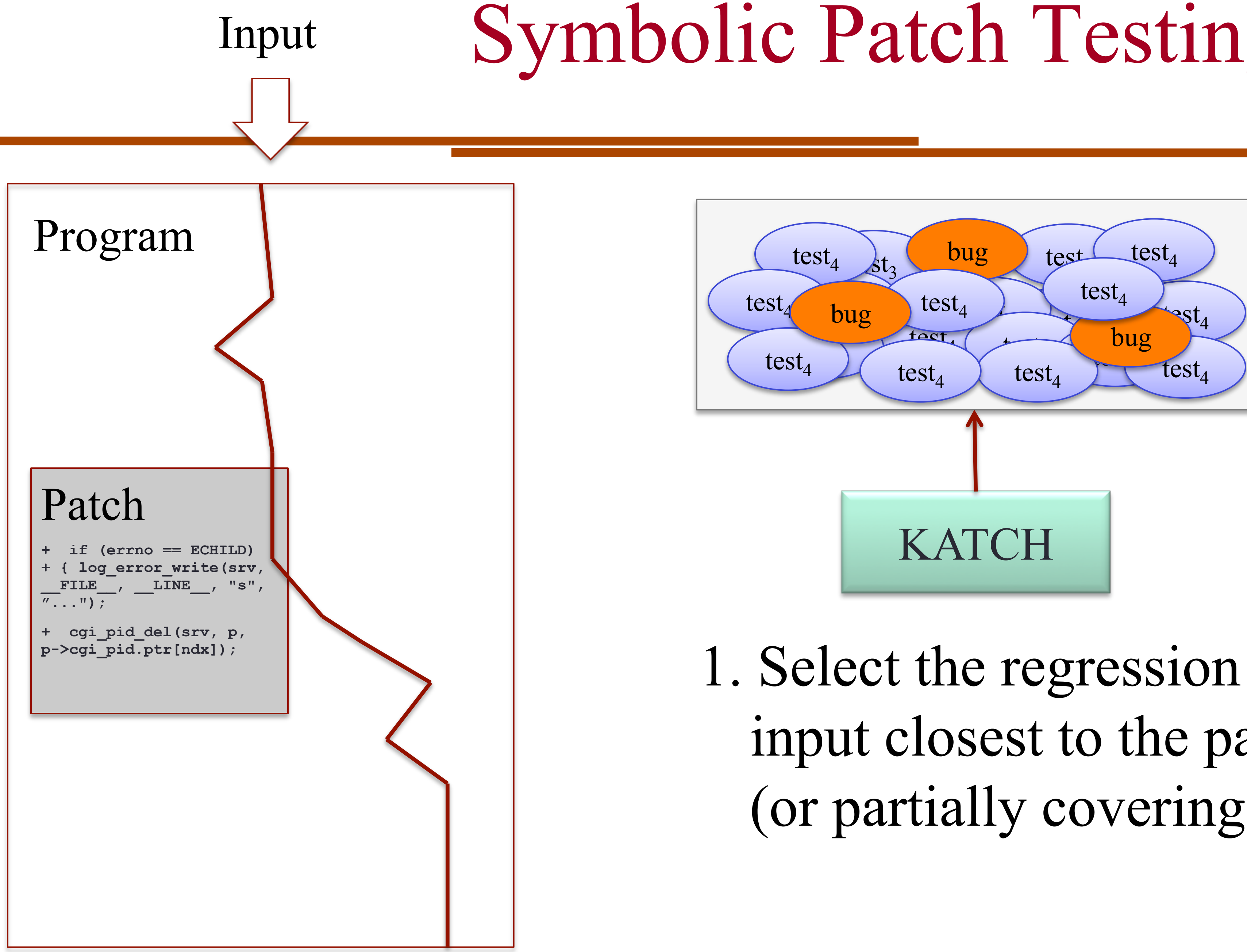
KATCH: High-Coverage Symbolic Patch Testing

```
--- klee/trunk/lib/Core/Executor.cpp  2009/08/01 22:31:44 77819
+++ klee/trunk/lib/Core/Executor.cpp  2009/08/02 23:09:31 77922
@@ -2422,8 +2424,11 @@
       info << "none\n";
     } else {
       const MemoryObject *mo = lower->first;
+      std::string alloc_info;
+      mo->getAllocInfo(alloc_info);
       info << "object at " << mo->address
-           << " of size " << mo->size << "\n";
+           << " of size " << mo->size << "\n"
+           << "\t\t" << alloc_info << "\n";
```



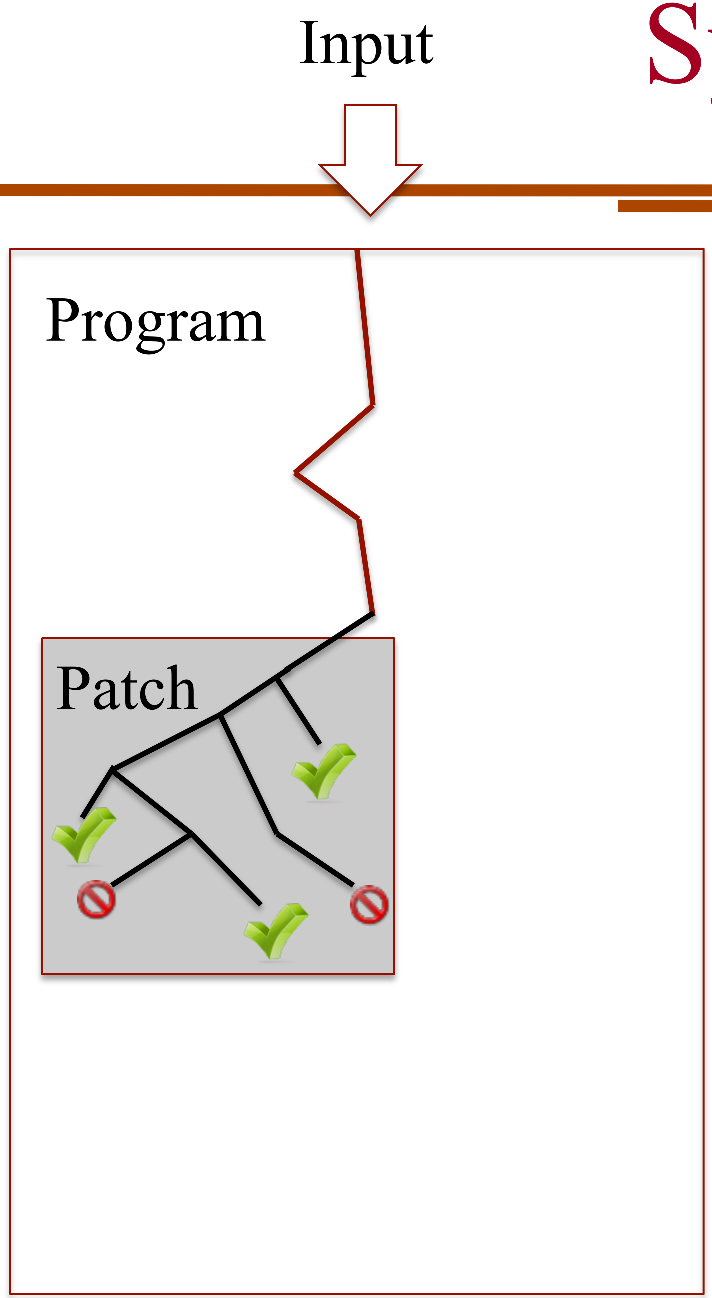


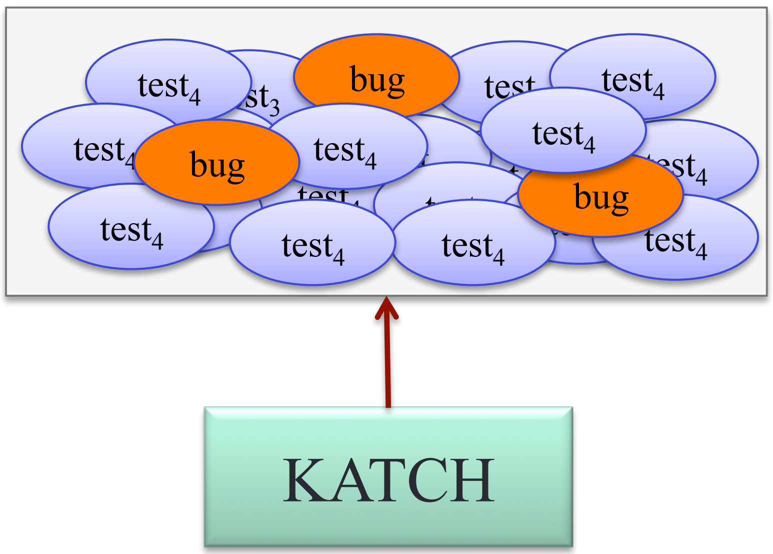
Symbolic Patch Testing



1. Select the regression input closest to the patch (or partially covering it)

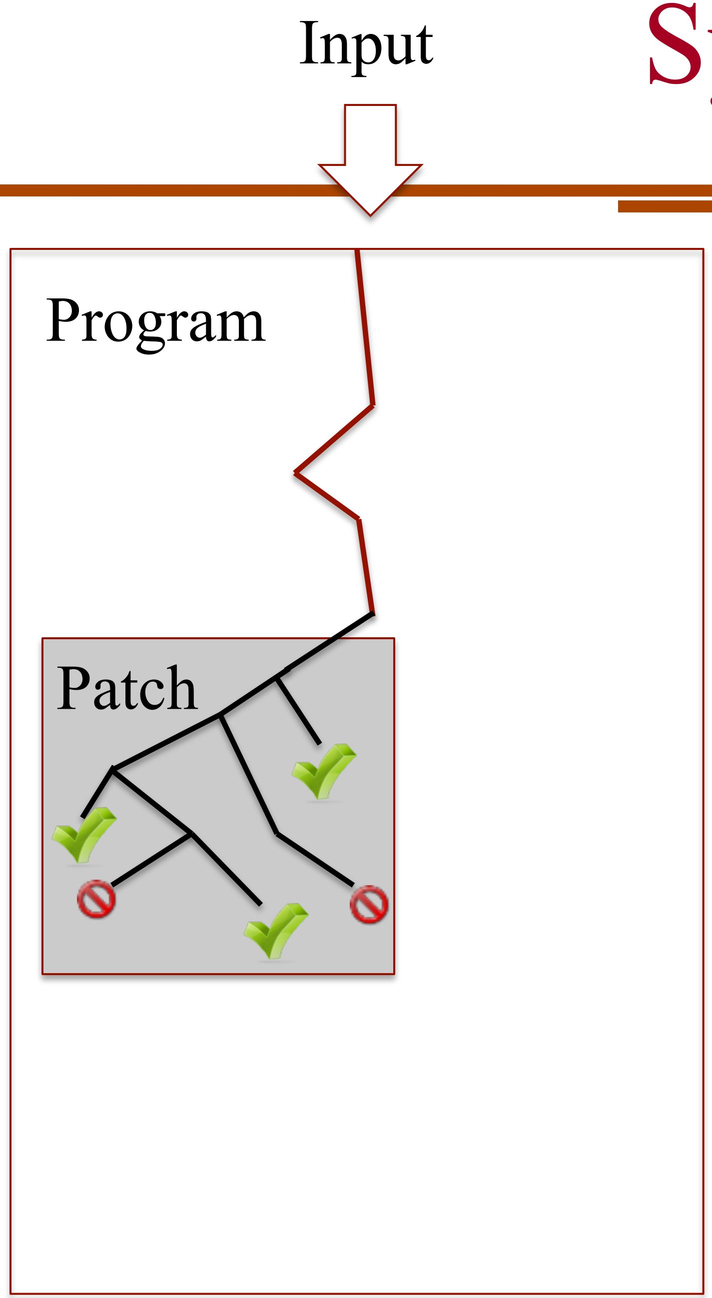
Symbolic Patch Testing

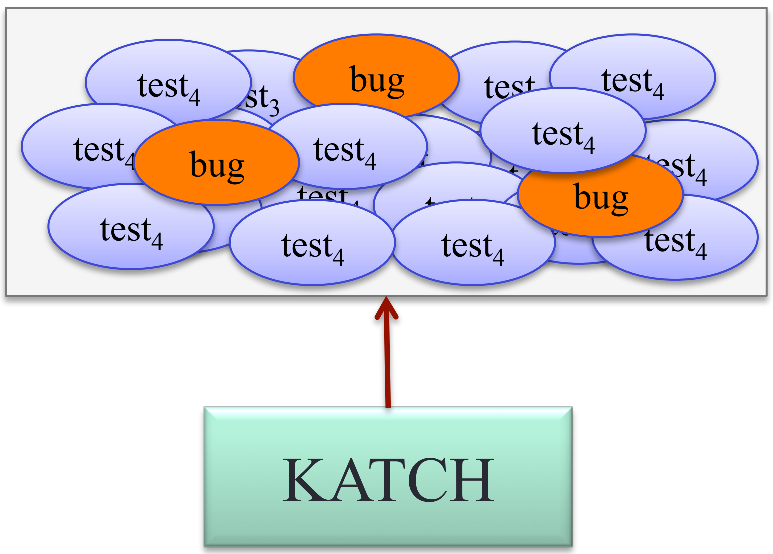




2. Greedily drive exploration toward uncovered statements in the patch

Symbolic Patch Testing





3. If stuck, identify the constraints that disallow execution to reach the patch, and backtrack

Greedy Exploration Step

```
void log(char input) {
    int file = open("access.log"…);
    if (input >= '␣' && input <= '~') {
        // printable characters
        write(file, &input, 1);
+   } else {
+       char escinput = escape(input);
+       write(file, &escinput, 1);
+   }
    close(file);
}
```

lighttpd r2660: patch modifies log() to escape sensitive characters

Greedy Exploration Step

```
void log(char input) {
    int file = open("access.log"...);
    if (input >= '␣' && input <= '~') {
        // printable characters
        write(file, &input, 1);
  + } else {
  +     char escinput = escape(input);
  +     write(file, &escinput, 1);
  + }
    close(file);
}
```

Available input: "t"
(or any printable char)

1. Greedy step: choose the symbolic branch point whose unexplored side is closest to the patch.
2. Explore this side!

Informed Path Regeneration

```
void log(char input) {
    if (input >= '␣' && input <= '~') {
        . . .
    } else {
+   . . .
    }
}

if (0 == strcmp(request, "GET")
    . . .
for (char* p = request; *p; p++)
    log(*p);
```

Available input: **"GET"**

Greedy step fails!

1. Backtrack to the last symbolic branch that disallows this side to be executed
2. Explore the other side of that branch

req[2] ≠ 'T'

Definition Switching

```
enum escape_t escape;
void log(char input) {
    if (escape == ESCAPE_ALL) {
+       . . .
 }

opt = getopt_long(argc, argv, ...);
switch (opt) {
    case 'a': escape = ESCAPE_SPACE;
              break;
    case 'b': escape = ESCAPE_ALL;
. . .
log(...);
```

Available test: opt = **'a'**

Patch guarded by concrete branch

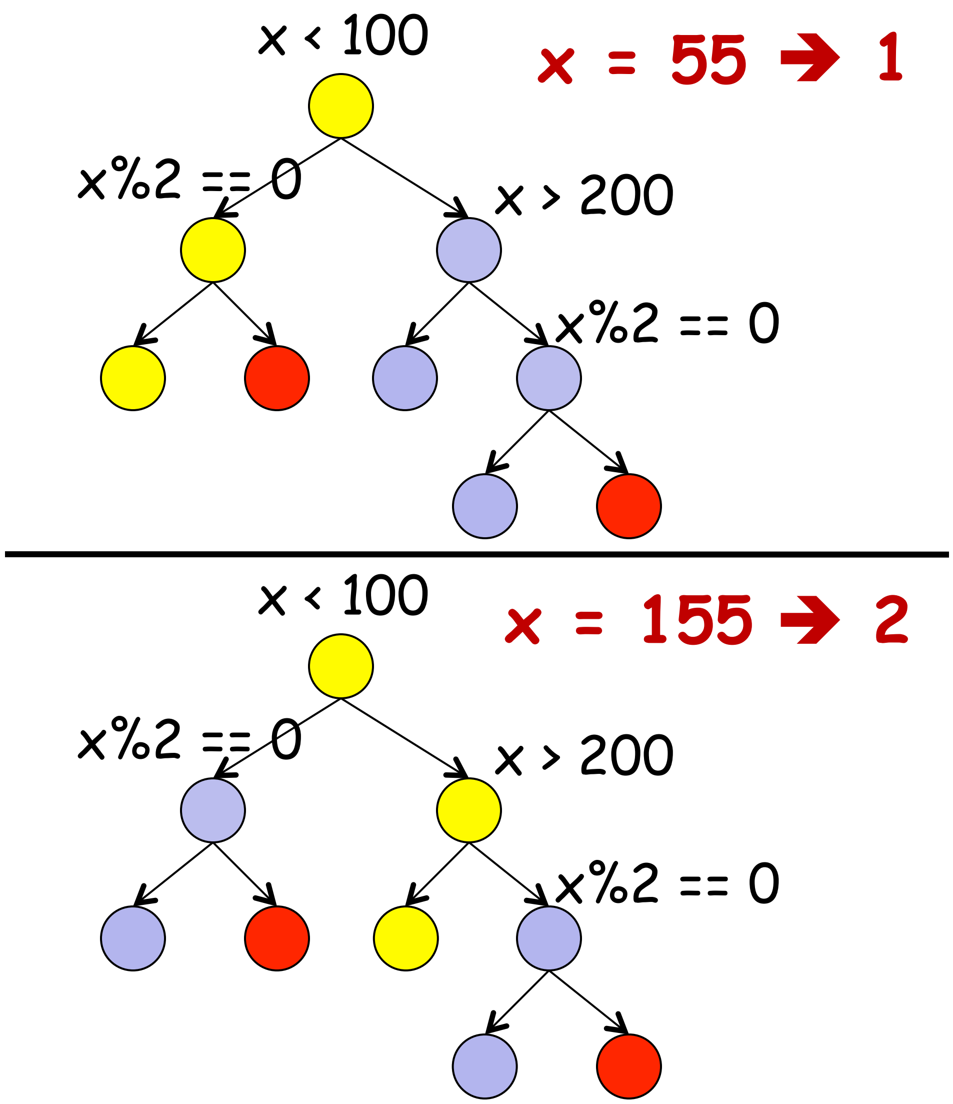
Backtracking step fails!

1. Find all reaching definitions for the variables involved and try to cover another one.
2. Favors definitions that can be statically shown to satisfy target, or unexecuted definitions

Input Selection

Naïve solution: calculate the context-sensitive static distance between the path executed by an input and the patch code

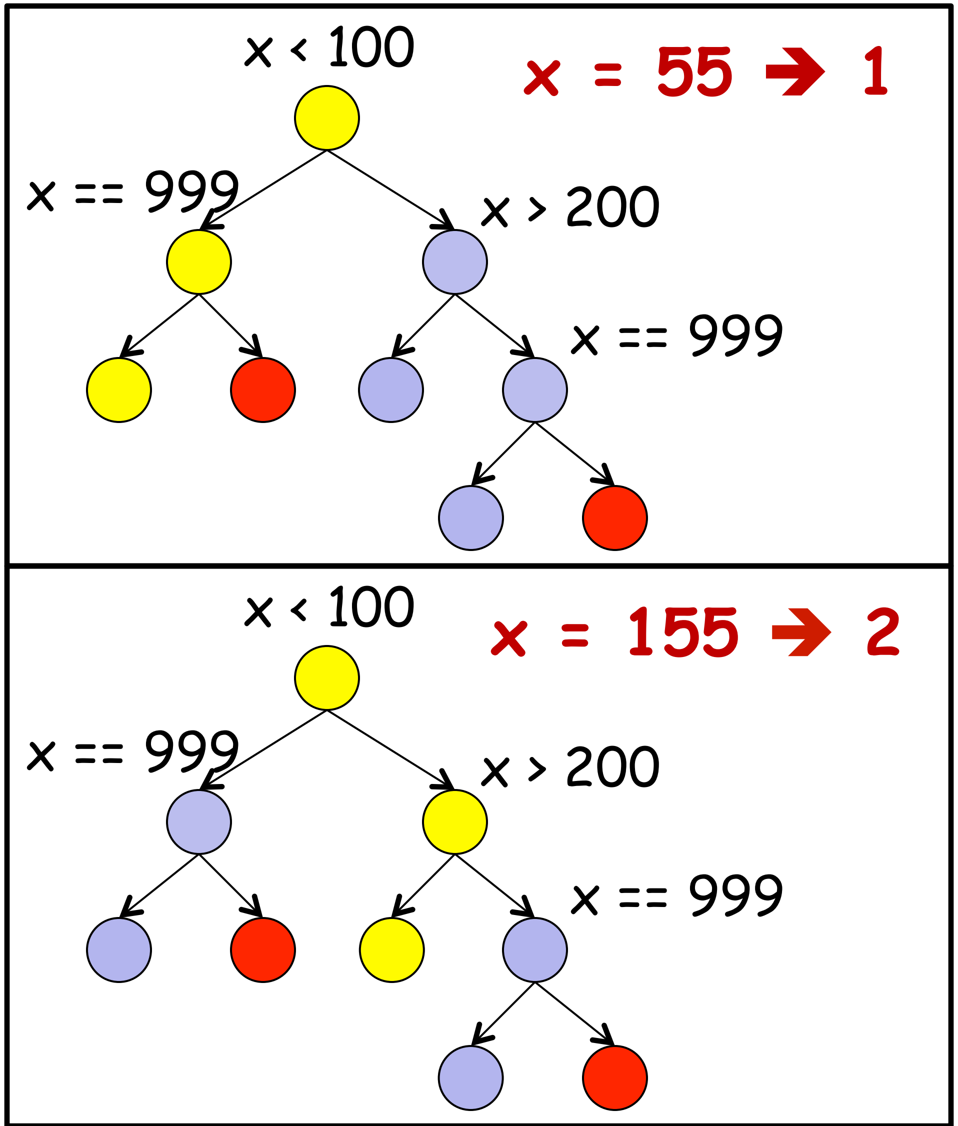
```
if (x < 100)
    f(x);
else
    if (x > 200)
        f(x+1);

void f(int x) {
   if (x % 2 == 0)
      PATCH;
   . . .
```



Input Selection

Naïve solution: calculate the context-sensitive static distance between the path executed by an input and the patch code

```
if (x < 100)
    f(x);
else
    if (x > 200)
        f(x+1);

void f(int x) {
   if (x == 999)
      PATCH;
   . . .
```

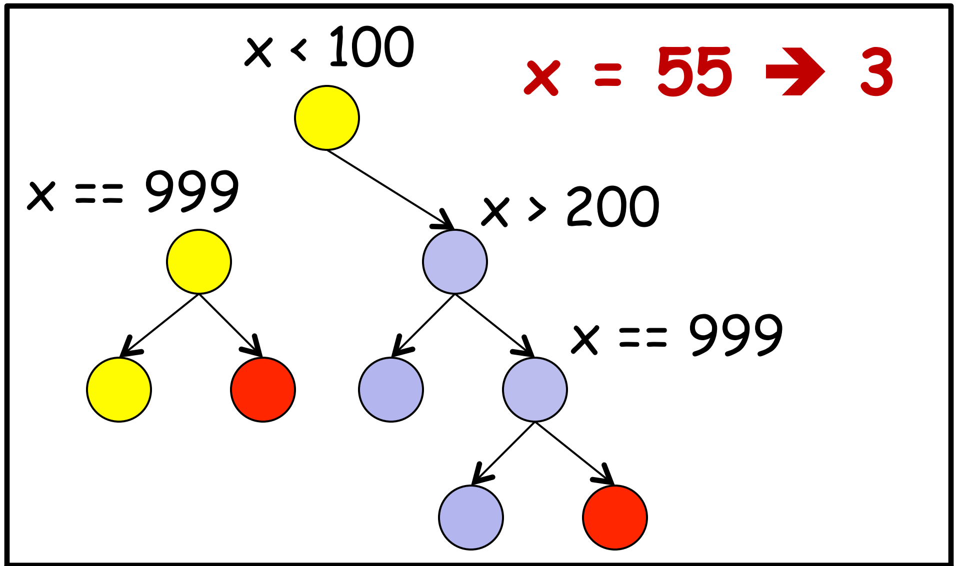


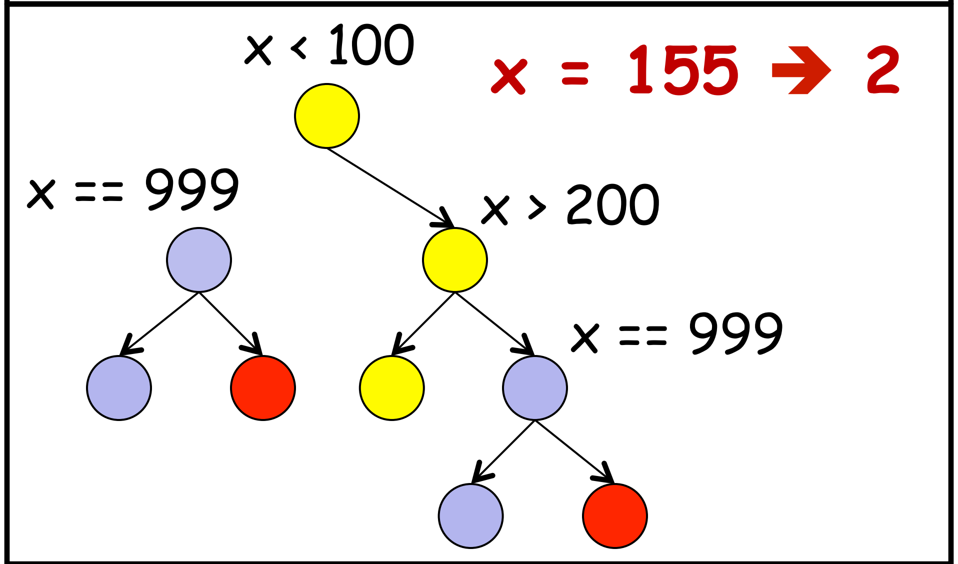
Input Selection: Weakest Preconditions

For which basic block in the program, compute a necessary condition for reaching the target.

Prune CFG edges which make the target unreachable.

```
if (x < 100)
   f(x);
else
   if (x > 200)
      f(x+1);

void f(int x) {
  if (x == 999)
     PATCH;
  . . .
```





Example: Lighttpd r2631

LIGHTTPD
fly light.

Powers several popular sites such as YouTube and Wikipedia

Revision	ELOC	Covered ELOC	
		Regression	KATCH
2631	20	15 (75%)	20 (100%)

http://zzz.example.com/ → KATCH → https://zz.example.com/

[SPIN 2012]

Lighttpd r2660

Revision	ELOC	Covered ELOC	
		Regression	KATCH
2660	33	9 (27%)	24 (72%)

```
if (str->ptr[i] >= '␣' && str->ptr[i] <= '~') {
    /* printable chars */
    buffer_append_string_len(dest,&str ->ptr[i],1);
} else switch (str->ptr[i]) {
case '"':
    BUFFER_APPEND_STRING_CONST(dest, "\\\"");
    break;
```

Bug reported and fixed promptly by developers

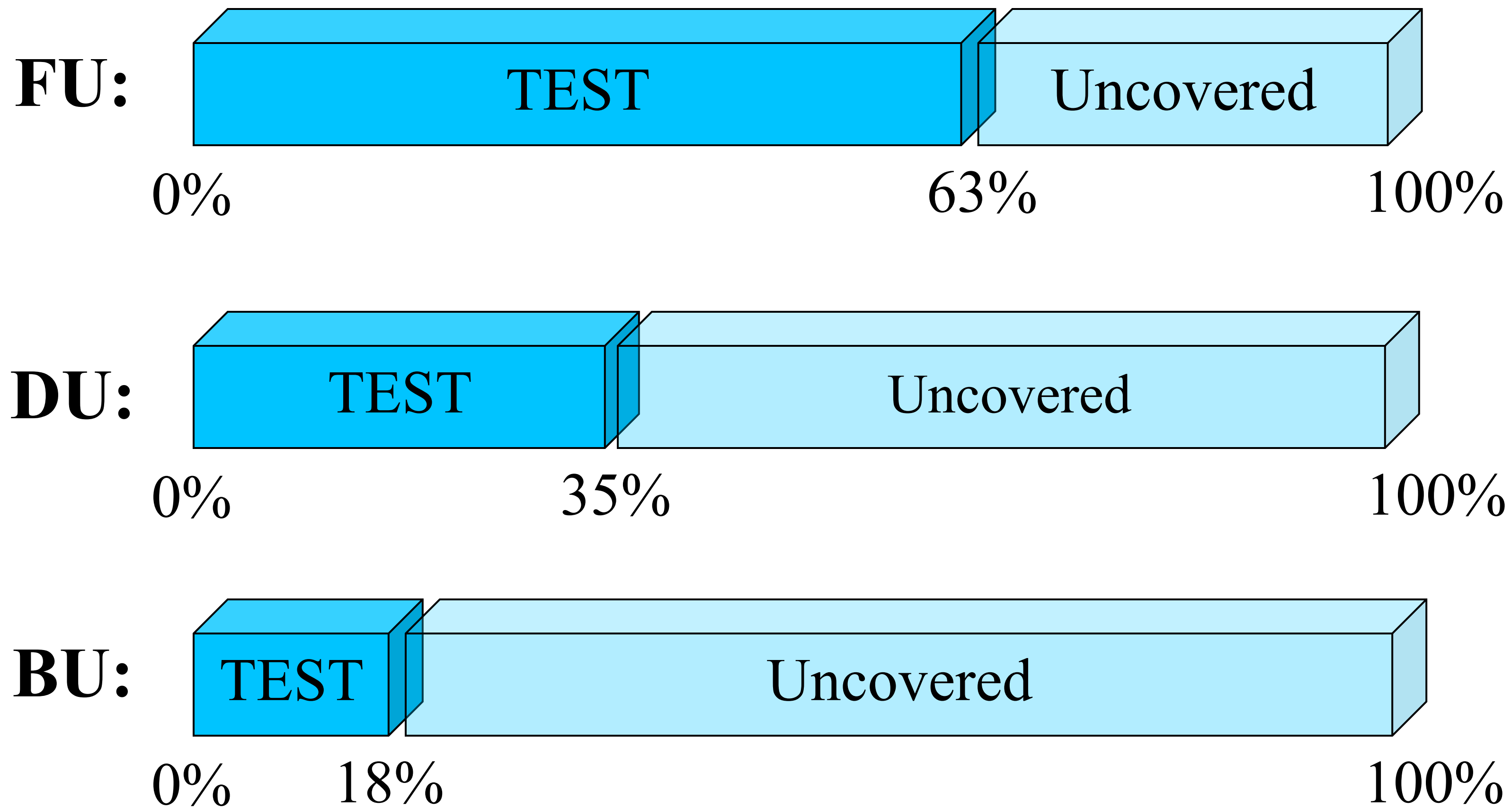
Extended Evaluation (WiP)

Key evaluation criteria: **no cherry picking!**

- choose all patches for an application over a contiguous time period

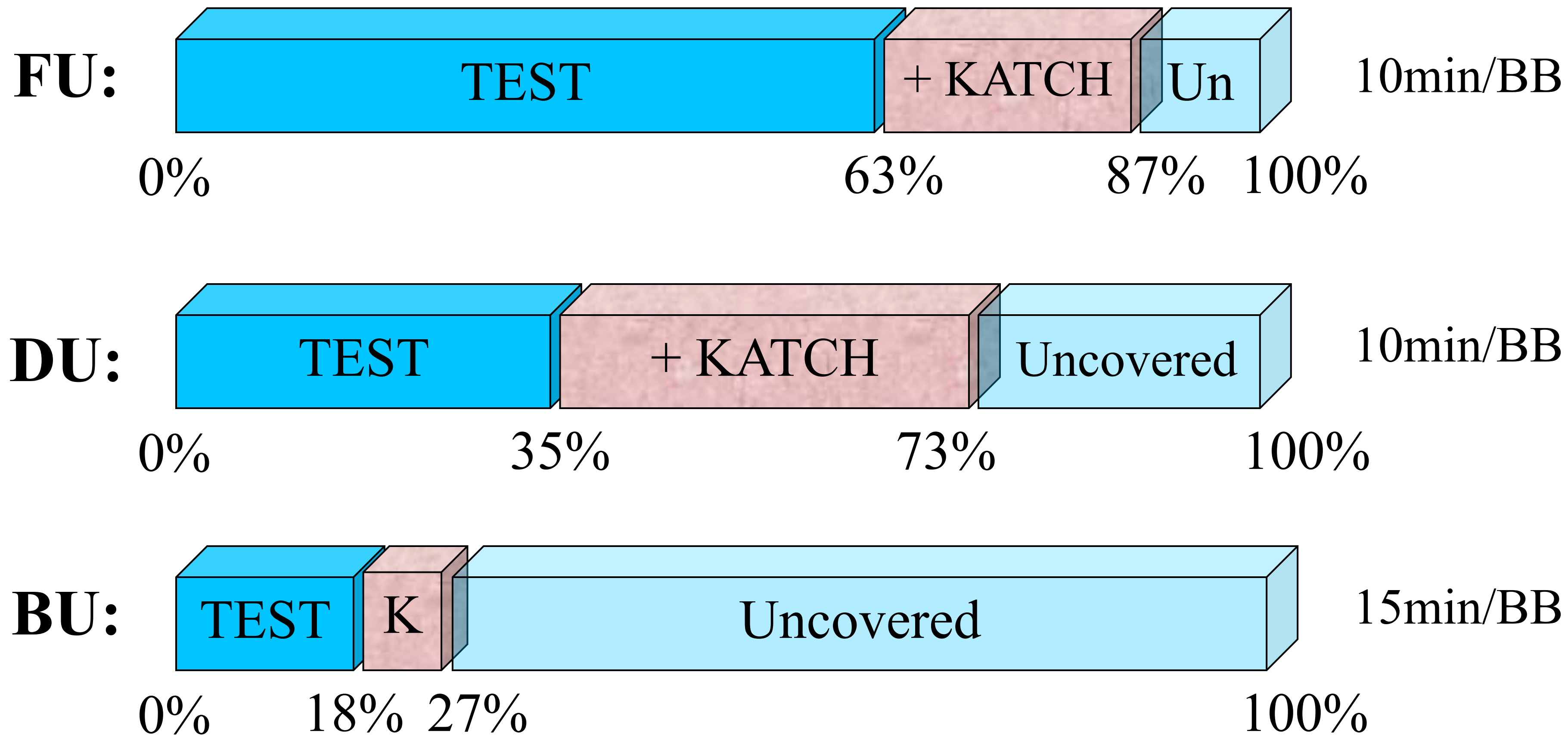
FindUtils suite (FU) **find, xargs, locate**	**12,648 ELOC**	**125 patches written over ~26 months**
DiffUtils suite (DU) **s/diff, diff3, cmp**	**55,655 ELOC** **+ 280,000 in libs**	**175 patches written over ~30 months**
BinUtils suite (BU) **ar, elfedit, nm, etc.**	**81,933 ELOC** **+ 800,000 in libs**	**181 patches written over ~16 months**

Patch Coverage (basic block level)



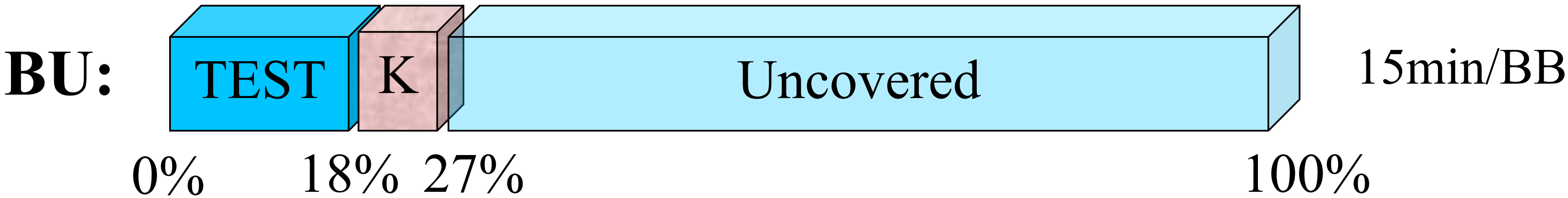
Standard symbolic execution (30min/BB) only added +1.2% to FU

Patch Coverage
(current results – ongoing work)



Standard symbolic execution (30min/BB) only added +1.2% to FU

Binutils: Coverage+Bugs (current results)



- Found 14 distinct crash bugs, all in BU
  - All unreachable by standard symbolic execution
- 12 bugs still present in latest version
  - Reported (some already fixed) by dev
- 10 bugs found in the patch code itself or affected by patch code

KLEE – Demo

http://klee.llvm.org

- bad_abs
- xcheck_abs
- squares

Dynamic Symbolic Execution

- Automatically reasons about program behavior and the interaction with users and environment
- Can generate inputs exposing both generic and semantic bugs in complex software
  - Including file systems, library code, utility applications, network servers, device drivers, computer vision code

KLEE: Freely Available as Open-Source

http://klee.llvm.org

- Over 250 subscribers to the klee-dev mailing list
- Extended in many interesting ways by several research groups, in the areas of:
  - wireless sensor networks/distributed systems
  - schedule memoization in multithreaded code
  - automated debugging
  - exploit generation
  - client-behavior verification in online gaming
  - GPU testing and verification
  - etc.