Targeted Program Transformations for Symbolic Execution

Cristian Cadar
Software Reliability Group
Department of Computing

Imperial College
London
Program analysis 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

Many dynamic symbolic execution/concolic tools available as open-source:

- CREST, KLEE, SYMBOLIC JPF, etc.

Started to be adopted/tried out in the industry:

- SAGE (Microsoft)
- SYMBOLIC JPF (NASA, Fujitsu, etc.)
- KLEE (Fujitsu, Hitachi, Citrix, etc.)
- APOLLO (IBM), etc. etc.
KLEE [http://klee.github.io]

Popular open-source engine:

- 1000+ clones per month
- 100+ forks on GitHub
- Many popular systems built on top of it (KleeNet, Cloud9, GKLEE, KLEE-MultiSolver, etc.)
- Lots of research ideas explored using KLEE as a platform
KLEE and LLVM

Weird phenomenon
Changing LLVM versions would sometimes result in HUGE performance differences

Performance of symbolic execution can vary dramatically across semantically-equivalent programs
To precompute or not to precompute

<table>
<thead>
<tr>
<th>Unoptimized</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>int get_value(int k){</td>
<td>int values[1000] = {0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, ... };</td>
</tr>
<tr>
<td>return k * k * k;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>int foo(unsigned k) {</td>
</tr>
<tr>
<td>// precond: k &lt; 1000</td>
<td>if (values[k] &gt; 100000</td>
</tr>
<tr>
<td>int foo(unsigned k) {</td>
<td>return 0;</td>
</tr>
<tr>
<td>if (get_value(k) &gt; 100000</td>
<td></td>
</tr>
<tr>
<td>return 0;</td>
<td>}</td>
</tr>
</tbody>
</table>

0.2s vs. 50s
250x slower!

\[ k^3 > 100,000 \] vs. \[ \text{values}[k] > 100,000 \land \text{values}[0] = 0 \land \text{values}[1] = 1 \land \ldots \]
To -O2 or not to -O2

**Compiler optimisations example**

```c
int bar(int a[10]) {
    int count=0, i;
    for (i=0; i<10; i++)
        if (a[i] > 0)
            count++;
    if (count == 10)
        printf("Success\n");
    return count;
}
```

- -O0: 23s
- -O2: 0.04s (575x faster!)

**Explanation:**

- -O2 transforms the if into a `select(a[i]>0, count+1, count)` which KLEE sends directly to the solver.

Essentially -O2 has merged the paths inside the loop.
How do I switch this?

Switch example

```c
int expensive(int x) {
    int bits = 0, i;
    for (i=0; i<< i))
        bits++;
    return bits;
}

int foo(int x, int y) {
    switch (x) {
        case 1: return expensive(y+1);
        case 2: return expensive(y+2);
        case 3: return expensive(y+3);
        case 4: return expensive(y+4);
        default: return x/y;
    }
}
```

Binary search: 23s to bug
Linear search: TIMEOUT 1h

KLEE 6118403fa4 with LLVM 2.9, BFS, STP 1668, Intel Core2 Duo CPU E8400 at 3.00GHz, Ubuntu 14.04
Testability transformations = key ingredients in symex
Testability Transformations for SymEx

Semantics-preserving

- Developers (optimisations, refactorings)
- Compilers (optimisations, code generation)
- Choice of abstraction (source, binary, intermediate language, etc.)

Semantics-altering

- Approximations (reals instead of FP)
- Shrinking large memory objects
- Assigning concrete values to part of the input

Testability transformations introduced in the context of SBST by Harman et al. [TSE 2004]
Could enable symex to scale to larger applications

Faster constraint solving
• E.g., precomputed lookup example

More targeted path exploration
• E.g., path merging examples

More application types
• E.g., floating point code

More generally, can we:
• write programs friendly to symex analysis?
• automatically transform programs to be symex-friendly?
Case study 1: paper reporting 10x improvement in performance on top of some prior KLEE experiments. Is this due to:

(a) the technique itself
(b) different LLVM versions
(c) different compiler options

Case study 2: study reporting a 10x performance difference between KLEE and CREST. Is this due to:

(a) the techniques in KLEE vs CREST
(b) the intermediate language used by LLVM and CIL
(c) different compiler optimisations being performed
Conclusion

Testability transformations should be a key ingredient in symex. We should:

• **Account for them**: essential for understanding ongoing research ideas and experiments in this area

• **Understand them**: improve performance by carefully enabling and disabling existing transformations such as compiler optimisations

• **Provide guidelines for writing symex-friendly code**: similar in spirit to existing interactive verifiers

• **Design targeted transformations**: both semantics-preserving and semantics-altering, and addressing both constraint solving and path exploration challenges
Looking for postdoc applicants to work in this area:
http://srg.doc.ic.ac.uk/vacancies/