Towards Efficient Data-flow Test Data Generation Using KLEE

Chengyu Zhang\textsuperscript{1}, Ting Su\textsuperscript{2}, Yichen Yan\textsuperscript{1}, Ke Wu\textsuperscript{3}, Geguang Pu\textsuperscript{1}

\textsuperscript{1}East China Normal University, China
\textsuperscript{2}Nanyang Technological University, Singapore
\textsuperscript{3}National Trusted Embedded Software Engineering Technology Research Center, China

2018.4.19
Outline

Introduction

Approach

Implementation

Evaluation

Application
Data-flow testing identifies potential bugs by checking the correctness of variable definitions by observing their corresponding uses. Several empirical studies have demonstrated that data-flow testing is more effective in fault detection than control-flow testing.
double power(int x, int y) {
    int exp; double res;
    if (y > 0) exp = y;
    else exp = -y;
    res = 1;
    while (exp != 0) {
        res *= x;
        exp -= 1;
    }
    if (y <= 0)
        if (x == 0)
            abort;
        else
            return 1.0 / res;
    return res;
}
An example

double power(int x, int y) {
    int exp; double res;
    if (y > 0) exp = y;
    else exp = -y;
    res = 1;  // definition
    while (exp != 0) {
        res *= x;
        exp -= 1;
    }
    if (y <= 0)
        if (x == 0)
            abort;
        else
            return 1.0/res;  // use
    return res;
}
double power(int x, int y) {
    int exp; double res;
    if (y > 0) exp = y;
    else exp = -y;
    res = 1; // definition
    while (exp != 0) {
        res *= x; // redefinition
        exp -= 1;
    }
    if (y <= 0) {
        if (x == 0)
            abort;
        else
            return 1.0/res; // use
    }
    return res;
}
Three heuristic search strategies:

- Cut point guided search
- Backtrack strategy
- Redefinition Path Pruning
Cut point guided search

Definition (Cut Point)

For $du(l_d, l_u, v)$
cut points are a sequence of critical control points $c_1, \ldots, c_i, \ldots, c_n$ that must be passed through in succession by any control flow paths that cover this pair.

$c_1 \gg I \ldots c_i \gg I l_d \gg I \ldots c_n \gg I l_u$.

Cut points: $\{l_1, l_3, l_d, l_6, l_u\}$
double power(int x, int y) {
    int exp; double res;
    if (y > 0) exp = y;
    else exp = -y;
    res = 1;
    while (exp != 0) {
        res *= x;
        exp -= 1;
    }
    if (y <= 0)
        if (x == 0)
            abort;
        else
            return 1.0 / res;
    return res;
}
Backtrack strategy

Definition (Backtrack formula)

\[ state\_weight(es) = \frac{1}{d^2} + \frac{1}{i^2} \]  (1)

- \(d\): instruction distance toward the next uncovered cut point
- \(i\): number of instructions since the last new instruction have been covered
```c
double power(int x, int y) {
    int exp; double res;
    if (y > 0) exp = y;
    else exp = -y;
    res = 1; // definition
    while (exp != 0) {
        res *= x; // redefinition
        exp -= 1;
    }
    if (y <= 0)
        if (x == 0)
            abort;
        else
            return 1.0/res; // use
    return res;
}
```
Approach Overview

- Cut-point Guided Search
- Backtrack
- Redefinition Path Pruning
- Enhanced Path Exploration

- Execution State
- Execution State with Redefinition
- Normal Exploration
- Exploration with Cut Point
- Exploration with Backtrack

valid path
Why We Choose KLEE?

CAUT: https://www.lab301.cn/caut/

- **Automated coverage-driven test data generation using dynamic symbolic execution**
  SERE 2014

- **Combining symbolic execution and model checking for data flow testing.**
  ICSE 2015
Why We Choose KLEE?

- Open-source
- Robust
- Scalable
Data-flow Testing Framework

- Source file
- CIL
- Data-flow Information
- Data-flow Information Table
- Executor
- Data-flow Searcher
- Special Function Handler
- Data-flow Test Suite
- Data-flow Coverage Results
## Experiment Setup

We evaluated the approach on 30 subjects.

<table>
<thead>
<tr>
<th>Subjects</th>
<th>#Sub</th>
<th>#LOC</th>
<th>#DU pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>Previous Literature</td>
<td>7</td>
<td>449</td>
<td>346</td>
</tr>
<tr>
<td>SIR</td>
<td>7</td>
<td>2,687</td>
<td>1,409</td>
</tr>
<tr>
<td>SV-COMP (ntdriver)</td>
<td>6</td>
<td>7,266</td>
<td>2,691</td>
</tr>
<tr>
<td>SV-COMP (ssh)</td>
<td>10</td>
<td>5,249</td>
<td>18,347</td>
</tr>
</tbody>
</table>
Performance Statistics of Different Search Strategies

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>Depth First Search</td>
</tr>
<tr>
<td>RSS</td>
<td>Random State Search</td>
</tr>
<tr>
<td>RSS-COS:md2u</td>
<td>RSS interleaved with Min-Dist-to-Uncovered heuristic</td>
</tr>
<tr>
<td>SDGS</td>
<td>Shortest Distance Guided Search</td>
</tr>
<tr>
<td>CPGS</td>
<td>Cut Point Guided Search</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>DFS (N)</th>
<th>M (SIQR)</th>
<th>RSS (N)</th>
<th>M (SIQR)</th>
<th>RSS-MD2U (N)</th>
<th>M (SIQR)</th>
<th>SDGS (N)</th>
<th>M (SIQR)</th>
<th>CPGS (N)</th>
<th>M (SIQR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>6218</td>
<td>9.13</td>
<td>7883</td>
<td>13.54</td>
<td>8091</td>
<td>21.89</td>
<td>8035</td>
<td>13.51</td>
<td>8300</td>
<td>12.96</td>
</tr>
</tbody>
</table>

CPGS achieves the **best performance** in the symbolic execution approach.
Model Checking based Approach

Towards Efficient Data-flow Test Data Generation
https://arxiv.org/abs/1803.10431

<table>
<thead>
<tr>
<th>Subject</th>
<th>BLAST</th>
<th>CPAchecker</th>
<th>CBMC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>I</td>
<td>U</td>
</tr>
<tr>
<td>Total</td>
<td>6720</td>
<td>10199</td>
<td>5874</td>
</tr>
</tbody>
</table>
# Model Checking VS. Symbolic Execution on Data-flow Testing

<table>
<thead>
<tr>
<th>Subjects</th>
<th>#Sub</th>
<th>#LOC</th>
<th>#DU pair</th>
<th>Average Coverage</th>
<th>Median Time (s/pair)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>KLEE</td>
<td>CPA</td>
</tr>
<tr>
<td>Previous Literature</td>
<td>7</td>
<td>449</td>
<td>346</td>
<td>60%</td>
<td>72%</td>
</tr>
<tr>
<td>SIR</td>
<td>7</td>
<td>2,687</td>
<td>1,409</td>
<td>57%</td>
<td>60%</td>
</tr>
<tr>
<td>SV-COMP (ntdriver)</td>
<td>6</td>
<td>7,266</td>
<td>2,691</td>
<td>75%</td>
<td>51%</td>
</tr>
<tr>
<td>SV-COMP (ssh)</td>
<td>10</td>
<td>5,249</td>
<td>18,347</td>
<td>29%</td>
<td>31%</td>
</tr>
</tbody>
</table>

KLEE can easily achieve nearly 60% of data-flow coverage within less than 1 second for each pair in the subjects from previous literature and SIR.
We planned to implement the data-flow testing approach in our cloud-based unit test framework.
SmartUnit (ICSE-SEIP’18)

- Automatically generate testing report.
- Automatically generate test case.
- Automatically insert function stub.
- Cloud-based platform for corporations.

It’s adapted for LDRA Testbed\(^1\), Tessy\(^2\) and other popular automated testing tools for embedded systems. It supports three common control-flow coverage criteria: statement coverage, branch coverage and MC/DC (Modified Condition Decision Coverage).

\(^1\)http://ldra.com/industrial-energy/products/ldra-testbed-tbvision/
SmartUnit (ICSE-SEIP’18)

<table>
<thead>
<tr>
<th>Statement Coverage (#Functions)</th>
<th>Branch Coverage (#Functions)</th>
<th>MC/DC Coverage (#Functions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%-50%</td>
<td>0%-50%</td>
<td>0%-50%</td>
</tr>
<tr>
<td>50%-99%</td>
<td>50%-99%</td>
<td>50%-99%</td>
</tr>
<tr>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>9.3%</td>
<td>14.3%</td>
<td>47.8%</td>
</tr>
<tr>
<td>17.5%</td>
<td>12.5%</td>
<td>13.6%</td>
</tr>
<tr>
<td>73.2%</td>
<td>73.2%</td>
<td>38.6%</td>
</tr>
</tbody>
</table>

SmartUnit achieves the 100% statement and branch coverage on most of the functions.
We have the cooperation with:

- China Academy of Space Technology
- CASCO Signal Ltd.
- The 32nd Institute of China Electronics Technology Group Corporation
- ...

SmartUnit: Empirical Evaluations for Automated Unit Testing of Embedded Software in Industry
ICSE-SEIP, May 27-June 3, 2018
Conclusion


Chengyu Zhang  2018.4.19
Towards Efficient Data-flow Test Data Generation Using KLEE

Chengyu Zhang\textsuperscript{1}, Ting Su\textsuperscript{2}, Yichen Yan\textsuperscript{1}, Ke Wu\textsuperscript{3}, Geguang Pu\textsuperscript{1}

\textsuperscript{1}East China Normal University, China  
\textsuperscript{2}Nanyang Technological University, Singapore  
\textsuperscript{3}National Trusted Embedded Software Engineering Technology Research Center, China

2018.4.19
Definition (Program Paths)

Two kinds of program paths, i.e., control flow paths and execution paths are distinguished during data-flow testing. Control flow paths are the paths from the control flow graph of the program under test, which abstract the flow of control. Execution paths are driven by concrete program inputs, which represent dynamic program executions. Both of them can be represented as a sequence of control points (denoted by line numbers), e.g., $l_1, \ldots, l_i, \ldots, l_n$. 
Definition (Def-use Pair)

The test objective of data-flow testing is referred as a def-use pair, denoted by $du(l_d, l_u, v)$. Such a pair appears when there exists a control flow path that starts from the variable definition statement $l_d$ (or the def statement in short), and then reaches the variable use statement $l_u$ (or the use statement in short), but no statements on the subpaths from $l_d$ to $l_u$ redefine the variable $v$. 
Definition (Data-flow Testing)

Data-flow testing requires to generate at least one test case $t$ for each def-use pair $(l_d, l_u, v)$ in the program $P$ under test. The test input $t$ should drive an execution path $p$ that covers the variable definition statement at $l_d$, and then covers variable use statement at $l_u$, but without covering any redefinition statements w.r.t $v$, i.e., the subpath from $l_d$ to $l_u$ is a def-clear path. Such a test adequacy requirement is called all def-use coverage\(^a\) in data-flow testing.

\(^a\)In this paper, we follow the all def-use coverage defined by Rapps and Weyuker, since almost all of the literature that followed uses or extends this definition.
**Definition (Cut Point)**

Given a def-use pair $du(l_d, l_u, v)$, its cut points are a sequence of critical control points $c_1, \ldots, c_i, \ldots, c_n$ that must be passed through in succession by any control flow paths that cover this pair. The latter control point is the immediate dominator of the former one, i.e., $c_1 \gg^I \ldots c_i \gg^I l_d \gg^I \ldots c_n \gg^I l_u$. Each control point in this sequence is called a cut point.