Combining Dynamic Symbolic Execution (DSE) with Search-Based Software Testing (SBST)

Cristian Cadar

Department of Computing

Imperial College London

Imperial College London

SBST Keynote

3 June 2014, Hyderabad, India


DSE
SBST

cu·ra·ble
re·search
in order to gather new
known about a subject

“Applying SBST in industry”

“Finding all these bugs has saved millions of dollars to Microsoft… The software running on your PC has been affected by SAGE”

Godefroid, Levin, Molnar
ACM Queue 2012

Diverging Roads?

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

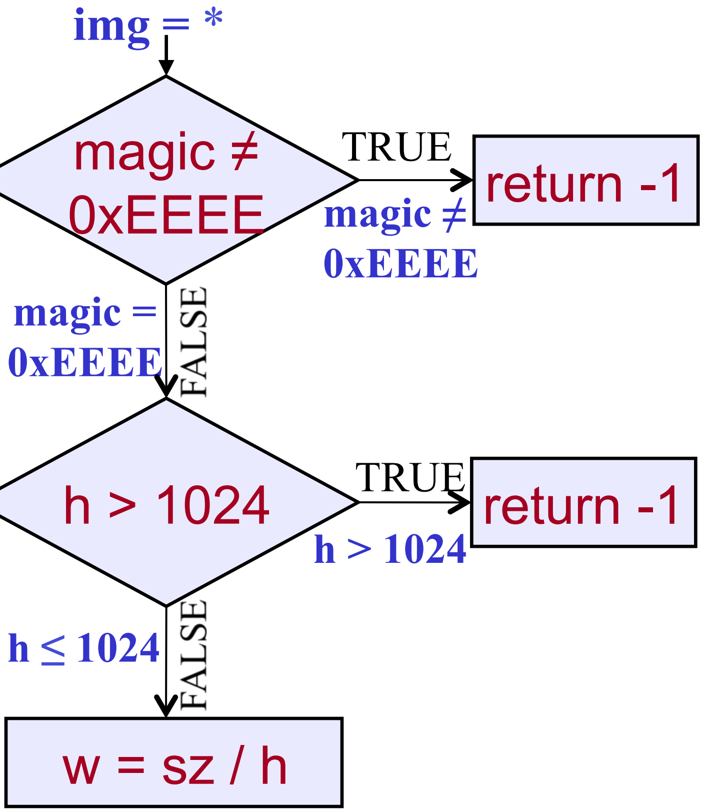
- Received significant interest in the last few years
- Many dynamic symbolic execution/concolic tools available as open-source:
  - **CREST, KLEE, SYMBOLIC JPF**, etc.
- Started to be adopted/tried out in the industry:
  - Microsoft (**SAGE, PEX**)
  - NASA (**SYMBOLIC JPF, KLEE**)
  - Fujitsu (**SYMBOLIC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

Symbolic Execution for Software Testing in Practice: Preliminary Assessment. Cadar, Godefroid, Khurshid, Pasareanu, Sen, Tillmann, Visser, **[ICSE Impact 2011]**

Toy Example

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

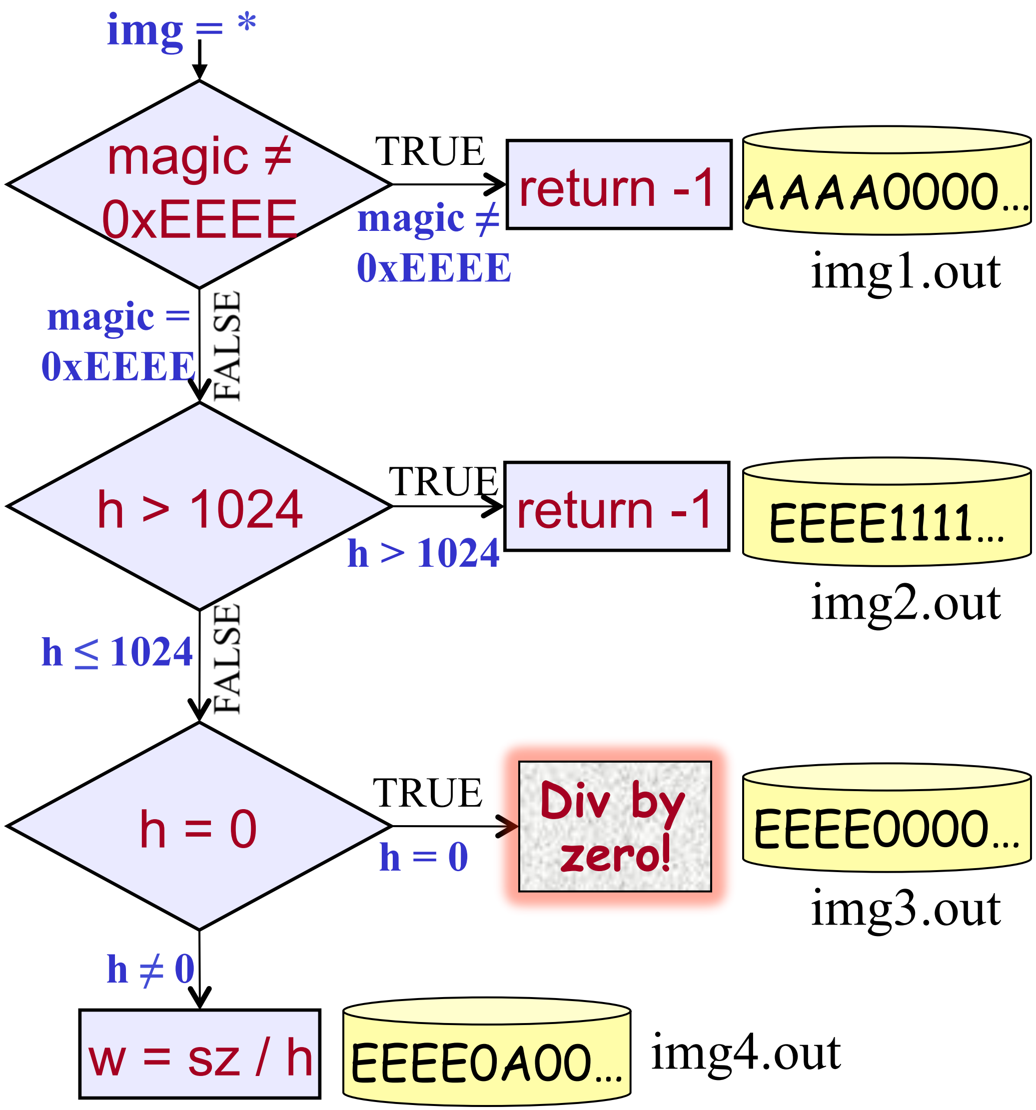
```
int main(int argc, char** argv) {
  ...
  image_t img = read_img(file);
  if (img.magic != 0xEEEE)
    return -1;
  if (img.h > 1024)
    return -1;
  w = img.sz / img.h;
  ...
}
```



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;
  ...
}
```



DSE Applications

Successfully used our DSE tools to:

- Automatically generate high-coverage test suites
- Discover generic bugs and security vulnerabilities in complex software
- Perform comprehensive patch testing
- Find semantic bugs via crosschecking
- Perform bounded verification

Some Applications We Tested
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

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**

[Oakland 2008]

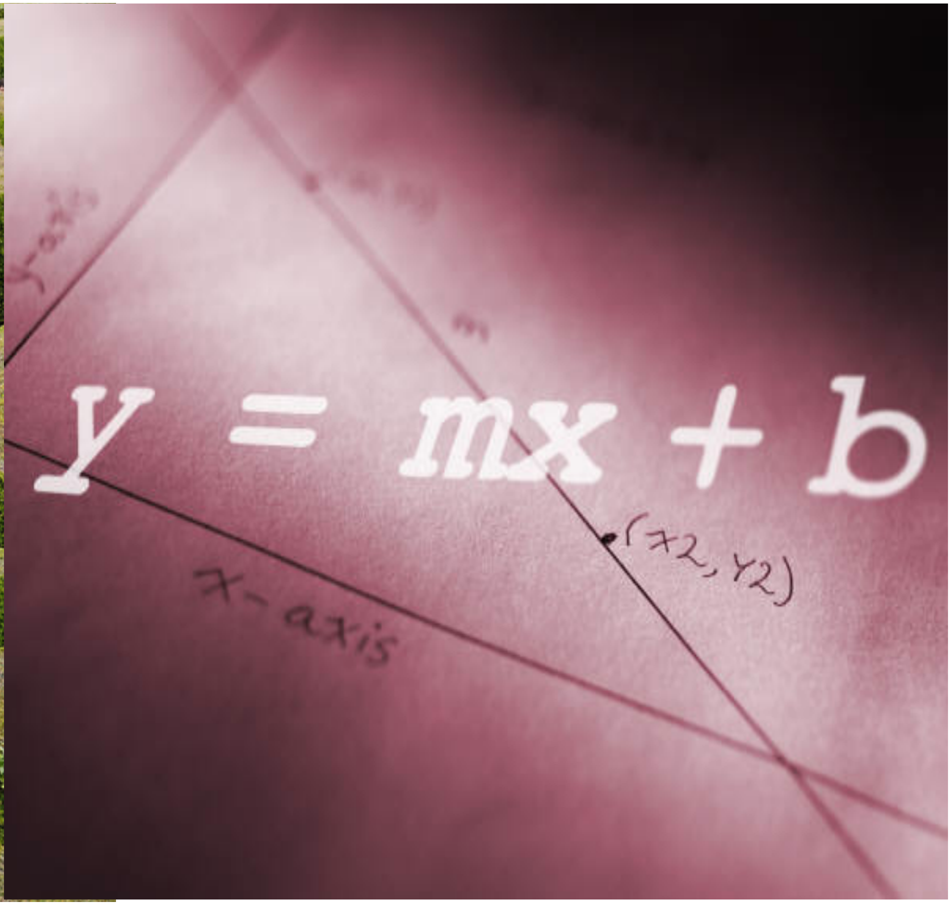
Packet of Death (Bonjour)

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 and fixed by Apple**

[ICCCN 2011]

Scalability Challenges



Path Exploration Challenges

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

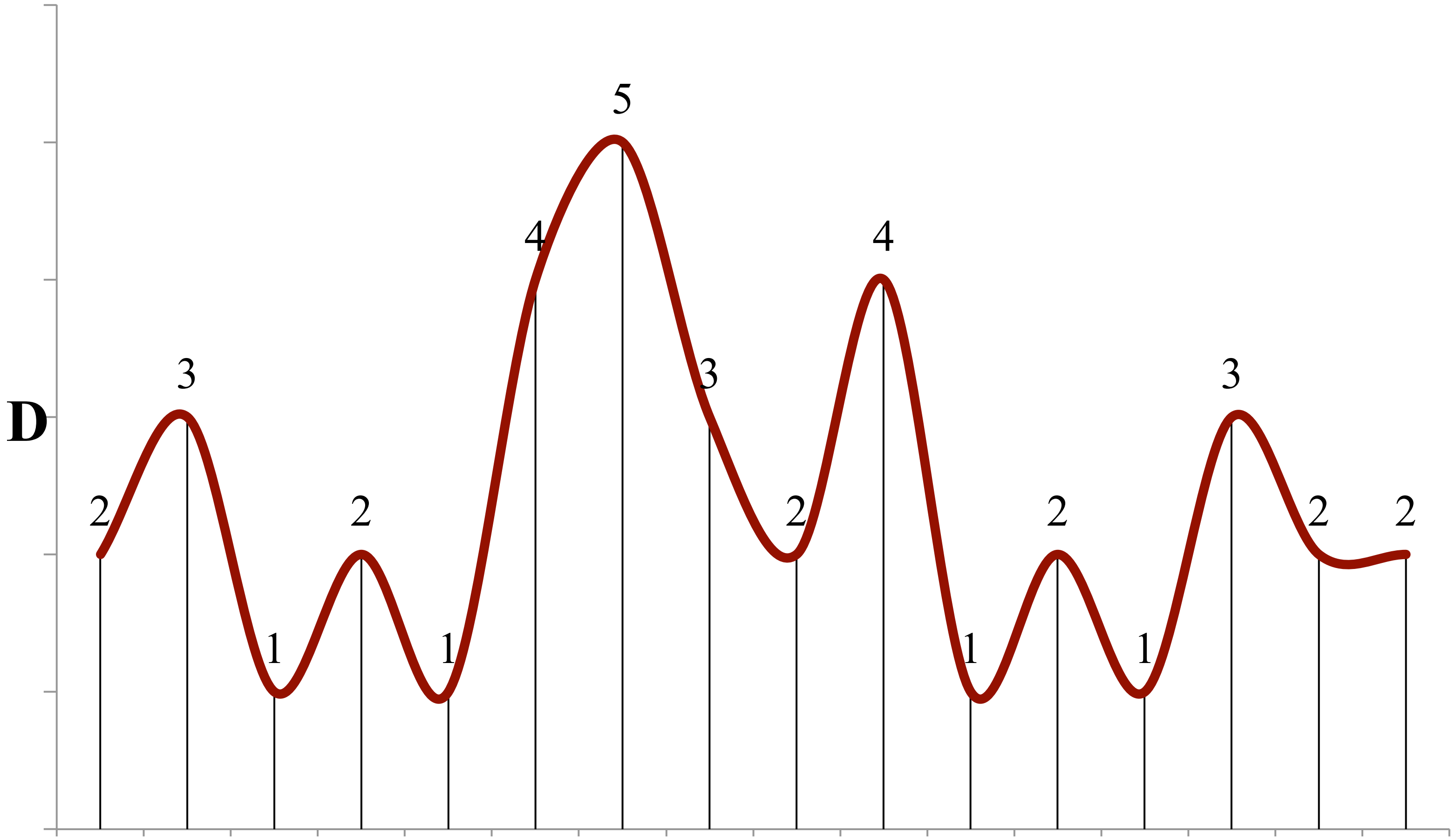
Search Heuristics

Which path should we explore next?

- Coverage-optimized search
- Query time-optimized search
- Best-first search
- Random path search
- etc.

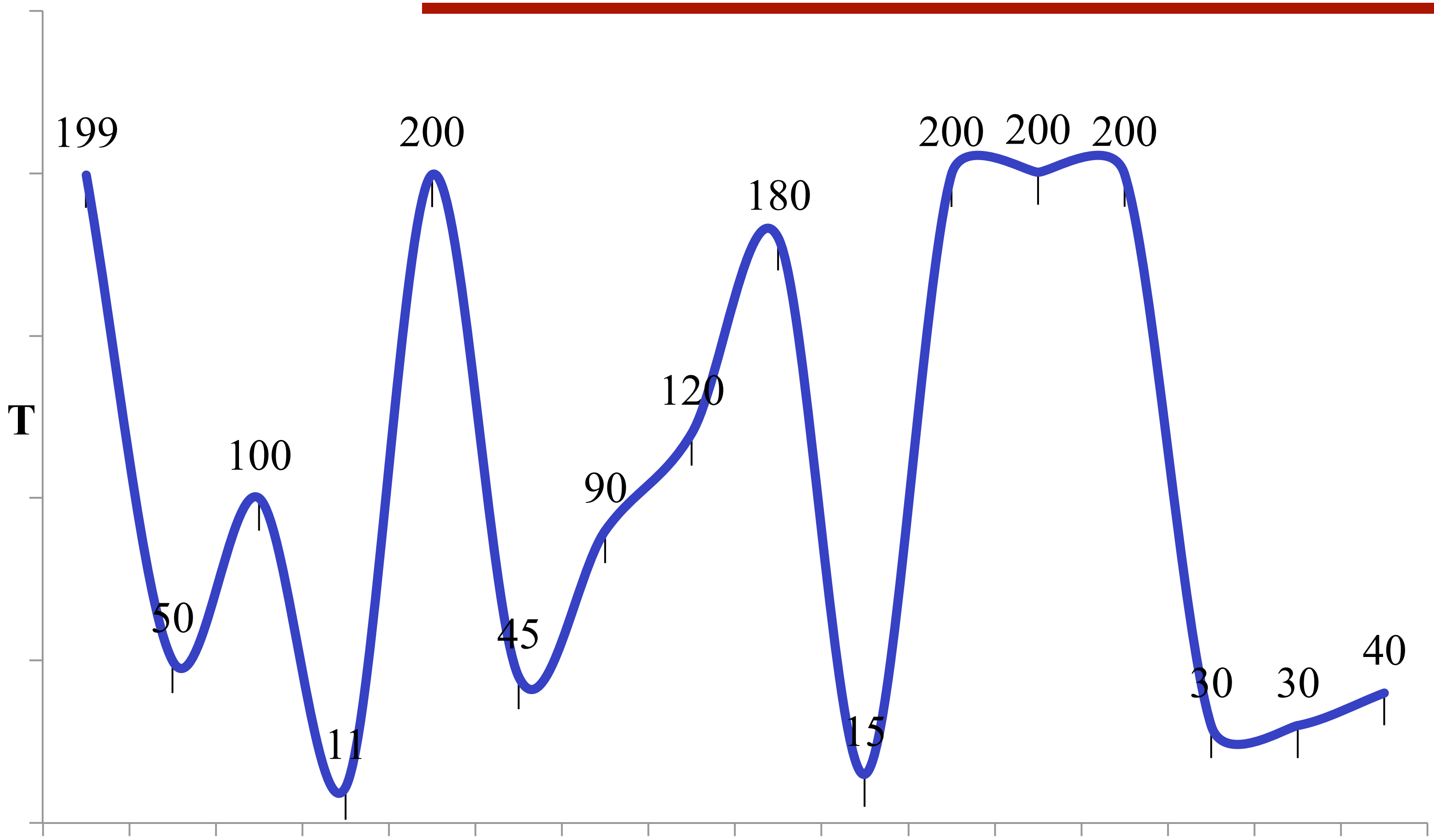
[CCS'06, OSDI'08, ICSE'12, etc.]

Coverage-optimized Search



D = distance to an uncovered instruction
Randomly select a path, with each path weighted by $1/D^2$

Solver Time-optimized Search



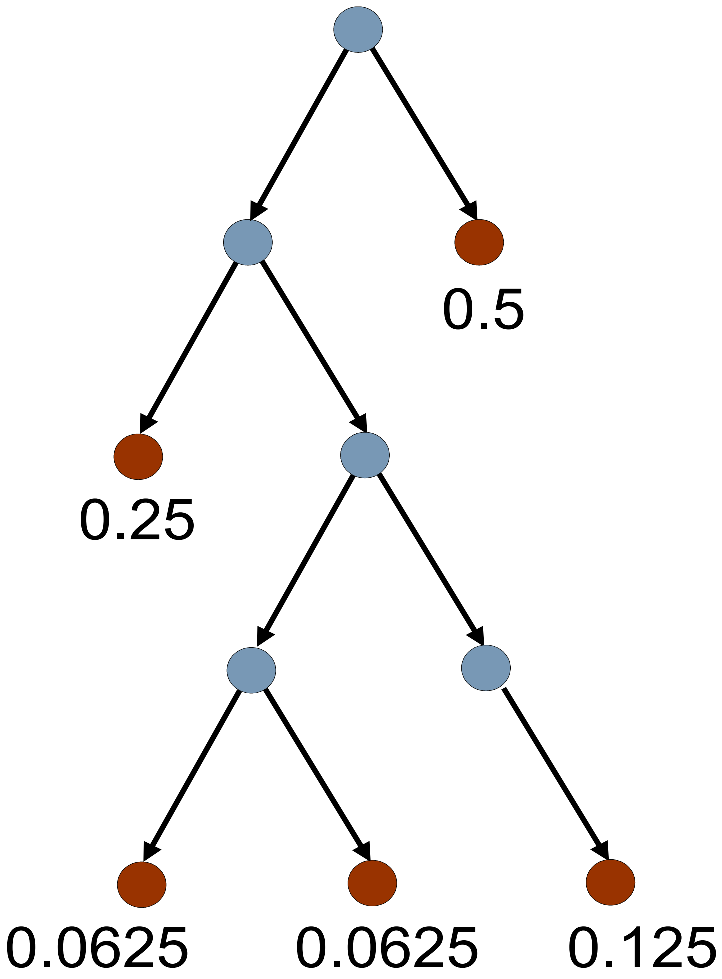
T = time spent in constraint solver
Randomly select a path, with each path weighted by1/T

Random Path Selection

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

---

- NOT randomly selecting a path
- Favors paths high in the tree
  - fewer constraints
- Avoid starvation
  - e.g. symbolic loop



Which Search Heuristic?

We typically use multiple search heuristics in a round-robin fashion, to protect against individual heuristics getting stuck in a local maximum.

Can SBST Help?

- Search heuristics key to the success of DSE
- Heuristics are at the very core of SBST

About the Workshop

Search-Based Software Testing (SBST) is the application of optimizing search techniques (for example, Genetic Algorithms) to solve problems in software testing. SBST is used to

- What are the SBST lessons applicable here?

Seeding in Symbolic Execution
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

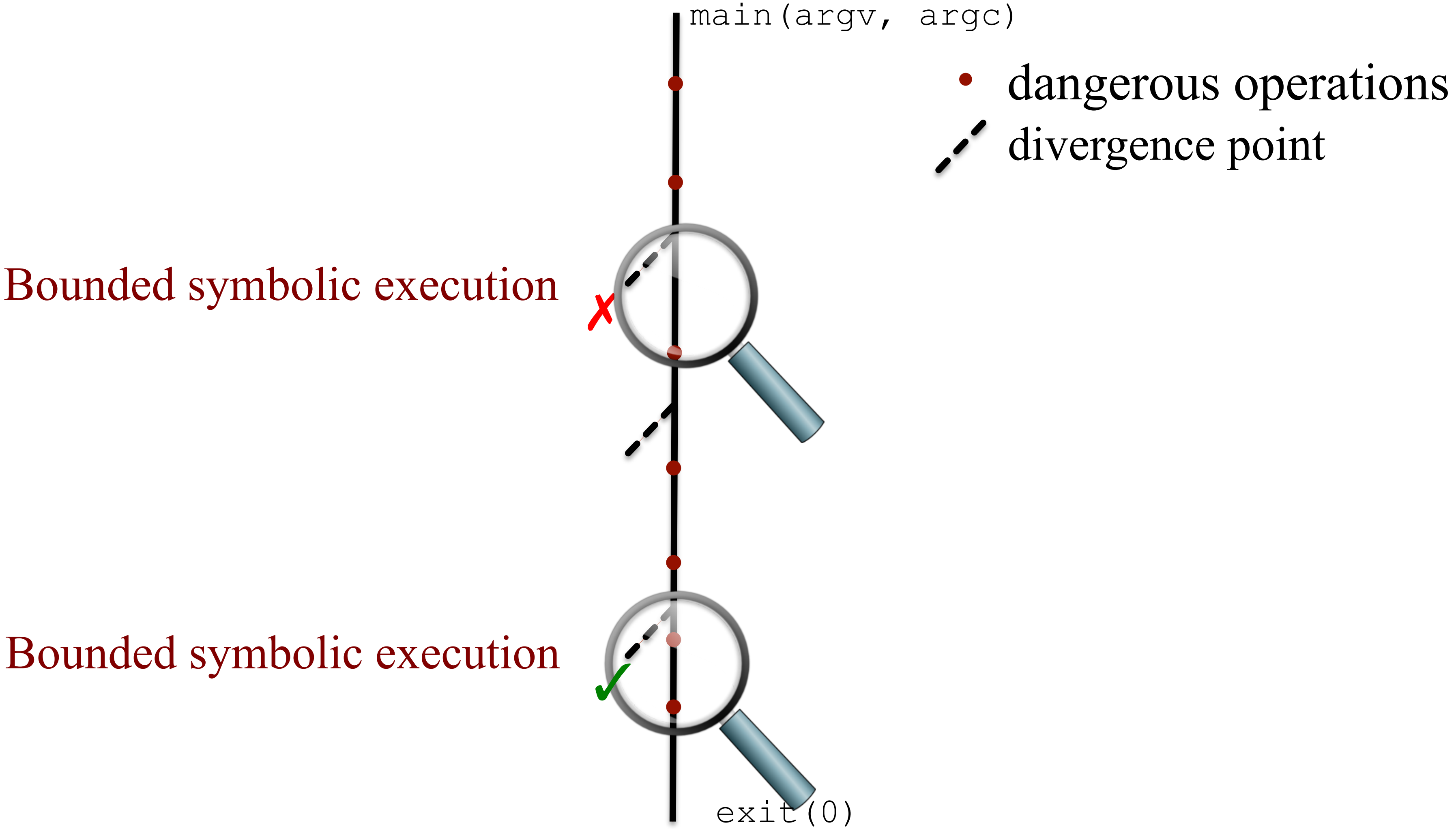
PROS	CONS
• Designed to execute interesting program paths	• Execute each path with a single set of inputs
• Often achieve good coverage of different program features	• Often exercise the general case of a program feature, missing corner cases

Seeding in Symbolic Execution

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]

ZESTI:
Bounded Symbolic Execution



ZESTI Results [ICSE'12]

- Found 52 previously unknown bugs, most of which are out of reach of standard DSE
- Additional advantage: generated inputs are close to those in the regression test suite

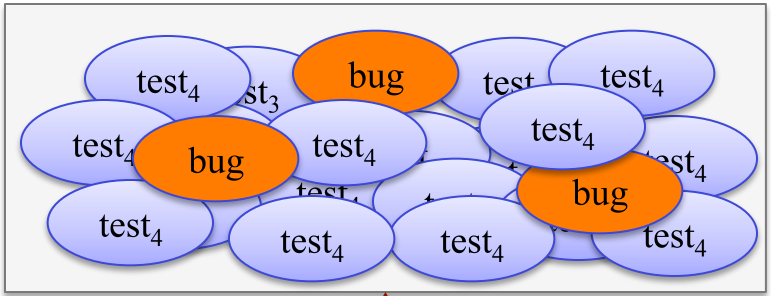
```
cut -c1-3,2-4,6- --output-d=: foo
```

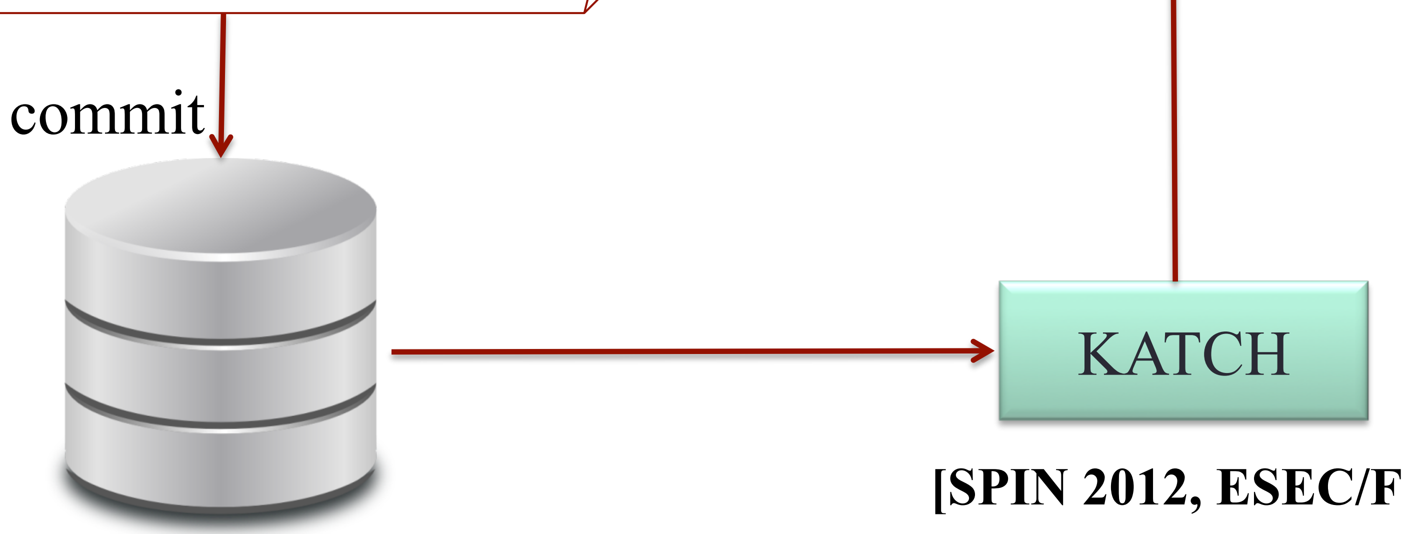
ZESTI

```
cut -c1-3,2-4,8- --output-d=: foo
```

KATCH: High-Coverage Symbolic Patch Testing

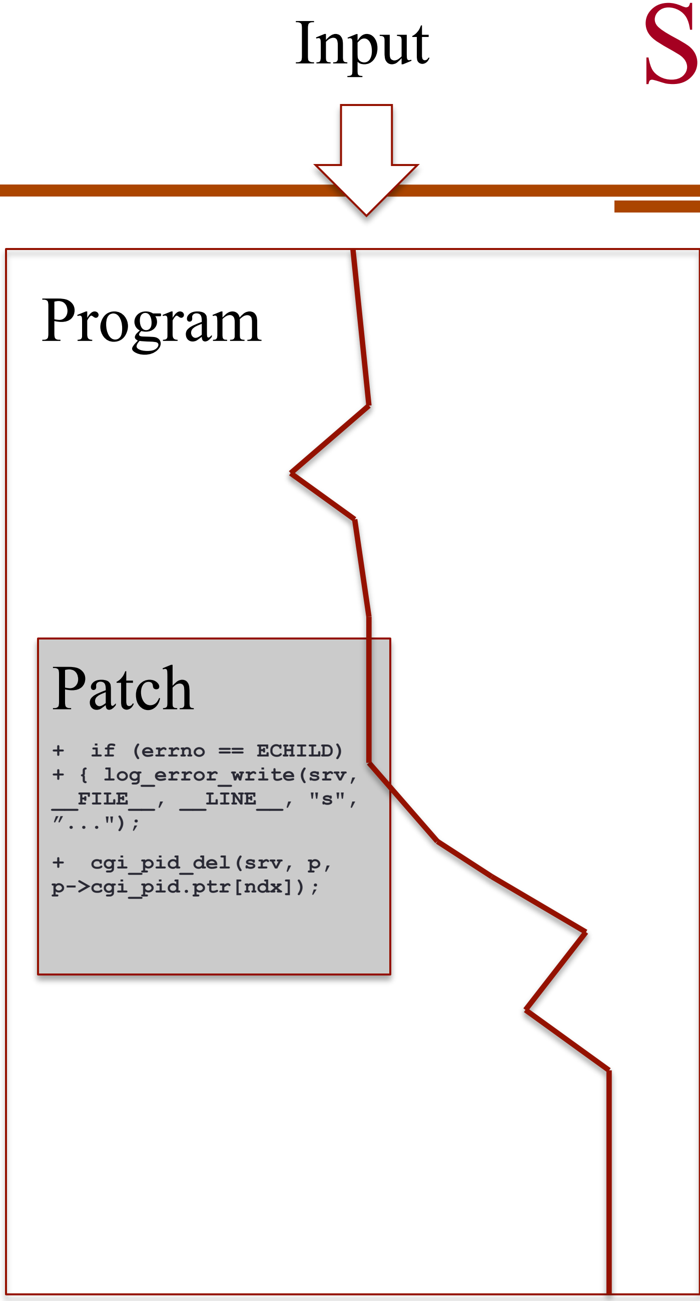
```
--- klee/trunk/lib/Core/Executor.cpp  2009/08/01 22:31:44 77819
+++ klee/trunk/lib/Core/Executor.cpp  2009/08/02 23:09:31 77922
@@ -2422,8 +2424,11 @@
       info << "none\n";
     } else {
       const MemoryObject *mo = lower->first;
+      std::string alloc_info;
+      mo->getAllocInfo(alloc_info);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

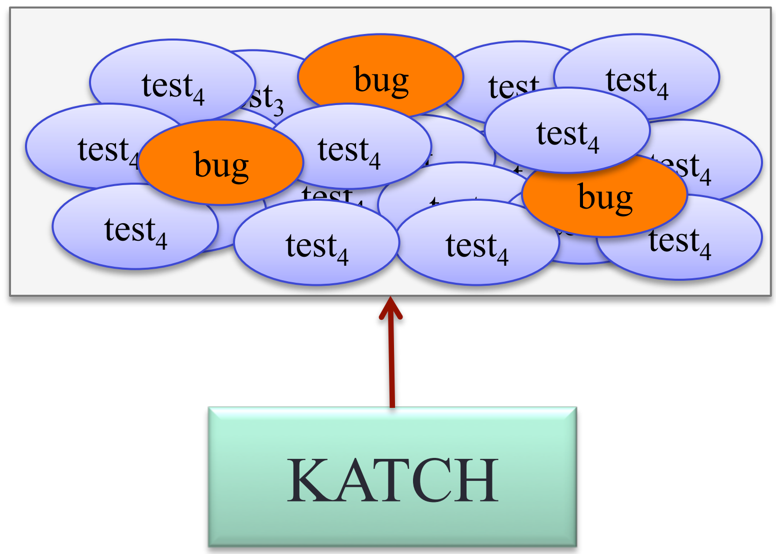




[SPIN 2012, ESEC/FSE 2013]

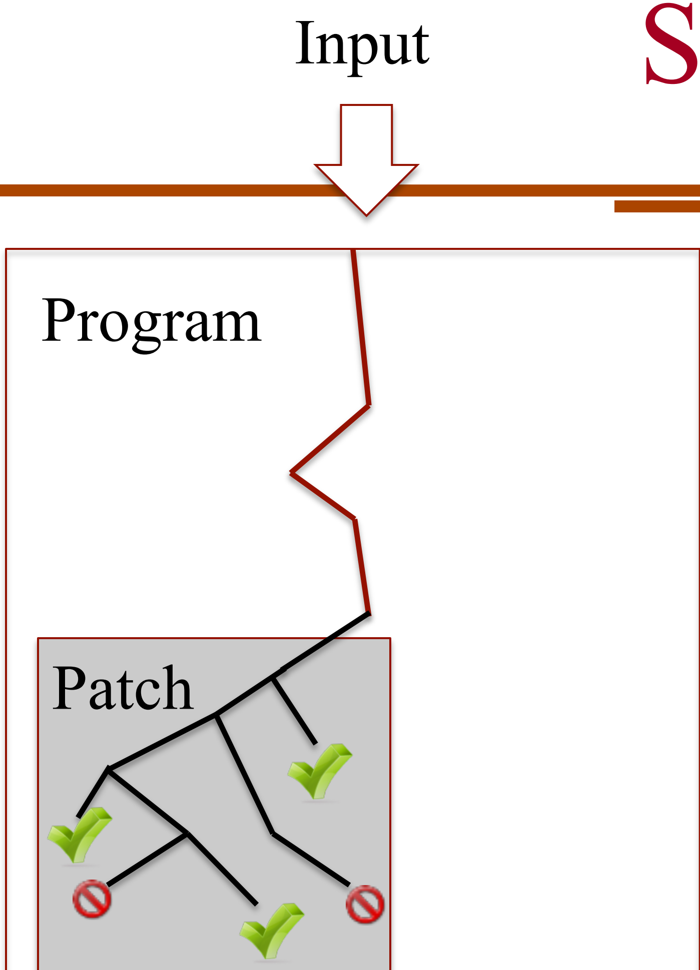
Symbolic Patch Testing



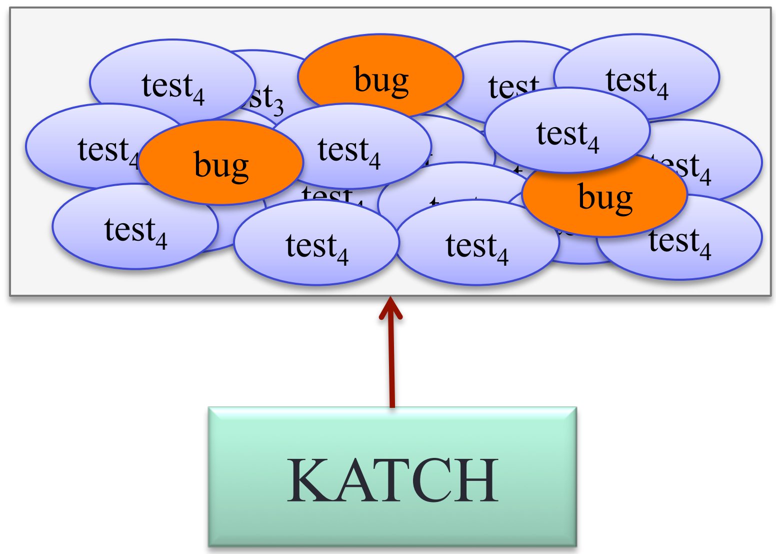


1. Select the regression input closest to the patch (or partially covering it)

Symbolic Patch Testing

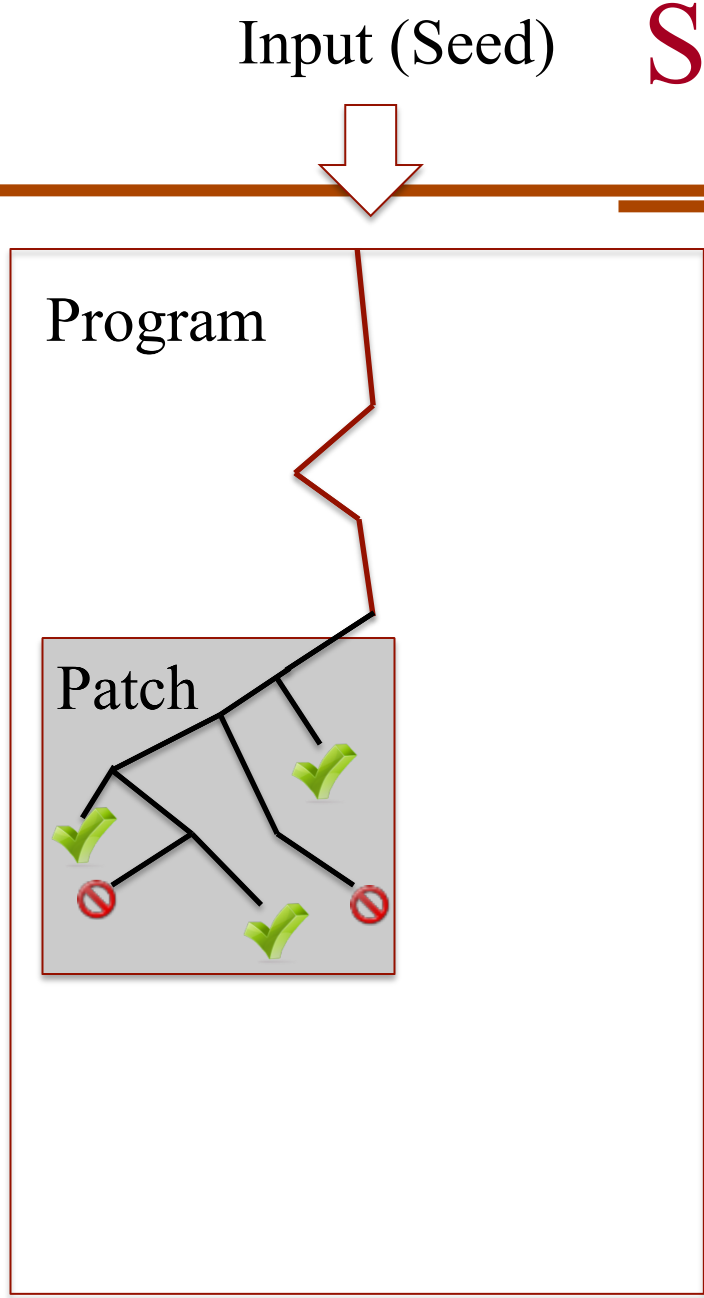


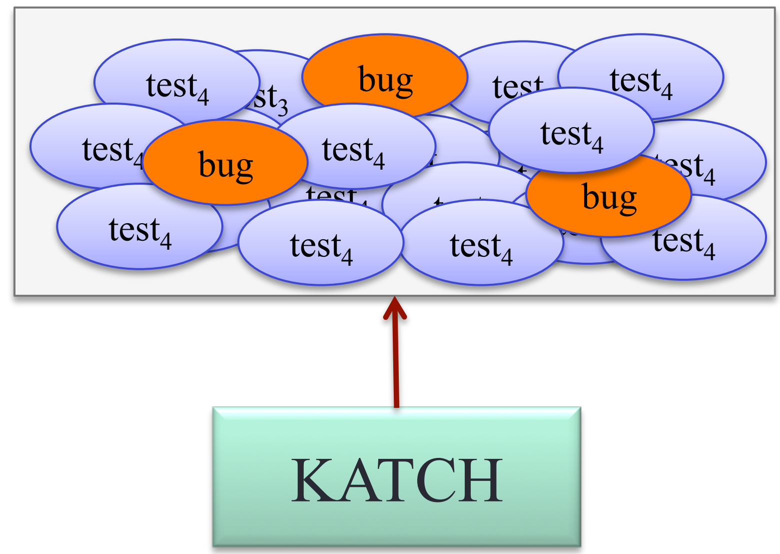
Our notion of estimated distance used to drive exploration is similar to that of fitness in SBST.



2. Greedily drive exploration toward uncovered statements in the patch

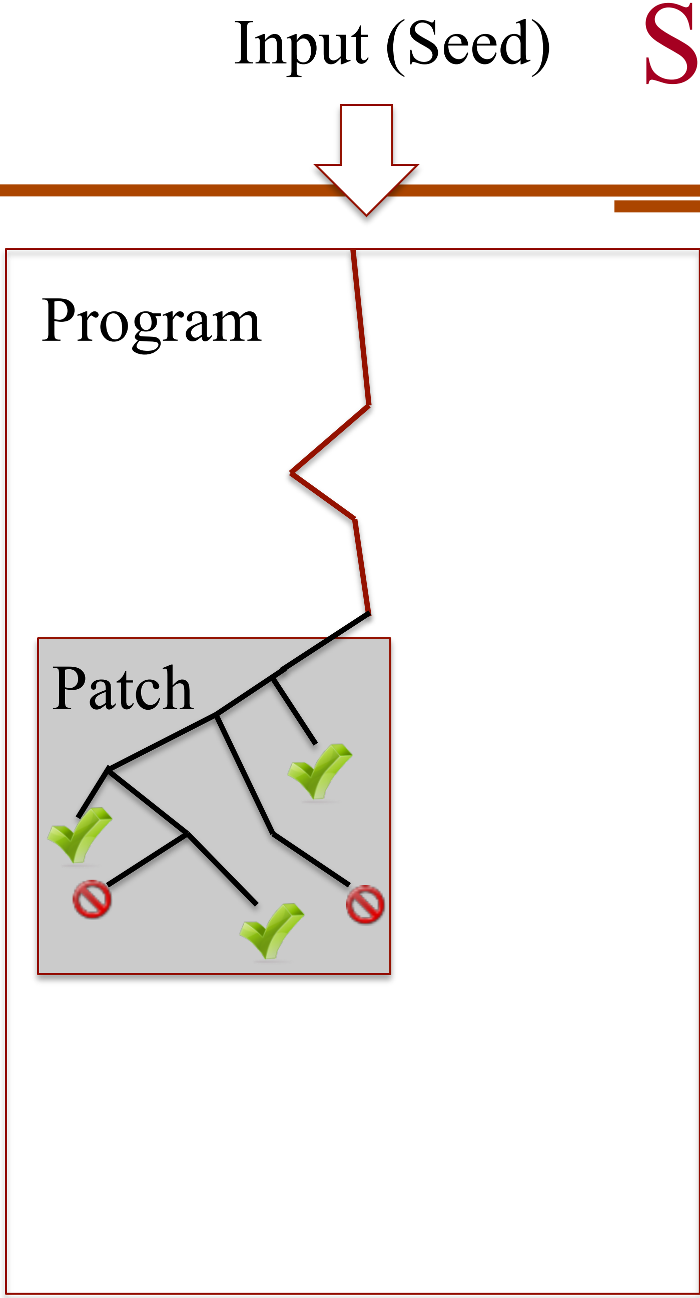
Symbolic Patch Testing

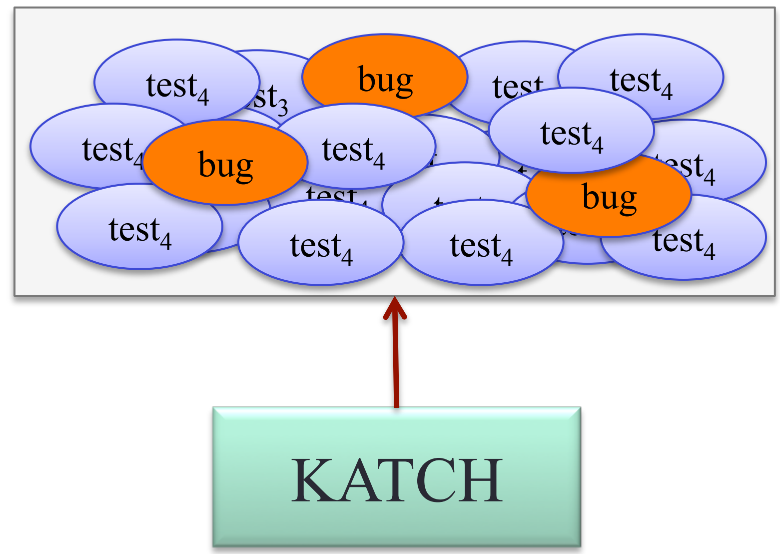




3. If stuck, identify the constraints/bytes that disallow execution to reach the patch, and backtrack

Symbolic Patch Testing





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

[ESEC/FSE 2013]

KATCH: Evaluation

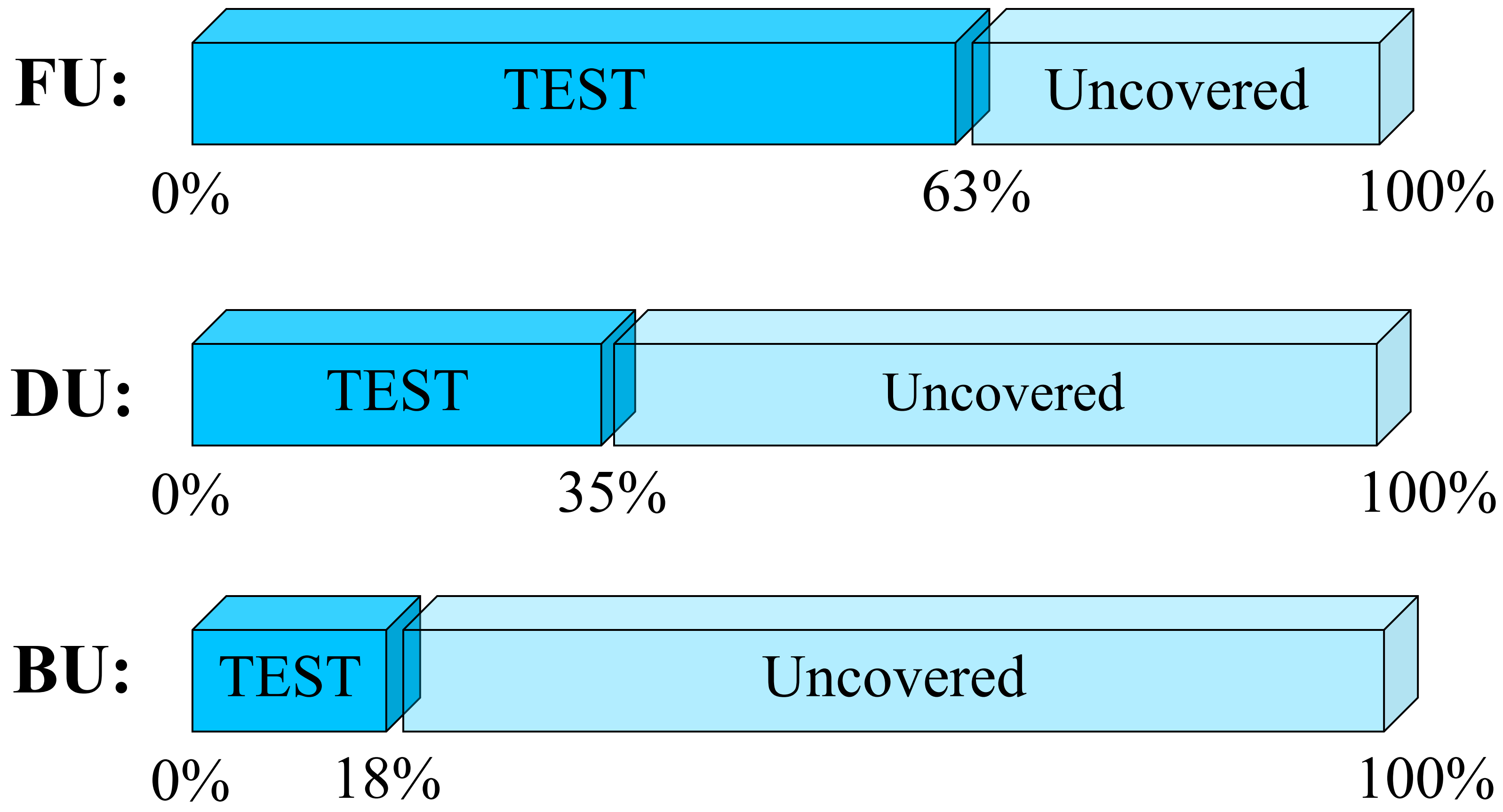
Key evaluation criteria: **no cherry picking!**

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

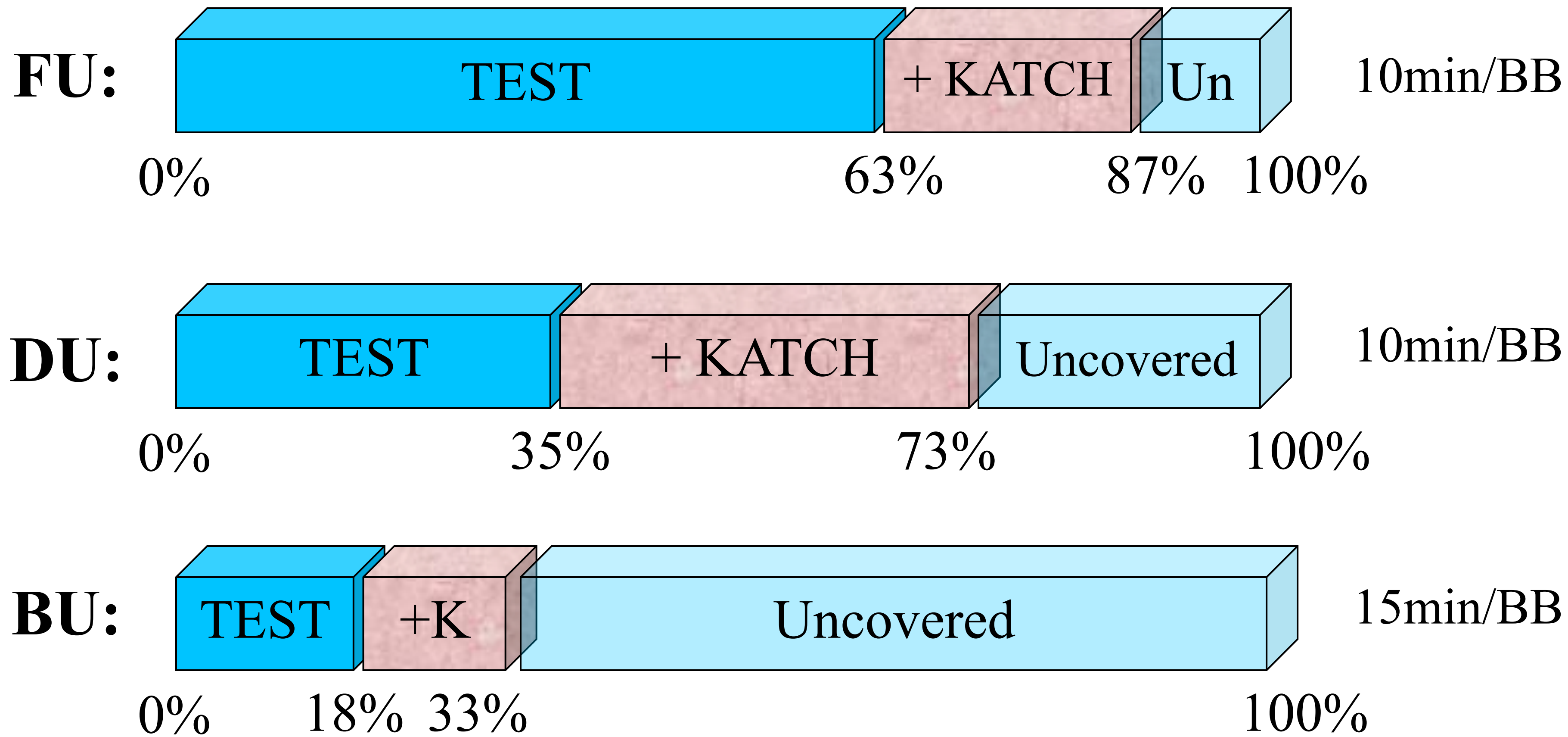
FindUtils suite (FU) **find, xargs, locate**	**12,648 ELOC**	**125 patches written over ~26 months**
DiffUtils suite (DU) **s/diff, diff3, cmp**	**55,655 ELOC** **+ 280,000 in libs**	**175 patches written over ~30 months**
BinUtils suite (BU) **ar, elfedit, nm, etc.**	**81,933 ELOC** **+ 800,000 in libs**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

[ESEC/FSE 2013]

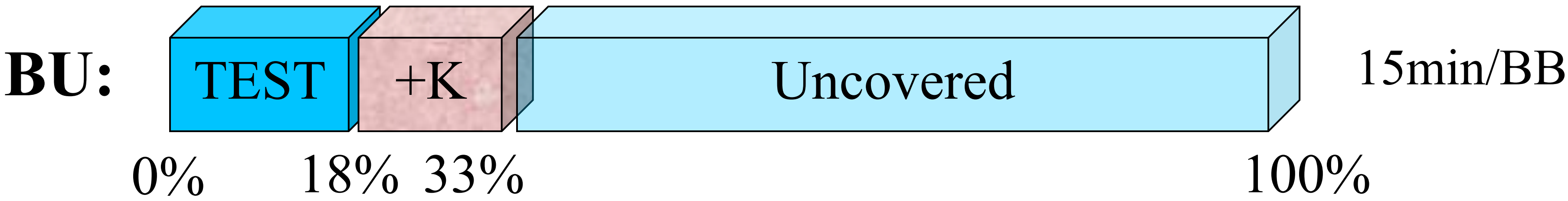
Patch Coverage (basic block level)



Patch Coverage (basic block level)



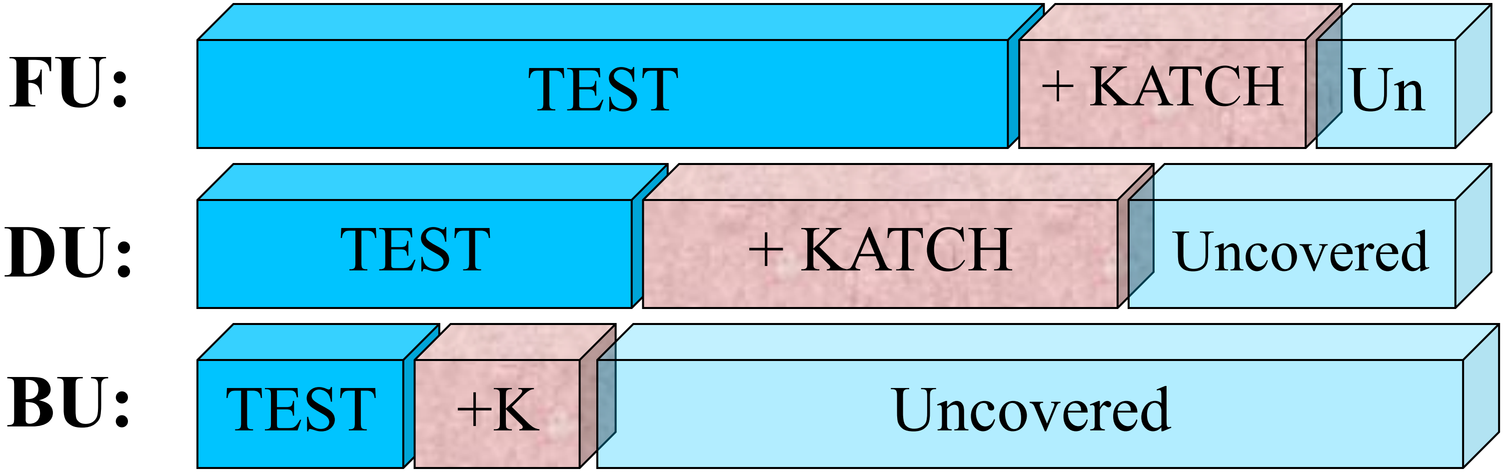
Binutils Bugs



- Found 14 distinct crash bugs
- 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

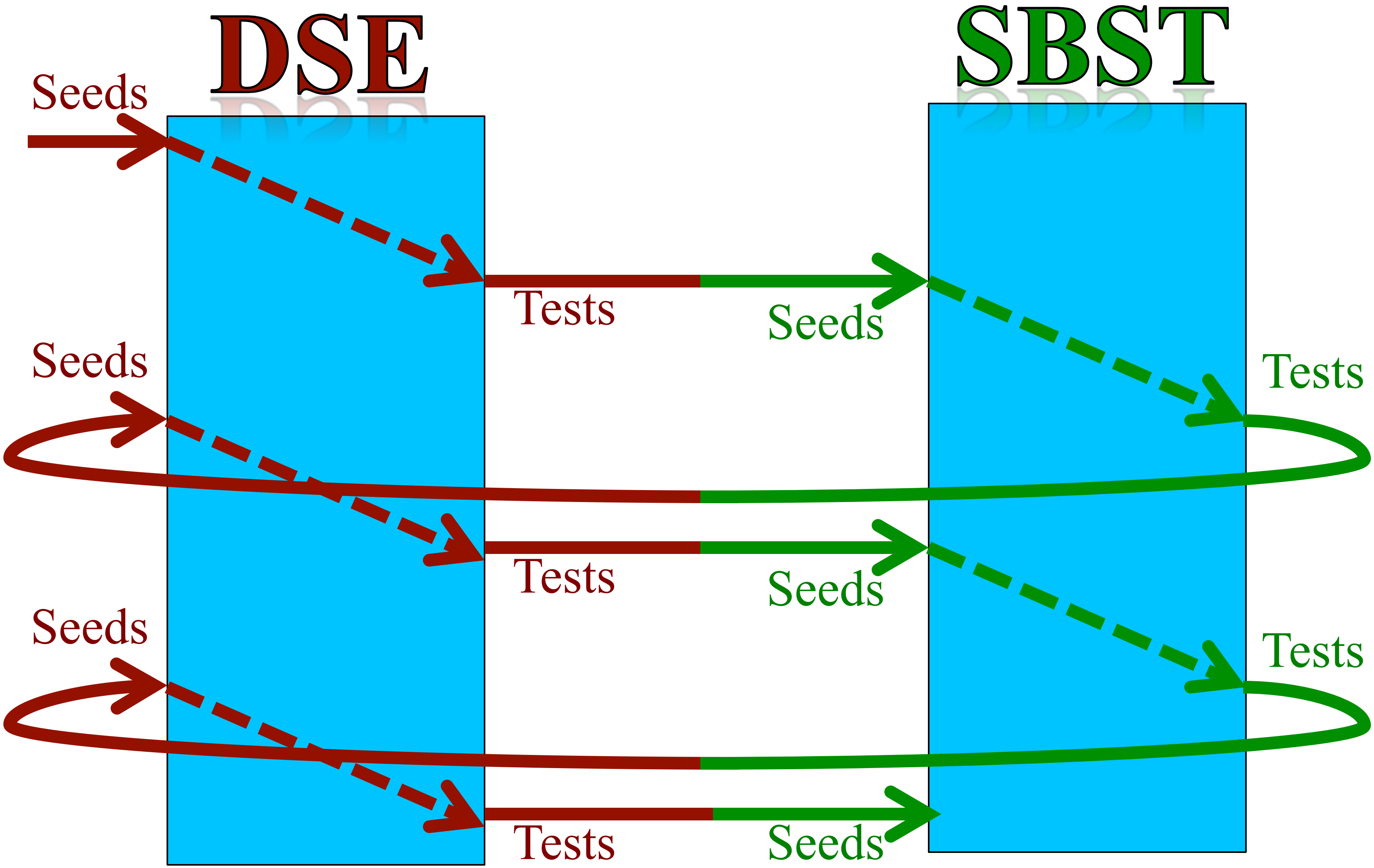
KATCH + SBST?

- Still lots of opportunities for improvement
- We make KATCH and all our experimental data available

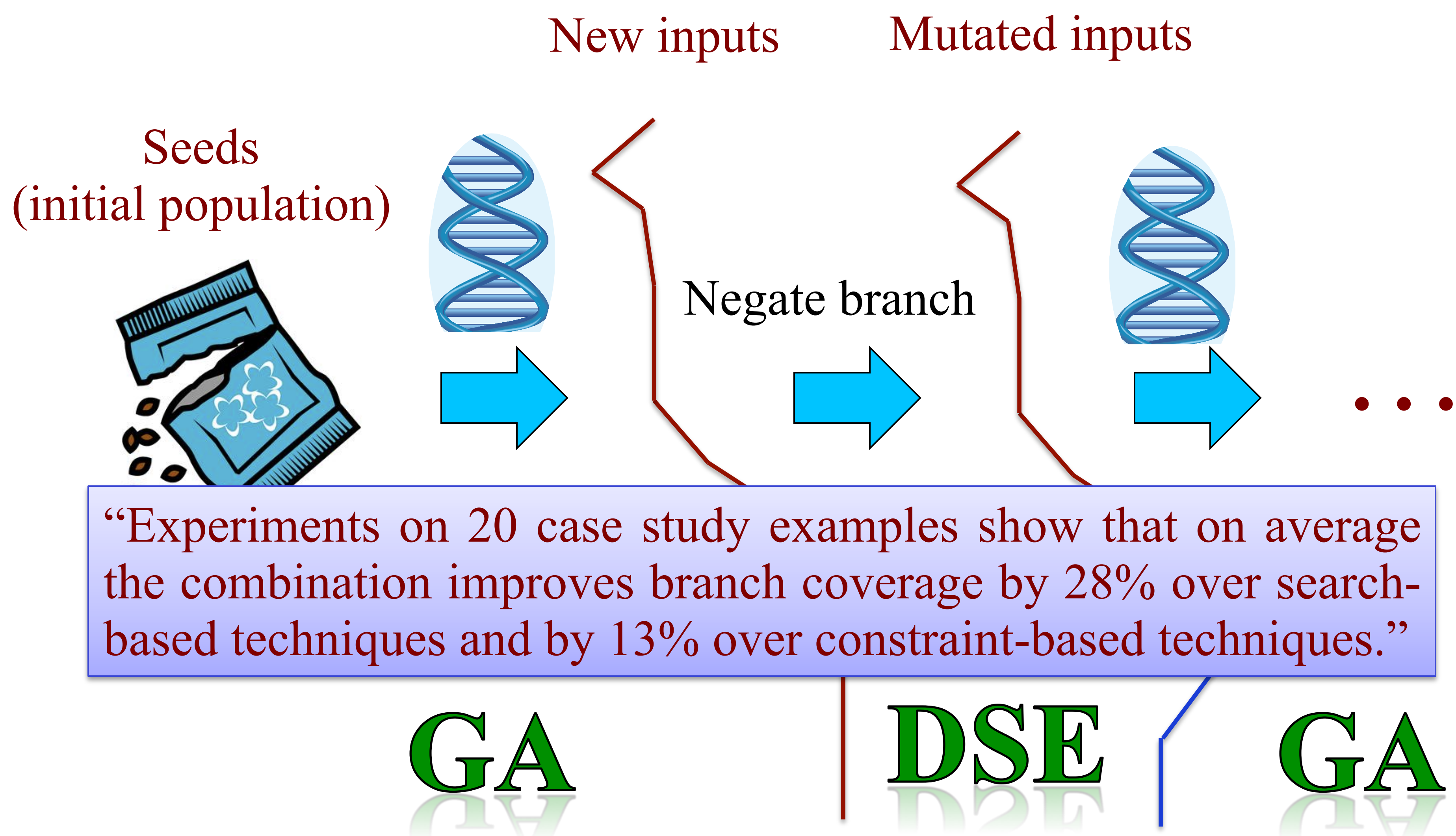


[*Best Artifact Award* at ESEC/FSE 2013, so should be relatively painless to reproduce our results]

Seeds as Communication Primitive?



DSE-based mutator operator [Malburg & Fraser, ASE 2011]



"Experiments on 20 case study examples show that on average the combination improves branch coverage by 28% over search-based techniques and by 13% over constraint-based techniques."

Symbolically-enhanced Fitness Function

[Baars, Harman, Hassoun, Lakhotia, McMinn, Tonella, Vos, ASE'11]

- "Traditional" code-level SBST fitness function:

Approach level + Branch distance

Essentially consider the shortest path to the target

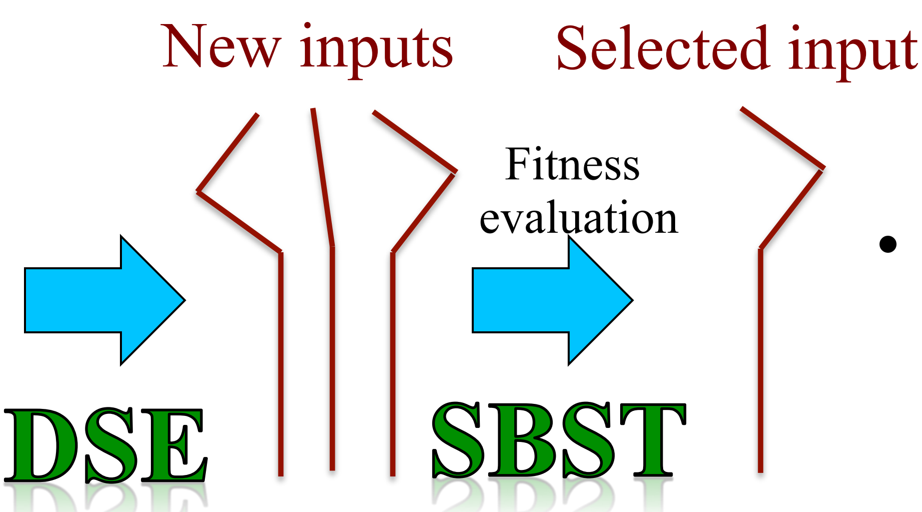
- Use (static) symbolic execution to consider all paths to the target (with some approximation for loops)

"On average, the local search requires 23.41% and the global search 7.78% fewer fitness evaluations when using a symbolic execution based fitness function"

- Can DSE be used instead? If so, what paths should be considered?

Fitness-guided Path Exploration
[Xie, Tillmann, de Halleux, Schulte, DSN'09]

```
a = symbolic
int nz = 0;
for (i=0; i < N; i++)
   if (a[i] == 0)
      nz++;
if (nz == 100)
   // BUG
```

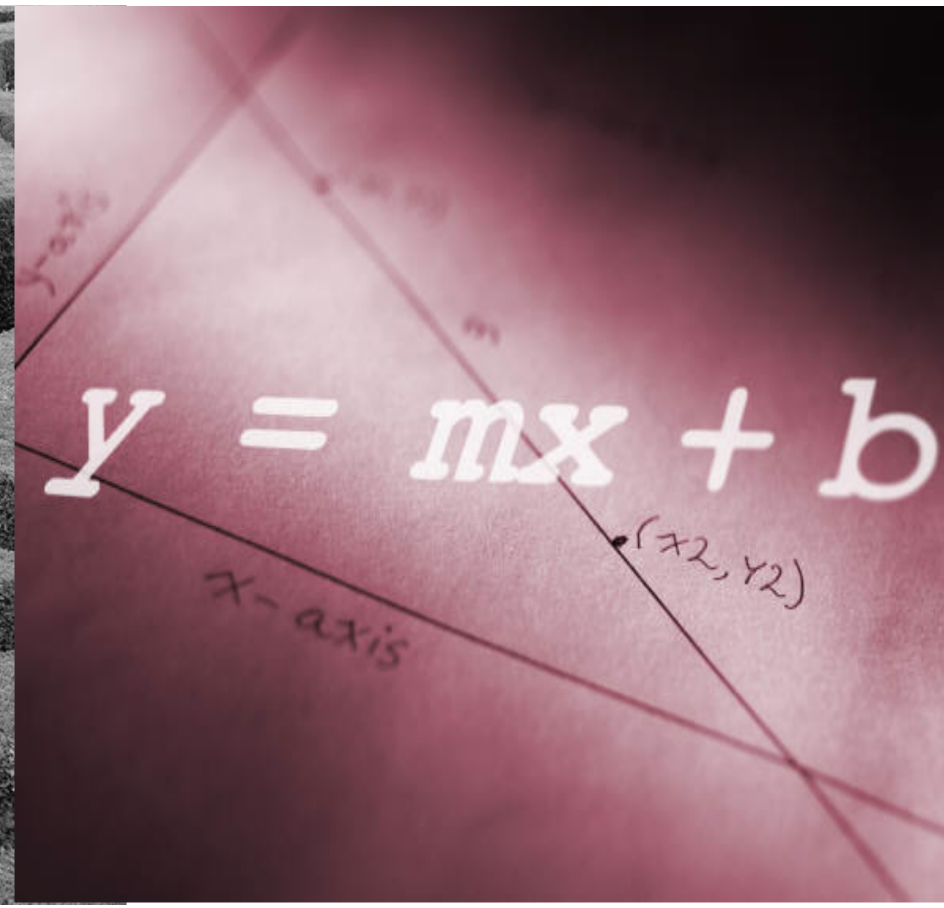


- The search heuristics discussed above struggle, because they ignore values on each path
- "Traditional" SBST fitness functions can help
- E.g., select path which minimizes *branch distance* (here |`nz-100`|)
- One additional problem is that fitness evaluations may result in symbolic values, which are expensive to compare

Fitness-guided Path Exploration
[Xie, Tillmann, de Halleux, Schulte, DSN'09]

- "our approach is effective since it consistently achieves high code coverage faster than existing search strategies" [on 30 case studies containing the "essence of an individual exploration problem" compared with random, DFS, BFS, Pex-Fitnex)]
- "integration of Fitnex and other strategies achieves the effect of getting the best of both in practice"

Scalability Challenges: Constraint Solving



Constraint Solving: Performance

- Inherently expensive
- Invoked at every branch

Optimisations can be implemented at several different levels:

- SAT solvers
- SMT solvers
- Symbolic execution tools

Search-Based Floating Point Constraint Solving
[Lakhotia, Tillmann, Harman, de Halleux, ICTSS'10]

SAT-based FP constraint solvers face serious scalability challenges

- SBST can help

"Results from a set of benchmark functions show that it is possible to increase the effectiveness of what might be called "vanilla DSE". However, a study on two open source programs also shows that for the solvers to be effective, they need to be given adequate resources in terms of wall clock execution time, as well as a large fitness budget."

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 →

$x = 5$
$y = 15$

[OSDI'08]

Caching Solutions

- How many sets should we keep?
- Which subsets should we try, and in what order?
  - Currently: all, in no particular order
- Should we try to see if any prior solution works (not just subsets)?

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

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 →

$x = 5$
$y = 15$

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!

◆◆◆

Can SBST help?

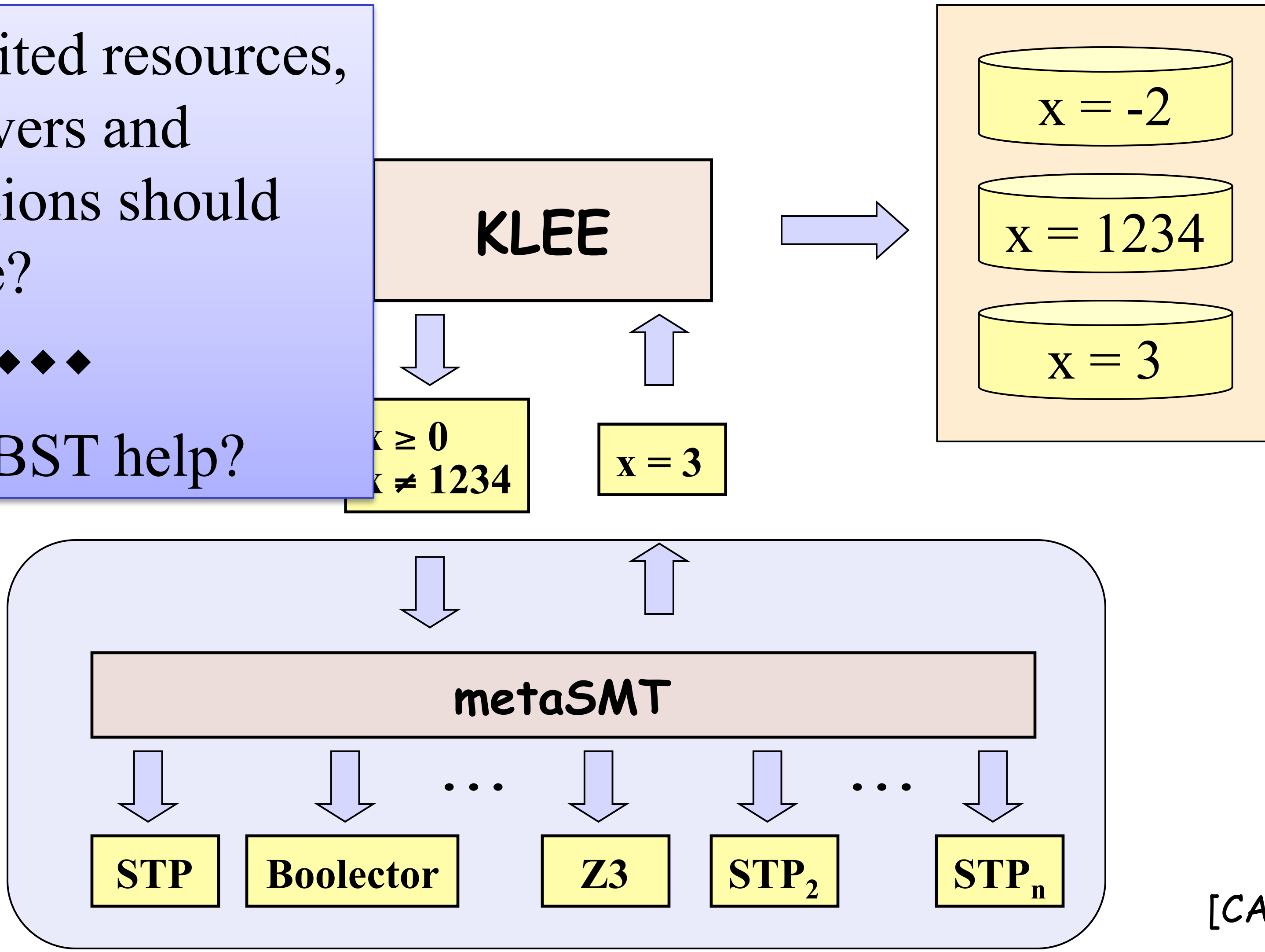
[CAV'13]

Portfolio of SMT Solvers

Given limited resources, which solvers and configurations should we choose?

◆◆◆

Can SBST help?



[CAV'13]


DSE
SBST

DSE+SBST
DSE
SBST