Automated Test Data Generation for Coverage: Haven’t We Solved This Problem Yet?

Kiran Lakhotia\(^1\) \hspace{1cm} Phil McMinn\(^2\) \hspace{1cm} Mark Harman\(^1\)

\(^1\)CREST  
Department of Computer Science  
King’s College London

\(^2\)Department of Computer Science  
University of Sheffield

CREST, London, 2009
Haven’t We Solved This Problem Yet?

Lakhotia et al.

Outline

Motivation

Background on AUSTIN and CUTE

- Search Based Testing
- Concolic Testing
- Pointers in CUTE and AUSTIN

Empirical Study

- Experimental Setup
- Test Subjects
- Results
- Answers To Research Questions

Summary
Motivation and Research Questions

Lately two very different automated test data generation techniques have gained a lot of attention:

- Search based testing (e.g. EvoTest [link to website], 3 year project worth $4m)
- Dynamic symbolic execution (e.g. Pex, CUTE, CREST)

But,

- How effective are concolic and search based test data generation for real world programs?
- What is the relative efficiency of both approaches?
- Which types of program structure does each technique fail to cover?
We transform the testing problem into an optimization problem; the assumption is that comparing two potential solutions is easier than generating a correct solution from scratch.

<table>
<thead>
<tr>
<th>Example</th>
<th>Candidate Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>void testme(int a, int b) {</td>
<td>a = 0, b = 10 random start</td>
</tr>
<tr>
<td>if( a * b &gt; 100 )</td>
<td></td>
</tr>
<tr>
<td>//target</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

Branch Distance Function:
if( 100 - (a * b) < 0 )
return 0;
else
return (100 - (a * b)) + constant;
Search Based Structural Testing

We transform the testing problem into an optimization problem; the assumption is that comparing two potential solutions is easier than generating a correct solution from scratch.

<table>
<thead>
<tr>
<th>Example</th>
<th>Candidate Solutions</th>
</tr>
</thead>
</table>
| void testme(int a, int b) 
  { 
    if( a * b > 100 ) 
      //target 
  } | a = 0, b = 10 random start |
| Branch Distance Function: 
  if( 100 - (a * b) < 0 ) 
  return 0; 
else 
  return ( 100 - (a * b) ) + constant; | a = -10, b = 30 😞 |
Search Based Structural Testing

We transform the testing problem into an optimization problem; the assumption is that comparing two potential solutions is easier than generating a correct solution from scratch.

**Example**

```c
void testme(int a, int b)
{
    if( a * b > 100 )
        //target
}
```

**Candidate Solutions**

- a = 0, b = 10 random start
- a = -10, b = 30 😞
- a = 5, b = 9 😊

**Branch Distance Function**

```c
if( 100 - (a * b) < 0 )
    return 0;
else
    return ( 100 - (a * b) ) + constant;
```
In search based testing we are not concerned about which path leads to the target. The path taken up to a target is an emergent property of the search process.

```c
void testme(int a, int b, int c) {
    if(a == 3) {
        ...
    }
    if(b == 2) {
        if(c == 1) //target
    }
```
In search based testing we are not concerned about which path leads to the target. The path taken up to a target is an emergent property of the search process.

```c
#include <stdio.h>

int main() {
    int a = 3, b = 2, c = 1;
    if(a == 3) {
        ... 
    }
    if(b == 2) {
        if(c == 1) {
            //target
        }
    }
    return 0;
}
```
In search based testing we are not concerned about which path leads to the target. The path taken up to a target is an emergent property of the search process.

```c
void testme(int a, int b, int c) {
    if(a == 3) {
        ...
    }
    if(b == 2) {
        if(c == 1) {
            //target
        }
    }
}
```
In search based testing we are not concerned about which path leads to the target. The path taken up to a target is an emergent property of the search process.

```c
void testme(int a, int b, int c)
{
    if(a == 3)
    {
        ...
    }
    if(b == 2)
        if(c == 1)
            //target
}
```
Search Based Structural Testing

In search based testing we are not concerned about which path leads to the target. The path taken up to a target is an emergent property of the search process.

```c
void testme(int a, int b, int c)
{
    if(a == 3)
    {
        ...
    }
    ...
}

if(b == 2)
    if(c == 1)
        //target
```

**Fitness Function:**

Approach Level + normalized( Branch Distance )
**AUSTIN:** Cover all goals (i.e. edges in a control flow graph)

- Execute program with a concrete input vector
- Symbolically execute concrete path taken by said input in parallel
- When execution follows an infeasible path w.r.t. current goal
  - Apply a hill climb algorithm to solve arithmetic constraints
  - Apply custom pointer rules to solve constraint over memory locations
Concolic Testing

The term concolic stands for **concrete** execution with **symbolic** execution. It is similar to dynamic symbolic execution:

- Explore all feasible execution paths
- Execute program with a concrete input vector
- Symbolically execute concrete path taken by said input in parallel
- Use concrete values to simplify symbolic expressions when necessary (*i.e.* regard them as constant values)
Concolic Testing

Suppose you want to find input values for a and b to traverse the path to the target.

<table>
<thead>
<tr>
<th>Example</th>
<th>Concrete Values</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>void testme(int a, int b)</td>
<td>a = -10, b = 50</td>
<td>(a₀ * b₀) ≤ 100</td>
</tr>
<tr>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if( a * b &gt; 100 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>//target</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Suppose you want to find input values for `a` and `b` to traverse the path to the target.

<table>
<thead>
<tr>
<th>Example</th>
<th>Concrete Values</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void testme(int a, int b)</code></td>
<td><code>a = -10, b = 50</code></td>
<td><code>(a_0 * b_0) &gt; 100</code></td>
</tr>
</tbody>
</table>
But your constraint solver cannot handle non-linear constraints.

<table>
<thead>
<tr>
<th>Example</th>
<th>Concrete Values</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void testme(int a, int b) { </code> if( a * b &gt; 100 ) //target }</td>
<td><code>a = -10, b = 50</code></td>
<td><code>(a_0 * b_0) &gt; 100</code></td>
</tr>
</tbody>
</table>
Concolic Testing

Simplify the path condition to make it amenable to your constraint solver.

<table>
<thead>
<tr>
<th>Example</th>
<th>Concrete Values</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>void testme(int a, int b) { if( a * b &gt; 100 ) //target }</td>
<td>a = -10, b = 50</td>
<td>(constant * b₀) &gt; 100</td>
</tr>
</tbody>
</table>
Substitute runtime values for symbolic expressions when necessary.

<table>
<thead>
<tr>
<th>Example</th>
<th>Concrete Values</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>void testme(int a, int b)</td>
<td>a = -10, b = 50</td>
<td>(-10 * b₀) &gt; 100</td>
</tr>
</tbody>
</table>
You can now use your constraint solver to find an input value for \( b \). You leave the value of \( a \) unchanged, and thus execution will traverse your desired path.

<table>
<thead>
<tr>
<th>Example</th>
<th>Concrete Values</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>void testme(int a, int b)</td>
<td>( a = -10, b = -11 )</td>
<td>((-10 \times b_0) &gt; 100)</td>
</tr>
</tbody>
</table>
**CUTE:** Explore all feasible execution paths

- Execute the program with a concrete input vector (all inputs set to zero, all pointers set to `NULL`)
- Simplify symbolic expressions with runtime values
- Invert last constraint in path condition (and use backtracking) to drive successive executions down different program paths
- Use `lp_solve` to obtain concrete input values
- Apply custom pointer rules to solve constraint over memory locations
Pointer Solving Rules

- Use CIL to simplify C code into more amenable form
  - Constraints are either of pointer or arithmetic type
- Represent each memory location reachable from the function under test with a symbolic variable
- Collect all constraints over memory locations from symbolic path condition
- Form equivalence classes of symbolic variables, based on the = operators in the path condition
- Construct an undirected graph whose nodes are above equivalence classes. Edges represent inequalities, defined by the ≠ operators in the path condition
### Pointer Handling in CUTE and AUSTIN

<table>
<thead>
<tr>
<th>Example</th>
<th>Concrete Values</th>
<th>Symbolic States</th>
<th>Path Condition</th>
</tr>
</thead>
</table>
| `typedef struct item{
  struct item* next;
} void testme(item* p, item* q)
{ if(p != NULL)
  if(p->next == q)
    //target
} ` | `p=NULL, q=NULL` | `p = p_0, q = q_0` |                |
<table>
<thead>
<tr>
<th>Example</th>
<th>Concrete Values</th>
<th>Symbolic States</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>`typedef struct item{</td>
<td><code>p=NULL,q=NULL</code></td>
<td><code>p = p0, q = q0</code></td>
<td><code>p0 = NULL</code></td>
</tr>
<tr>
<td>struct item* next;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>void testme(item* p, item* q) {</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(p != NULL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(p-&gt;next == q)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>//target</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Pointer Handling in CUTE and AUSTIN

<table>
<thead>
<tr>
<th>Example</th>
<th>Concrete Values</th>
<th>Symbolic States</th>
<th>Path Condition</th>
</tr>
</thead>
</table>
| ```
typedef struct item{
    struct item* next;
}
void testme(item* p, item* q)
{
    if(p != NULL)
        p=NULL,
        q=NULL
    if(p->next == q)
        //target
}
``` | p=NULL,q=NULL | p = p₀, q = q₀ | Solve: p₀ ≠ NULL |
**Pointer Handling in CUTE and AUSTIN**

<table>
<thead>
<tr>
<th>Example</th>
<th>Concrete Values</th>
<th>Symbolic States</th>
<th>Path Condition</th>
</tr>
</thead>
</table>
| ```
typedef struct item{
    struct item* next;
}
void testme(item* p, item* q)
{
    if(p != NULL)
        if(p->next == q)
            //target
    p=0x...q=NULL
    p0, q = q0, next = next0
    p0 \neq NULL
``` | | | |
## Pointer Handling in CUTE and AUSTIN

<table>
<thead>
<tr>
<th>Example</th>
<th>Concrete Values</th>
<th>Symbolic States</th>
<th>Path Condition</th>
</tr>
</thead>
</table>
| ```
typedef struct item{
    struct item* next;
}
void testme(item* p, item* q)
{
    if(p != NULL)
        if(p->next == q)
            p=0x..., q=NULL
            p = p₀, q = q₀, next = next₀
            p₀ \neq NULL \land next₀ = q₀
```
| p=0x..., q=NULL | p = p₀, q = q₀, next = next₀ | p₀ \neq NULL \land next₀ = q₀ |
**Outline**

- Motivation
- Background
  - Search Based Testing
  - Concolic Testing
  - Pointers in CUTE and AUSTIN
- Empirical Study
  - Experimental Setup
  - Test Subjects
  - Results
  - Answers
- Summary

---

**Example**

```c
typedef struct item{
    struct item* next;
} item;

void testme(item* p, item* q)
{
    if(p != NULL)
        if(p->next == q)
            //target
            p = 0x..., q = NULL

    return;
}
```

<table>
<thead>
<tr>
<th>Example</th>
<th>Concrete Values</th>
<th>Symbolic States</th>
<th>Path Condition</th>
</tr>
</thead>
</table>
| typedef struct item{
    struct item* next;
} item;
void testme(item* p, item* q)
{
    if(p != NULL)
        if(p->next == q)
            //target
            p = 0x..., q = NULL
}
| p = p0, q = q0, next = next0 | p0 ≠ NULL ∧ next0 = q0 |
Haven't We Solved This Problem Yet?

Lakhotia et al.

Outline

Motivation

Background

Search Based Testing

Concolic Testing

Pointers in CUTE and AUSTIN

Empirical Study

Experimental Setup

Test Subjects

Results

Answers

Summary

Pointer Handling in CUTE and AUSTIN

Example

typedef struct item{
    struct item* next;
}
void testme(item* p, item* q)
{
    if(p != NULL)
    {
        if(p->next == q)
            //target
    }

Equivalence Graph
Experimental Setup

- Use default settings for each tool
- Generate test driver for each (non–trivial) function, for both tools without specifying any preconditions
- Instrument the source file containing function under test for each tool (other source files remain un–instrumented)
- Limit each tool to a maximum wall clock runtime of 2min for each function
- Repeate experiments 30 times for each tool
Test Subjects

We used a total of 169,161 pre–processed lines of C code contained within five open–source programs to compare CUTE against AUSTIN.

<table>
<thead>
<tr>
<th>Test Object</th>
<th>Lines of Code</th>
<th>Functions</th>
<th>Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
<td>Tested</td>
</tr>
<tr>
<td>libogg</td>
<td>2,552</td>
<td>68</td>
<td>33</td>
</tr>
<tr>
<td>plot2d</td>
<td>6,062</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>time</td>
<td>5,503</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>vi</td>
<td>81,572</td>
<td>474</td>
<td>405</td>
</tr>
<tr>
<td>zile</td>
<td>73,472</td>
<td>446</td>
<td>340</td>
</tr>
<tr>
<td>Total</td>
<td>169,161</td>
<td>1,035</td>
<td>823</td>
</tr>
</tbody>
</table>

**Functions Total**: total number of functions in the program  
**Functions Tested**: number of functions that were testable by the tools  
**Branches Total**: total number of branches in the program  
**Branches Tested**: number of branches contained within the tested functions
Branch Coverage of Function Under Test Only

<table>
<thead>
<tr>
<th>Library</th>
<th>Austin only</th>
<th>Cute only</th>
<th>Both</th>
<th>Neither</th>
</tr>
</thead>
<tbody>
<tr>
<td>libogg</td>
<td>2%</td>
<td>4%</td>
<td>33%</td>
<td>62%</td>
</tr>
<tr>
<td>plot2d</td>
<td>38%</td>
<td>1%</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>2%</td>
<td>36%</td>
<td>63%</td>
<td></td>
</tr>
<tr>
<td>vi</td>
<td>7%</td>
<td>1%</td>
<td>15%</td>
<td>77%</td>
</tr>
<tr>
<td>zile</td>
<td>12%</td>
<td>1%</td>
<td>13%</td>
<td>74%</td>
</tr>
</tbody>
</table>
Branch Coverage of Function Under Test And Interprocedurally

libogg
- Austin only: 2%
- Cute only: 5%
- Both: 34%
- Neither: 58%

plot2d
- Austin only: 49%
- Cute only: 1%
- Both: 28%
- Neither: 22%

time
- Austin only: 1%
- Cute only: 39%
- Both: 60%

vi
- Austin only: 8%
- Cute only: 1%
- Both: 16%
- Neither: 75%

zile
- Austin only: 13%
- Cute only: 13%
- Both: 73%
- Neither: 13%
Branch Coverage of Functions CUTE Can Handle Only

- **libogg**
  - Austin only: 2%
  - Cute only: 5%
  - Both: 34%
  - Neither: 58%

- **plot2d**
  - Austin only: 49%
  - Cute only: 1%
  - Both: 28%
  - Neither: 22%

- **time**
  - Austin only: 1%
  - Cute only: 39%
  - Both: 60%

- **vi**
  - Austin only: 7%
  - Cute only: 1%
  - Both: 17%
  - Neither: 75%

- **zile**
  - Austin only: 6%
  - Cute only: 1%
  - Both: 20%
  - Neither: 74%
### Efficiency Measured on Sample Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>CUTE</th>
<th>AUSTIN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (s)</td>
<td>Branches covered</td>
</tr>
<tr>
<td></td>
<td>FUT (Inter-procedural)</td>
<td></td>
</tr>
<tr>
<td>ogg_stream_clear</td>
<td>0.84</td>
<td>7/8 (8)/(8)</td>
</tr>
<tr>
<td>oggpack_read</td>
<td>0.24</td>
<td>2/14 (2)/(14)</td>
</tr>
<tr>
<td>CPLTOT_BYTE_MTX.Fill</td>
<td>131.25</td>
<td>4/4 (4)/(8)</td>
</tr>
<tr>
<td>CPLTOT_DrawDashedLine</td>
<td>130.43</td>
<td>13/56 (13)/(56)</td>
</tr>
<tr>
<td>CPLTOT_DrawPoint</td>
<td>0.51</td>
<td>13/16 (13)/(16)</td>
</tr>
<tr>
<td>resuse_end</td>
<td>0.42</td>
<td>4/6 (4)/(6)</td>
</tr>
<tr>
<td>_compile</td>
<td>2.11</td>
<td>26/348 (26)/(348)</td>
</tr>
<tr>
<td>exitex</td>
<td>146.47</td>
<td>1/4 (1)/(4)</td>
</tr>
<tr>
<td>main</td>
<td>0.45</td>
<td>0/152 (0)/(174)</td>
</tr>
<tr>
<td>plod</td>
<td>1.58</td>
<td>18/174 (18)/(174)</td>
</tr>
<tr>
<td>pofix</td>
<td>0.41</td>
<td>1/4 (1)/(4)</td>
</tr>
<tr>
<td>vappend</td>
<td>0.53</td>
<td>6/134 (6)/(426)</td>
</tr>
<tr>
<td>vgetline</td>
<td>0.81</td>
<td>3/220 (3)/(228)</td>
</tr>
<tr>
<td>vmain</td>
<td>0.27</td>
<td>3/404 (3)/(480)</td>
</tr>
<tr>
<td>vmove</td>
<td>0.79</td>
<td>3/30 (3)/(30)</td>
</tr>
<tr>
<td>vnnpins</td>
<td>0.22</td>
<td>2/6 (2)/(108)</td>
</tr>
<tr>
<td>vputchar</td>
<td>0.20</td>
<td>4/148 (4)/(212)</td>
</tr>
<tr>
<td>astr_rfind_cstr</td>
<td>0.45</td>
<td>2/6 (2)/(6)</td>
</tr>
<tr>
<td>check_case</td>
<td>0.38</td>
<td>1/6 (1)/(6)</td>
</tr>
<tr>
<td>expand_path</td>
<td>0.37</td>
<td>0/82 (1)/(84)</td>
</tr>
<tr>
<td>find_window</td>
<td>0.40</td>
<td>1/20 (1)/(20)</td>
</tr>
<tr>
<td>line_beginning_position</td>
<td>0.27</td>
<td>0/8 (0)/(8)</td>
</tr>
<tr>
<td>setcase_word</td>
<td>0.31</td>
<td>0/40 (0)/(74)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>419.71</strong></td>
<td><strong>114/1890</strong></td>
</tr>
</tbody>
</table>
**RQ 1:** How effective are concolic and search based test data generation for real world programs?
Not very. For all test subjects both tools achieve less than 50% C/DC coverage.

**RQ 2:** What is the relative efficiency of each individual approach?
Overall one cannot say either CUTE or AUSTIN is more efficient. When run on code which is amenable to concolic testing and CUTE’s implementation of it, CUTE is more efficient than AUSTIN.
RQ 3: *Which types of program structure did each technique fail to cover?*

Both tools performed badly for:

- functions that use pointers without checking if they have been initialized.
- functions that contain a `void` pointer, function pointer, or string input.
- programs which take another process or program as input.
- variable argument length functions.
Problems for CUTE

- Its unbounded depth first search often gets stuck in loops
- Implementation of pointer solving rules does not precisely follow their description
- Cannot handle floating type inputs
- Cannot test code containing bitfields
Problems for CUTE

- Its unbounded depth first search often gets stuck in loops
- Implementation of pointer solving rules does not precisely follow their description
- Cannot handle floating type inputs
- Cannot test code containing bitfields
- Its symbolic engine is not powerful enough for a range of software

Example

```c
void testme(int a, int b){
    int x = (int)(a / b);
    int y = b << 1;
    if( x == y)
        //target
}
```

Symbolic State

```
a = a_0, b = b_0, x = constant, y = constant
```
Problems for CUTE

- Its unbounded depth first search often gets stuck in loops
- Implementation of pointer solving rules does not precisely follow their description
- Cannot handle floating type inputs
- Cannot test code containing bitfields
- Its symbolic engine is not powerful enough for a range of software

Example

```c
void testme(int a, int b)
{
    int x = (int)(a / b);
    int y = b << 1;
    if( x == y)
        //target
}
```

Symbolic State

| a = a₀, b = b₀, x = constant, y = constant |
| Solve: constant = constant |

Haven't We Solved This Problem Yet?
Lakhotia et al.
Outline
Motivation
Background
Search Based Testing
Concolic Testing
Pointers in CUTE and AUSTIN
Empirical Study
Experimental Setup
Test Subjects
Results
Answers
Summary
Problems for AUSTIN

- Does not use any input domain reduction; very large search space can render the hill climb ineffective, leading to violation of 2min timeout
- Does not easily discover links between input variables (e.g. argc, argv)
- Does not use any data flow information, thus search gets easily stuck on plateaus (equal to flag problem)
- Pointer constraints are only solved if they appear in critical branching nodes; i.e. constraints which influence critical branching nodes remain unsolved
Problems for AUSTIN

```c
int foo(int * p) {
    if ( p != NULL)
        return 1;
    else
        return 0;
}

void testme(int* p) {
    if( ! foo(p) ) return;
    //unreachable for AUSTIN
}
```
Problems for AUSTIN

- Cannot handle strings automatically (when is a char* a null terminated string, when an array of chars, and when a pointer to a single character?)
- Cannot handle a number of stdlib functions, e.g. memset, memmove, memcpy, calloc
Problems for AUSTIN

- Cannot handle strings automatically (when is a char* a null terminated string, when an array of chars, and when a pointer to a single character?)
- Cannot handle a number of stdlib functions, e.g. memset, memmove, memcpy, calloc
- Cannot handle union structures correctly, lacks analysis of which member is used

```
union nums {
    char ch;
    int i;
    long l;
};
```
Summary of Observed Problems

- Cannot recover from segmentation faults
  - Cannot infer preconditions to functions
- Cannot handle void pointers
- Cannot handle function pointers
- Cannot handle variable argument length functions
Use search to
- solve constraints over floating point inputs
- handle black box function calls

Use dynamic symbolic execution to
- prune infeasible paths
- reduce the search space
- solve constraints whenever possible
void foo(int* a, int* b, int* c, int* d)
{
    if(a != null)
        if(a == b)
            if(c == d)
                if(a == c)
                    assert( false );
}