Symbolic Execution for Evolving Software

Cristian Cadar

Department of Computing

Imperial College London

SOFTWARE RELIABILITY
GROUP

Joint work with

Peter Collingbourne, Paul Kelly, Tomek Kuchta,
Paul Marinescu, Hristina Palikareva

Imperial College
London

CREST Open Workshop
UCL, London, UK, 30 January 2017

Motivation

Software evolves, with new versions and patches being released frequently

Unfortunately, patches are notoriously unreliable

- E.g., many users refuse to upgrade their software…
- …relying instead on outdated versions flawed with vulnerabilities or missing useful features and bug fixes

Many admins (70% of those interviewed) refuse to upgrade

Crameri, O., Knezevic, N., Kostic, D., Bianchini, R., Zwaenepoel, W. *Staged deployment in Mirage, an integrated software upgrade testing and distribution system.* SOSP'07

Automatically-Generated Patches

- Research community has recently started to look at automatically-generated patches for
  - Program repair / bug fixing
  - Improving non-functional properties such as performance and energy consumption
  - Porting to other hardware/software environments

Symbolic Execution for Evolving Software

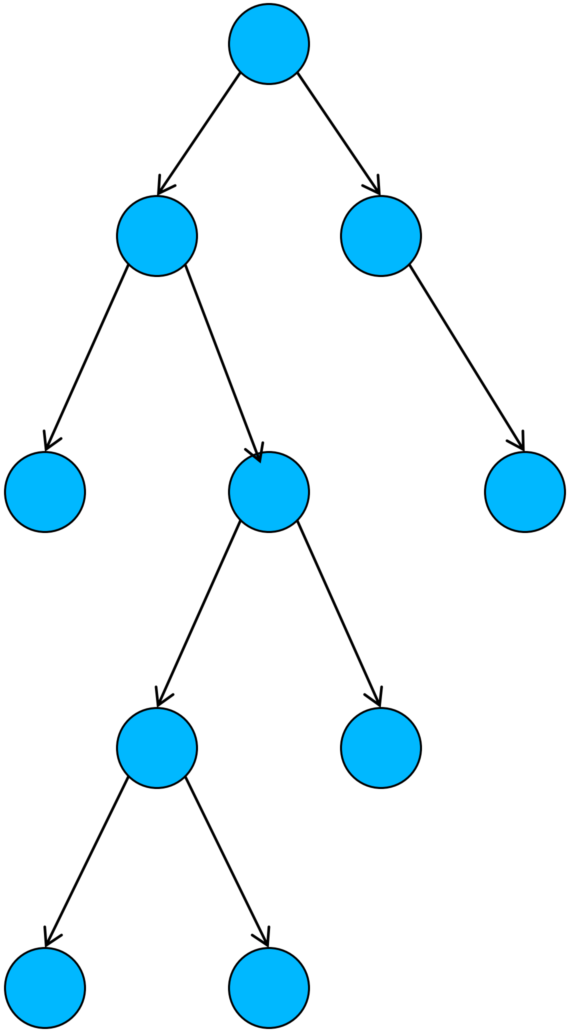
- Active area of research in the Software Reliability Group at Imperial
- Three main directions so far:
  - Testing/verifying semantics-preserving changes, such as performance optimizations and porting to different platforms
  - Coverage-testing of arbitrary software patches
  - Behaviour-testing of arbitrary software patches
- We have only looked at manual changes
  - Are automatically-generated testing any different?

Symbolic Execution
or Dynamic Symbolic Execution (DSE)

Symbolic execution is a program analysis technique for ***automatically exploring paths*** through a program

Reasons about the feasibility of individual paths using a ***constraint solver***

Can generate ***test inputs*** for each path explored



Symbolic Execution for Evolving Software

Evolving software offer the potential to:

- Prune a large part of the search space
- Perform incremental reasoning/analysis
- Use previous version as an oracle

SymEx for Evolving Software

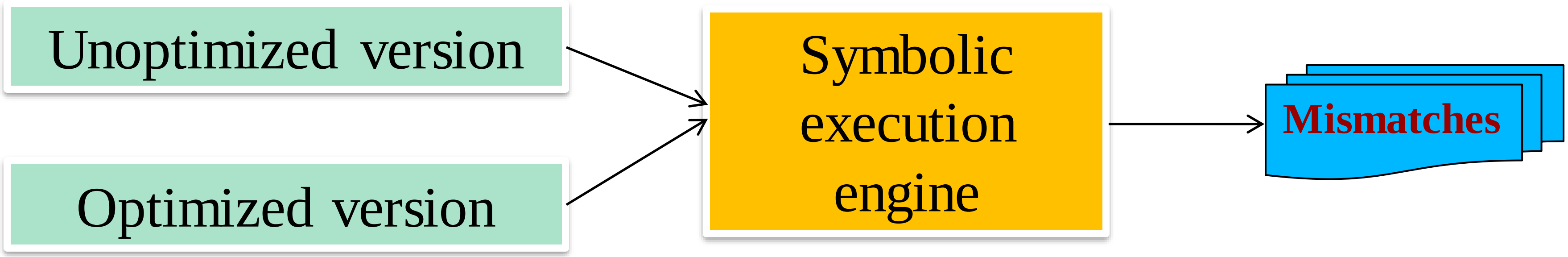
TESTING AND VERIFYING OPTIMIZATIONS

Testing Semantics-Preserving Evolution via Crosschecking

Lots of available opportunities as code is:

Optimized frequently Refactored frequently

Ported to new platforms



Unoptimized version

Optimized version

Symbolic execution engine

Mismatches

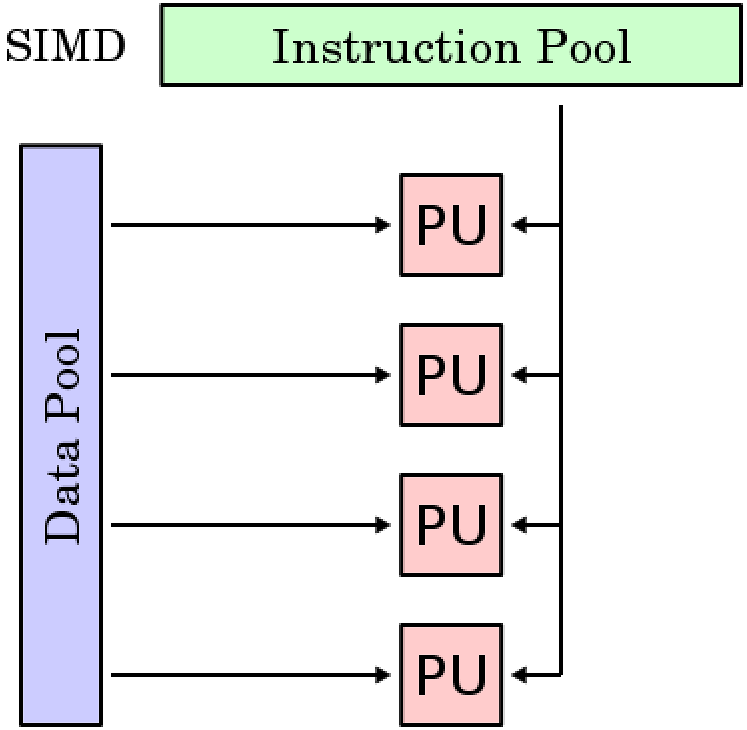
We can find any mismatches in their behavior by:

1. Use symbolic execution to explore multiple paths in version 1
2. For each explored path, explore corresponding path(s) in version 2
3. Comparing the (symbolic) output b/w versions

SIMD Optimizations

Most processors offer support for SIMD instructions

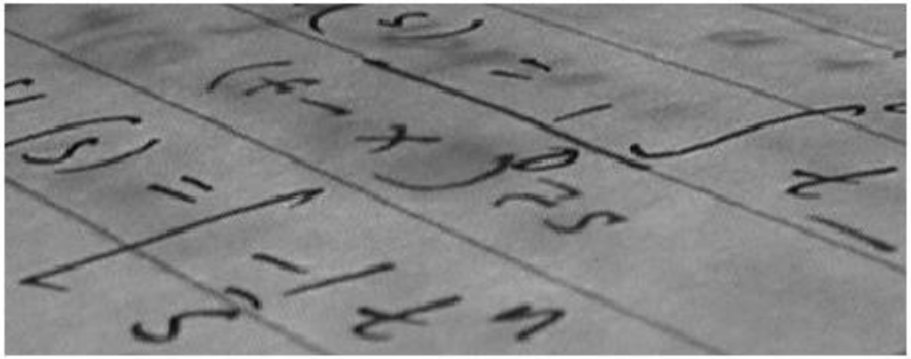
- Can operate on multiple data concurrently
- Many algorithms can make use of them (e.g., computer vision algorithms)



[EuroSys 2011]

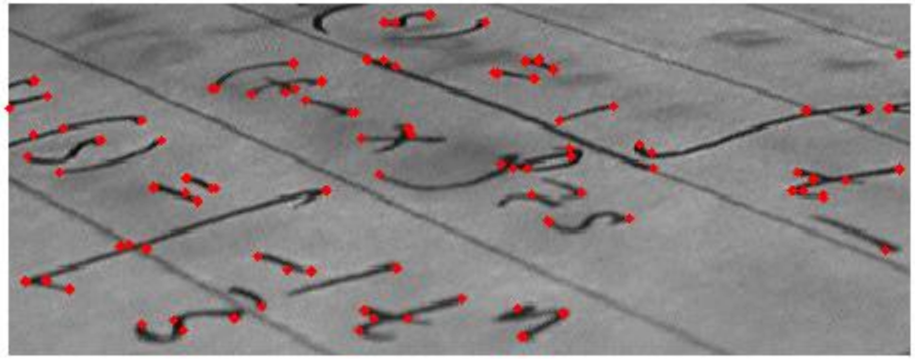
OpenCV

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
  - Verified the correctness of 41 of them up to a certain image size (*bounded verification*)
  - Found mismatches in 10
- Most mismatches due to tricky FP-related issues:
  - Precision, rounding, associativity, distributivity, NaN values

[EuroSys 2011]

GPGPU Optimizations

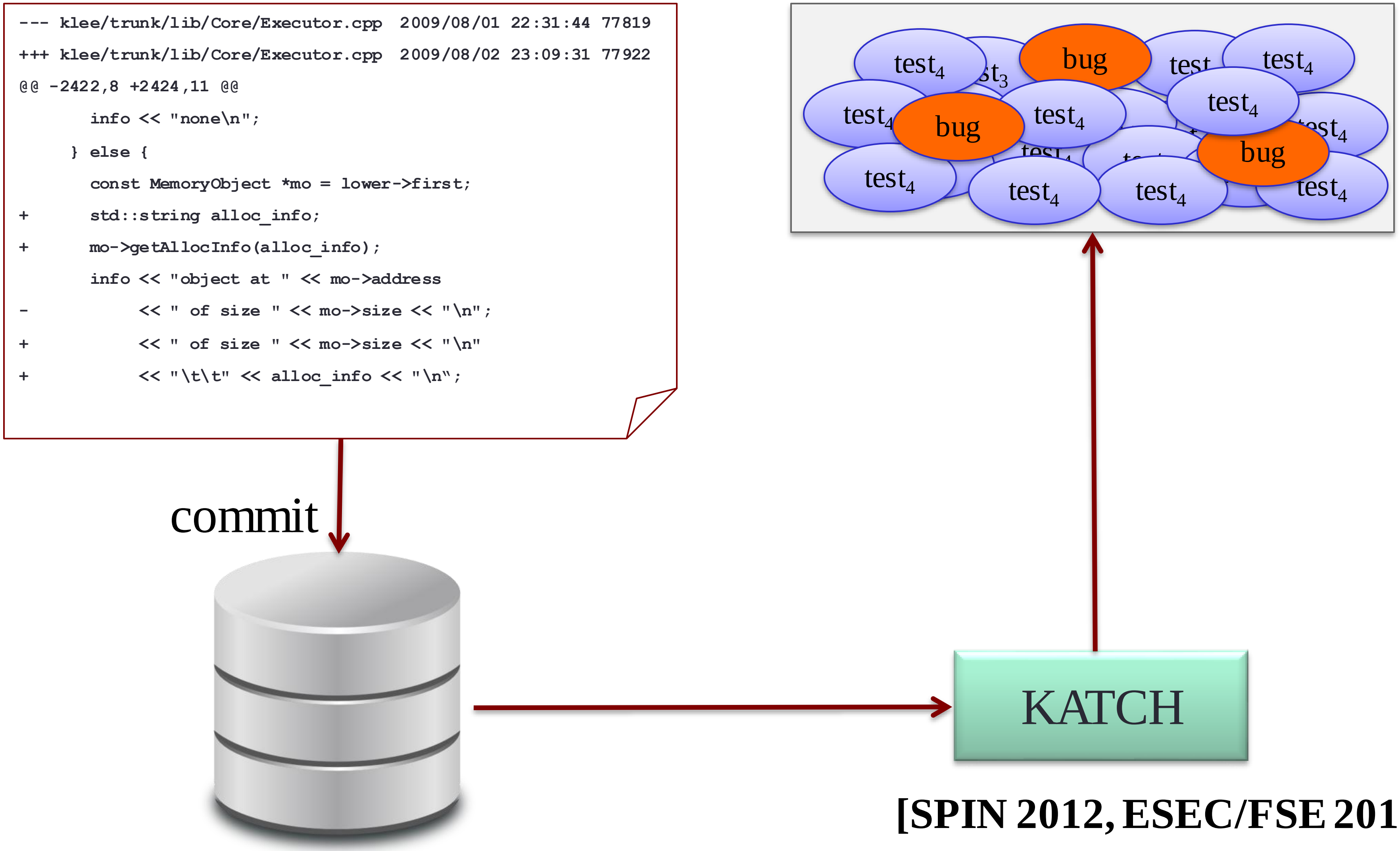
Scalar vs. GPGPU code

[HVC 2011]

SymEx for Evolving Software

HIGH-COVERAGE PATCH TESTING WITH KATCH

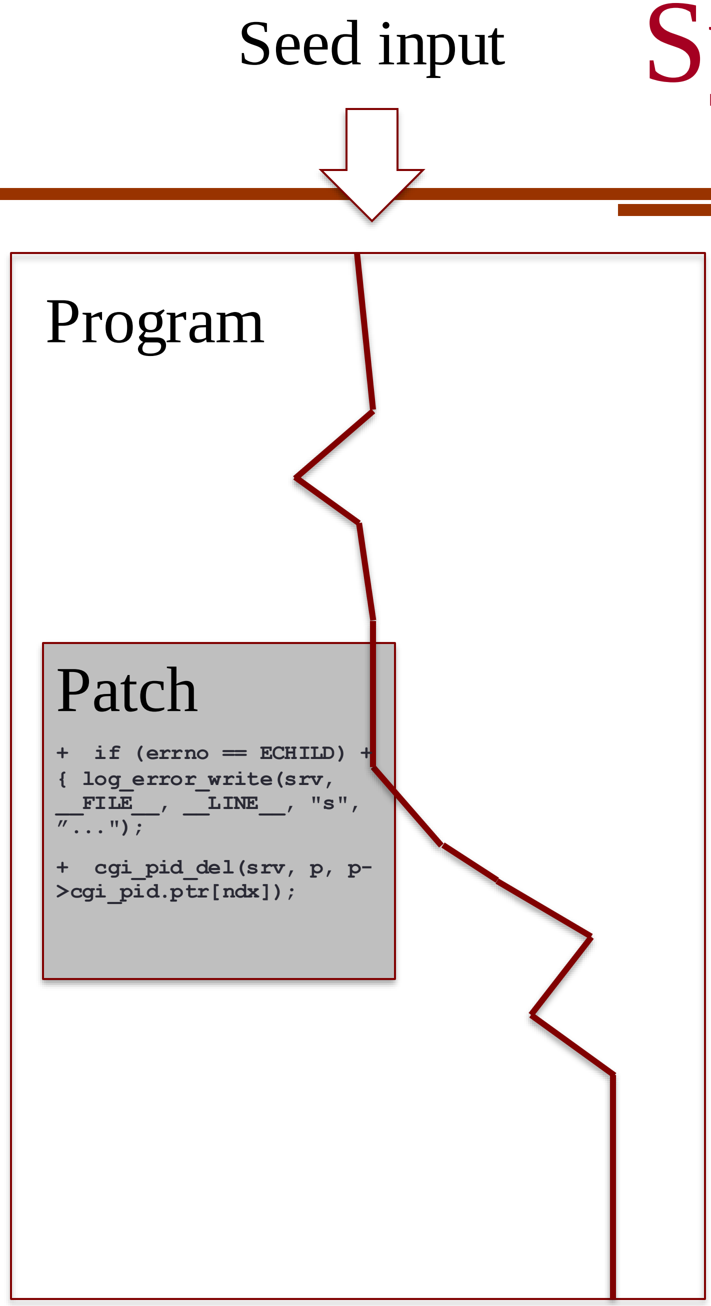
KATCH: High-Coverage Symbolic Patch Testing



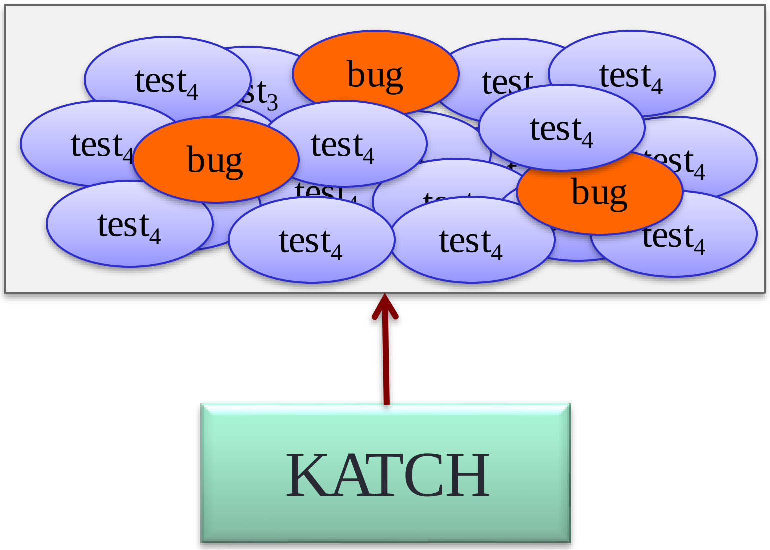
```
--- klee/trunk/lib/Core/Executor.cpp  2009/08/01 22:31:44 77819
+++ klee/trunk/lib/Core/Executor.cpp  2009/08/02 23:09:31 77922
@@ -2422,8 +2424,11 @@
       info << "none\n";
     } else {
       const MemoryObject *mo = lower->first;
+      std::string alloc_info;
+      mo->getAllocInfo(alloc_info);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

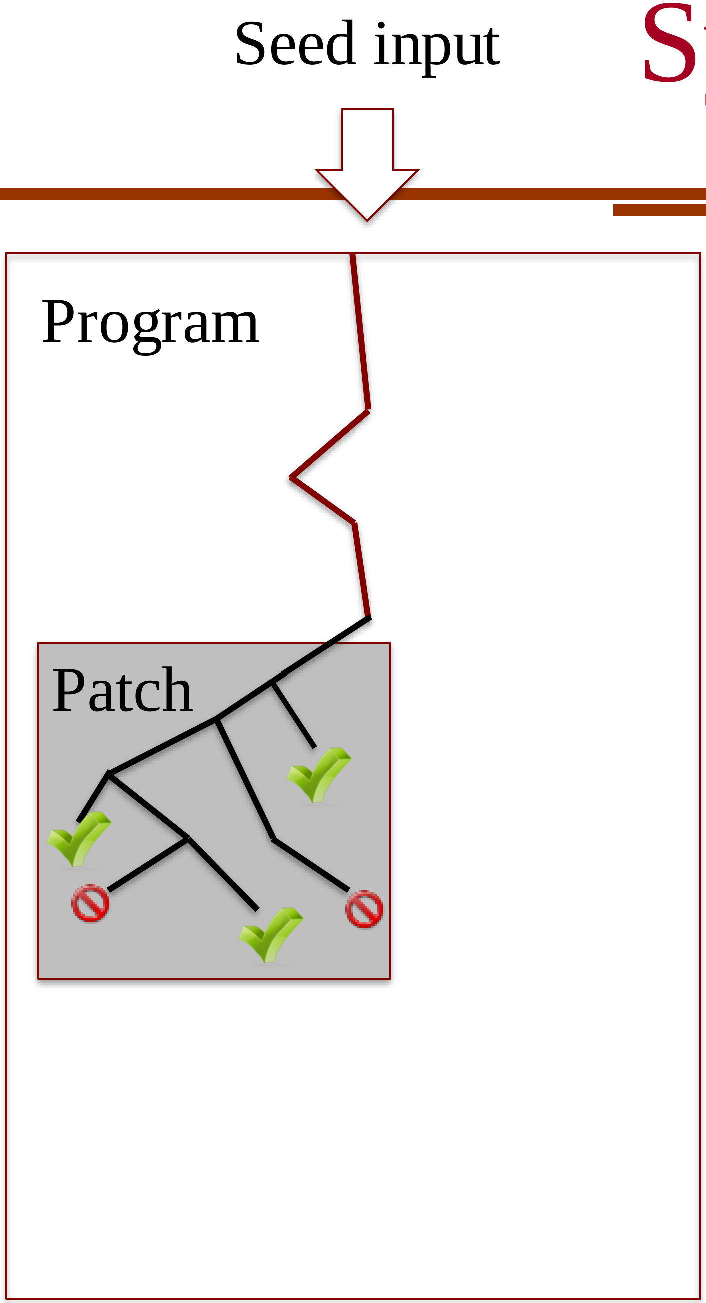


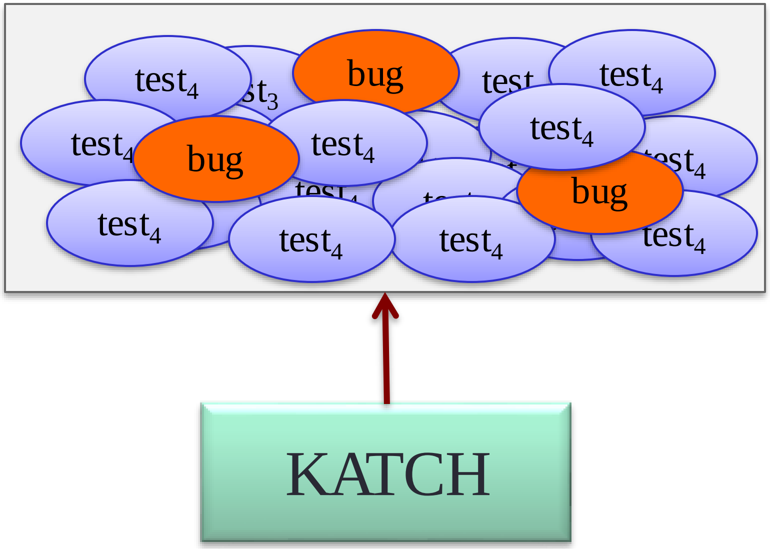
```
+  if (errno == ECHILD) +
{ log_error_write(srv,
__FILE__, __LINE__, "s",
"...");

+  cgi_pid_del(srv, p, p-
>cgi_pid.ptr[ndx]);
```



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

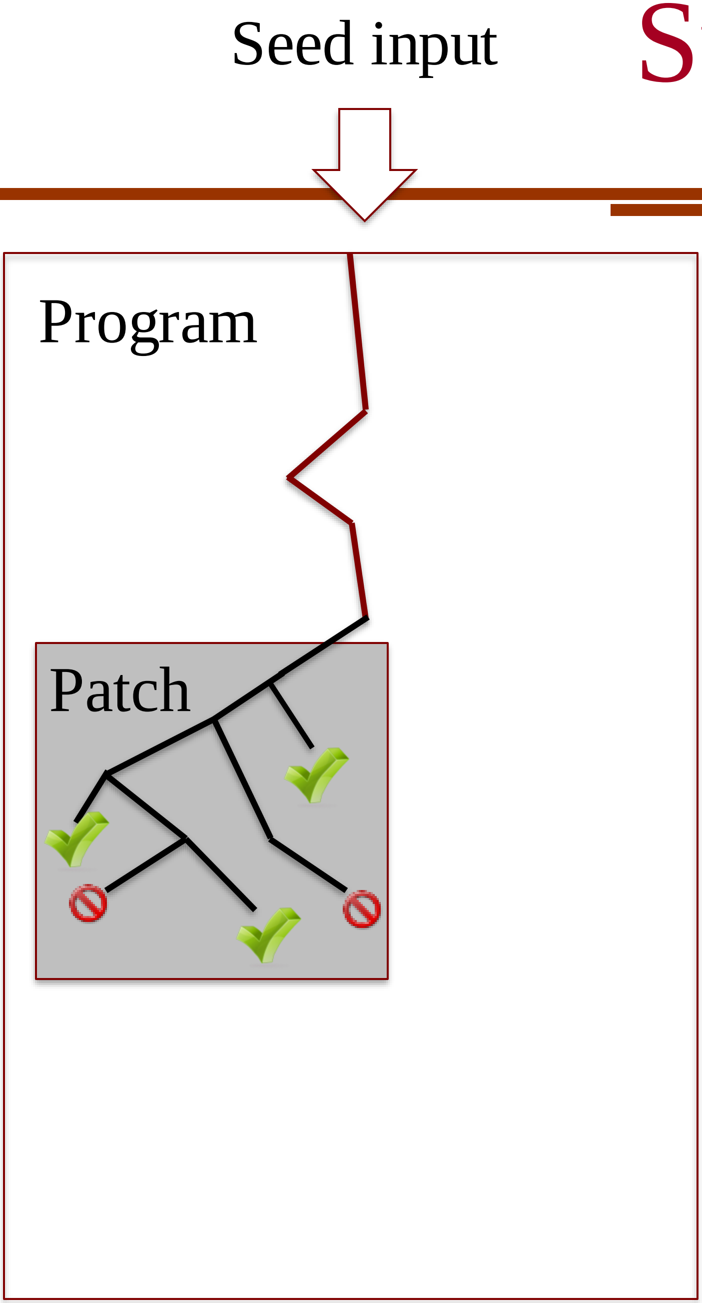
Symbolic Patch Testing

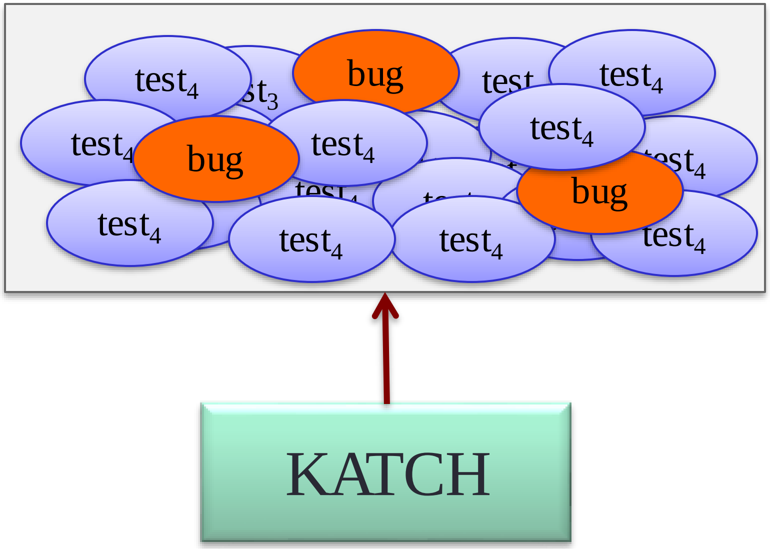




2. Greedily drive exploration toward uncovered basic blocks in the patch

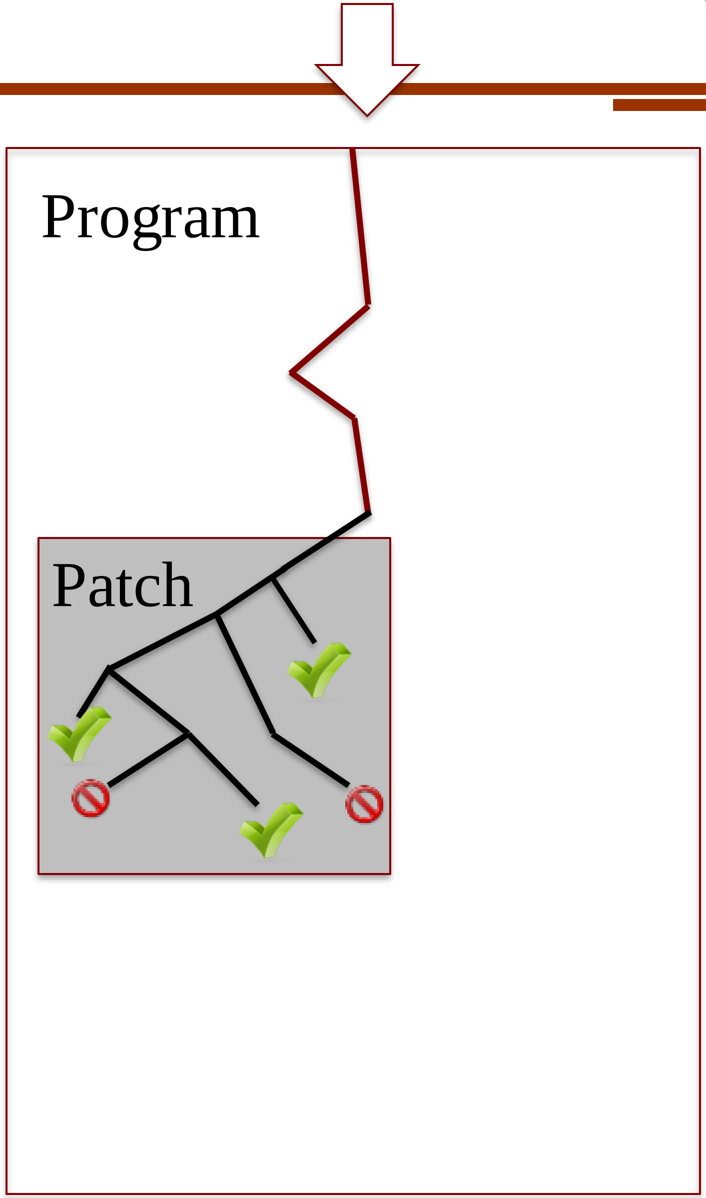
Symbolic Patch Testing

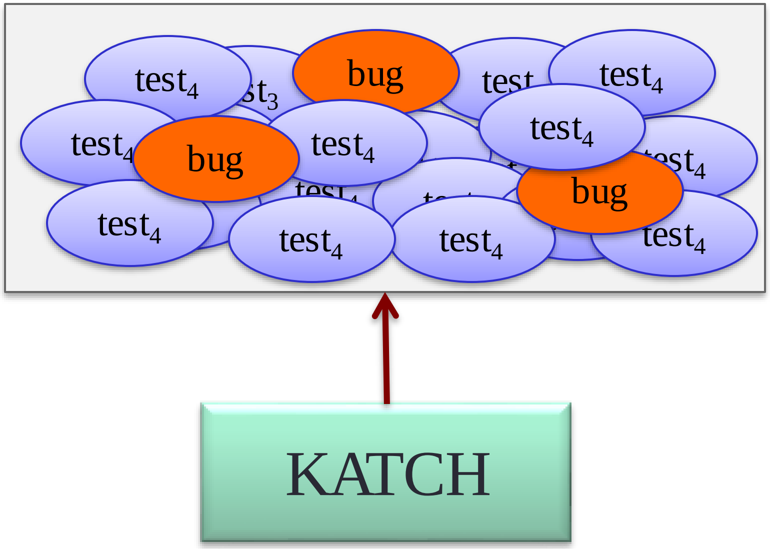




3. If stuck, identify the constraints/bytes that disallow execution to reach the patch, and backtrack

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]

Extended Evaluation

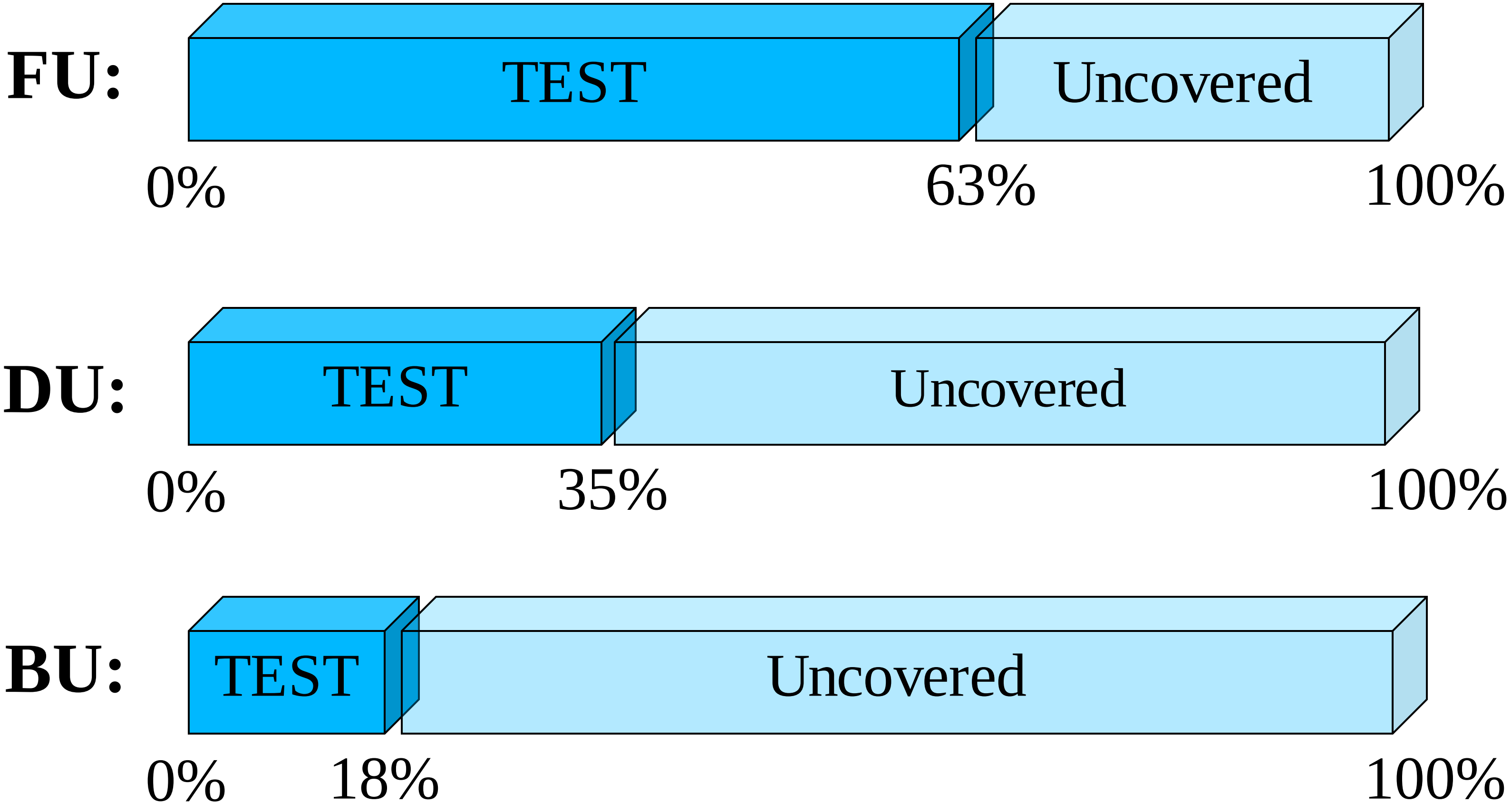
Key evaluation criteria: **no cherry picking!**

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

App. Suite	ELOC	Patches	#BBs
FindUtils (FU) **find, locate, xargs**	**~12k**	**125 written over ~26 months**	**344**
DiffUtils (DU) **cmp, (s)diff, diff3**	**~55k** **+ 280k in libs**	**175 written over ~30 months**	**166**
BinUtils (BU) **ar, elfedit, nm, etc.**	**82k** **+ 800k in libs**	**181 written over ~16 months**	**852**

[ESEC/FSE 2013]

Patch Coverage (basic block level)



FU: TEST | Uncovered

0% 63% 100%

DU: TEST | Uncovered

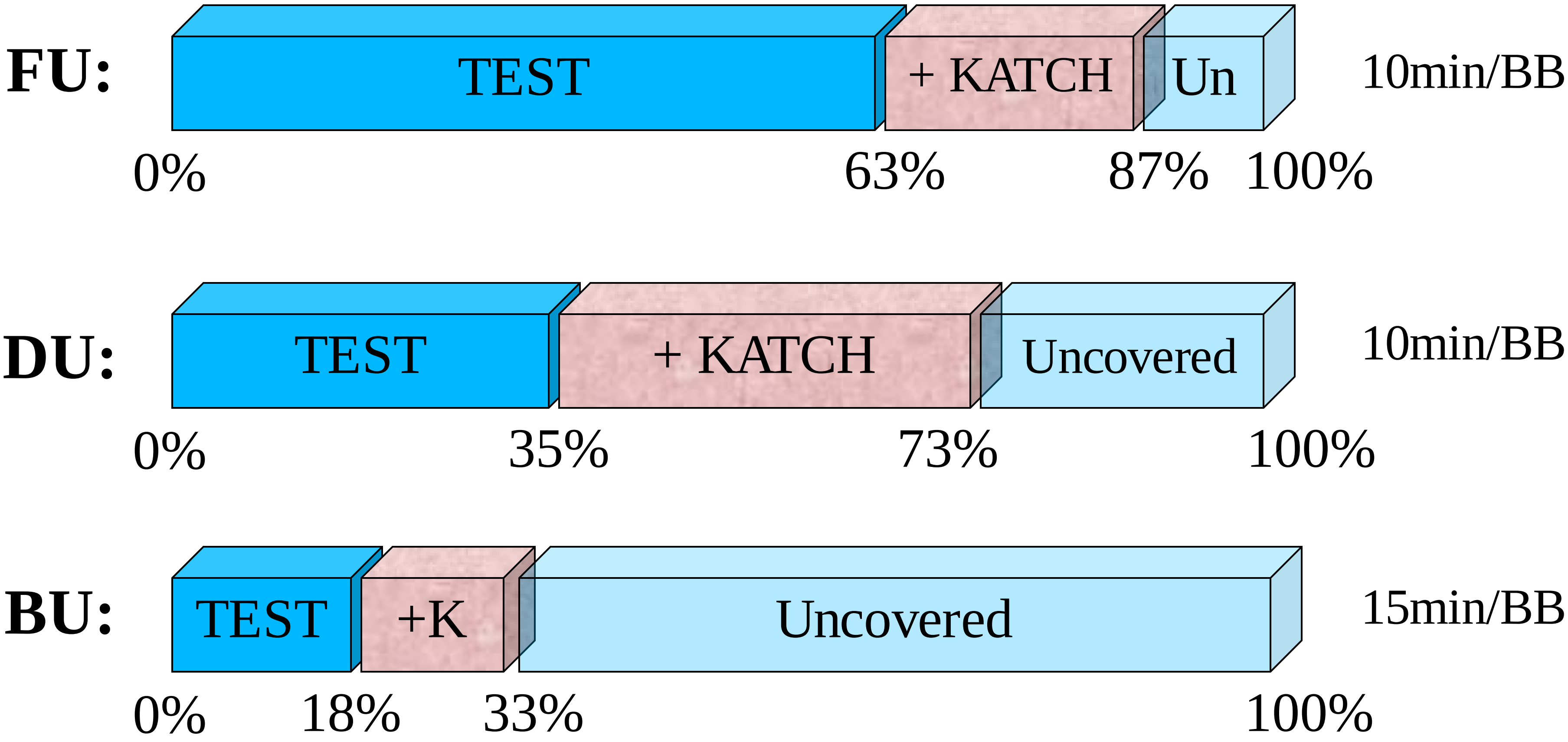
0% 35% 100%

BU: TEST | Uncovered

0% 18% 100%

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

Patch Coverage (basic block level)



FU: TEST | + KATCH | Un — 10min/BB

0% 63% 87% 100%

DU: TEST | + KATCH | Uncovered — 10min/BB

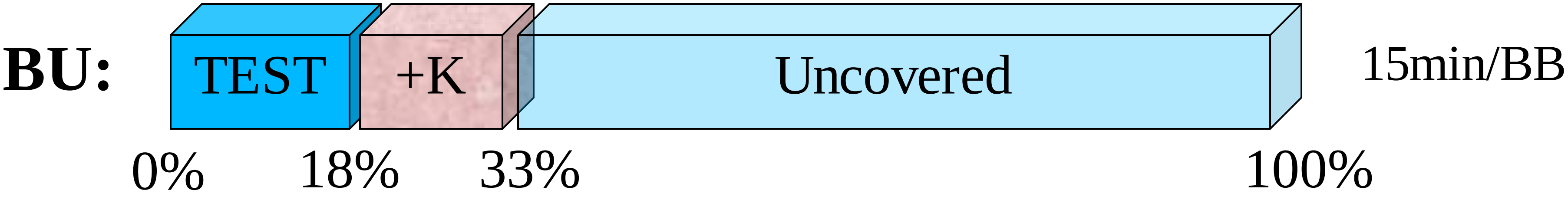
0% 35% 73% 100%

BU: TEST | +K | Uncovered — 15min/BB

0% 18% 33% 100%

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

Binutils Bugs



BU: TEST +K Uncovered 15min/BB

0% 18% 33% 100%

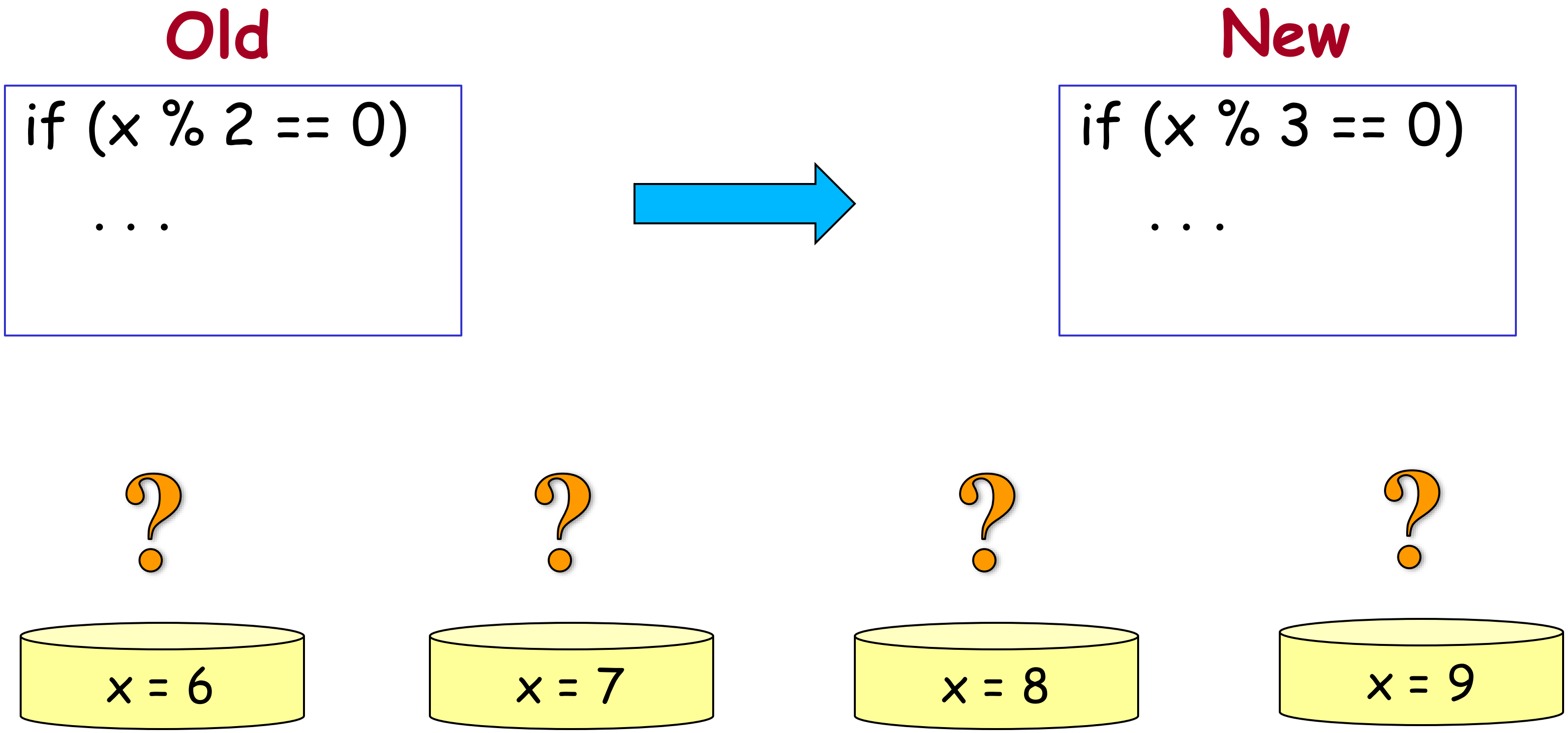
- Found 14 distinct crash bugs
- 12 bugs still present in latest version of BU
  - Reported and fixed by developers
- 10 bugs found in the patch code itself or in code affected by patch code

SymEx for Evolving Software

BEHAVIOURAL PATCH TESTING
VIA SHADOW SYMBOLIC EXECUTION

Is Basic Block Coverage Enough?

- If I change a statement, what tests should I add?



Old

```
if (x % 2 == 0)
    . . .
```

New

```
if (x % 3 == 0)
    . . .
```

Is High Coverage Enough?

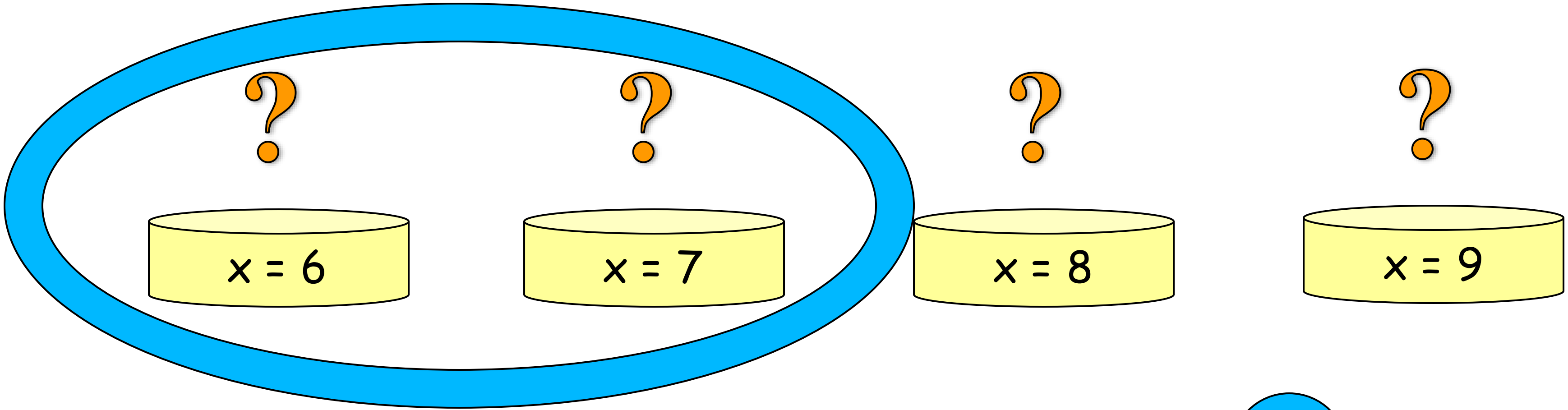
- If I change a statement, what tests should I add?

Old

```
if (x % 2 == 0)
    . . .
```

New

```
if (x % 3 == 0)
    . . .
```



?

?

?

?

x = 6

x = 7

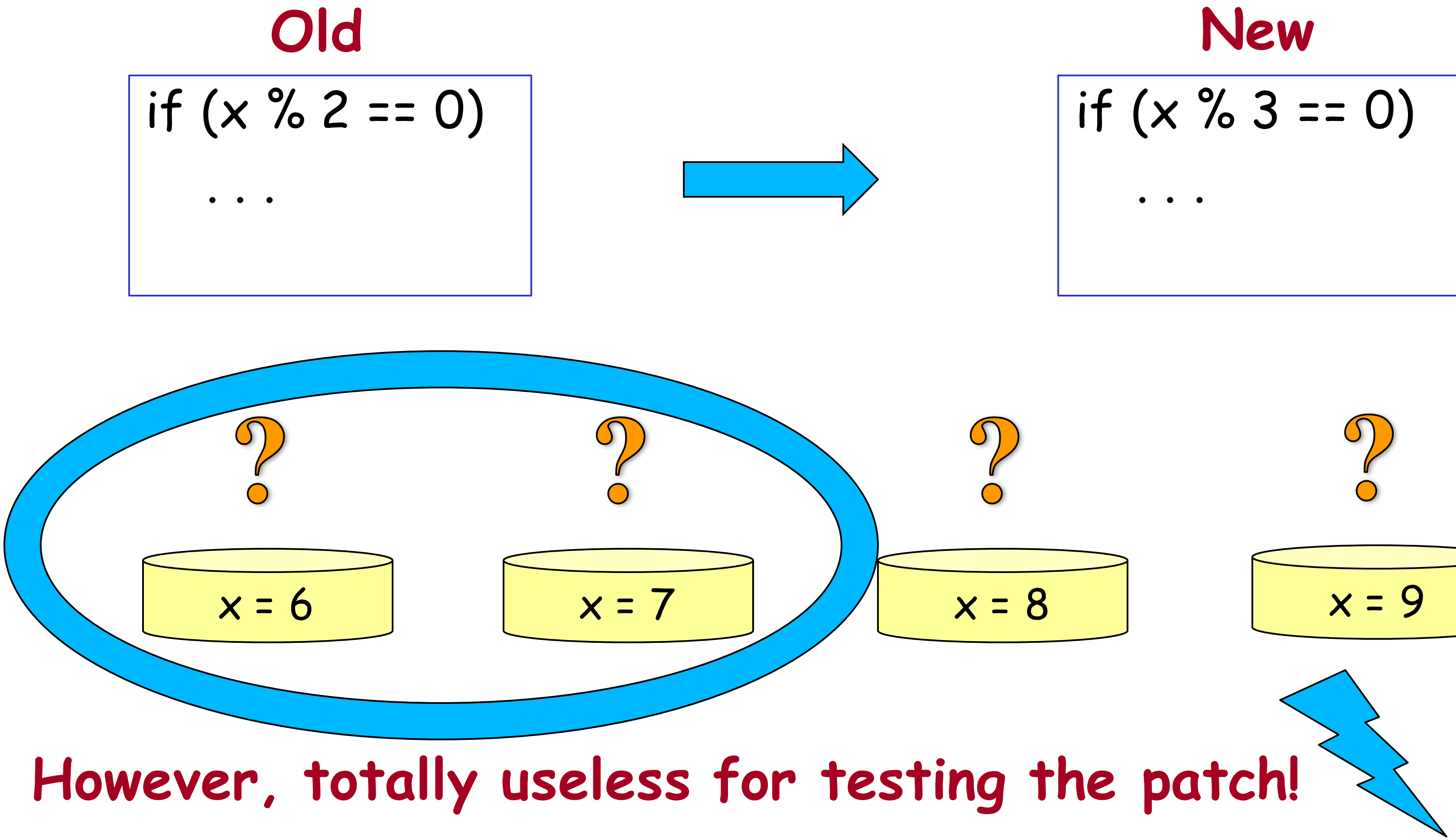
x = 8

x = 9

Full branch coverage in the new version

Is High Coverage Enough?

- If I change a statement, what tests should I add?



Old

```
if (x % 2 == 0)
    . . .
```

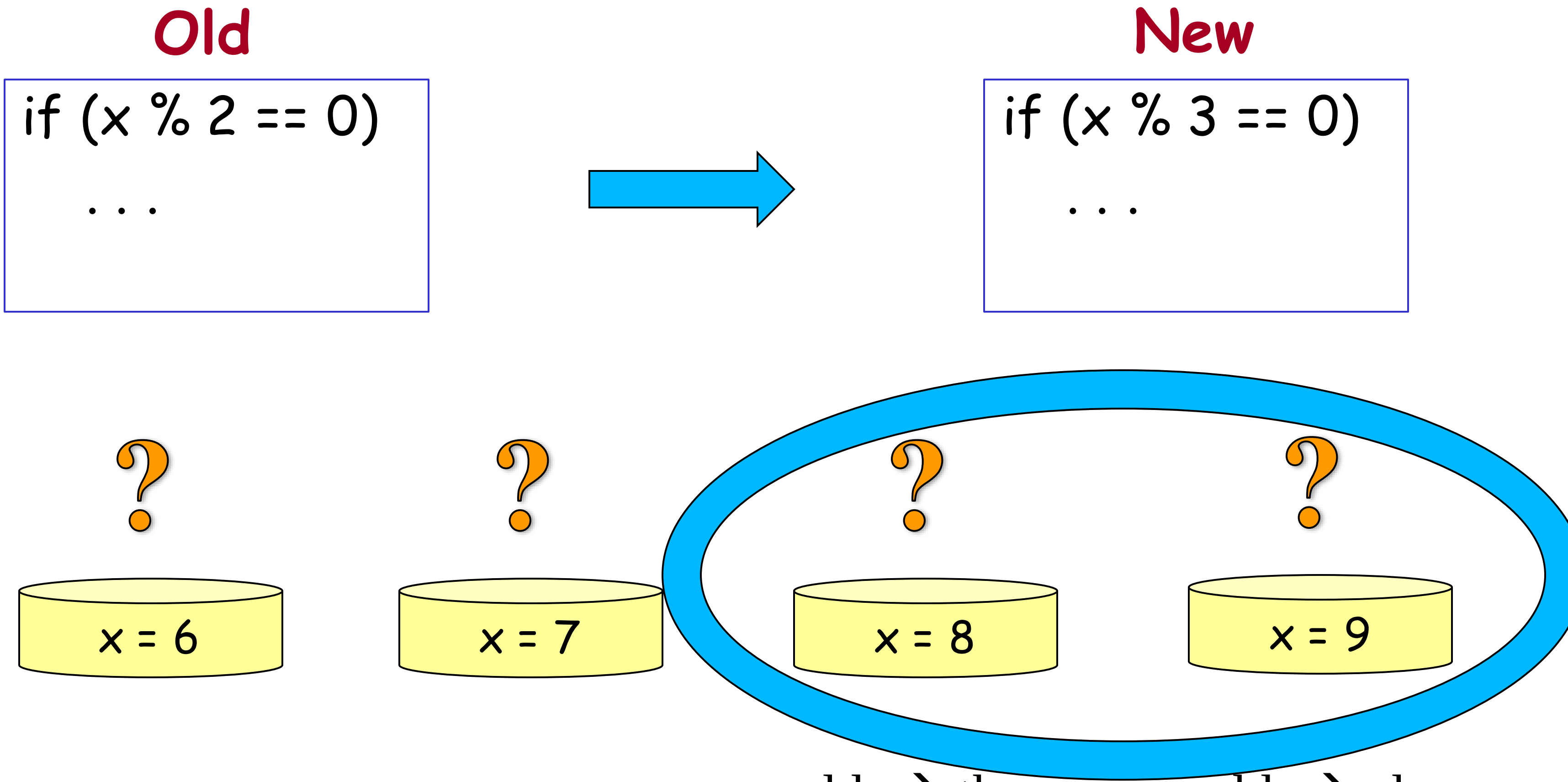
New

```
if (x % 3 == 0)
    . . .
```

However, totally useless for testing the patch!

Is High Coverage Enough?

- If I change a statement, what tests should I add?



Old

```
if (x % 2 == 0)
    . . .
```

New

```
if (x % 3 == 0)
    . . .
```

old → then
new → else

old → else
new → then

Shadow Symbolic Execution

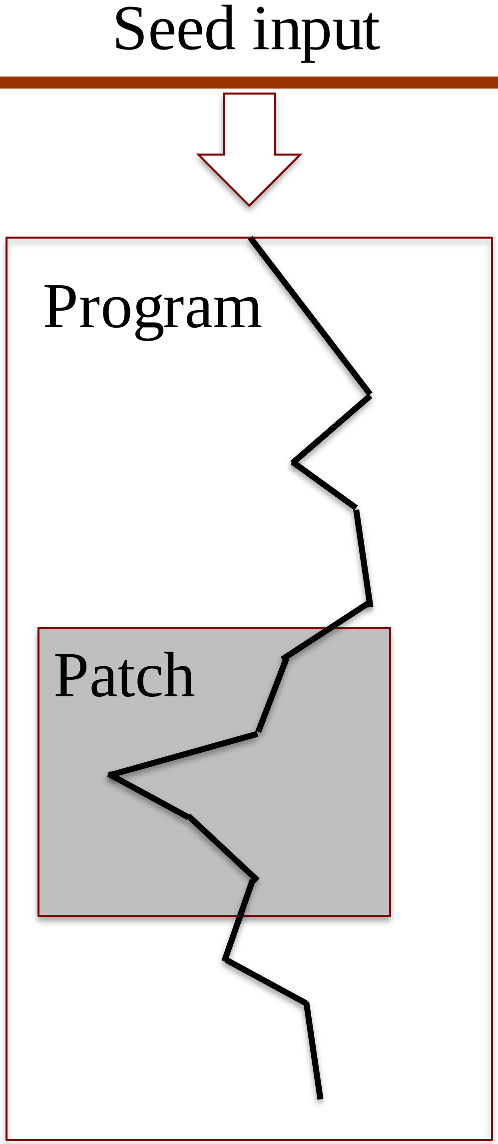
Automatically generate inputs that trigger different behaviors in the two versions

The novelty of shadow symbolic execution is to run the two versions together (in the same symbolic execution instance), with the old version shadowing the new

- Can prune large parts of the search space, for which the two versions behave identically
- Provides the ability to reason about specific values leading to simpler path constraints
- Is memory-efficient by sharing large parts of the symbolic constraints
- Does not execute unchanged computations twice

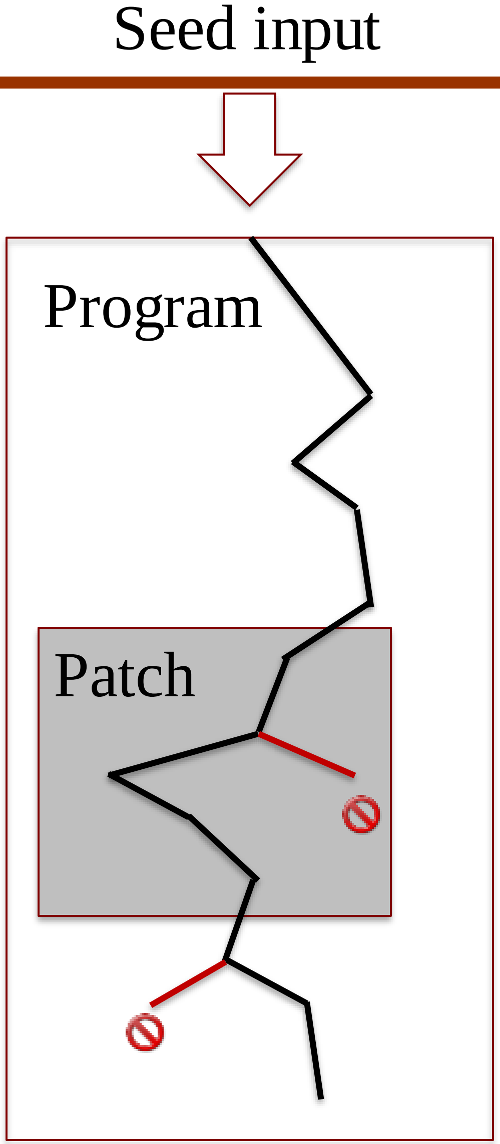
Behavioural Testing: Algorithm

1) Start with seed inputs covering patch
   - Or use KATCH if one is not available



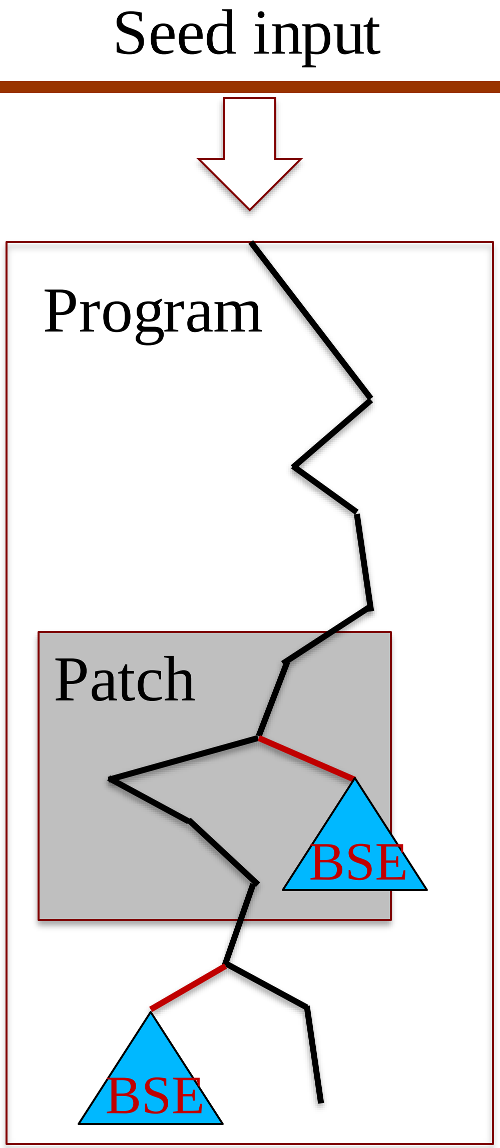
Behavioural Testing: Algorithm

1) Start with seed inputs covering patch
    - Or use KATCH if one is not available
2) Whenever a possible divergence found on those paths, generate a test case



Behavioural Testing: Methodology

1) Start with seed inputs covering patch
   - ➢ Or use KATCH if one is not available
2) Whenever a possible divergence found on those paths, generate a test input
3) Start bounded symbolic execution at each divergence point, to generate more divergent test inputs



Mismatches Found in `cut`

Input	Old	New
cut –c1-3,8- -output-d=: file (file is "abcdefg")	abc	abc + *buffer overflow*
cut -c1-7,8- --output-d=: file file contains "abcdefg"	abcdef	abcdef + *buffer overflow*
cut -b0-2,2- --output-d=: file file contains "abc"	abc	signal abort
cut -s -d: -f0- file (file is ":::\n:1")	:::\n:1	\n\n
cut –d: -f1,0- file (file is "a:b:c")	a:b:c	a

[Palikareva, Kuchta, Cadar, ICSE 2016]

Symbolic Execution for Evolving Software

- Testing and bounded verification of optimizations via crosschecking (equivalence checking)
  - Found semantic errors and performed bounded verification of SIMD and GPGPU optimizations
- KATCH: automatic patch testing guided by heuristics and program analyses
  - Automatically improved patch coverage and found errors in FindUtils, DiffUtils, BinUtils and Lighttpd
- Shadow symbolic execution: behavioral patch testing
  - Revealed regression bugs and expected divergences in complex Coreutils patches

Symbolic Execution for Automatically-Generated Patches

- Do automatically-generated patches present any additional challenges?

- Can patch generation and testing benefit from collaborating with each other?
  - Can patches be generated so that they are more easily tested?
  - Can testing technique take advantage of the structure of automatically-generated patches?