Dynamic Symbolic Execution

Cristian Cadar

Department of Computing

Imperial College London

Based on 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

Computer Aided Verification (CAV'13)
St Petersburg, 15 July 2013

Dynamic Symbolic Execution

- Dynamic symbolic execution is a technique for ***automatically exploring paths*** through a program
  - Determines the feasibility of each explored path using a ***constraint solver***
  - Checks if there are ***any*** values that can cause an error on each explored path
  - For each path, can generate a ***concrete input triggering the path***

Dynamic Symbolic Execution

- Received significant interest in the last few years
- Many dynamic symbolic execution/concolic tools available as open-source:
  - **CREST, KLEE, SYMBOLIC JPF**, etc.
- Started to be adopted/tried out in the industry:
  - Microsoft (**SAGE, PEX**)
  - NASA (**SYMBOLIC JPF, KLEE**)
  - Fujitsu (**SYMBOLIC JPF, KLEE/KLOVER**)
  - IBM (**APOLLO**)
  - etc.

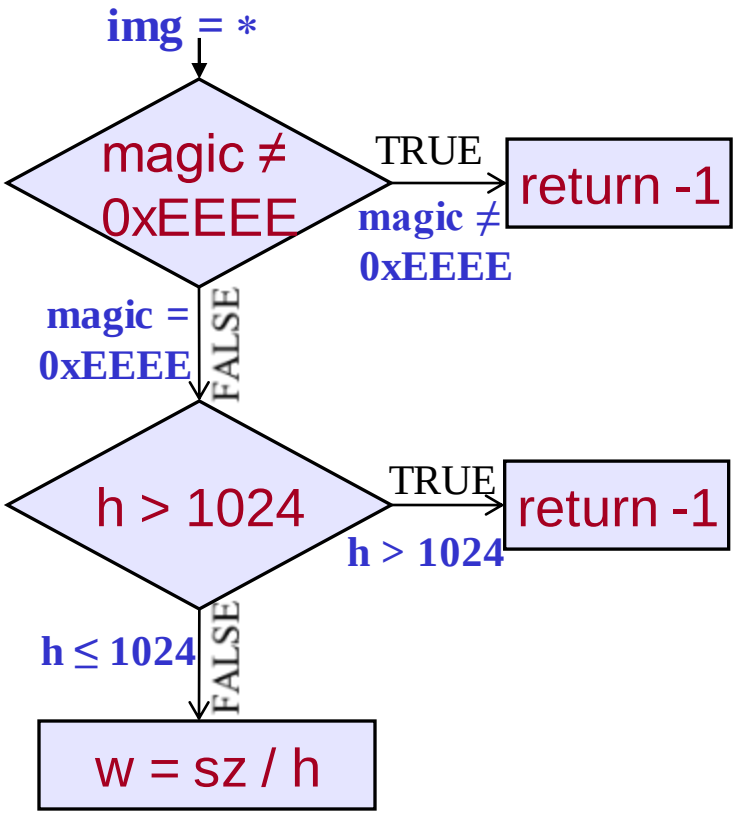
Dynamic Symbolic Execution

- **Toy example and demo**
- **Scalability challenges**
  - Path explosion challenges
  - Constraint solving challenges
- **Generic bug finding**
  - User-level utilities, kernel code, drivers, libraries, etc.
  - Attack generation against file systems and network servers
- **Patch testing**
  - Testing six years of patches in three application suite
- **Semantic errors via crosschecking**
  - Server interoperability, SIMD optimizations, 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;
  ...
}
```

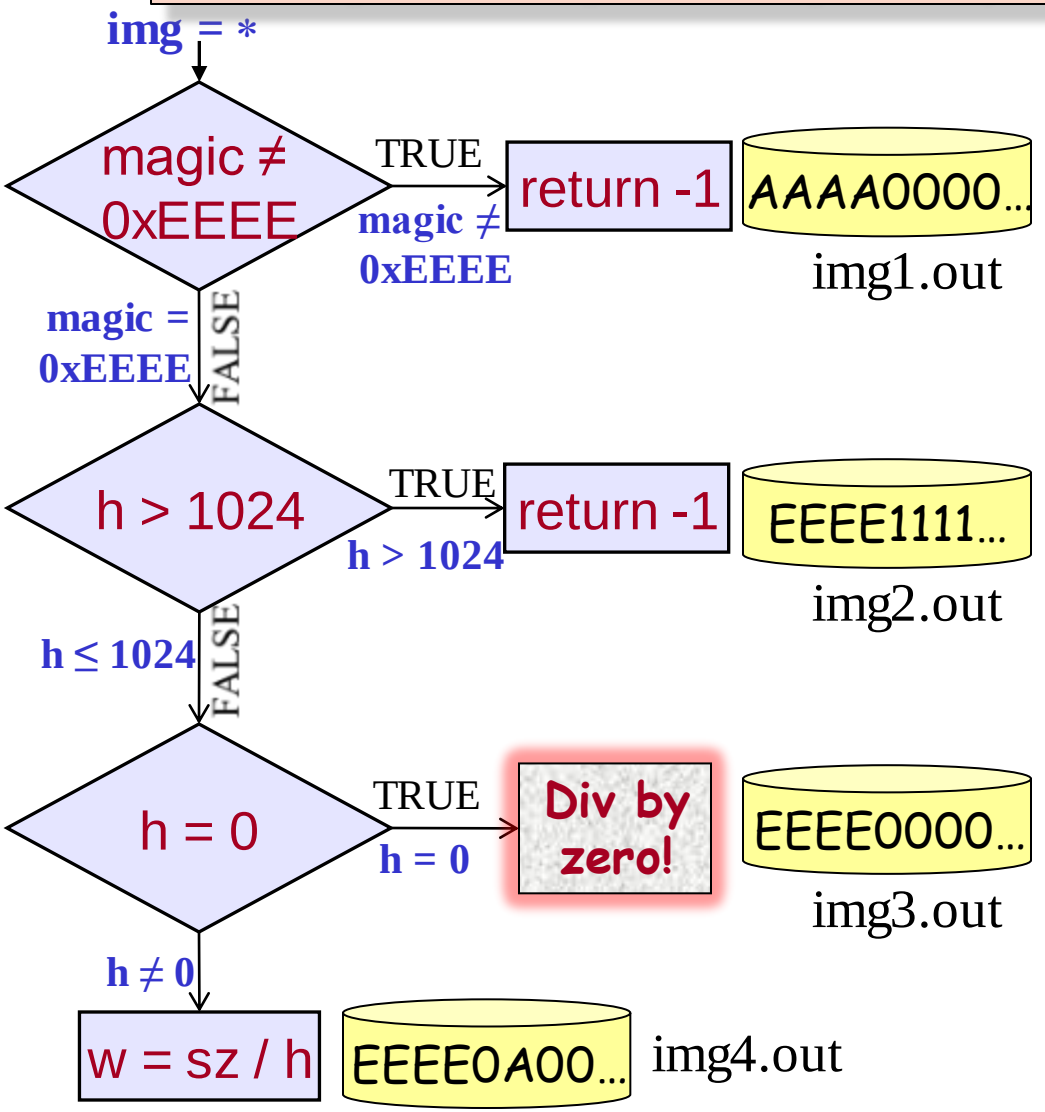


Toy Example

Each path is explored separately!

```
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;
  ...
}
```



All-Value Checks

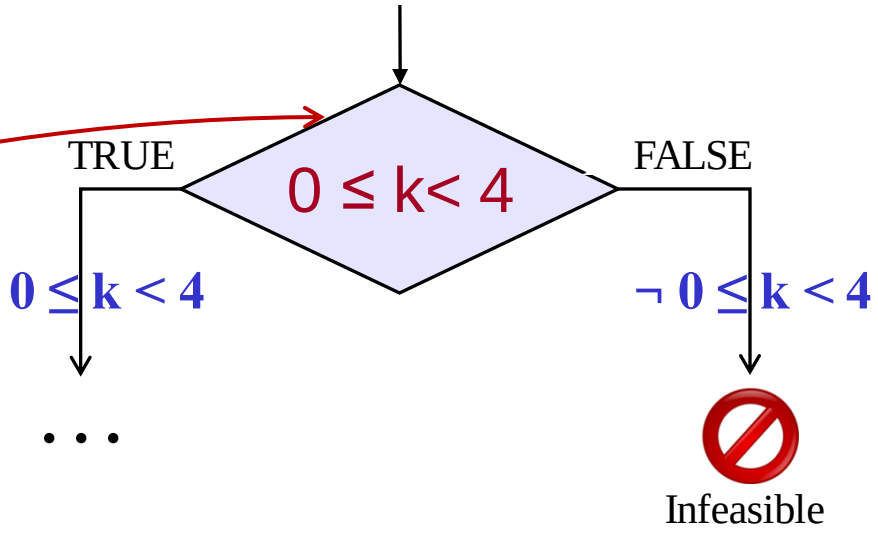
Implicit checks before each dangerous operation

- 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

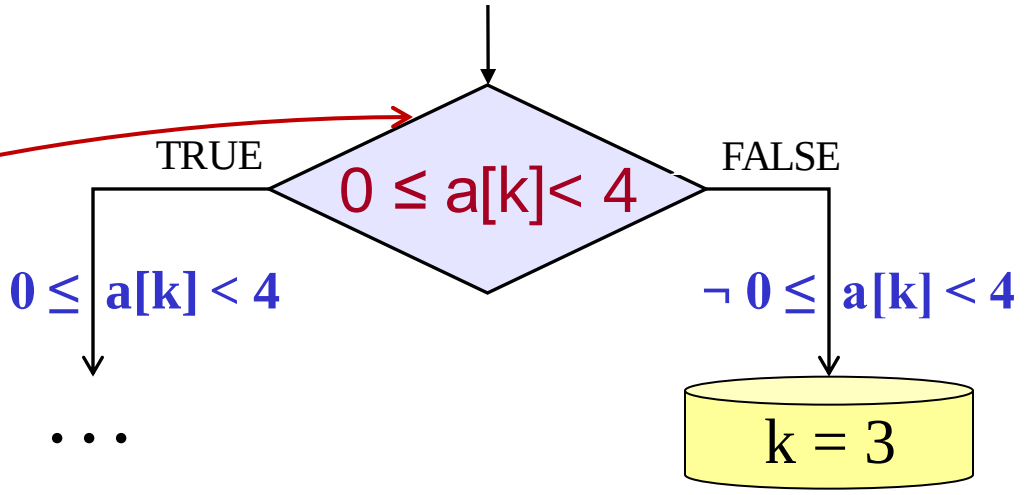
Implicit checks before each dangerous operation

- 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]];
}
```



Buffer overflow!

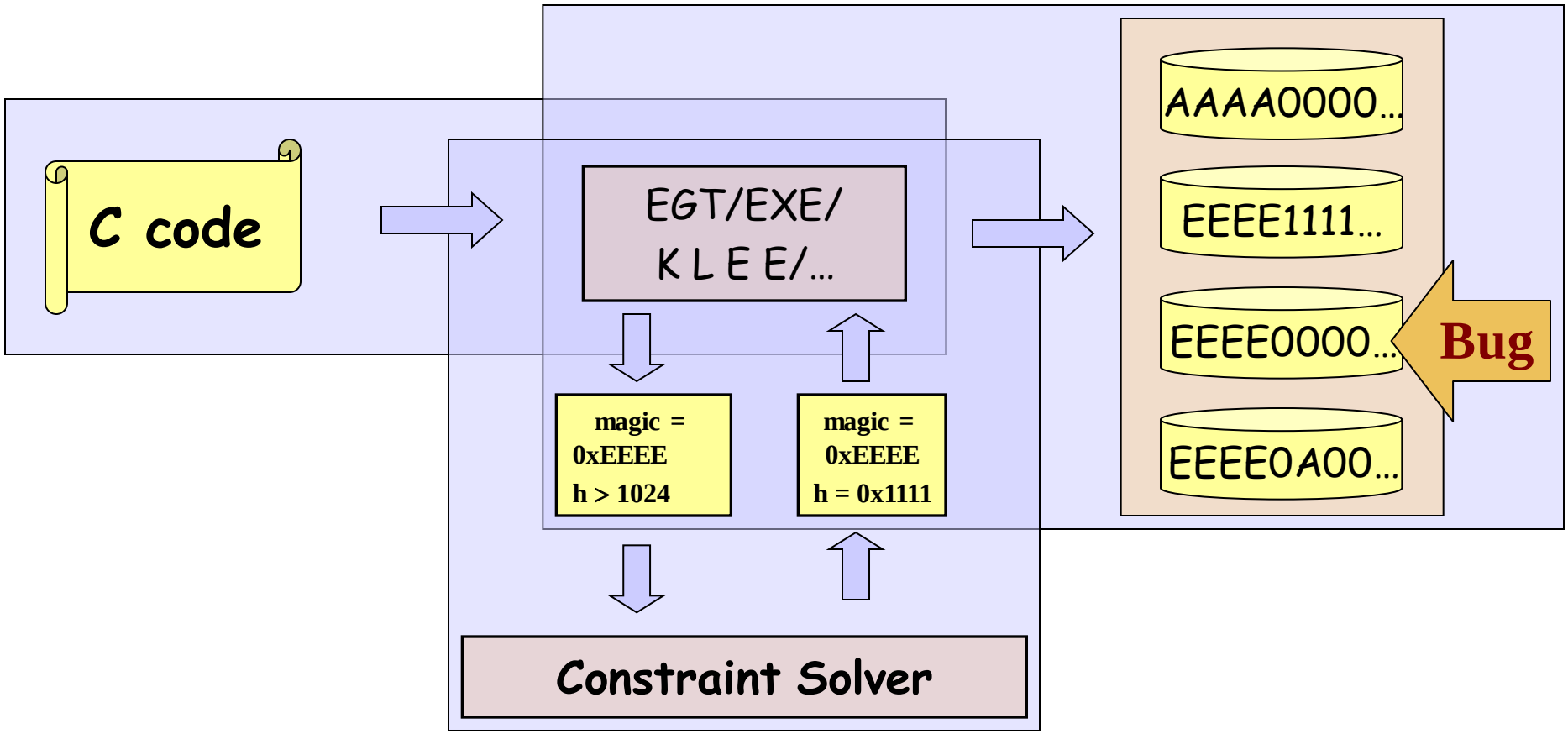
Mixed Concrete/Symbolic Execution

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

Tools: EGT, EXE, KLEE, ZESTI, etc.



KLEE [http://klee.llvm.org]

- Symbolic execution tool based on the LLVM compiler infrastructure and the STP constraint solver
- Works as a mixed concrete/symbolic interpreter for LLVM bitcode
- Extensible platform, used and extended by many groups in academia and industry

DEMO

KLEE [http://klee.llvm.org]

```
// #include directives
struct image_t {
  unsigned short magic;
  unsigned short h, sz; // height, size
  char pixels[1018];
};
int main(int argc, char** argv) {
  struct image_t img;
  int fd = open(argv[1], O_RDONLY);
  read(fd, &img, 1024);

  if (img.magic != 0xEEEE)
    return -1;
  if (img.h > 1024)
    return -1;
  unsigned short w = img.sz / img.h;

  return w;
}
```

```
$ klee-gcc -c -g image_viewer.c
$ klee --posix-runtime -write-pcs
  image_viewer.o --sym-files 1 1024 A
...
KLEE: output directory = klee-out-1
(klee-last)
...
KLEE: ERROR: ... divide by zero
...
KLEE: done: generated tests = 4
```

KLEE [http://klee.llvm.org]

```
$ ls klee-last
assembly.ll   test000001.ktest    test000003.ktest
info          test000001.pc       test000003.pc
messages.txt  test000002.ktest    test000004.ktest
run.istats    test000002.pc       test000004.pc
run.stats     test000003.div.err  warnings.txt

$ cat klee-last/test000003.div.err
Error: divide by zero
...
Stack:
...
#1 00000171 in main (argc=5, argv=28823328) at image_viewer.c:29
```

KLEE [http://klee.llvm.org]

```
$ cat klee-last/test000003.pc
...
array A-data[1024] : w32 -> w8 = symbolic
(query [
         ...
         (Eq  61166
             (ReadLSB w16 0 A-data))
         (Eq  0
             (ReadLSB w16 2 A-data))
         ...
)
```

KLEE [http://klee.llvm.org]

```
$ klee-replay --create-files-only klee-last/test000003.ktest
[File A created]

$ xxd -g 1 -l 10 A
0000000: ee ee 00 00 00 00 00 00 00 00                    ..........

$ gcc -o image_viewer image_viewer.c
[image_viewer created]

$ ./image_viewer A
Floating point exception (core dumped)
```

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.

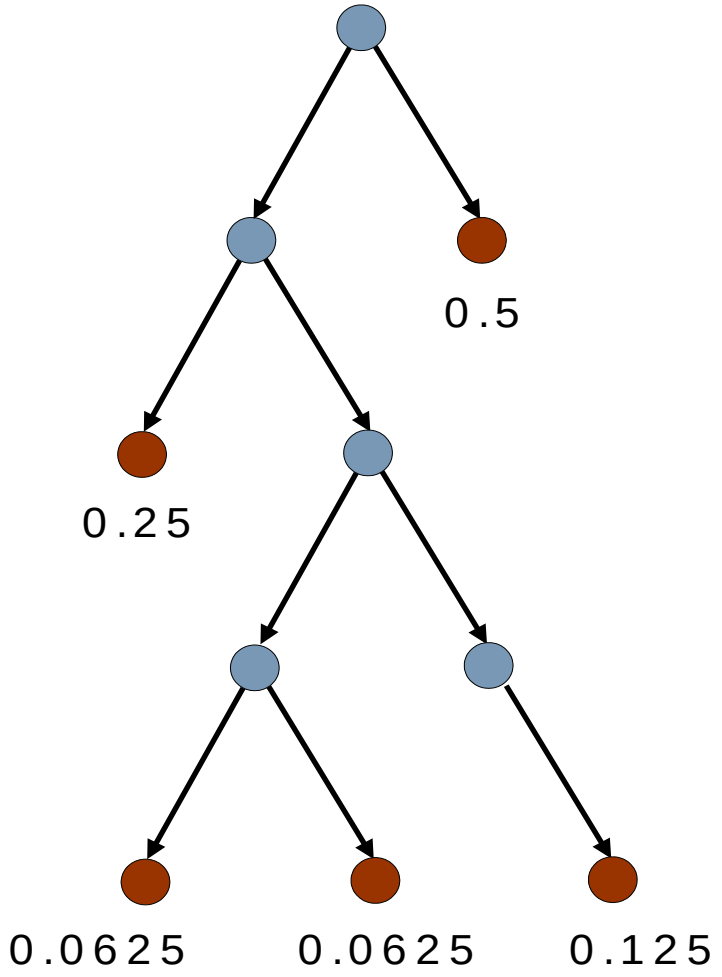
[CCS'06, OSDI'08, ICSE'12, etc.]

Random Path Selection

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

---

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



Which Search Heuristic?

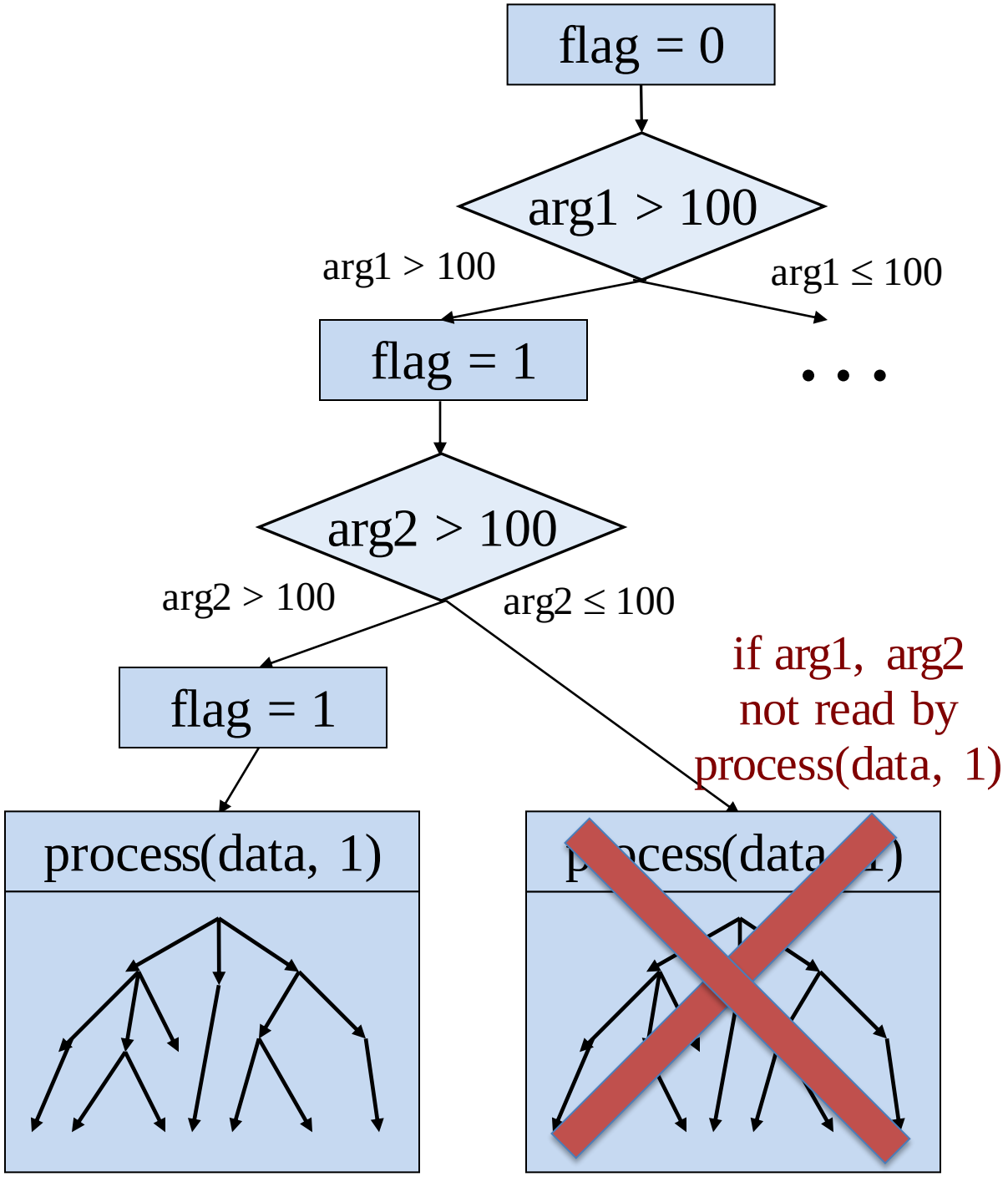
Our latest tool KLEE uses multiple heuristics in a round-robin fashion, to protect against individual heuristics getting stuck in a local maximum.

Eliminating Redundant Paths

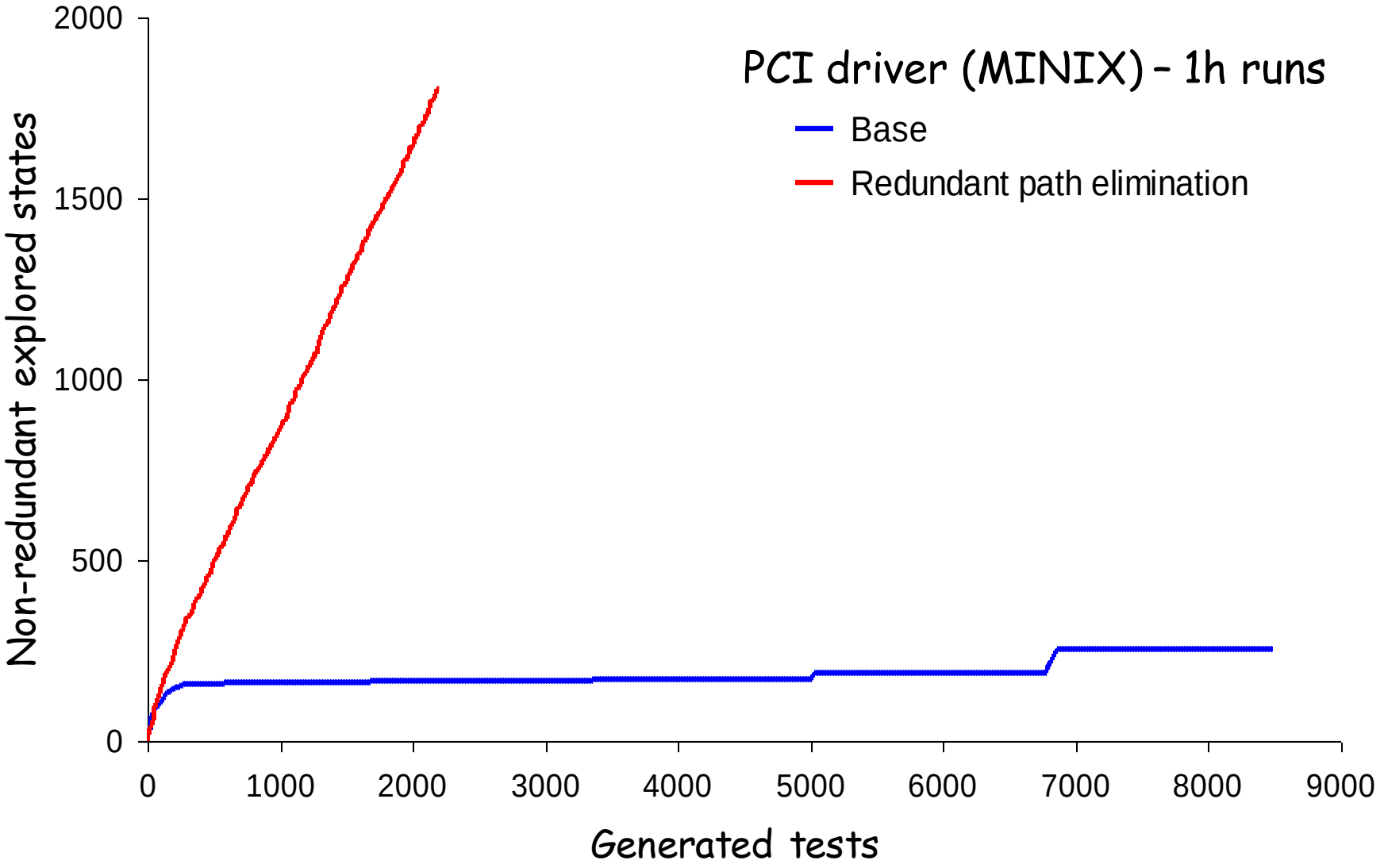
- If two paths reach the same program point with the same constraint sets, we can prune one of them
- We can discard from the constraint sets of each path those constraints involving memory which is never read again

[TACAS'08]

```
data, arg1, arg2 = *

flag =  0;

if (arg1 > 100)
    flag = 1;

if (arg2 > 100)
    flag = 1;

process(data, flag);
```

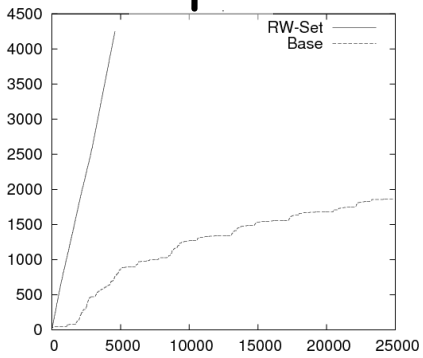


Many Redundant Paths

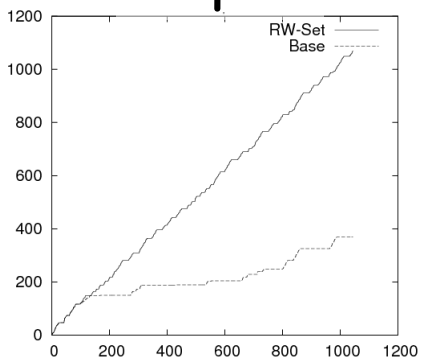


Lots of Redundant Paths

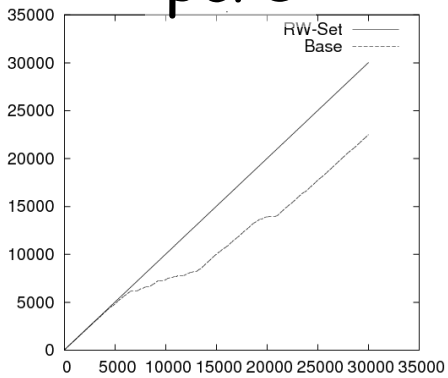
bpf



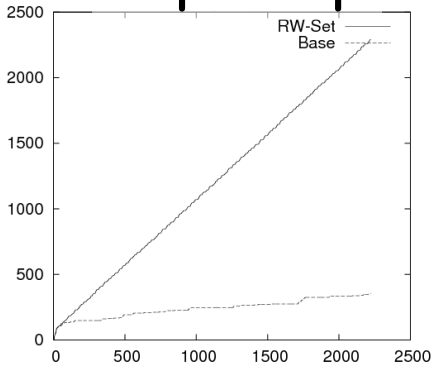
expat



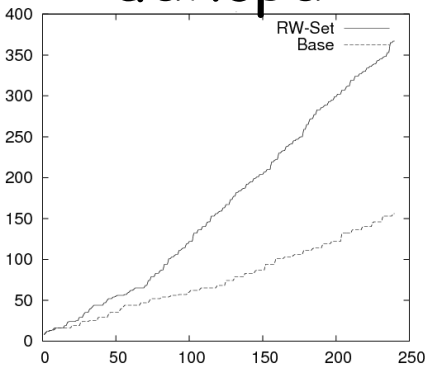
pcre



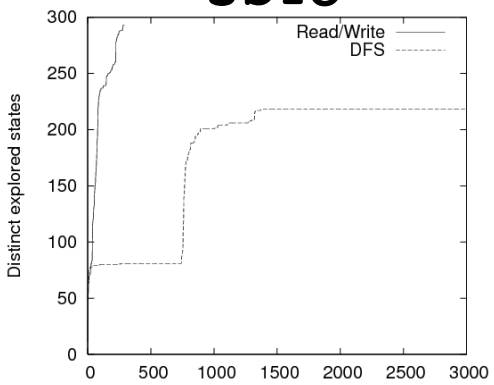
tcpdump



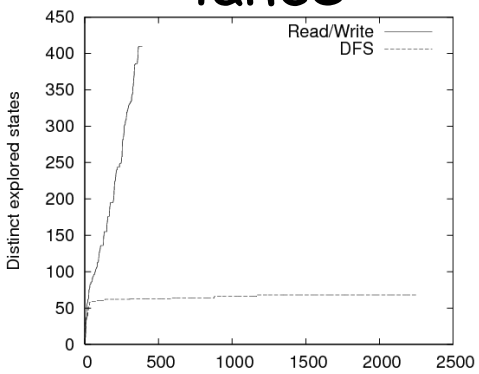
udhcpd



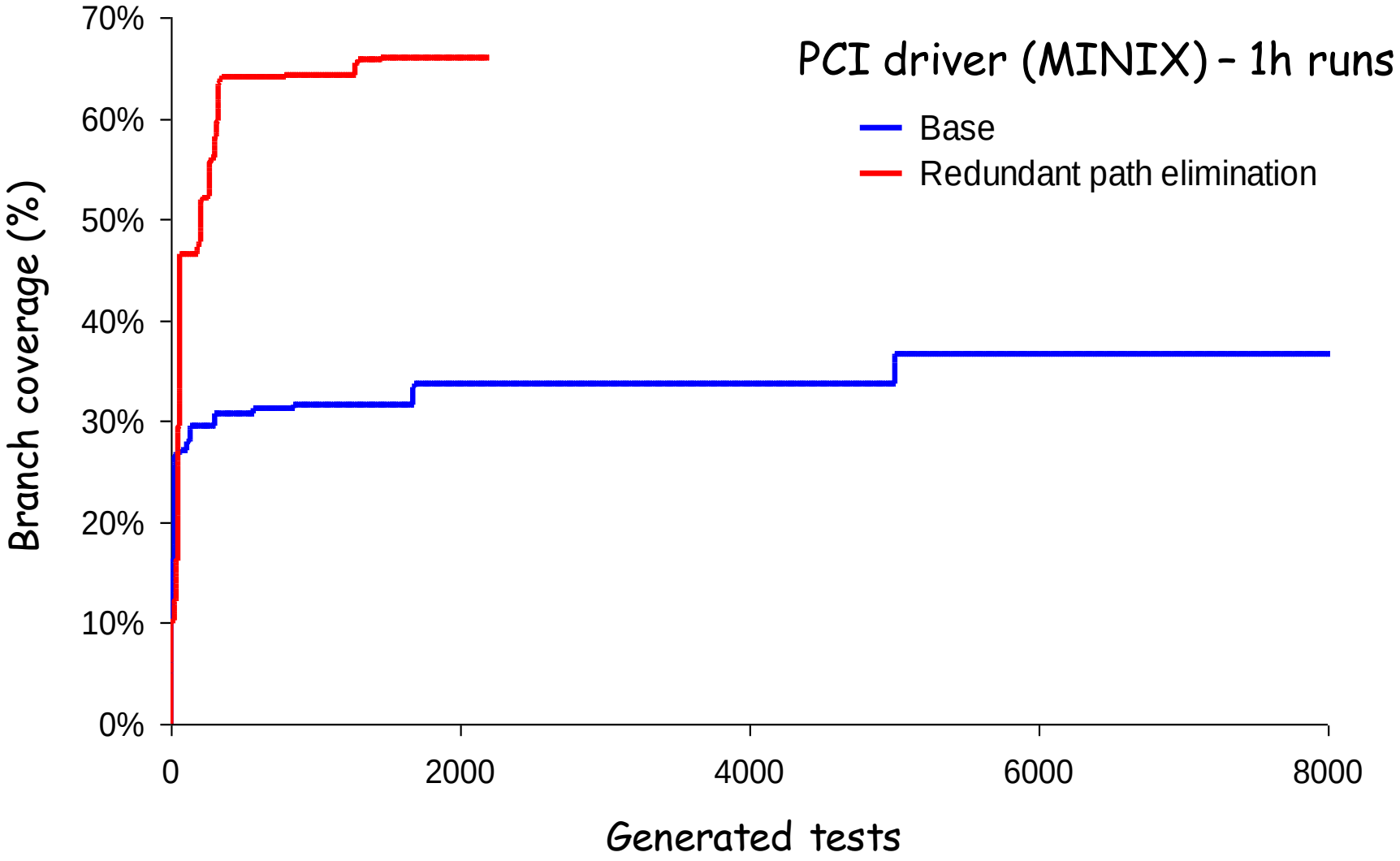
sb16



lance



Redundant Path Elimination



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'12]

Regression Suites

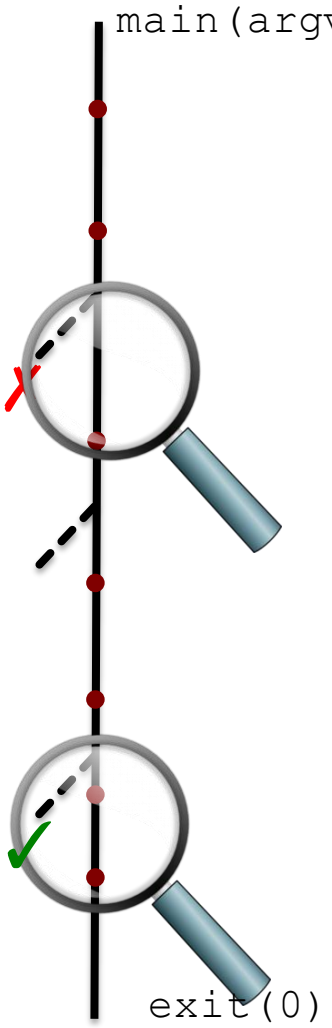
PROS	CONS
• Designed to execute interesting program paths	• Execute each path with a single set of inputs
• Often achieve good coverage of different program features	• Often exercise the general case of a program feature, missing corner cases

ZESTI: 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'12]

Multipath Analysis



- dangerous operations
- divergence point

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
- 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

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

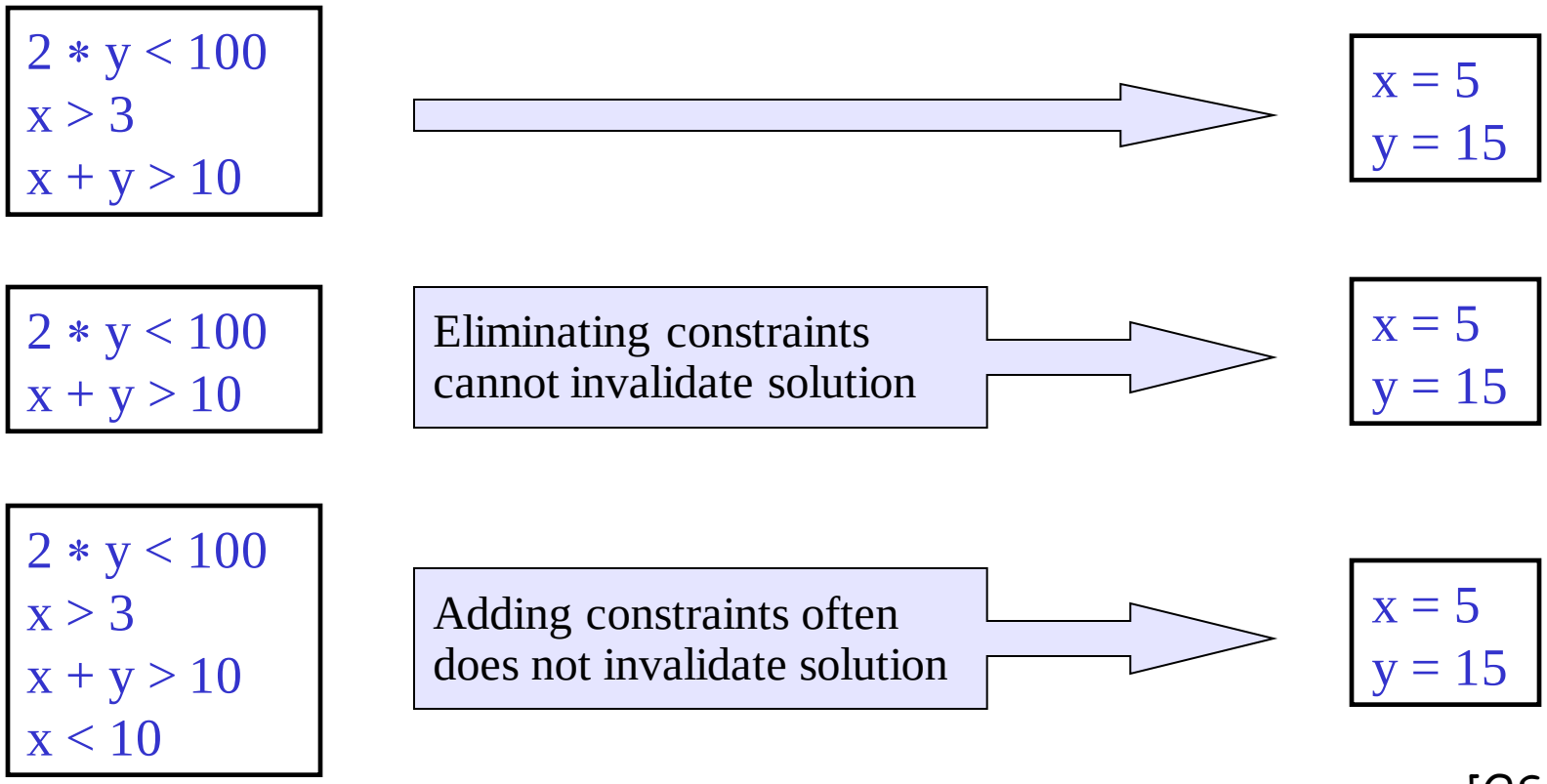
~~z & ~z = z~~

x < 10 ?

[CCS'06]

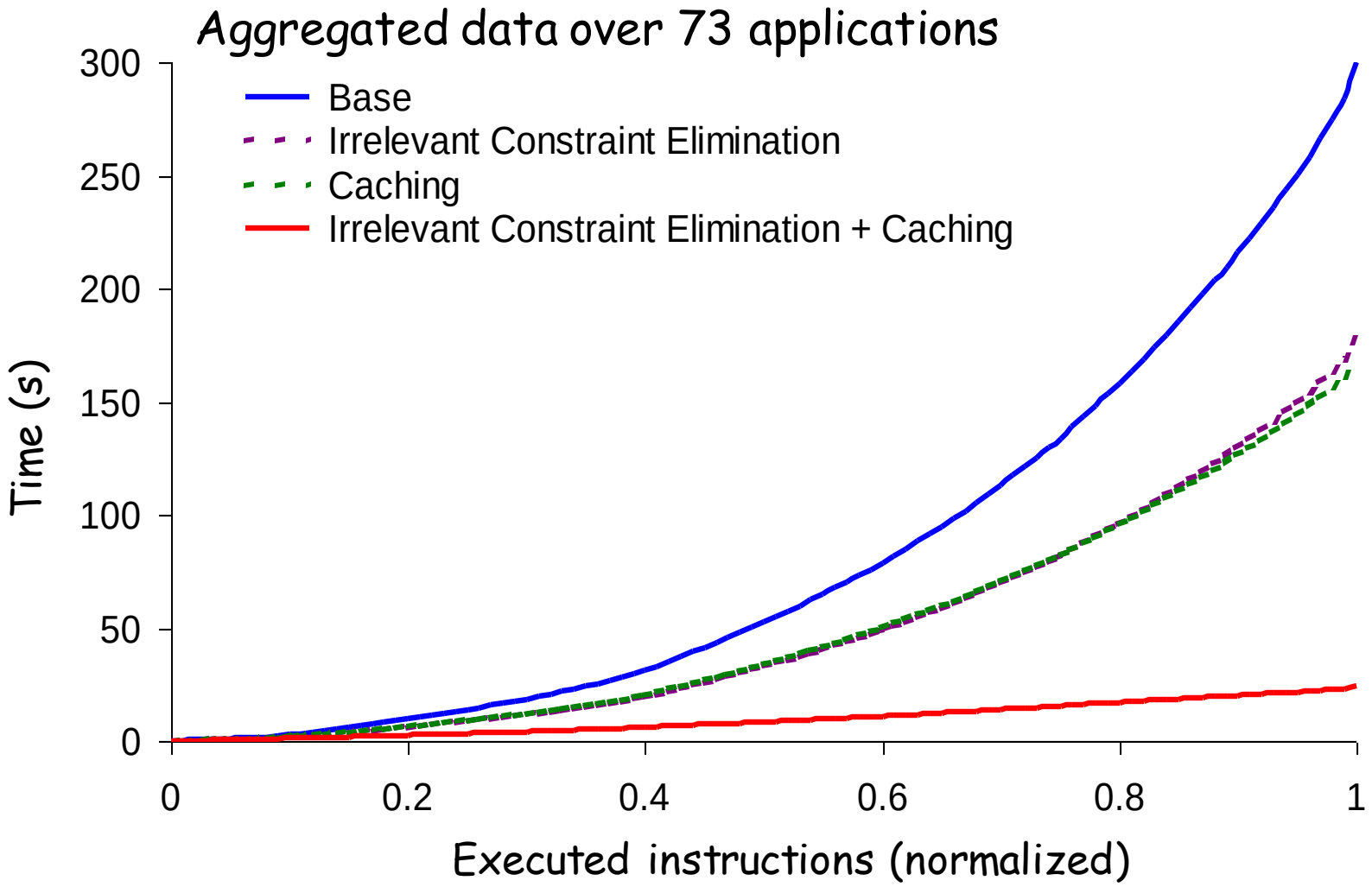
Caching Solutions

- Static set of branches: lots of similar constraint sets



[OSDI'08]

Speedup


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

More on Caching: Instrs/Sec

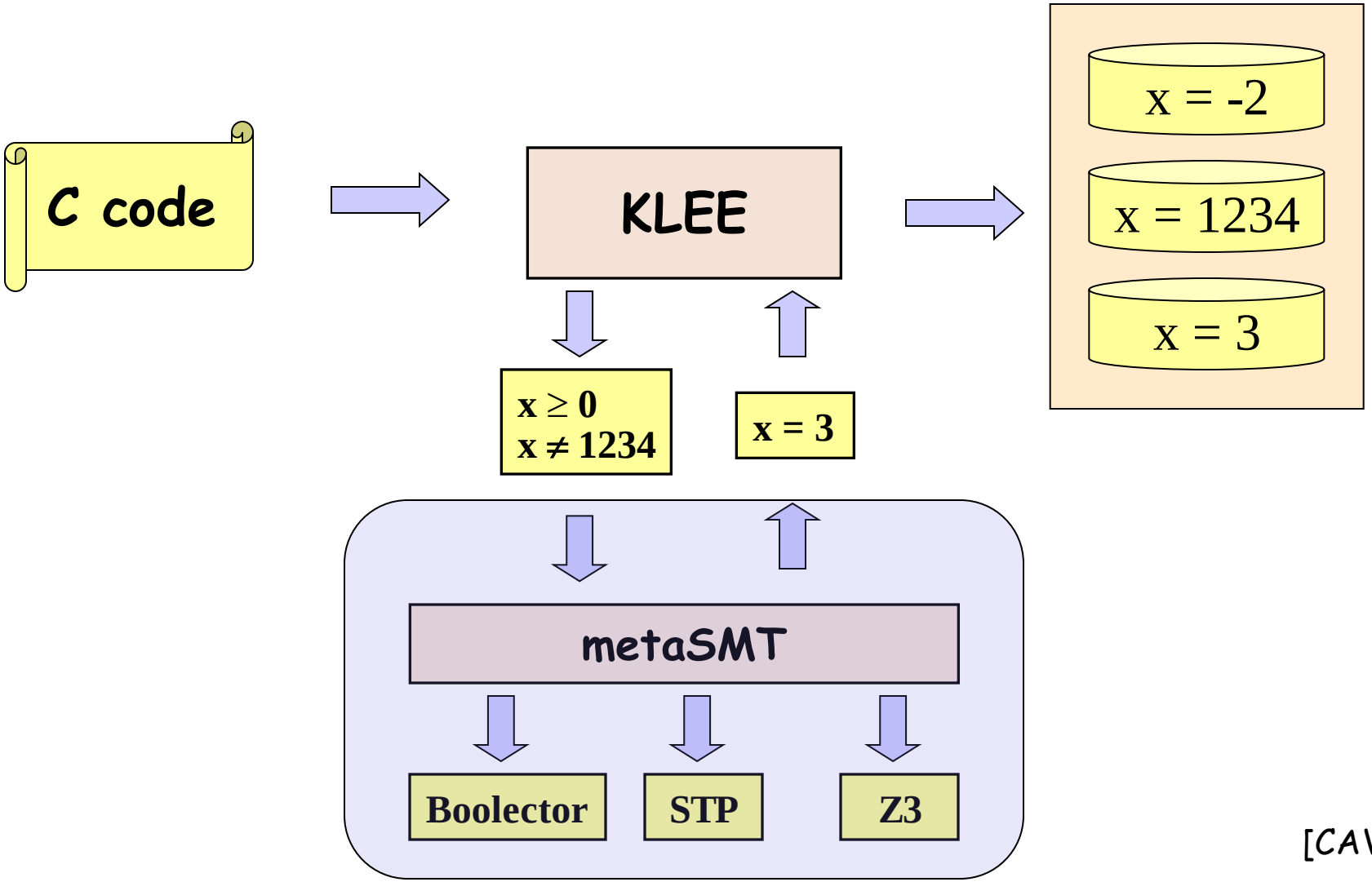
Application	No caching	Caching	Speedup
[	3,914	695	0.17
base64	18,840	20,520	1.08
chmod	12,060	5,360	0.44
comm	73,064	222,113	3.03
csplit	10,682	19,132	1.79
dircolors	8,090	1,019,795	126.05
echo	227	52	0.22
env	21,995	13,246	0.60
factor	1,897	12,119	6.38
join	12,649	1,033,022	81.66
ln	13,420	2,986	0.22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

Usage Scenarios

Successfully used our tools to:

- Automatically generate high-coverage test suites
- Discover generic bugs and security vulnerabilities in complex software
- Perform comprehensive patch testing
- Flag potential semantic bugs via crosschecking
- 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, etc.
Library code	libdwarf, libelf, PCRE, uClibc, etc.
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(`

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

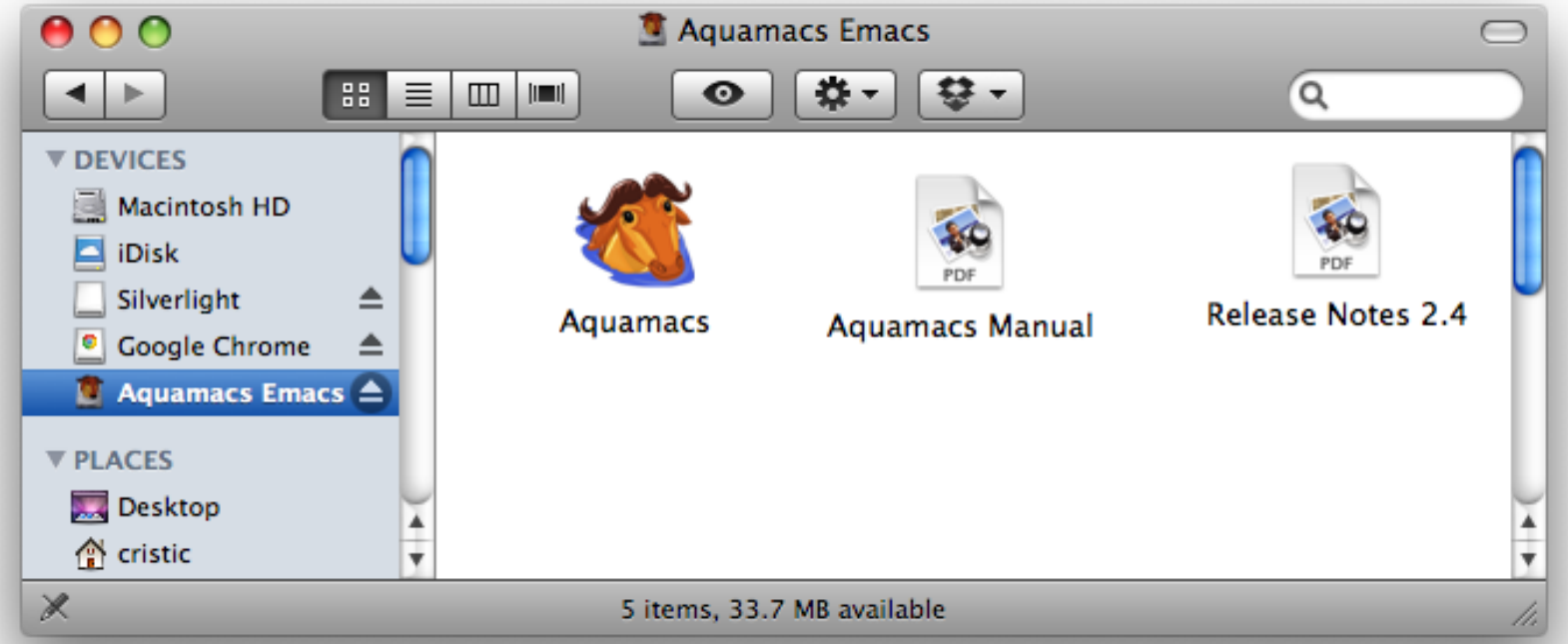
t3.txt: `\n`

t4.txt: `A`

[OSDI 2008, ICSE 2012]

Attack Generation: File Systems

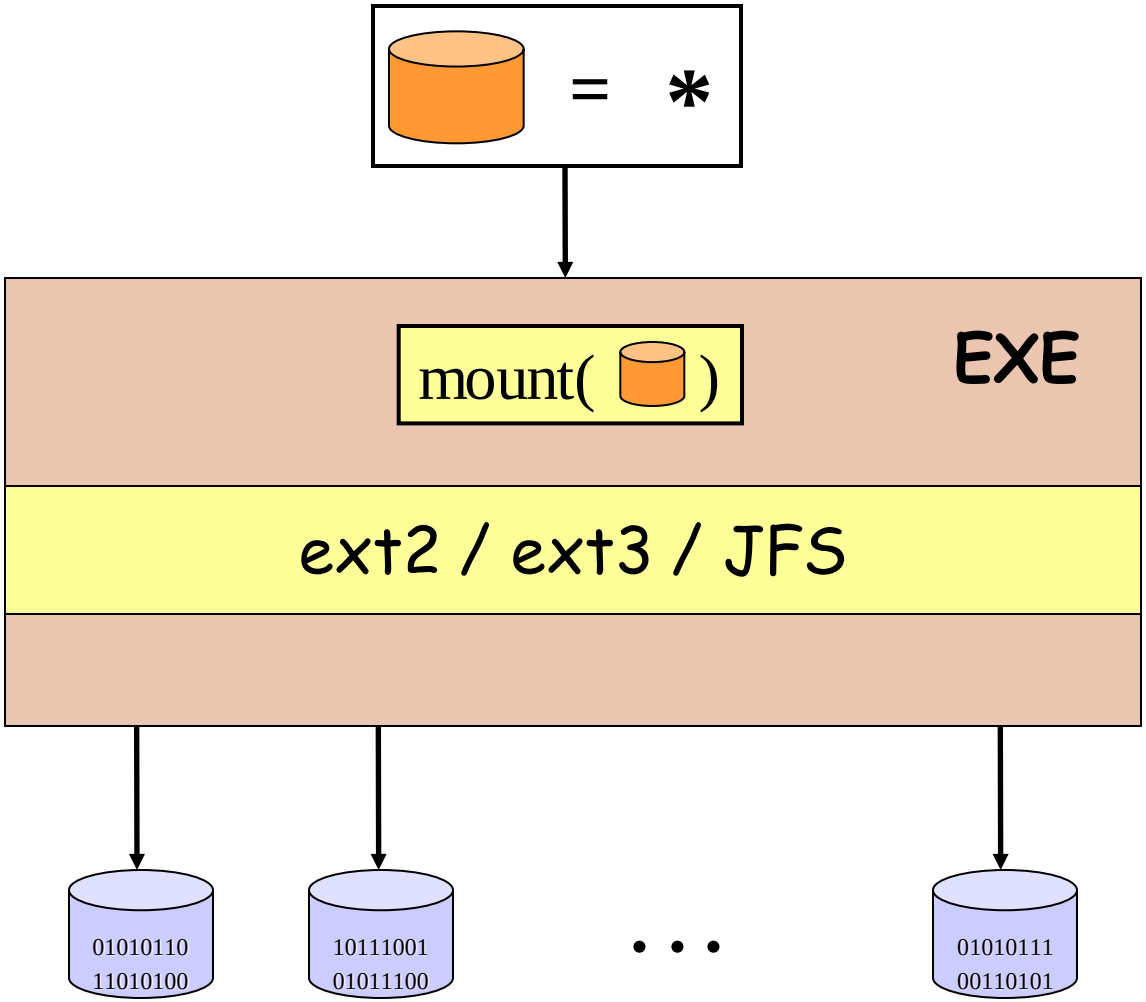
Some modern operating systems allow untrusted users to mount regular files as disk images!



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]

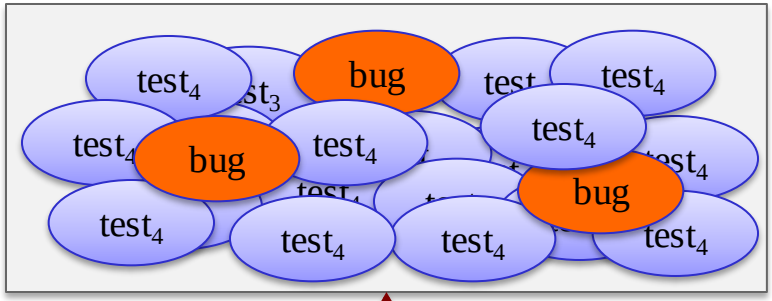
Disk of death (JFS, Linux 2.6.10)

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	. . .				

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

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";
```

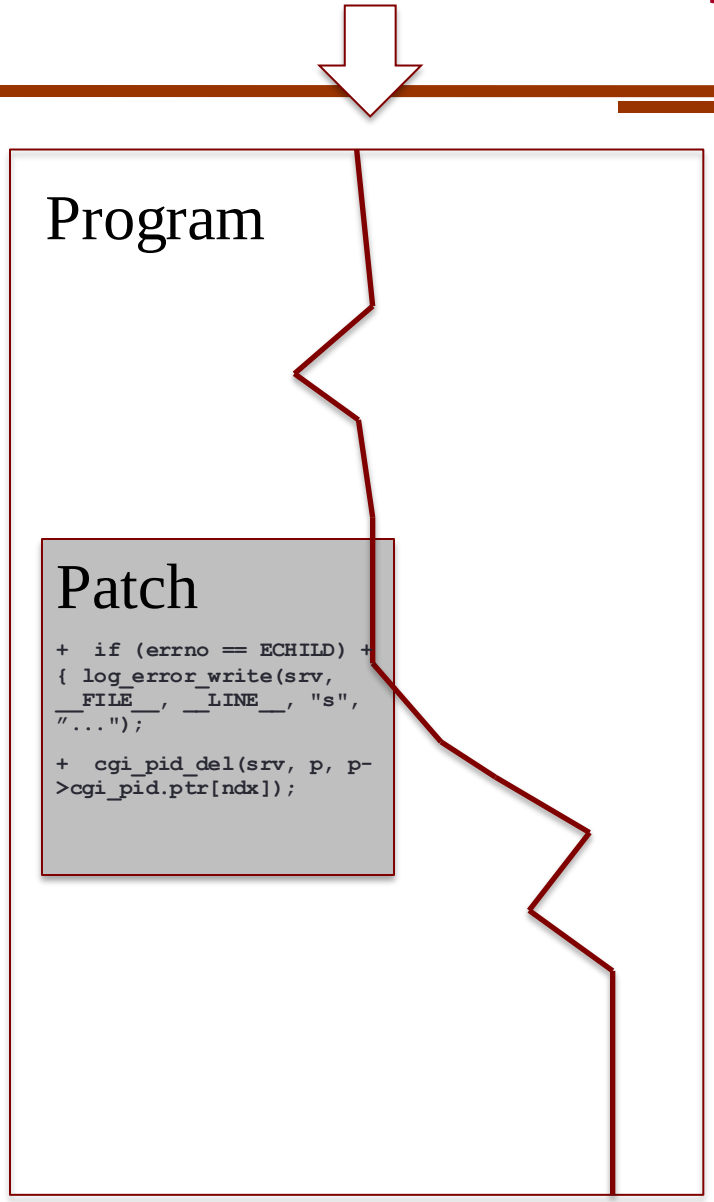


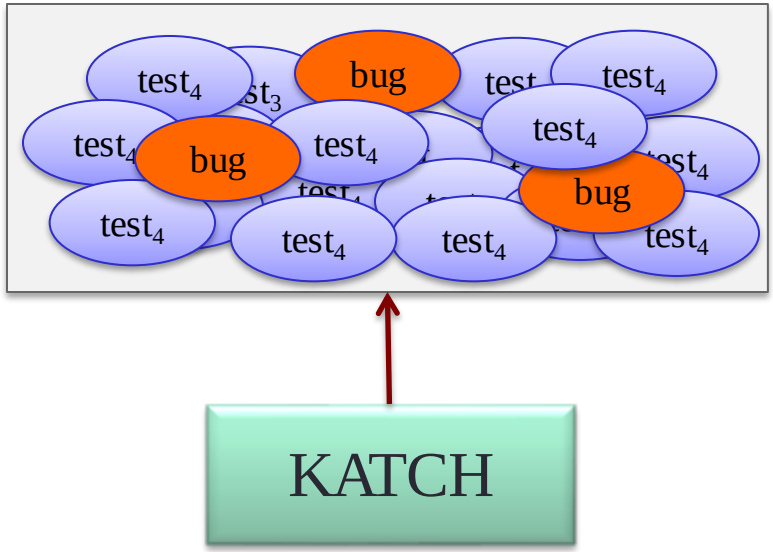
commit

KATCH

[SPIN 2012, ESEC/FSE 2013]

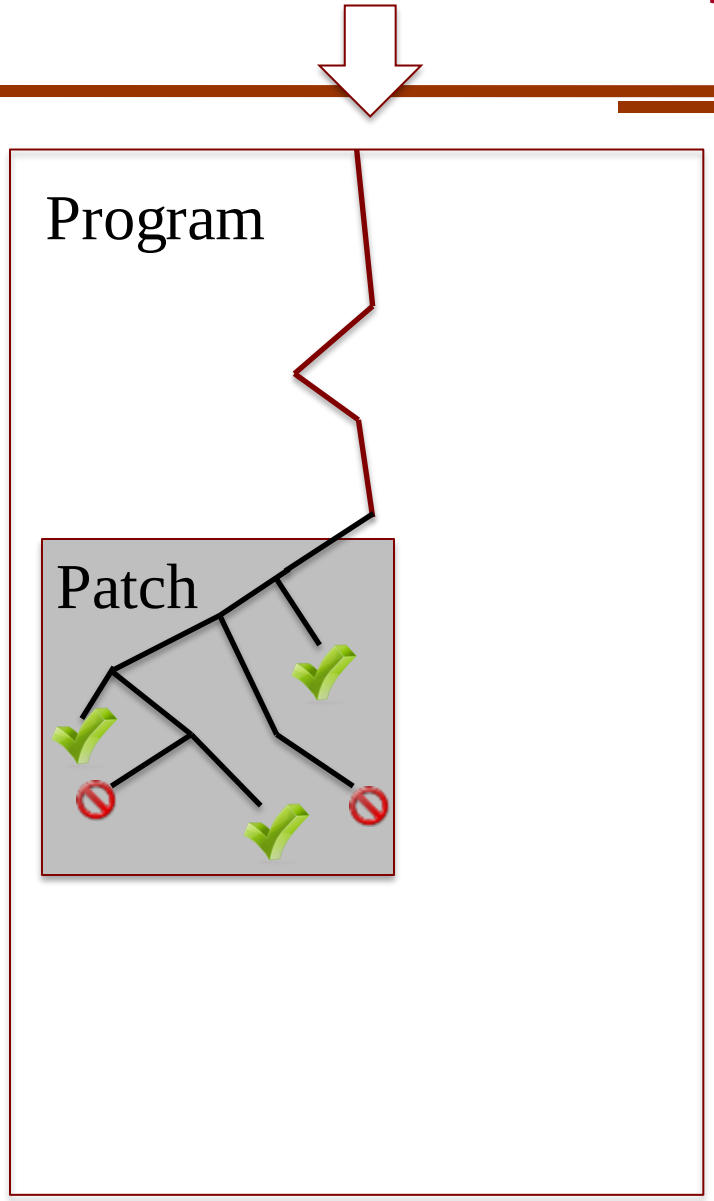
Symbolic Patch Testing

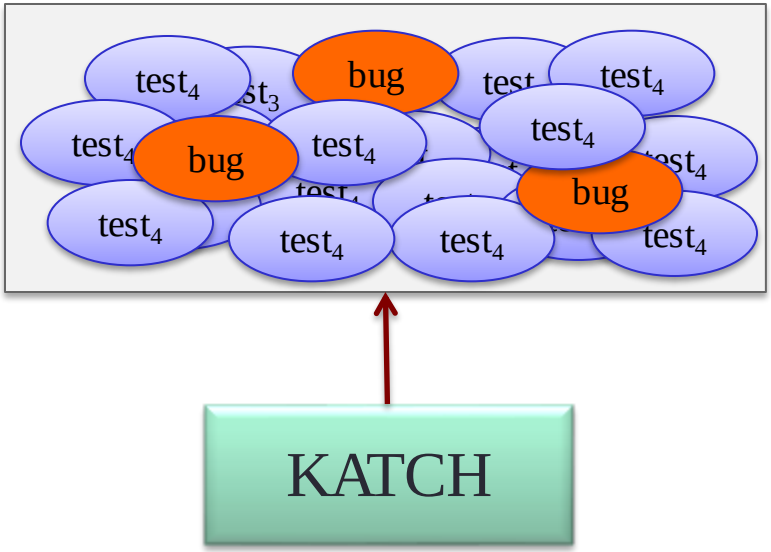




1. Select the regression input closest to the patch (or partially covering it)

Symbolic Patch Testing

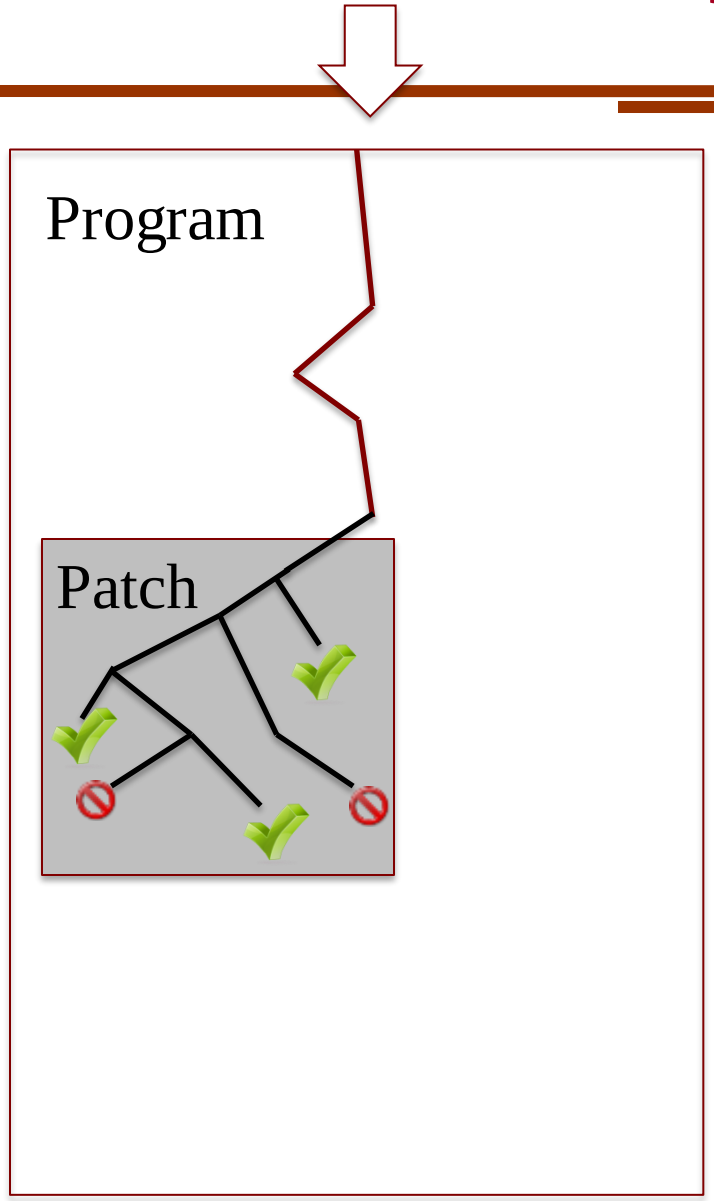


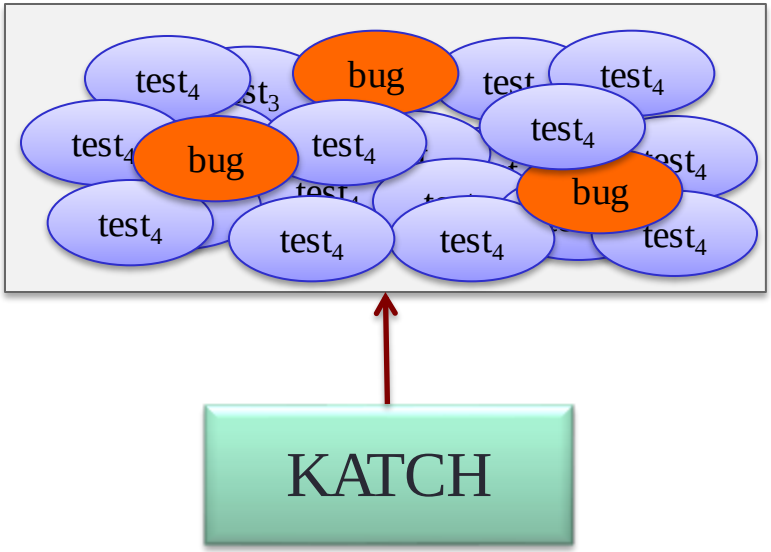


KATCH

2. Greedily drive exploration toward uncovered statements in the patch

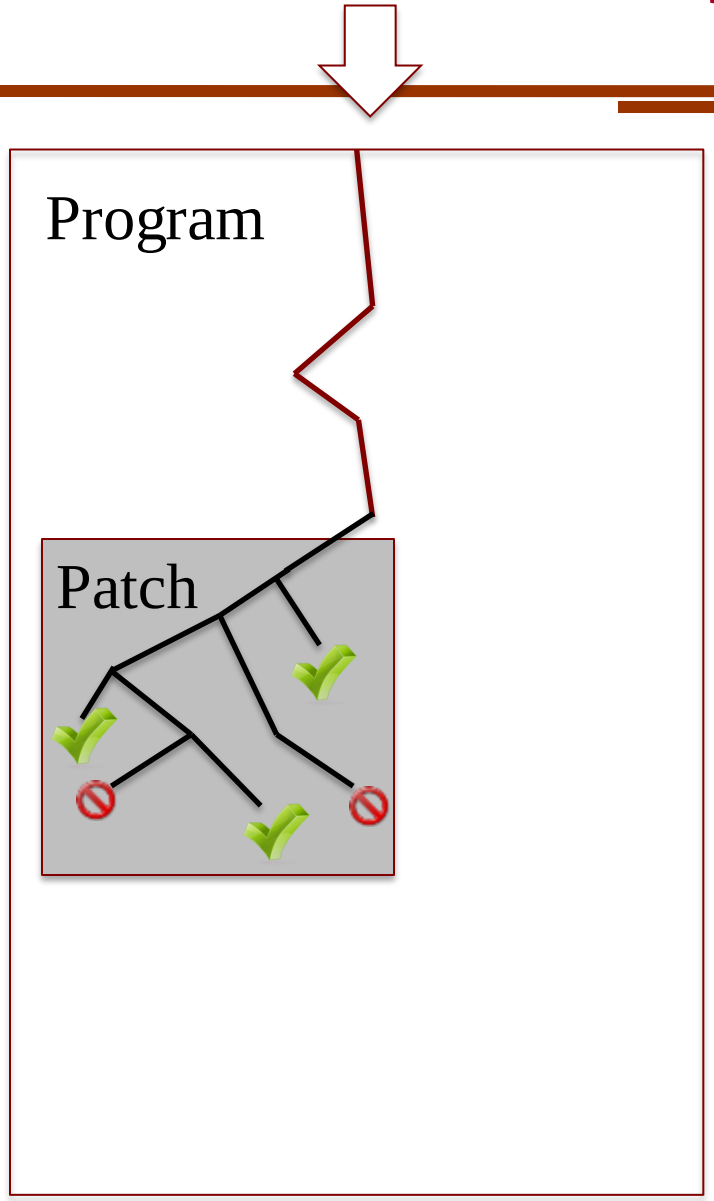
Symbolic Patch Testing

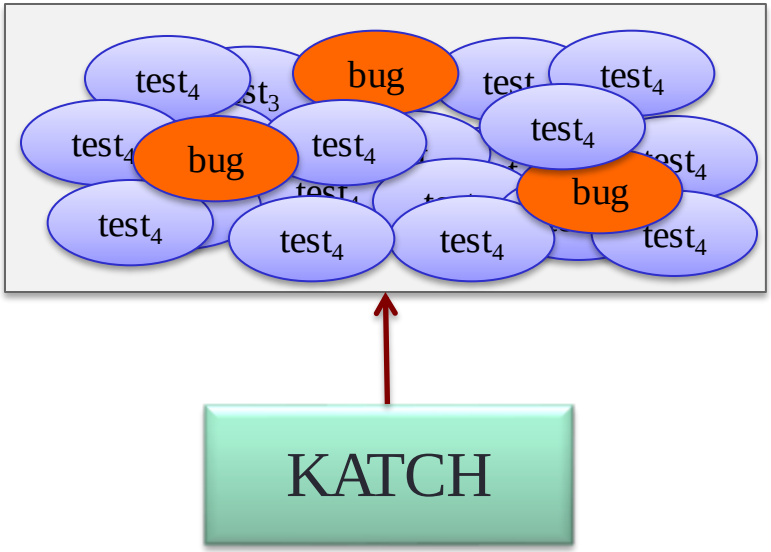




3. If stuck, identify the constraints/bytes that disallow execution to reach the patch, and backtrack

Symbolic Patch Testing





KATCH

Combines **symbolic execution** with various program analyses such as **weakest preconditions** for input selection, and **definition switching** for backtracking

[ESEC/FSE 2013]

Example: Lighttpd r2631

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%)

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

Bug reported and fixed promptly by developers

Extended Evaluation (WiP)

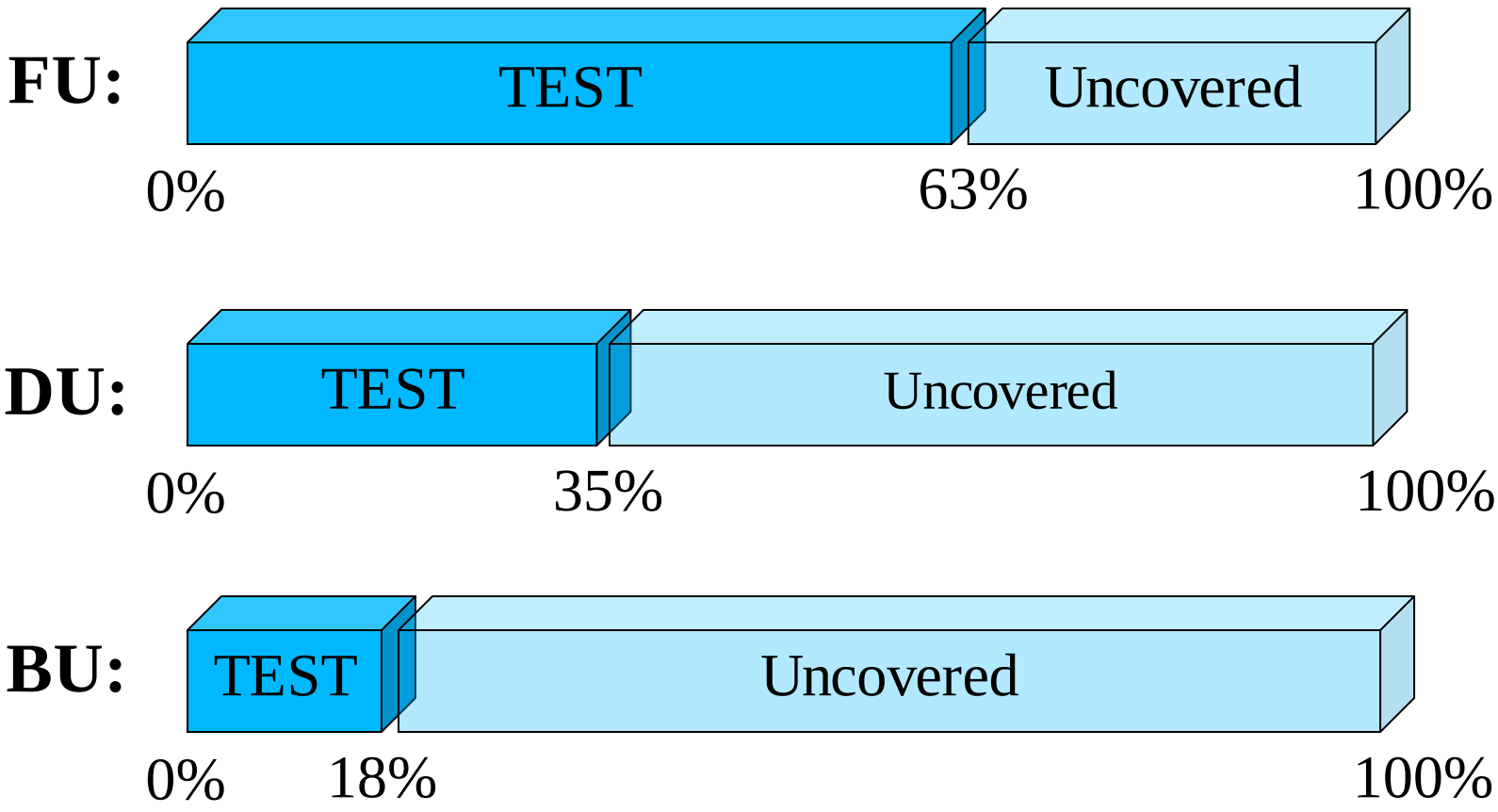
Key evaluation criteria: **no cherry picking!**

- 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**

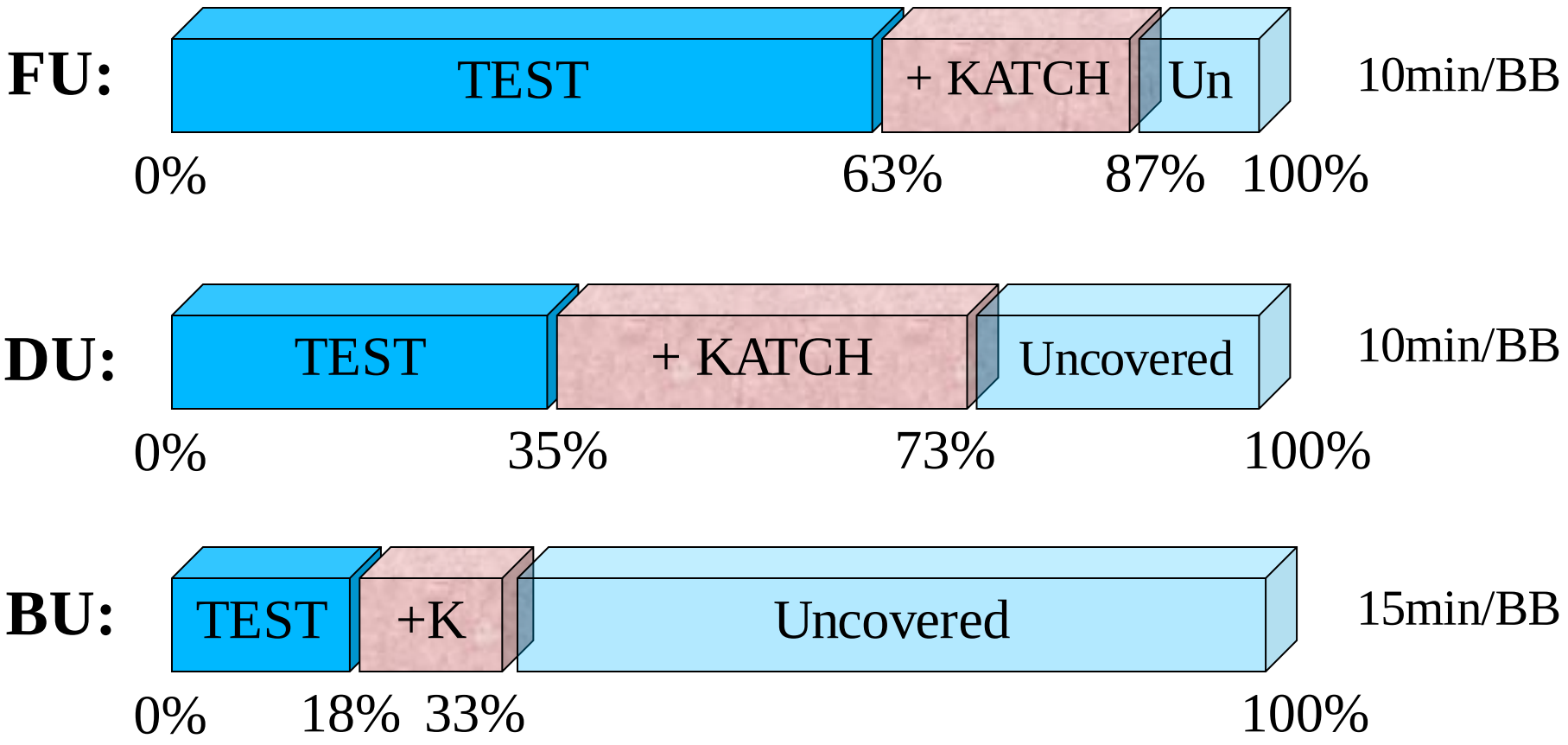
[ESEC/FSE 2013]

Patch Coverage (basic block level)



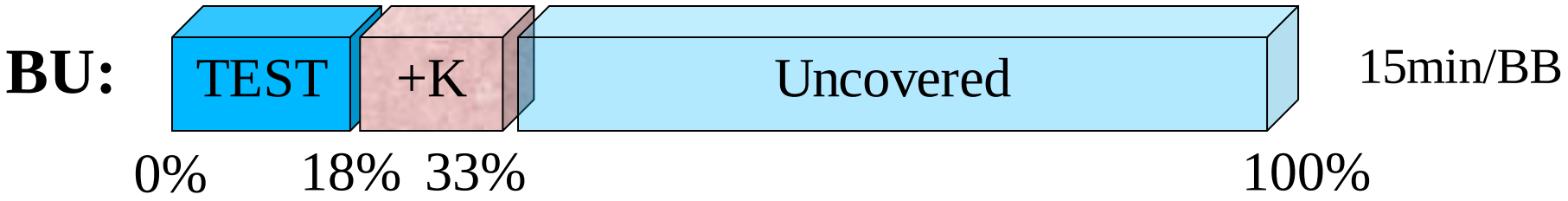
Standard symbolic execution (30min/BB) only added +1.2% to FU

Patch Coverage (basic block level)

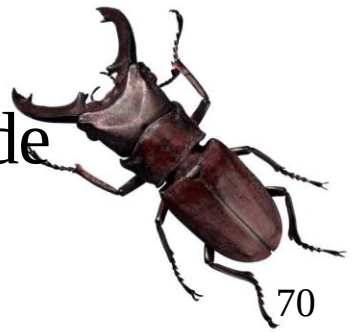


Standard symbolic execution (30min/BB) only added +1.2% to FU

Binutils Bugs



- Found 14 distinct crash bugs
- Unreachable by standard symbolic execution given similar time budget
- 12 bugs still present in latest version of BU
  - Reported and fixed by developers
- 10 bugs found in the patch code itself or in code affected by patch code

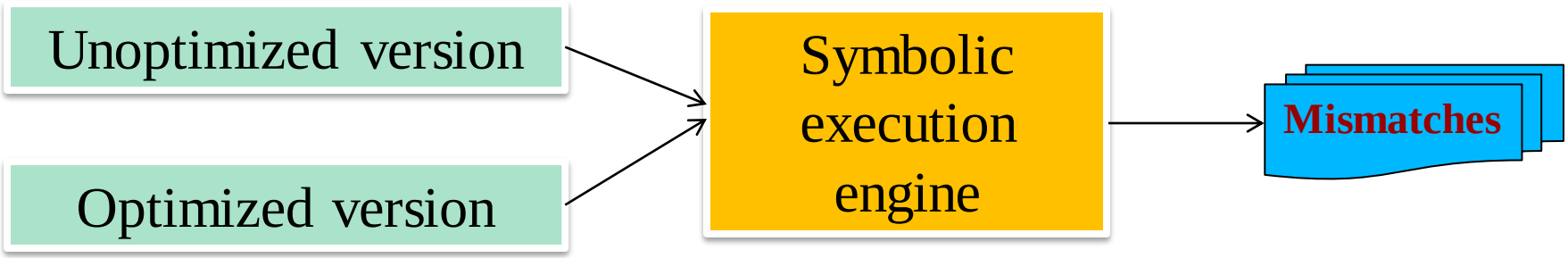


Semantic Errors via Crosschecking (Equivalence Checking)

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 (symbolic) input/output pairs across implementations

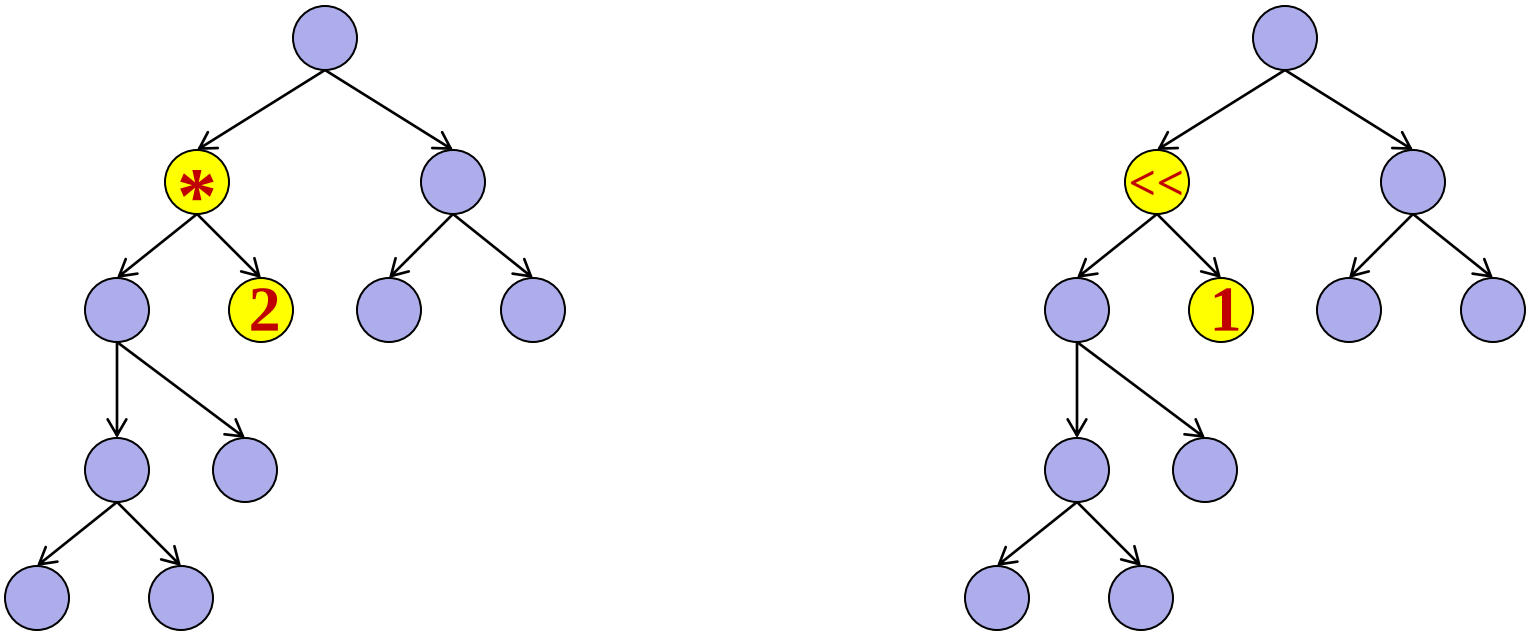
Crosschecking: Advantages

- 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

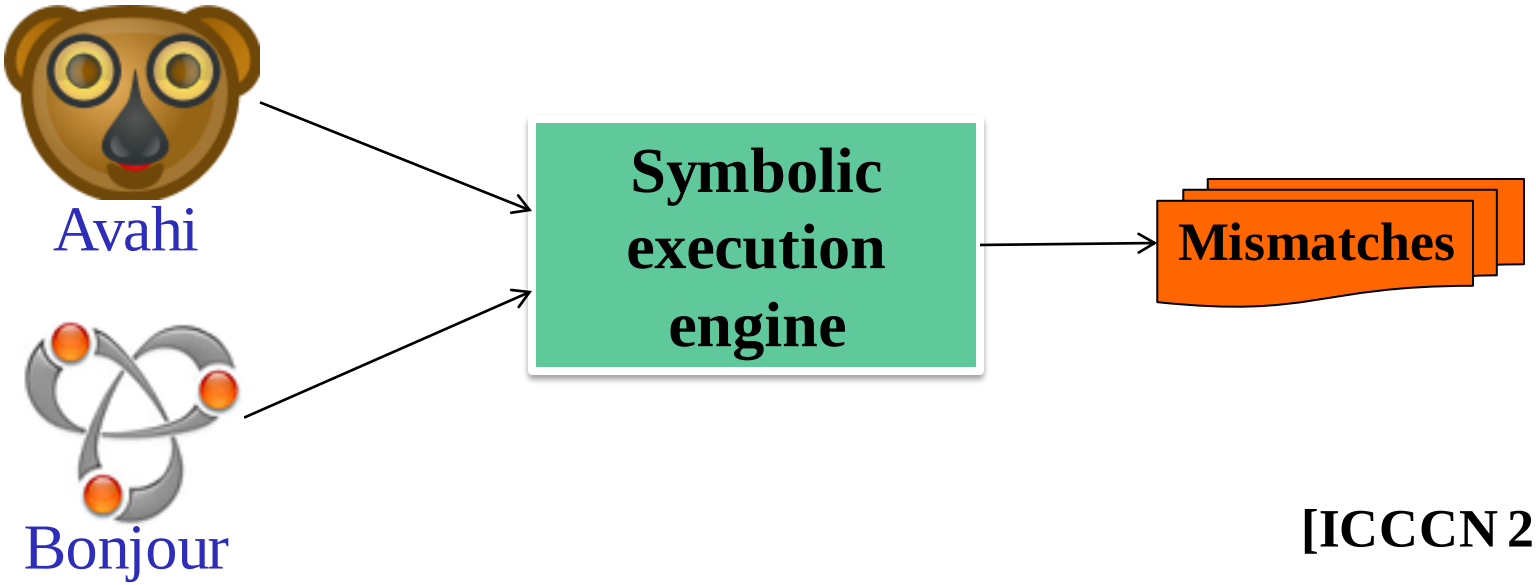
Crosschecking: Advantages



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	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**

Other Crosschecking Studies

UNIX utilities:
desktop vs. embedded

[OSDI 2008]

SIMD/GPU Optimizations:
Scalar vs. SIMD/GPGPU code

[EuroSys 2011, HVC 2011]

DHCP servers:
reference vs. embedded

[WiP]

uDHCPD

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.