Shadow Symbolic Execution for Better Testing of Evolving Software

Cristian Cadar and Hristina Palikareva

Department of Computing

Imperial College London

ICSE NIER 2014

4 June 2014, Hyderabad, India

Patches, patches, patches…

- Software evolves, with new versions and patches being released frequently
- Patches add new features, fix existing bugs, improve performance, usability, etc.
- But are usually poorly tested, and oftentimes introduce new bugs and vulnerabilities

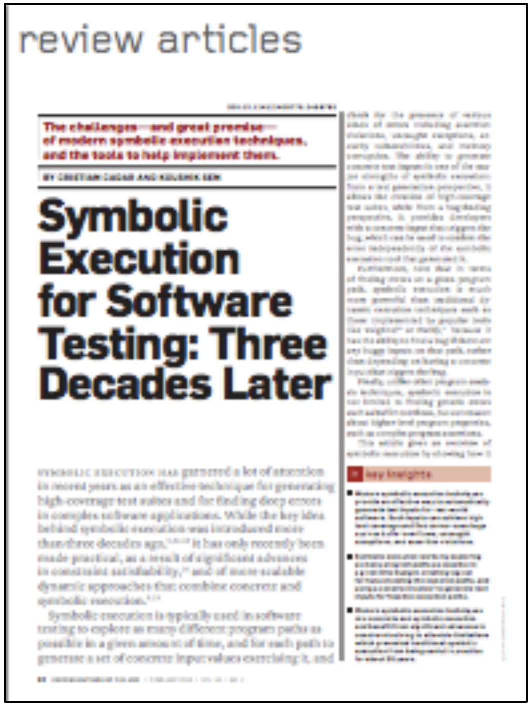
70% of the sys admins interviewed refuse to upgrade

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

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***
  - For each path, can generate a ***concrete input triggering the path***

Dynamic Symbolic Execution



review articles

The challenges—and great promise—of modern symbolic execution techniques, and the tools to help implement them.

BY CRISTIAN CADAR AND KOUSHIK SEN

Symbolic Execution for Software Testing: Three Decades Later

SYMBOLIC EXECUTION HAS garnered a lot of attention in recent years as an effective technique for generating high-coverage test suites and for finding deep errors in complex software applications. While the key idea behind symbolic execution was introduced more than three decades ago, it has only recently been made practical, as a result of significant advances in constraint satisfiability, and of more scalable dynamic approaches that combine concrete and symbolic execution.

Symbolic execution is typically used in software testing to explore as many different program paths as possible in a given amount of time, and for each path to generate a set of concrete input values exercising it, and [illegible]

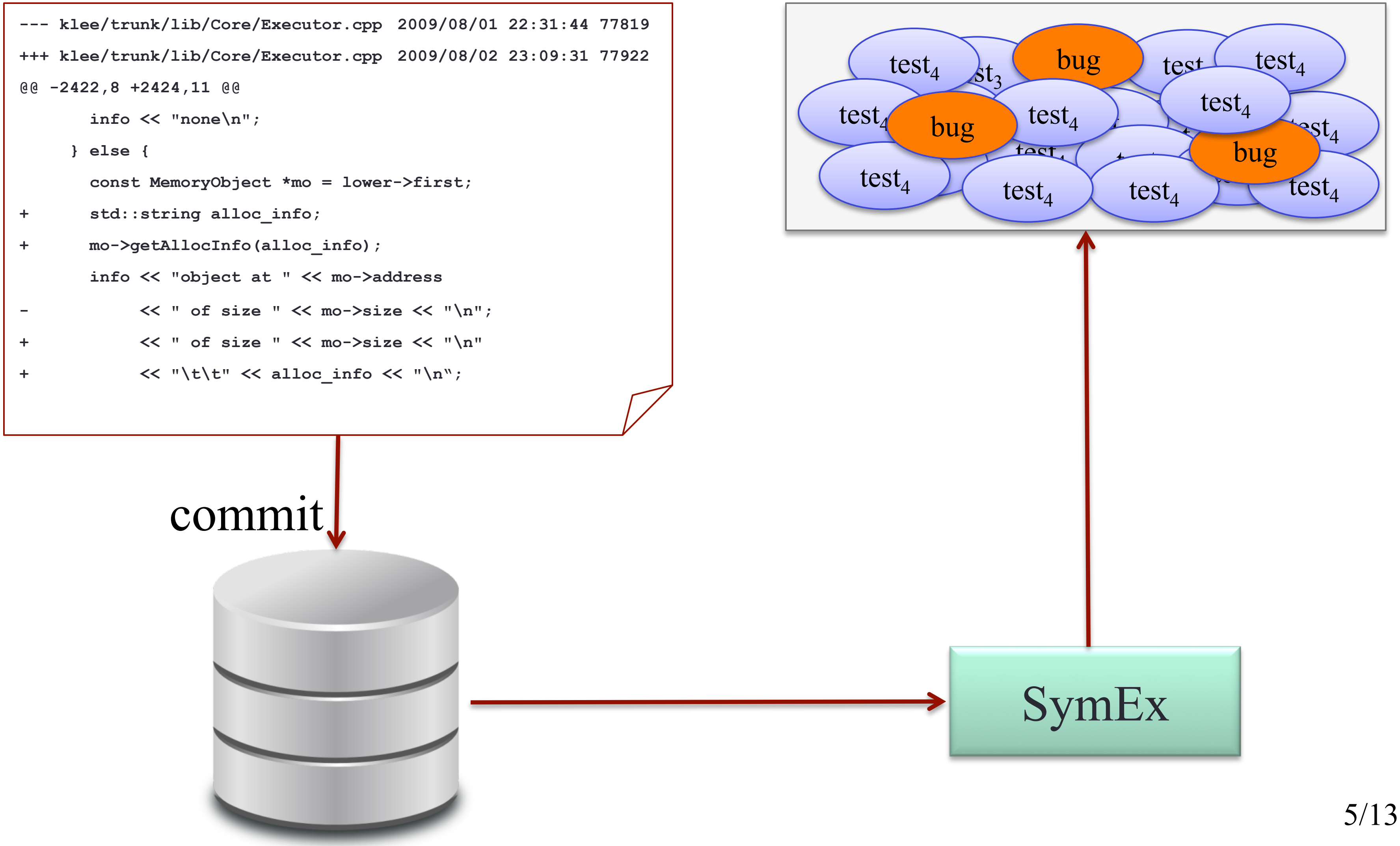
key insights

[illegible]

[CACM 2013]

- Received significant interest in the last few years
- Most work on whole program testing/bug-finding
- Recent focus on evolving software
  - Person et al. FSE’08, PLDI’11
  - Babic et al, ISSTA’11
  - Bohme et al. ICSE’13, FSE’13
  - Marinescu and Cadar, SPIN’12, FSE’13
  - etc.

SymEx for Testing Software Patches



```
--- 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";
```

Generate Inputs to Cover Each Line in the Patch

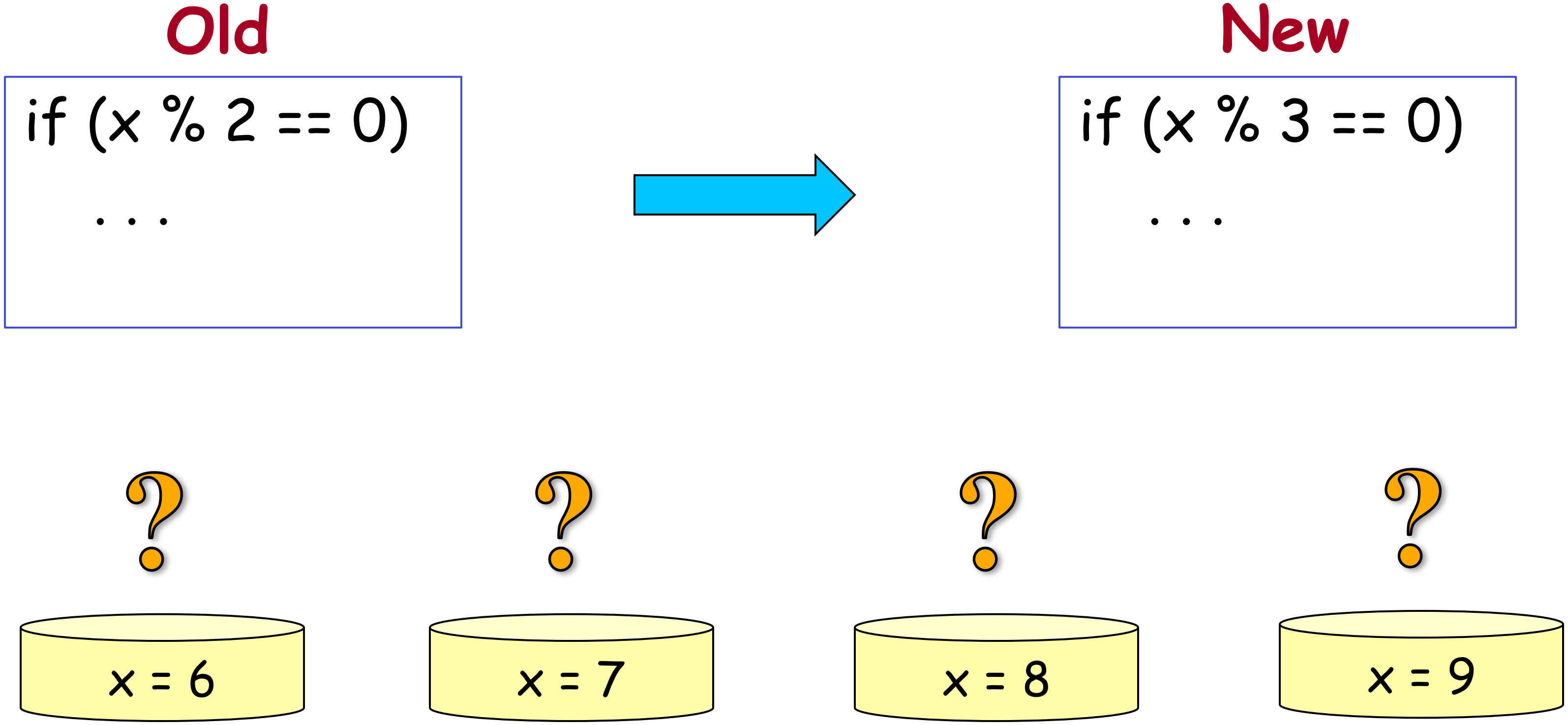
Our symex tool KATCH

- Tested several hundreds patches
- Significantly increased patch coverage
- Found (crash) bugs in the process
  - Unreachable by standard symbolic execution given similar time budget

[Marinescu and Cadar, SPIN'12, ESEC/FSE'13]

Is Line Coverage Enough?

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



Old

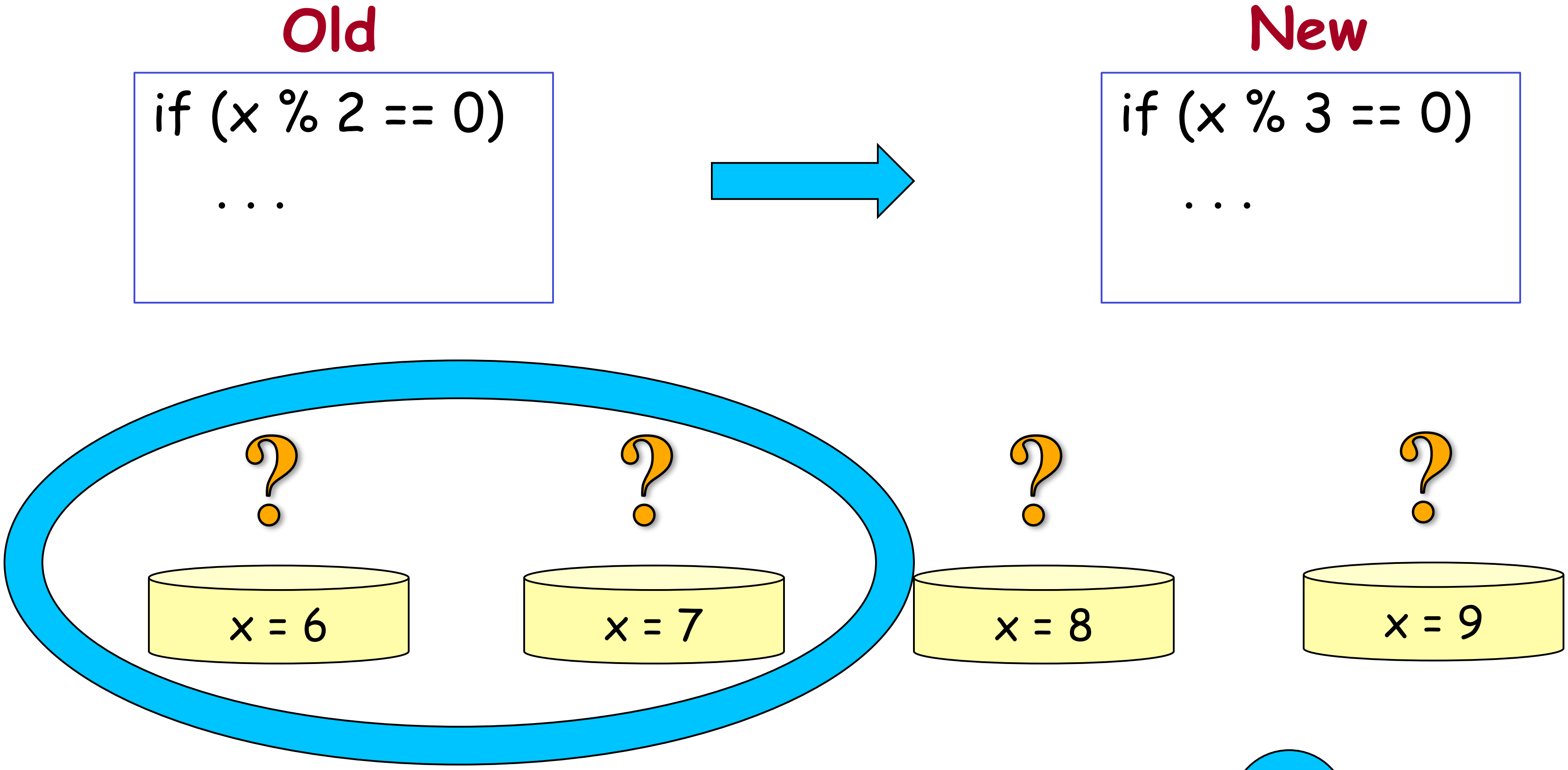
```
if (x % 2 == 0)
   . . .
```

New

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

Is Line Coverage Enough?

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



Old

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

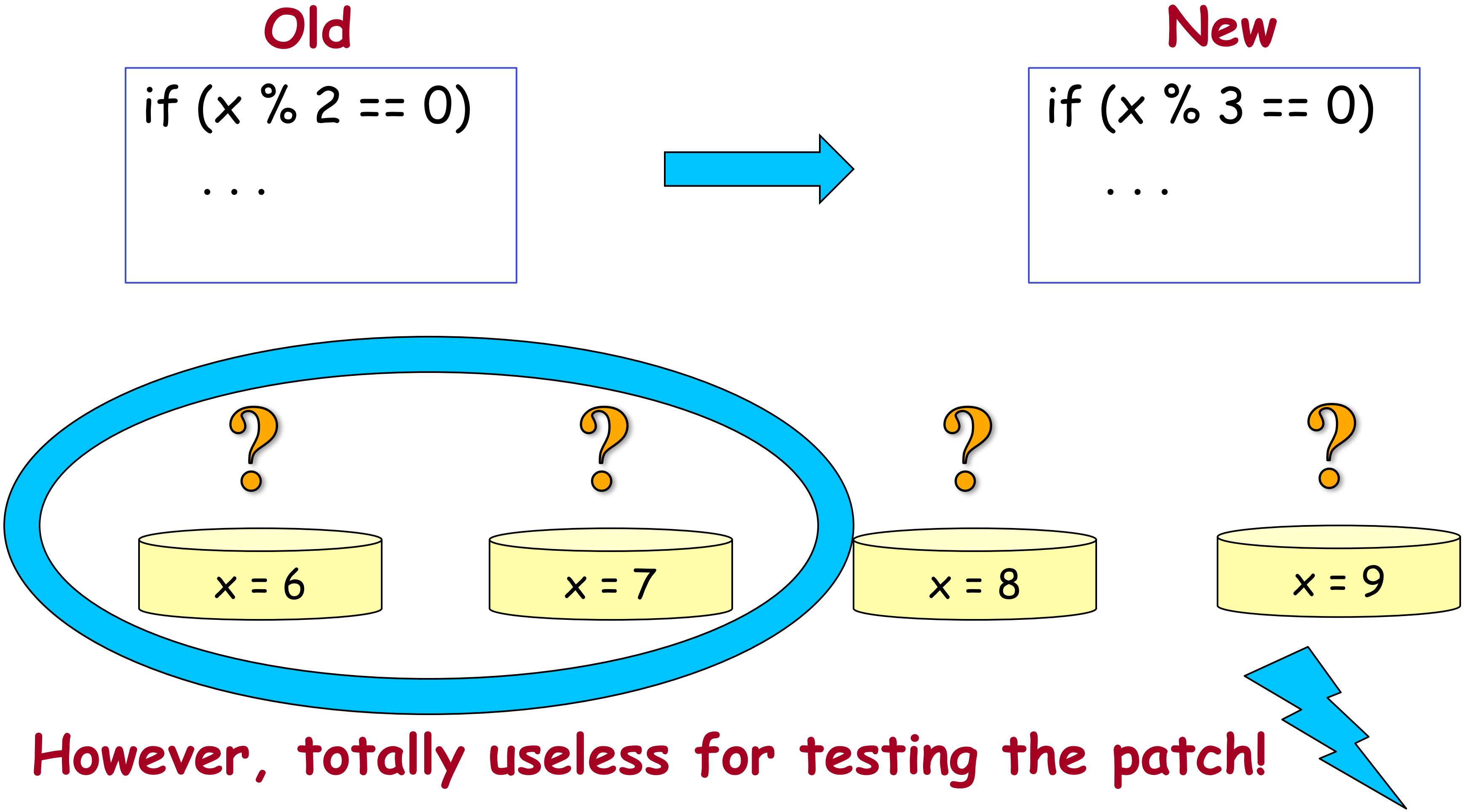
New

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

Full branch coverage in the new version

Is Line Coverage Enough?

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



Old

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

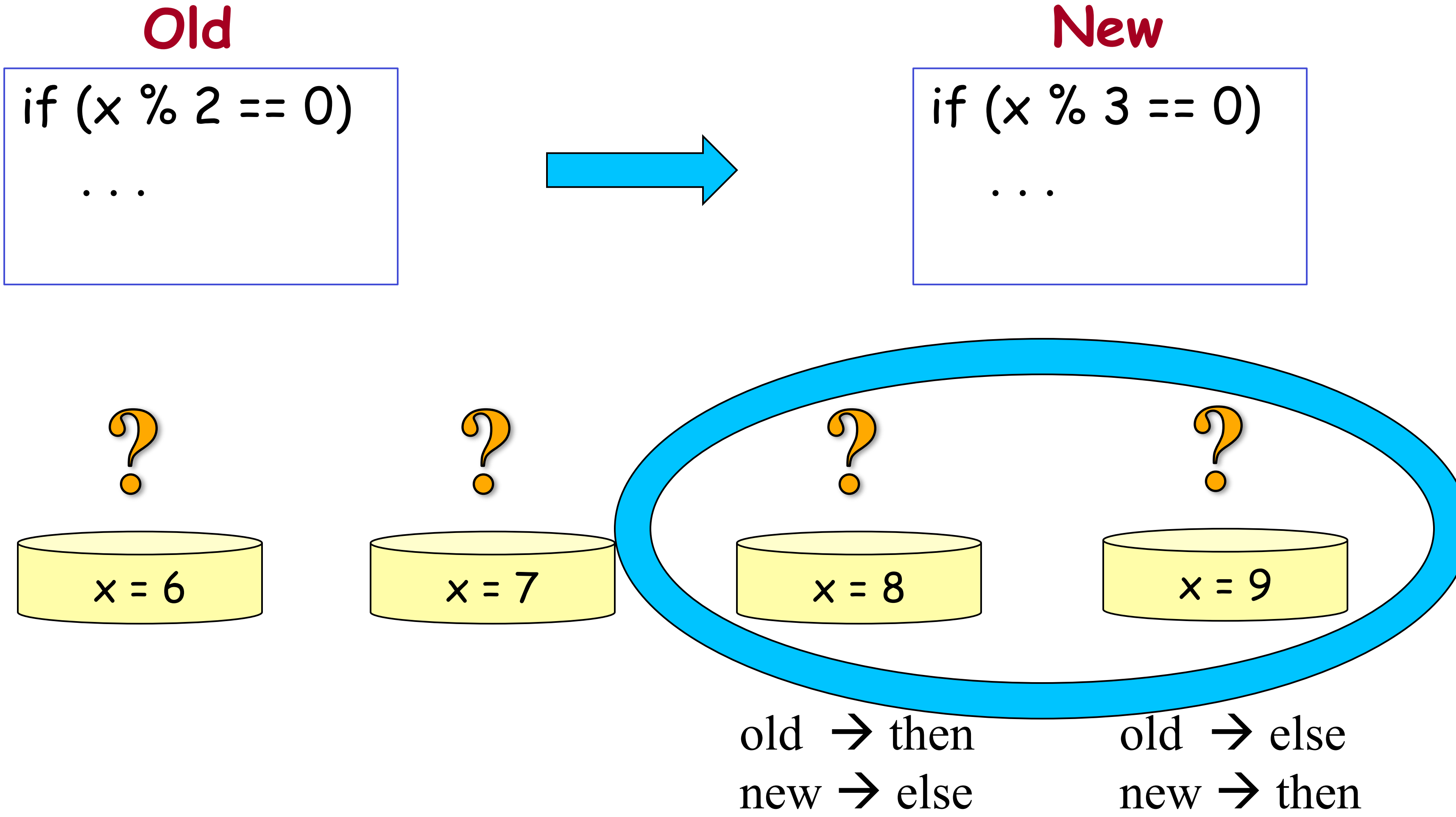
New

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

However, totally useless for testing the patch!

Is Line Coverage Enough?

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



Old

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

New

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

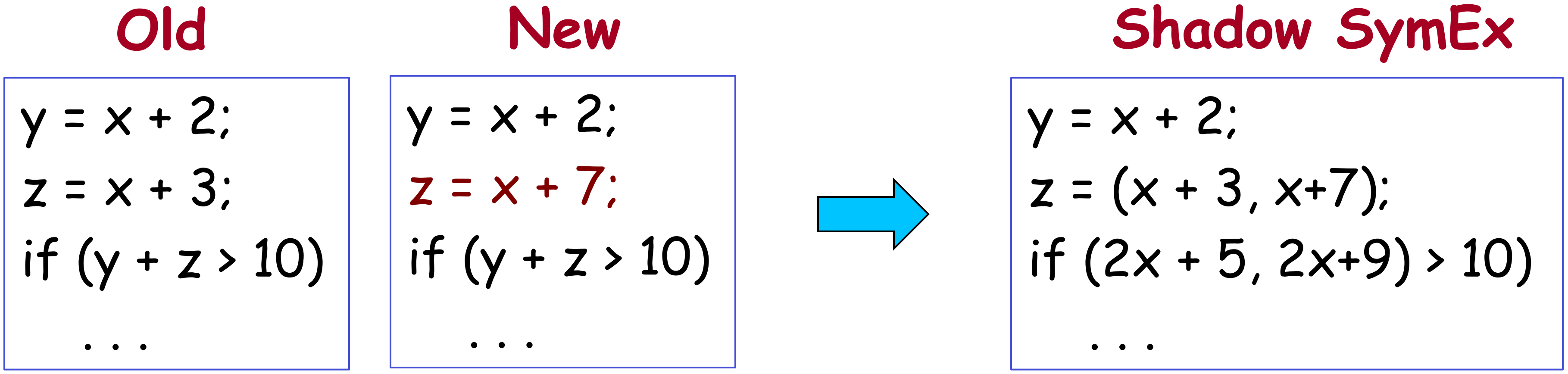
old → then
new → else

old → else
new → then

Shadow Symbolic Execution

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

- Provides the ability to reason about specific values and prune large parts of the search space



Old

```
y = x + 2;
z = x + 3;
if (y + z > 10)
   . . .
```

New

```
y = x + 2;
z = x + 7;
if (y + z > 10)
   . . .
```

Shadow SymEx

```
y = x + 2;
z = (x + 3, x+7);
if (2x + 5, 2x+9) > 10)
   . . .
```

Shadow Symbolic Execution

```
y = x + 2;
z = (x + 3, x + 7);
if (2x + 5, 2x + 9) > 10)
    . . .
```

No need to explore the else side of the branch, potentially pruning a huge # of paths.

We only need to explore the then path under the constraint **$1 \le x \le 2$**

$(2x+5 > 10) \wedge (2x+9 \le 10)$

$(2x+5 \le 10) \wedge (2x+9 > 10)$

No solutions

$x = 1$

$1 \le x \le 2$

*Assumes the current path constraints allow no arithmetic overflow, and no further uses of z

Shadow Symbolic Execution

Challenges

- Map statements from one version to another (static +dynamic analysis)
- Deal with changes in multiple parts of the program (when can we still prune?)

Opportunities (Potential impact)

- Prune large parts of the search space, for which the two versions behave identically
- Obtain simpler constraints
- Save memory by sharing large parts of the symbolic store (symbolic constraints)
- Find bugs in patches quicker, add relevant inputs to the regression test suite

Shadow Symbolic Execution for Better Testing of Evolving Software

Cristian Cadar and Hristina Palikareva

Department of Computing

Imperial College London

ICSE NIER 2014
4 June 2014, Hyderabad, India