Probabilistic Symbolic Execution
A New Hammer

Willem Visser
Stellenbosch University

Joint work with Matt Dwyer, Jaco Geldenhuys, Corina Pasareanu, Antonio Filieri, ...

Probabilistic Symbolic Execution

Symbolic Execution + Model Counting
Saving the Whooping Crane
void test(int x, int y) {
    if (y == x*10)
        S0;
    else
        S1;
    if (x > 3 && y > 10)
        S2;
    else
        S3;
}

Symbolic Execution

Test(1,10) reaches S0,S3
Test(0,1)  reaches S1,S3
Test(4,11) reaches S1,S2
void test(int x, int y: 0..99) {
    if (y == x*10)
        S0;
    else
        S1;
    if (x > 3 && y > 10)
        S2;
    else
        S3;
}
LattE Model Counter

http://www.math.ucdavis.edu/~latte/

Count solutions for conjunction of Linear Inequalities
Things we can handle...

• Usage profiles (ICSE 2013)

• Domains
  – Linear Integer Arithmetic (ISSTA2012)
  – Floating point and non-linear (PLDI2014)
    • approximate
  – Data structures (SPIN2015)
  – Strings (CAV2015 by Tevfik Bultan)
A Path Condition defines the constraints on the inputs to execute a path

How likely is a PC to be satisfied?

# solutions to the PC

Domain Size

Assuming uniform distribution of values
Conditional and Path Probabilities

\[ P_c = \text{Prob} (c \mid PC) \]

\[ \frac{\text{Prob} (c \& PC)}{\text{Prob} (PC)} = P_c \]

\[ P'' = (1 - P_c) \times P \]

\[ P' = P_c \times P \]
Information Leakage via Side Channels
Pasareanu and Bultan

- Side channels produce a set of observables that partition a secret
  - Classically: execution time
- Shannon Entropy
  - Expected amount of information gain in terms of bits
- Probabilistic Symbolic Execution

\[ O = \{o_1, o_2, \ldots, o_m\} \]

\[ H(P) = - \sum_{i=1}^{m} p(o_i) \log_2(p(o_i)) \]

the probability of observing \( o_i \) is:

\[ p(o_i) = \frac{\sum_{\text{cost}(\pi_j)=o_i} \#(PC_j(h, l))}{\#D} \]
Information Leakage Example
from slides by Tevfik Bultan

```java
// binary 4-digit pin, D = 256

bool checkPIN(guess[]) {
    for (int I = 0; I < 4; i++)
        if (guess[i] != PIN[i])
            return false;
    return true;
}

PATHS:
1. Return false; 128 values
2. Return false; 64 values
3. Return false; 32 values
4. Return false; 16 values
5. Return true; 16 values

bool checkPINBetter(guess[]) {
    matched = true;
    for (int i = 0; i < 4; i++)
        if (guess[i] != PIN[i])
            matched = false;
        else
            matched = matched;
    return matched;
}

Assuming observable is time
H = 1.875

Assuming observable is output
H = 0.33729
```
(Java) Probabilistic Programming

• Combine general purpose programming with probability distributions to answer interesting questions.
  – (Easily) encode Bayesian Networks, Hidden Markov Models, etc. as a (Java) program with a few special keywords
    – probability(loc), observe(cond), flip(ratio)

• Using Probabilistic Symbolic Execution for inference
public static void FOSE() {
    boolean c1 = flip(0.5);
    boolean c2 = flip(0.5);
    observe(c1 || c2);
    if (c1) probability(1); 0.6667
}

public static void PRISMDiceExample() {
    int s = 0;
    int d = 0; // dice value
    while (true) {
        if (s==0) { s = flip(0.5) ? 1 : 2; }
        else if (s == 1) { s = flip(0.5) ? 3 : 4;}
        else if (s == 2) { s = flip(0.5) ? 5 : 6;}
        else if (s == 3) { if (flip(0.5)) { s = 1; }
            else { s = 7; d = 1; }}
        else if (s == 4) { s = 7; d = flip(0.5) ? 2 : 3;}
        else if (s == 5) { s = 7; d = flip(0.5) ? 4 : 5;}
        else if (s == 6) { if (flip(0.5)) { s = 2;}
            else { s = 7; d = 6;}}
        else { /* s = 7 */ break; }
    }
    probability(d); // probability of seeing each value for d
    0.16667 for all d
“Semantic” Difference Between Programs

On what percentage of the input space does \( P \) and \( P' \) give different outputs?

```java
public static void check(int a, int b, int c) {
    assert P(a, b, c) == P'(a, b, c);
}
```

Record path conditions when assertion fails and count their sizes then divide by total domain size to get \% difference
Difference Example

Boolean PP(int i, int j) {
    return i > j;
}
100% different

Boolean PP(int i, int j) {
    return i >= j;
}
99% different

Boolean PP(int i, int j) {
    return i != j;
}
50.5% different

Boolean PP(int i, int j) {
    return i == j;
}
49.5% different

Boolean PP(int i, int j) {
    return i < j;
}
1% different

Boolean P(int i, int j : 0..99) {
    return i <= j;
}
Taking an analytical look

Mutations

especially when used to seed faults
Mutation is Killed if there exist a test that fails on it

**Mutation Score** = \[ \frac{\# \text{ Killed}}{\# \text{ Mutations}} \]
Killing Mutations == Finding real errors?
Mutations have found another use

FAULT SEEDING

How good is my super-duper new bug finding tool at finding seeded faults?
How hard is it to kill a mutant?

Previous work: fixed the test suite

We consider **ALL** test inputs and show the influence of varying the oracle.
How hard is it to kill a mutant?

Birthplace more important than chicken or bull

Not hard at all

Spoiler Alert
What

How easy or hard is it to kill a mutant?

How

On what percentage of the input space does the oracle for the reference version and mutated version give different outputs?

diff == 0% => Equivalent Mutant
diff < threshold% => Stubborn Mutant
Implementation

• Listener for Symbolic PathFinder (SPF)
  – Traps calls to every bytecode instruction executed
• Collects path conditions when oracle differs
• Count the solutions to these with Green and Barvinok
• Also collects path conditions at the point of mutation and counts the sizes
  – Special NOP bytecode is pushed at this point
• Dumps a CSV file with the output
• Dockerfile to recreate image to run experiments
In the initial results

We saw something interesting
public static int classify(int i, int j, int k) {
    if ((i <= 0) || (j <= 0) || (k <= 0))
        return 4;
    int type = 0;
    if (i == j) type = type + 1;
    if (i == k) type = type + 2;
    if (j == k) type = type + 3;
    if (type == 0) {
        if ((i + j <= k) || (j + k <= i) || (i + k <= j)) type = 4;
        else type = 1;
        return type;
    }
    if (type > 3) type = 3;
    else if ((type == 1) && (i + j > k)) type = 2;
    else if ((type == 2) && (i + k > j)) type = 2;
    else if ((type == 3) && (j + k > i)) type = 2;
    else type = 4;
    return type;
}
public static int classify(int i, int j, int k) {
    if (((i <= 0) || (j <= 0) || (k <= 0))
        return 4;
    int type = 0;
    if (i == j) type = type + 1;
    if (i == k) type = type + 2;
    if (j == k) type = type + 3;
    if (type == 0) {
        if (((i + j <= k) || (j + k <= i) || (i + k <= j))
            type = 4;
        else type = 1;
        return type;
    }
    if (type > 3) type = 3;
    else if (((type == 1) && (i + j > k))
        type = 2;
    else if (((type == 2) && (i + k > j))
        type = 2;
    else if (((type == 3) && (j + k > i))
        type = 2;
    else type = 4;
    return type;
}

Only 3% of inputs pass here
## Results with Reachability
### Arithmetic + Constant Replacement

<table>
<thead>
<tr>
<th>Programs</th>
<th>Muts</th>
<th>Stubborn &lt; 0.1%</th>
<th>Really &lt; 0.1%</th>
<th>Always 100%</th>
<th>Easy &gt; 33%</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRI-YHJ</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>TRI-V1</td>
<td>19</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>TRI-V2</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>TCAS</td>
<td>38</td>
<td>8</td>
<td>4</td>
<td>9</td>
<td>28</td>
</tr>
</tbody>
</table>

Reach it ... kill it
## Results with Reachability
### Relational Operators

<table>
<thead>
<tr>
<th>Programs</th>
<th>Muts</th>
<th>Stubborn &lt; 0.1%</th>
<th>Really &lt; 0.1%</th>
<th>Always 100%</th>
<th>Easy &gt; 33%</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRI-YHJ</td>
<td>40</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>TRI-V1</td>
<td>85</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>61</td>
</tr>
<tr>
<td>TRI-V2</td>
<td>55</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>38</td>
</tr>
<tr>
<td>TCAS</td>
<td>185</td>
<td>32</td>
<td>24</td>
<td>12</td>
<td>46</td>
</tr>
</tbody>
</table>

Reach it …good chance of killing it
Luckily

not all relational operators

behave the same
## Results by Relational Operator

<table>
<thead>
<tr>
<th>Operator</th>
<th>Muts</th>
<th>Equiv</th>
<th>Stubborn</th>
<th>Always</th>
<th>Easy</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=,==</td>
<td>17</td>
<td>0.00%</td>
<td>5.88%</td>
<td>23.53%</td>
<td>64.71%</td>
</tr>
<tr>
<td>&lt;,&gt;=</td>
<td>5</td>
<td>80.00%</td>
<td>0.00%</td>
<td>20.00%</td>
<td>20.00%</td>
</tr>
<tr>
<td>&lt;=,&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>==,!=</td>
<td>24</td>
<td>0.00%</td>
<td>8.33%</td>
<td>20.83%</td>
<td>62.50%</td>
</tr>
<tr>
<td>==,&gt;</td>
<td>24</td>
<td>0.00%</td>
<td>8.33%</td>
<td>20.83%</td>
<td>62.50%</td>
</tr>
<tr>
<td>&gt;, &lt;=</td>
<td>6</td>
<td>0.00%</td>
<td>0.00%</td>
<td>50.00%</td>
<td>83.33%</td>
</tr>
<tr>
<td>&gt;=,&lt;</td>
<td>3</td>
<td>0.00%</td>
<td>0.00%</td>
<td>33.33%</td>
<td>100.00%</td>
</tr>
<tr>
<td>&lt;.,&lt;=</td>
<td>5</td>
<td>80.00%</td>
<td>20.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>&lt;=,&lt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;,&gt;=</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;=,&gt;</td>
<td>3</td>
<td>0.00%</td>
<td>100.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

**NEGATION** operators are good at creating easy to kill mutants.

**OFF BY ONE** operators are good at creating hard to kill mutants.
Unfortunately so far we were looking at an ideal situation: we used a “perfect” oracle that can reliably detect mutations.

Let’s see what happens if we vary the precision of the oracle.
public boolean repOK() {
    return checkTree(root, 0, 9);
}

private boolean checkTree(Node n, int min, int max) {
    if (n == null) return true;
    if (n.value < min || n.value > max)
        return false;
    boolean resL = checkTree(n.left, min, n.value - 1);
    if (!resL) return false;
    else
        return checkTree(n.right, n.value + 1, max);
}

public String linearize() {
    if (!repOK()) return "NotABST";
    return linearize(root);
}

private String linearize(Node n) {
    StringBuilder b = new StringBuilder();
    b.append("(");
    if (n != null) {
        b.append(n.value).append(' ');
        b.append(linearize(n.left)).append(' ');
        b.append(linearize(n.right)).append(' ');
    }
    b.append(")");
    return b.toString();
}
## Linearize vs repOK for BST

<table>
<thead>
<tr>
<th>Operator</th>
<th>Muts</th>
<th>Equivalent Linearize</th>
<th>Equivalent repOK</th>
<th>Easy Linearize</th>
<th>Easy repOK</th>
<th>Always Linearize</th>
<th>Always repOK</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>67</td>
<td>30%</td>
<td>66%</td>
<td>57%</td>
<td>31%</td>
<td>21%</td>
<td>15%</td>
</tr>
<tr>
<td>AOR+Const</td>
<td>12</td>
<td>83%</td>
<td>83%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>ROR</td>
<td>55</td>
<td>18%</td>
<td>62%</td>
<td>69%</td>
<td>38%</td>
<td>25%</td>
<td>18%</td>
</tr>
<tr>
<td>Negation</td>
<td>23</td>
<td>4%</td>
<td>47%</td>
<td>78%</td>
<td>52%</td>
<td>48%</td>
<td>34%</td>
</tr>
</tbody>
</table>

Precise Oracle, less Equivalent, but more easily killed
Imprecise Oracle, more Equivalent, but less easily killed
They found for the Relational Operators you get stubborn and equivalent mutants in almost equal amounts (other classes had no such connection).

They also found that more mutations implied more equivalent mutations, but no such correlation with stubborn mutations.

Beware of Empirical Software Engineering!
WARNING!!!

Can we find an analytical link between coverage and fault detection?
If we assume we know nothing about the distribution of test inputs, then…

For a given program P, calculate the probability of achieving X% coverage with a test suite of size k

For a faulty program P, calculate the probability of observing the bug with a test suite of size k
Step 1: Probabilistic Symbolic Execution

```java
public int simple(int x, int y) {
    int a = 0;
    if (x < 4) { // 25
        a = 0;
    } else {
        a = x;
    }
    if (y < 4) { // 30
        return a + y;
    } else {
        return x + y;
    }
}
```

Collect all paths with coverage and probability (x,y:0..9):

- [30T, 25T] 0.36
- [30T, 25F] 0.24
- [30F, 25T] 0.24
- [30F, 25F] 0.16

For 100% coverage:
- 30T, 30F, 25T and 25F
Step 2: Sample and Calculate

1. Sample $k$-paths $M$ times based on the probability (with replacement)
2. For these $k$-paths calculate coverage, based on number of samples that gets the coverage, lets say $c$
3. $c/M$ gives the probability

Assume $k=2$ & 100% coverage

<table>
<thead>
<tr>
<th>Option</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>[30T, 25T]</td>
<td>0.36</td>
</tr>
<tr>
<td>[30T, 25F]</td>
<td>0.24</td>
</tr>
<tr>
<td>[30F, 25T]</td>
<td>0.24</td>
</tr>
<tr>
<td>[30F, 25F]</td>
<td>0.16</td>
</tr>
</tbody>
</table>

Pick $10^6$ 2-tests, see on how many do you cover all 4 options, if 230k times, then probability is 23%.

Probability of getting full coverage with 2-tests, is 23%
Step 3: Calculate Probability of Bug

1. Use previous stuff to calculate on what percentage of inputs can an oracle observe the bug, call this probability $p$

2. $\text{Prob}(\text{bug} \mid \text{for a given } k) = 1 - (1 - p)^k$

//spec simple(x,y) = x+y
Public int simple(int x, int y) {
    int a = 0;
    if (x < 4) { // 25
        a = 0;
    } else {
        a = x;
    }
    if (y < 4) { // 30
        return a + y;
    } else {
        return x + y;
    }
}

Prob(bug) = 12/100
PC for bug: $y! = y + x \land y < 4 \land x < 4$
then $\text{Prog}(\text{bug} \mid k = 2) = 22.6\%$

Probability of seeing the bug and obtaining coverage is therefore about the same, and thus one can argue they will correlate
Broken BinaryTree Example

Very much uncorrelated

High coverage doesn’t mean you will find the bug

3% Probability of a bug

Size of the test suite (each test 4 add/remove)
TRI-YHJ, i.e. broken TriangleClassify

Uncorrelated with high coverage

High Coverage means very good chance of finding the bug
Still working on this...

- Need more faults, the two shown were real errors not mutations
- Can create mutations and repeat all of this
- Need to see if we can find real examples from literature and analyze them
- Note that empirical work in this setting can easily be skewed to show whatever you want; only if you analyze truly large datasets with very good tests can you say something useful
- Even though this will probably only work for small programs it might give some interesting insights
Other ongoing work

• Probabilistic Java Programming
  – Including parametric analysis
  – Add sampling to scale to larger examples

Monte-Carlo Tree Search for WCET
  – Works much better than Monte-Carlo or Reinforcement Learning

• Whitebox Fuzzing revisited
  – Infer input grammars by iterative symbolic execution, i.e. derive seed-file structure on-the-fly?