Shadow Symbolic Execution for Better Testing of Evolving Software

Cristian Cadar and Hristina Palikareva
Department of Computing
Imperial College London
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

Cameri, 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

- 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" << 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)
  ...
```

- If I change a statement, what tests should I add?
Is Line Coverage Enough?

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

Old

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

```java
x = 6
x = 7
```

New

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

```java
x = 8
x = 9
```

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)
...

x = 6
x = 7
x = 8
x = 9

However, totally useless for testing the patch!
Is Line Coverage Enough?

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

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x % 2 == 0)</td>
<td>if (x % 3 == 0)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>x = 6</td>
<td>x = 8</td>
</tr>
<tr>
<td>x = 7</td>
<td>x = 9</td>
</tr>
<tr>
<td>old $\rightarrow$ then</td>
<td>old $\rightarrow$ else</td>
</tr>
<tr>
<td>new $\rightarrow$ else</td>
<td>new $\rightarrow$ then</td>
</tr>
</tbody>
</table>
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

\[ y = x + 2; \]
\[ z = x + 3; \]
\[ \text{if } (y + z > 10) \]

\[ \ldots \]

New

\[ y = x + 2; \]
\[ z = x + 7; \]
\[ \text{if } (y + z > 10) \]

\[ \ldots \]

Shadow SymEx

\[ y = x + 2; \]
\[ z = (x + 3, x+7); \]
\[ \text{if } (2x + 5, 2x+9) > 10) \]

\[ \ldots \]
Shadow Symbolic Execution

\[ y = x + 2; \]
\[ z = (x + 3, x + 7); \]
\[ \text{if } (2x + 5, 2x + 9) > 10) \]

\[ (2x+5 > 10) \land (2x+9 \leq 10) \]

\[ (2x+5 \leq 10) \land (2x+9 > 10) \]

\( x = 1 \)

\[ 1 \leq x \leq 2 \]

No solutions

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

<table>
<thead>
<tr>
<th>Challenges</th>
<th>Opportunities (Potential impact)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Map statements from one version to another (static + dynamic analysis)</td>
<td>• Prune large parts of the search space, for which the two versions behave identically</td>
</tr>
<tr>
<td>• Deal with changes in multiple parts of the program (when can we still prune?)</td>
<td>• Obtain simpler constraints</td>
</tr>
<tr>
<td></td>
<td>• Save memory by sharing large parts of the symbolic store (symbolic constraints)</td>
</tr>
<tr>
<td></td>
<td>• Find bugs in patches quicker, add relevant inputs to the regression test suite</td>
</tr>
</tbody>
</table>
Shadow Symbolic Execution for Better Testing of Evolving Software

Cristian Cadar and Hristina Palikareva
Department of Computing
Imperial College London