CREST

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

Kiran Lakhotia[1] Phil McMinn[2] Mark Harman[1]

[1]CREST
Department of Computer Science
King's College London

[2]Department of Computer Science
University of Sheffield

CREST, London, 2009

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 www.evotest.eu, 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?

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	Candidate Solutions
`void testme(int a, int b) { if( a * b > 100 ) //target }`	• a = 0, b = 10 random start

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

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.

Example	Candidate Solutions
`void testme(int a, int b) { if( a * b > 100 ) //target }`	• `a = 0, b = 10` random start • `a = -10, b = 30` ☹

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.

Example	Candidate Solutions
`void testme(int a, int b) { if( a * b > 100 ) //target }`	• a = 0, b = 10 random start • a = -10, b = 30 ☹ • a = 5, b = 9 ☺

Branch Distance Function:

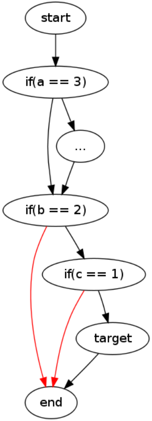
```
if( 100 - (a * b) < 0 )
  return 0;
else
  return ( 100 - (a * b) ) + constant;
```

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.

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

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

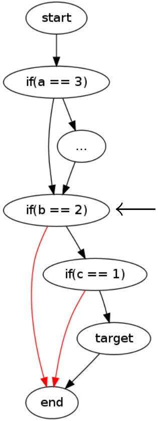


Search Based Structural Testing

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.

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

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

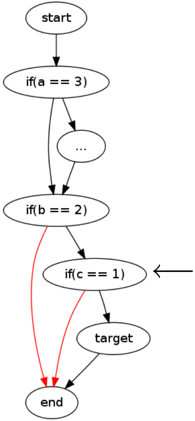


Search Based Structural Testing

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.

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

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

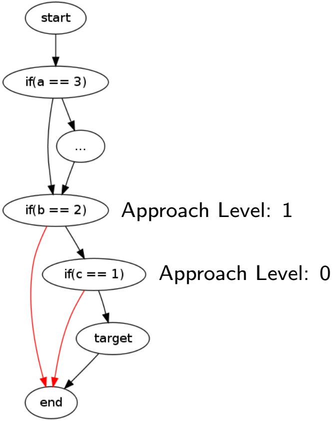


Search Based Structural Testing

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.

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

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

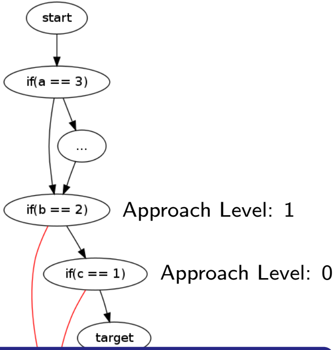


Search Based Structural Testing

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.

```
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

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 **conc**rete execution with symb**olic** 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.

Example	Concrete Values	Path Condition
`void testme(int a, int b)`	a = -10, b = 50	
`{`		
`if( a * b > 100 )`		$(a_0 * b_0) \leq 100$
`//target`		
`}`		

Concolic Testing

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

Example	Concrete Values	Path Condition
`void testme(int a, int b)`	`a = -10, b = 50`	
`{`		
`if( a * b > 100 )`		$(a_0 * b_0) > 100$
`//target`		
`}`		

Concolic Testing

But your constraint solver cannot handle non–linear constraints.

Example	Concrete Values	Path Condition
`void testme(int a, int b)`	`a = -10, b = 50`	
`{`		
`if( a * b > 100 )`		$(a_0 * b_0) > 100$
`//target`		
`}`		

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

Example	Concrete Values	Path Condition
`void testme(int a, int b)` `{` `if( a * b > 100 )` `//target` `}`	`a = -10, b = 50`	$(constant * b_0) > 100$

Concolic Testing

Substitute runtime values for symbolic expressions when necessary.

Example	Concrete Values	Path Condition
`void testme(int a, int b)`	`a = -10, b = 50`	
`{`		
`if( a * b > 100 )`		$(-10 * b_0) > 100$
`//target`		
`}`		

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.

Example	Concrete Values	Path Condition
`void testme(int a, int b)`	`a = -10, b = -11`	
`{`		
`if( a * b > 100 )`		$(-10 * b_0) > 100$
`//target`		
`}`		

CUTE

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 backtracktracking) 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 $\neq$ operators in the path condition

Pointer Handling in CUTE and AUSTIN

Example	Concrete Values	Symbolic States	Path Condition
`typedef struct item{`			
`  struct item* next;`			
`}`			
`void testme(item* p, item* q)`	p=NULL,q=NULL	$p = p_0, q = q_0$	
`{`			
`  if(p != NULL)`			
`    if(p->next == q)`			
`        //target`			
`}`			

Pointer Handling in CUTE and AUSTIN

Example	Concrete Values	Symbolic States	Path Condition
`typedef struct item{ struct item* next; } void testme(item* p, item* q) { if(p != NULL) if(p->next == q) //target }`	p=NULL,q=NULL	$p = p_0, q = q_0$	$p_0 = NULL$

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

Pointer Handling in CUTE and AUSTIN

Example	Concrete Values	Symbolic States	Path Condition
`typedef struct item{`			
`  struct item* next;`			
`}`			
`void testme(item* p, item* q)`			
`{`			
`  if(p != NULL)`	p=NULL,q=NULL	$p = p_0, q = q_0$	
`    if(p->next == q)`			
`        //target`			
`}`			

Solve: $p_0 \neq NULL$

Pointer Handling in CUTE and AUSTIN

Example	Concrete Values	Symbolic States	Path Condition
`typedef struct item{`			
`  struct item* next;`			
`}`			
`void testme(item* p, item* q)`			
`{`			
`  if(p != NULL)`	p=0x...,q=NULL	$p = p_0, q = q_0, next = next_0$	$p_0 \neq NULL$
`    if(p->next == q)`			
`      //target`			
`}`			

Pointer Handling in CUTE and AUSTIN

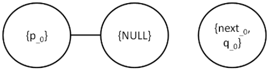
Example	Concrete Values	Symbolic States	Path Condition
`typedef struct item{ struct item* next; } void testme(item* p, item* q) { if(p != NULL) if(p->next == q) //target }`	p=0x...,q=NULL	$p = p_0, q = q_0, next = next_0$	$p_0 \neq NULL \wedge next_0 = q_0$

Pointer Handling in CUTE and AUSTIN

Example	Concrete Values	Symbolic States	Path Condition

`typedef struct item{`
`  struct item* next;`
`}`
`void testme(item* p, item* q)`
`{`
`  if(p != NULL)`
`    if(p->next == q)`
`      //target`
`}`
	p=0x...,q=NULL	$p = p_0, q = q_0, next = next_0$	$p_0 \neq NULL \wedge next_0 = q_0$

Pointer Handling in CUTE and AUSTIN

Example	Equivalence Graph
(code below)	

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

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

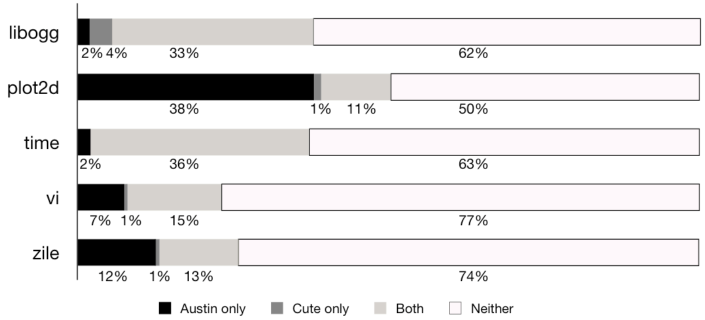
Test Subjects

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

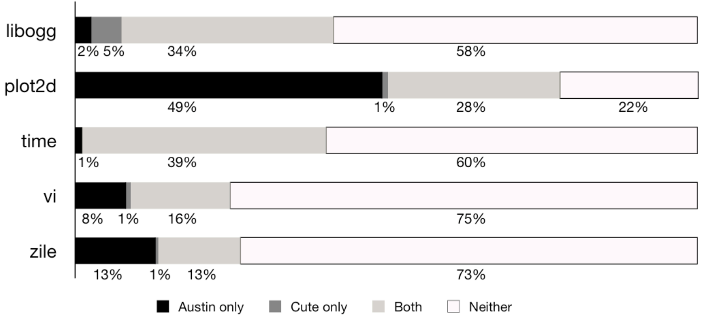
Test Object	Lines of Code	Functions		Branches	
		Total	Tested	Total	Tested
libogg	2, 552	68	33	290	284
plot2d	6, 062	35	35	1, 524	1, 522
time	5, 503	12	10	202	198
vi	81, 572	474	405	8, 950	8, 372
zile	73, 472	446	340	3, 630	3, 348
Total	169, 161	1, 035	823	14, 596	14, 084

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

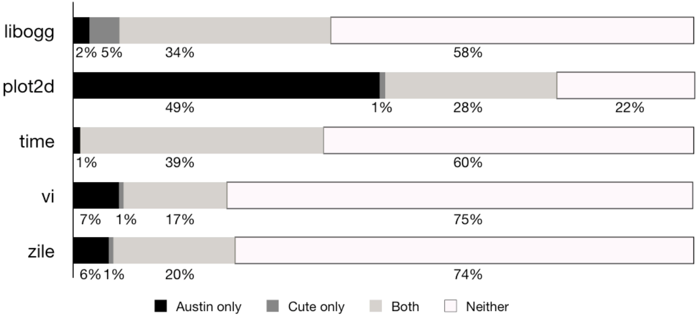
Branch Coverage of Function Under Test Only



Branch Coverage of Function Under Test And Interprocedurally



Branch Coverage of Functions CUTE Can Handle Only



Efficiency Measured on Sample Functions

Function	CUTE			AUSTIN		
	Time (s)	Branches covered FUT	Branches covered (Inter-procedural)	Time (s)	Branches covered FUT	Branches covered (Inter-procedural)
ogg_stream_clear	0.84	7/8	(8)/(8)	134.75	5/8	(7)/(8)
oggpack_read	0.24	2/14	(2)/(14)	0.18	2/14	(2)/(14)
CPLOT_BYTE_MTX_Fill	131.25	4/4	(4)/(8)	130.05	1/4	(1)/(8)
CPLOT_DrawDashedLine	130.43	13/56	(13)/(56)	130.40	37/56	(37)/(56)
CPLOT_DrawPoint	0.51	13/16	(13)/(16)	131.82	13/16	(13)/(16)
resuse_end	0.42	4/6	(4)/(6)	131.84	5/6	(5)/(6)
_compile	2.11	26/348	(26)/(348)	34.14	24/348	(24)/(348)
exitex	146.47	1/4	(1)/(4)	0.22	1/4	(1)/(4)
main	0.45	0/152	(0)/(174)	0.31	0/152	(0)/(174)
plod	1.58	18/174	(18)/(174)	137.17	18/174	(18)/(174)
pofix	0.41	1/4	(1)/(4)	135.08	1/4	(1)/(4)
vappend	0.53	6/134	(6)/(426)	0.26	6/134	(6)/(426)
vgetline	0.81	3/220	(3)/(228)	0.26	3/220	(3)/(228)
vmain	0.27	3/404	(3)/(480)	0.30	3/404	(3)/(480)
vmove	0.79	3/30	(3)/(30)	0.24	3/30	(3)/(30)
vnpins	0.22	2/6	(2)/(108)	0.25	2/6	(2)/(108)
vputchar	0.20	4/148	(4)/(212)	0.25	4/148	(4)/(212)
astr_rfind_cstr	0.45	2/6	(2)/(6)	0.17	2/6	(2)/(6)
check_case	0.38	1/6	(1)/(6)	130.49	6/6	(6)/(6)
expand_path	0.37	0/82	(1)/(84)	0.16	0/82	(1)/(84)
find_window	0.40	1/20	(1)/(20)	131.40	1/20	(1)/(20)
line_beginning_position	0.27	0/8	(0)/(8)	0.18	0/8	(0)/(8)
setcase_word	0.31	0/40	(0)/(74)	0.18	0/40	(0)/(74)
Total	419.71	114/1890	116/2494	1230.10	137/1890	140/2494

Answers To Research Questions

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.

Answers To Research Questions cont.

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

<table>
<tr><th>Example</th><th>Symbolic State</th></tr>
<tr><td><pre>
void testme(int a, int b){
  int x = (int)(a / b);
  int y = b << 1;
  if(x == y)
    //target
}
</pre></td><td>$a = a_0, b = b_0, x =$ constant, $y =$ constant</td></tr>
</table>

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	Symbolic State
`void testme(int a, int b){`	$a = a_0, b = b_0, x =$constant$, y =$constant
`  int x = (int)(a / b);`	
`  int y = b << 1;`	
`  if( x == y)`	Solve: constant = constant
`    //target`	
`}`	

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

```
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;
};
```

CREST

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 );
}
```