Symbolic execution \([x=\alpha; y=\beta]\)

What's really going on?
- Create a symbolic execution tree
- Explicitly track path conditions
- Solve path conditions — “how do you get to this point in the execution tree?” to define test inputs
- Goal: define test inputs that reach all reachable statements

CFG for (edge) coverage
- Test inputs \(\{[x=1, y=0], [x=0, y=1]\}\)
- Cover 6/7 edges, 86%
- Which edge isn't covered?
- Can it be covered?
- With what test input?
Another example (Sen and Agha)

```
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Error: possible by solving equations

```
2*beta = alpha \land alpha > beta + 10
\Rightarrow 2*beta = alpha \land beta > 10
\Rightarrow 2*beta = alpha \land beta > 10
```

Any solution to this will cause the error state to be reached

\{x=22, y=11\}, \{x=200, y=100\}, ...

OK, do this in small groups for...

```
if x ≠ 0 then
    y := 5;
else
    z := z - x;
endif;
if z > 1 then
    z := z / x;
else
    z := 0;
end
```
Way cool – we’re done!

- First example can’t reach `assert(false)`, and it’s easy to reach `end` via both possible paths
- Second example: can reach `error` and `end` via both possible paths
- Third example: can avoid edge coverage weakness
- Well, what if we can’t solve the path conditions?
  - Some arithmetic, some recursion, some loops, some pointer expressions, etc.
- We’ll see an example
- What if we want specific test cases?

Concolic testing: Sen et al.

- Basically, combine concrete and symbolic execution
- More precisely...
  - Generate a random concrete input
  - Execute the program on that input both concretely and symbolically simultaneously
  - Follow the concrete execution and maintain the path conditions along with the corresponding symbolic execution
  - If and when the symbolic constraints cannot be solved by a solver, use the path conditions collected by this guided process to constrain the generation of inputs for the next iteration
  - Repeat until test inputs are produced to exercise all feasible paths

```c
int double (int v){
    return 2*v;
}
void testme(int x, int y){
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

2nd example redux
1st iteration x=22, y=7

- `x<22&&y<7`  
- `x=22&&y=7`  
- `z=14` as solution

2nd iteration x=1, y=2
- `z=2*2=4`  
- `x>1+10`  
- Solve `2^x<8` to force the other branch
- `x=1; y=2` is one solution

```
int double (int v){
    return 2*v;
}
void testme(int x, int y){
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```
Concolic testing example: P. Sağlam

- Random seed
  - x=-3; y=7
- Concrete
  - z=9
- Symbolic
  - z=x^3+3x^2+9
  - Take then branch with constraint
    - x^3+3x^2+9\neq y
- Take else branch with constraint
  - x^3+3x^2+9=y

Three concrete test cases

- 3rd example
  - 3rd iteration x=30, y=15

Concolic testing example: P. Sağlam

- Solving is hard for x^4+3x^3+9\neq y
  - So use z’s concrete value, which is currently 9, and continue concretely
  - 9!=7 so then is good
  - Symbolically solve 9=y for else clause
    - Execute next run with x=-3; y=9
      - so else is bad

- When symbolic expression becomes unmanageable (e.g., non-linear) replace it by concrete value
Example

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}

• Random Test Driver:
  • random memory graph reachable from p
  • random value for x

• Probability of reaching abort() is extremely low

CUTE Approach

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}

Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

Following N slides from Sağlam

Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

x > 0

Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

x > 0

Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

x > 0

Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

x > 0
CUTE Approach

typedef struct cell {
    int v;
    struct cell *next;
} cell;

testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}

solve: \(x > 0\) and \(p \neq \text{NULL}\)

\(x = 0\), \(p = 0\), \(p = \text{NULL}\)

Concrete Execution

Symbolic Execution

constraints
CUTE Approach

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
    if (p != NULL)
        if (f(x) == p->v)
            if (p->next == p)
                abort();
    return 0;
}

Concrete Execution

symbolic Execution

Concrete state

symbolic state

Constraints
CUTE Approach

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}

Concrete Execution

Symbolic Execution

Concrete state

Symbolic state

Constraints

solve: x_0>0 and p_0=NULL and 2x_0+1=v_0

solve: x_0>0 and p_0=NULL and 2x_0+1=v_0

CUTE Approach

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}

Concrete Execution

Symbolic Execution

Concrete state

Symbolic state

Constraints

solve: x_0>0 and p_0=NULL and 2x_0+1=v_0

solve: x_0>0 and p_0=NULL and 2x_0+1=v_0

CUTE Approach

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}

Concrete Execution

Symbolic Execution

Concrete state

Symbolic state

Constraints

solve: x_0>0 and p_0=NULL and 2x_0+1=v_0

solve: x_0>0 and p_0=NULL and 2x_0+1=v_0
CUTE Approach

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
CUTE Approach

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
CUTE Approach

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}

Concrete Execution

Symbolic Execution

CUTE Approach

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}

Concrete Execution

Symbolic Execution

Program Error

Concrete Execution

Symbolic Execution

Cute Approach

Concrete Execution

Symbolic Execution

Cute Approach

Concrete Execution

Symbolic Execution

Concolic testing example: P. Sağlam

- Random
  - Random memory graph reachable from p
  - Random value for x
  - Probability of reaching abort is extremely low

- (Why is this a somewhat misleading motivation?)

random memory graph
reachable from p
random value for x
probability of reaching abort is extremely low

random memory graph
reachable from p
random value for x
probability of reaching abort is extremely low

## Concolic: status

- The jury is still out on concolic testing – but it surely has potential
- There are many papers on the general topic
- Here’s one that is somewhat high-level Microsoft-oriented
  - They tend to call the approach DART – Dynamic Automated Random Testing