CS409
Software Testing

TAN, Shin Hwei
陈馨慧
Southern University of Science and Technology

Slides adapted from cs4218 in NUS

Administrative Info

- Project Progress Report was due on December 4
- No bug report posted in GitHub discussion so far!
  - Use your app frequently and test it to find more bugs for the bonus!

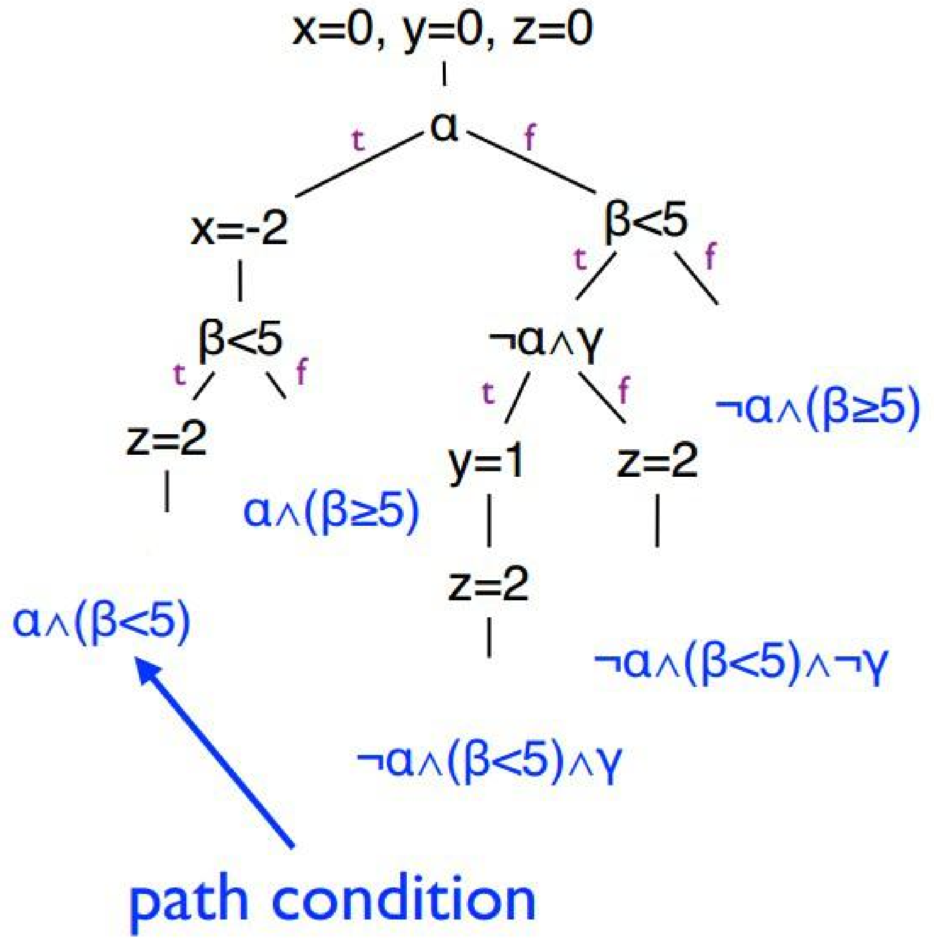
Symbolic Techniques for Debugging and Testing

Some Insights about Symbolic Execution

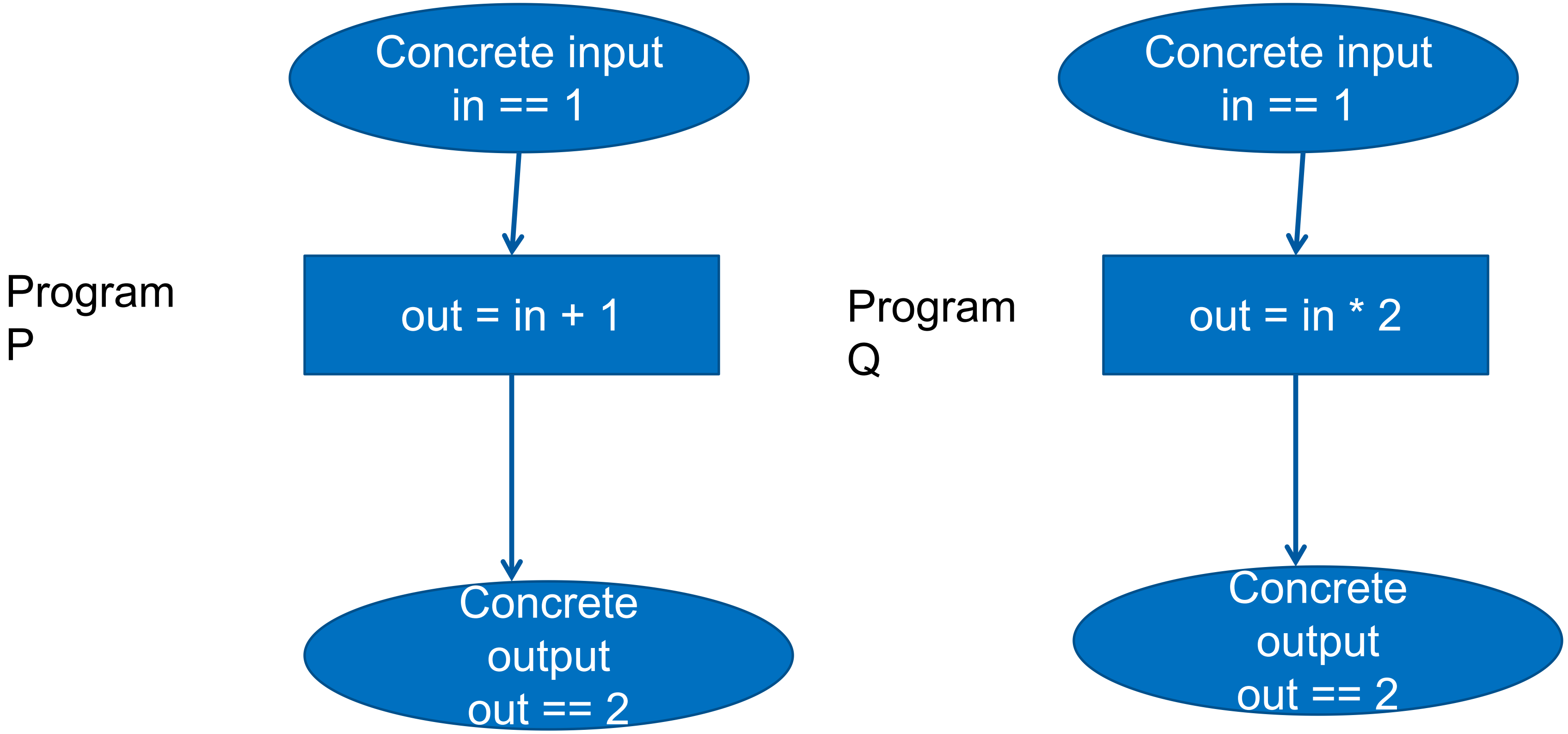
- ”Execute” programs with symbols: we track symbolic state rather than concrete input
- ”Execute” many program paths simultaneously: when execution path diverges, fork and add constraints on symbolic values
- When ”execute” one path, we actually simulate many test runs, since we are considering all the inputs that can exercise the same path

Symbolic Execution Tree

```
int a = α, b = β, c = γ;
                // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```

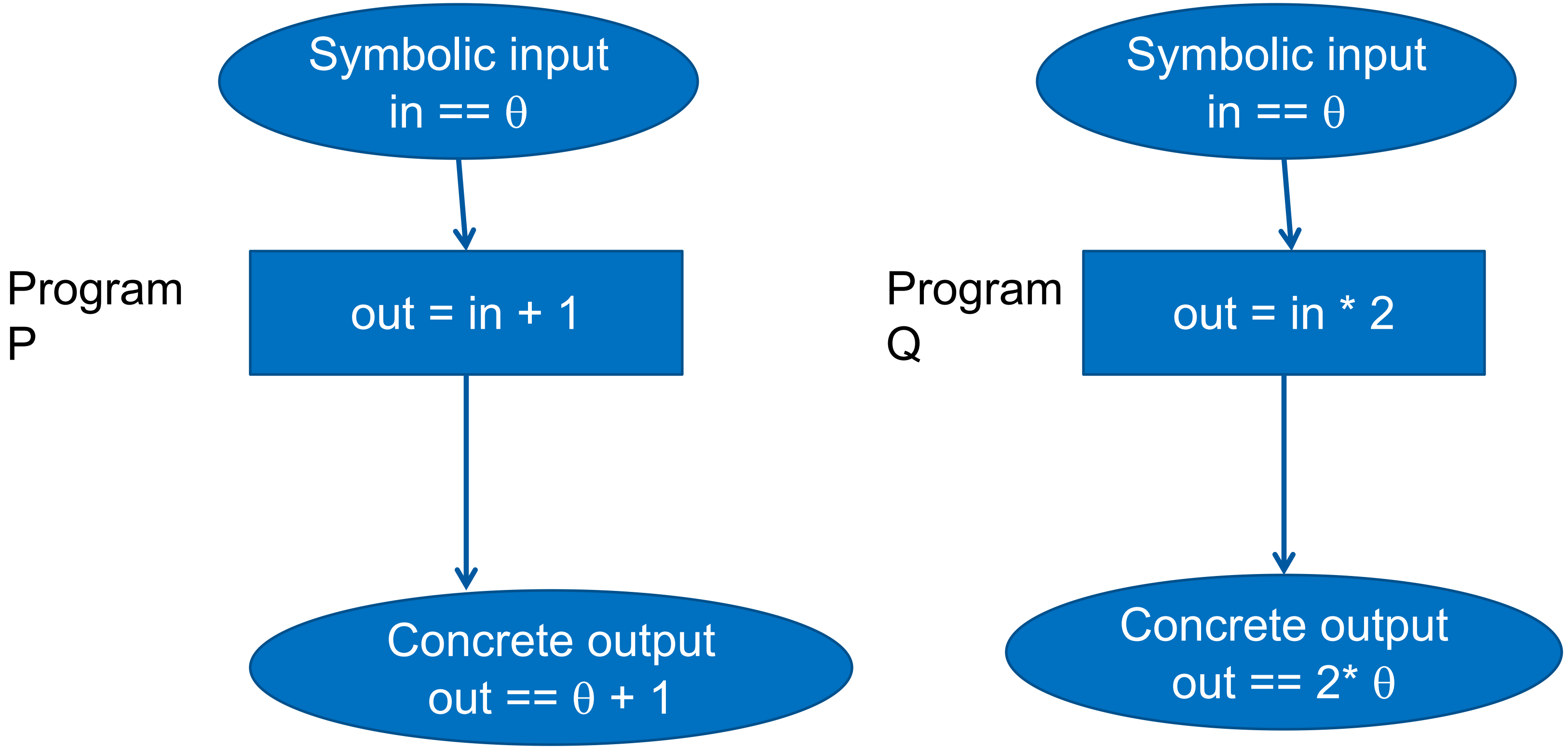


Concrete execution



No observable difference!

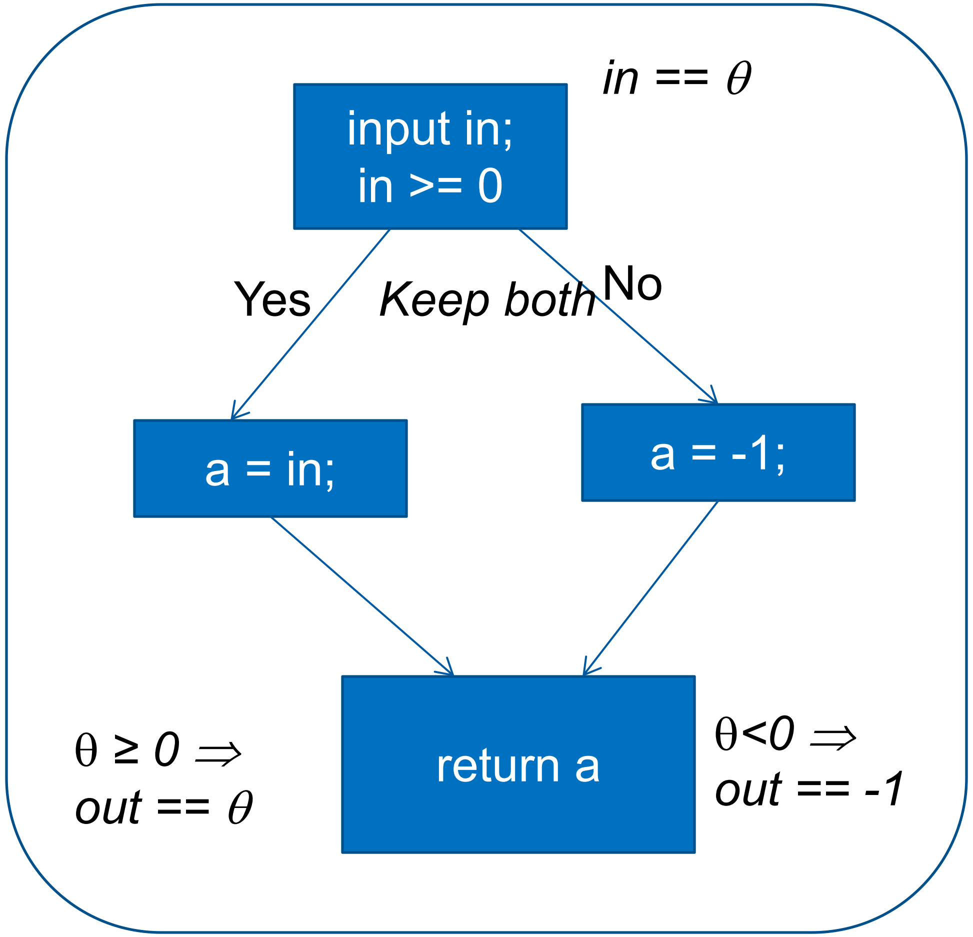
Execution with symbolic inputs



To expose difference, try to find θ such that $\theta + 1 \neq 2 * \theta$

Path exploration based symbolic execution

```
input in;

if  (in >= 0)
      a = in;
else
      a = -1;

return a;
```



On-the-fly path exploration

Instead of analyzing the whole program, shift from one program path to another.

in == 0

in == 5

```
input in;
z = 0; x = 0;
if  (in > 0){
    z = in *2;
    x = in +2;
    x = x + 2;
}
else  …
if ( z  > x){
   return error;
}
```

Sample exploration: *Continue the search for failing inputs. Try those which do not go through the "same" path.*

How to perform symbolic execution along a single path?

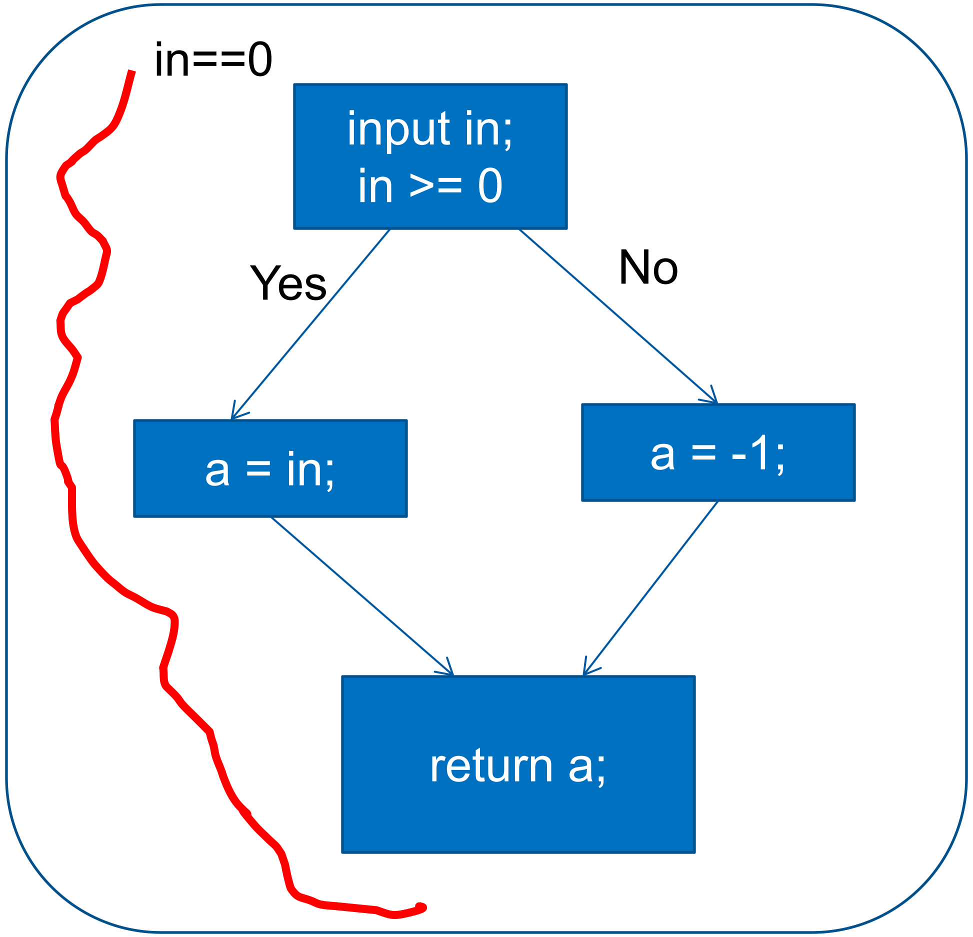
Exploring one path

Useful to find:

“the set of all inputs which trace a given path”

-> *Path condition*

in ≥ 0



Path condition computation

in == 5

```
1 input in;
2 z = 0; x = 0;
3 if  (in > 0){
4     z = in *2;
5   x = in +2;
6   x = x + 2;
7  }
8 else  …
9 if ( z  > x){
   return error;
}
```

Line#	Assignment store	Path condition
1	{}	*true*
2	{(z,0),(x,0)}	*true*
3	{(z,0),(x,0)}	*in > 0*
4	{(z,2*in), (x,0)}	*in > 0*
5	{(z,2*in), (x,in+2)}	*in > 0*
6	{(z,2*in), (x, in+4)}	*in > 0*
7	{(z, 2*in), (x, in+4)}	*in > 0*
9	{(z, 2*in), (x, in+4)}	$in>0 \wedge (2*in > in +4)$

Use of path conditions – (1)

in == 5

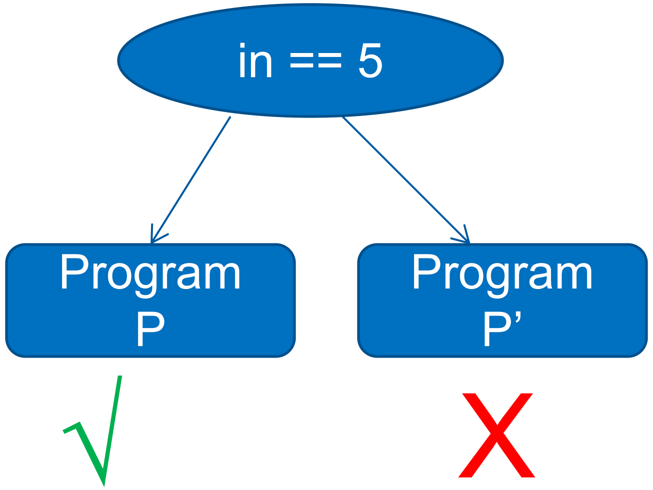
```
input in;
z = 0; x = 0;
if  (in > 0){
    z = in *2;
    x = in +2;
    x = x + 2;
}
else  …
if ( z  > x){
   return error;
}
```

X

Failing inputs: *Which other inputs follow the "same" error path?*

Path condition of in == 5

$in>0 \wedge (2*in > in +4) \equiv in > 4$



Can help in

Regression debugging

DARWIN (later)

Use of path conditions – (2)

x == 0, y == 0

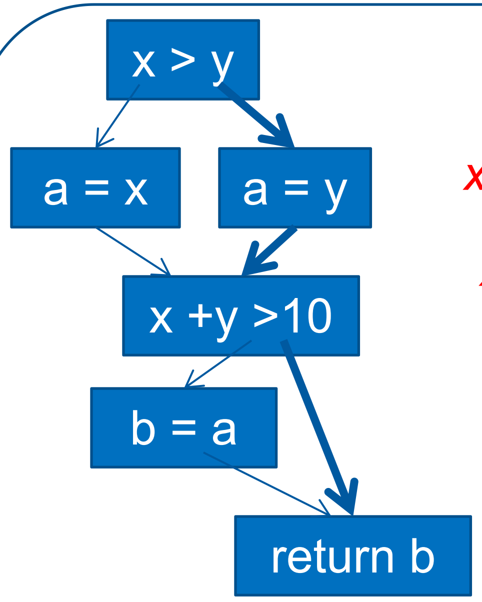
```
input x, y;
a = 0; b = 0;
if (x > y)
    a = x;
else
    a = y;
if (x + y > 10)
    b = a;
return b;
```

√

Passing inputs: *Continue the search for failing inputs, those which do not go through the "same" path.*

Path condition of (x == 0, y == 0)

$x \leq y \wedge x + y \leq 10$



Cover more paths

$x \leq y \wedge x + y \leq 10$

$x \leq y \wedge \neg x + y \leq 10$

$\neg x \leq y$

Directed D
Automated A
Random R
Testing T

Outline of today's briefing

- Symbolic execution
- Debugging
- Sample symbolic debugging techniques
    - Regression errors [FSE09, 10, …]
    - Cause clue clauses [PLDI11]
    - Error invariants [FM12]
    - Angelic debugging [ICSE11]

A quote from many years ago

"Even today, debugging remains very much of an art. Much of the computer science community has largely ignored the debugging problem….. over 50 percent of the problems resulted from the time and space chasm between symptom and root cause or inadequate debugging tools."

Hailpern & Santhanam, IBM Systems Journal, 41(1), 2002

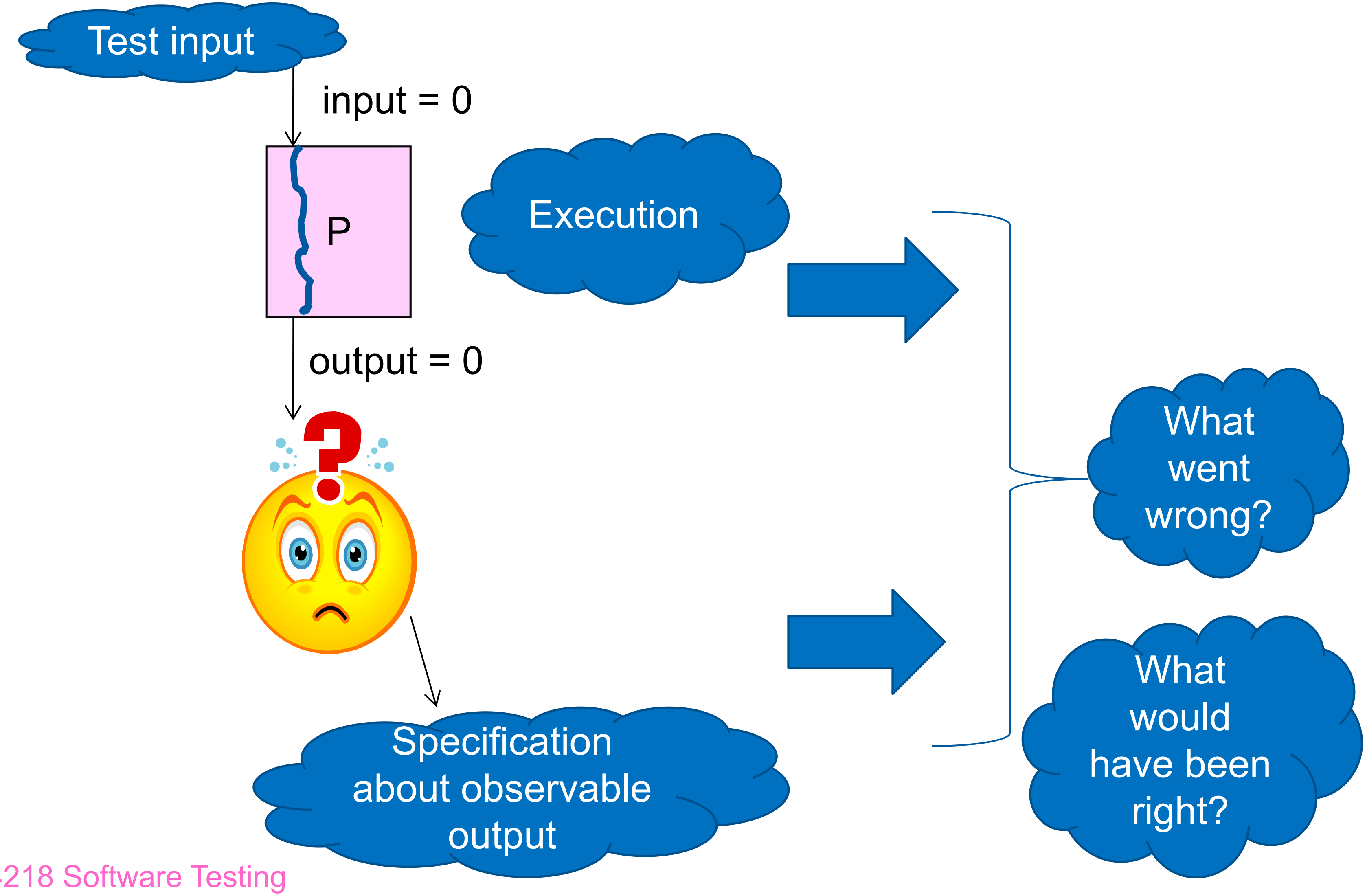
Any progress in 2002 – 2014?

How can symbolic execution help?

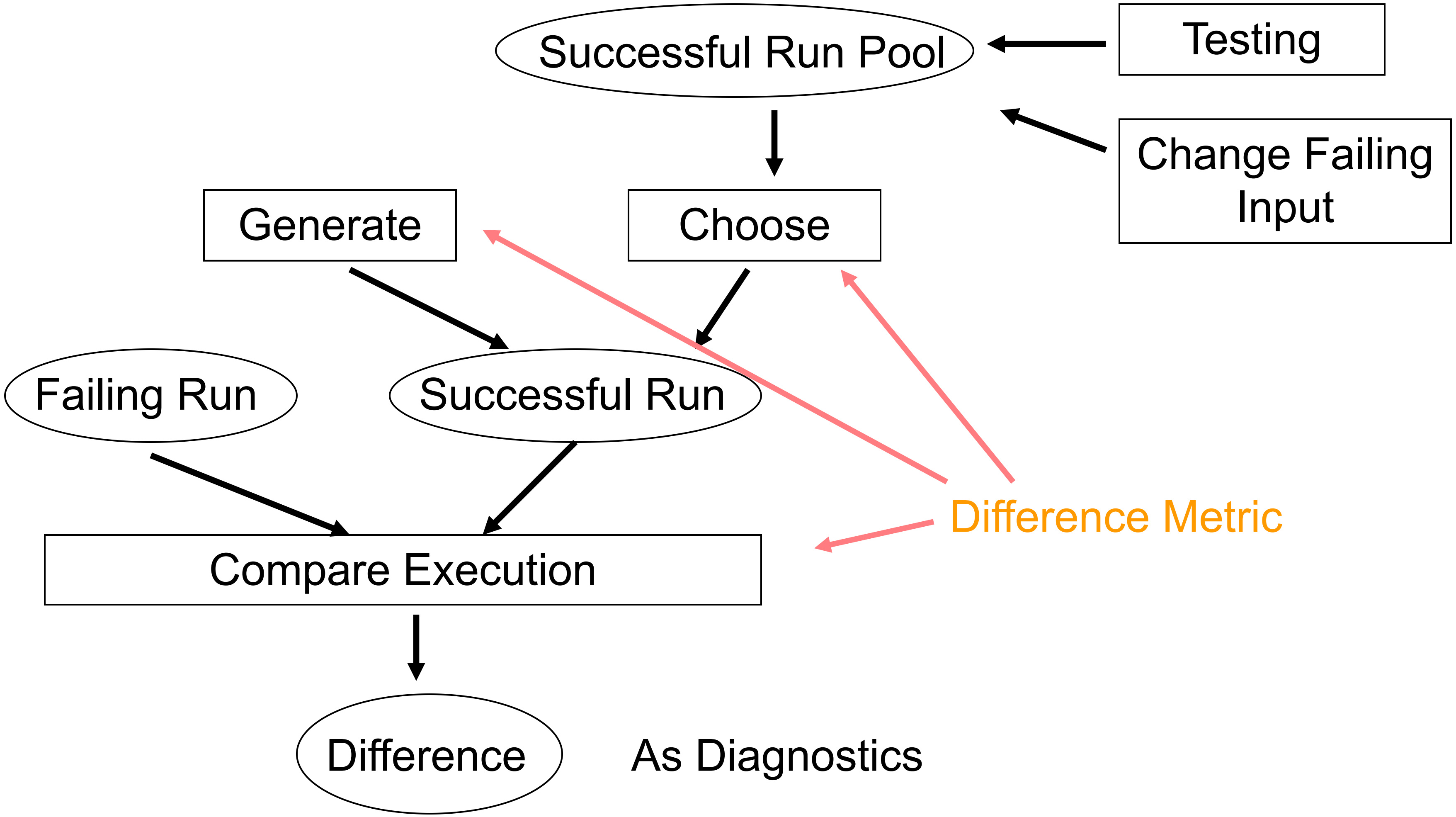
Debugging vs. Bug Hunting

- Debugging
  - Have a problematic input i, or ``counter-example” trace.
  - Does not match expected output for i.
  - Not sure what desired “property” is violated.
  - Amounts to implicitly alerting programmer about program’s intended specifications as well.
- Bug Hunting via Model Checking / SE
  - Have a desired “property”
  - Tries to find a counter-example trace, and hence an input which violates the property.

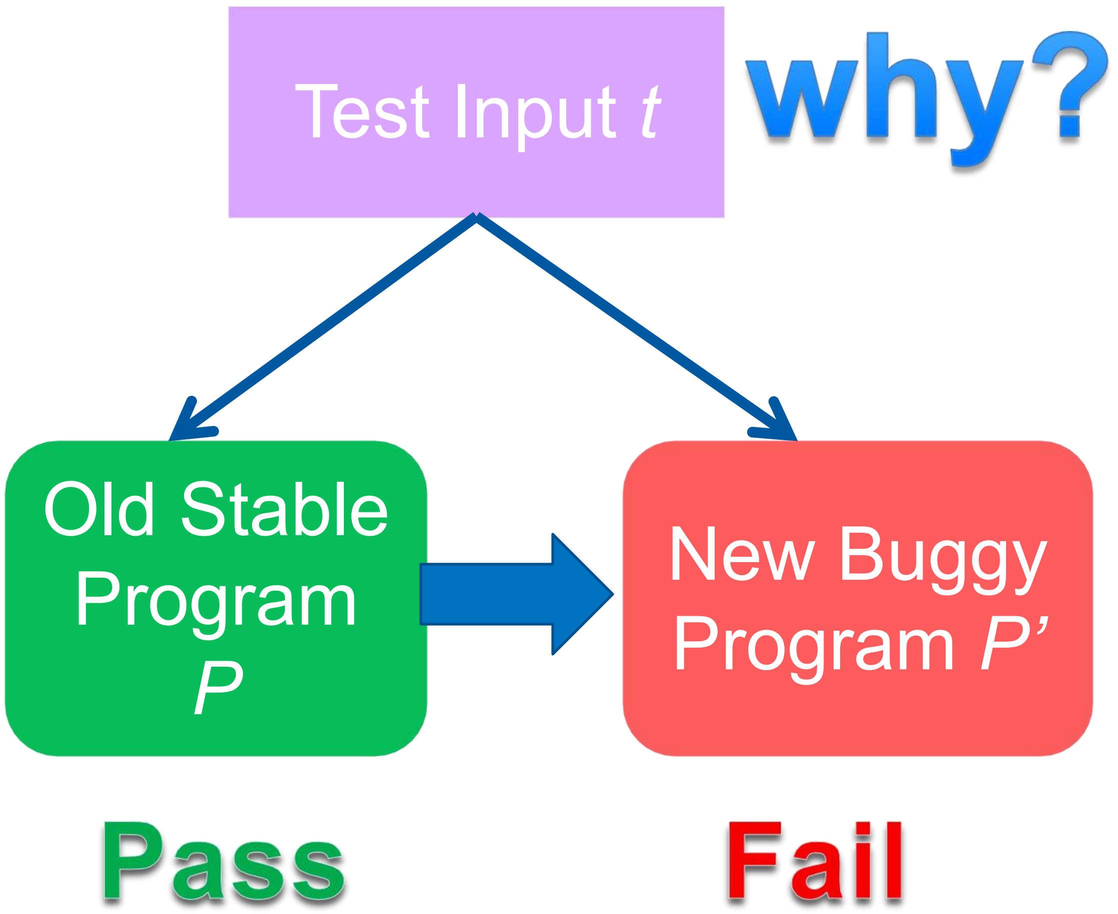
Debugging: comparing to the intended behavior



Trace Comparison based Debugging



Regression debugging

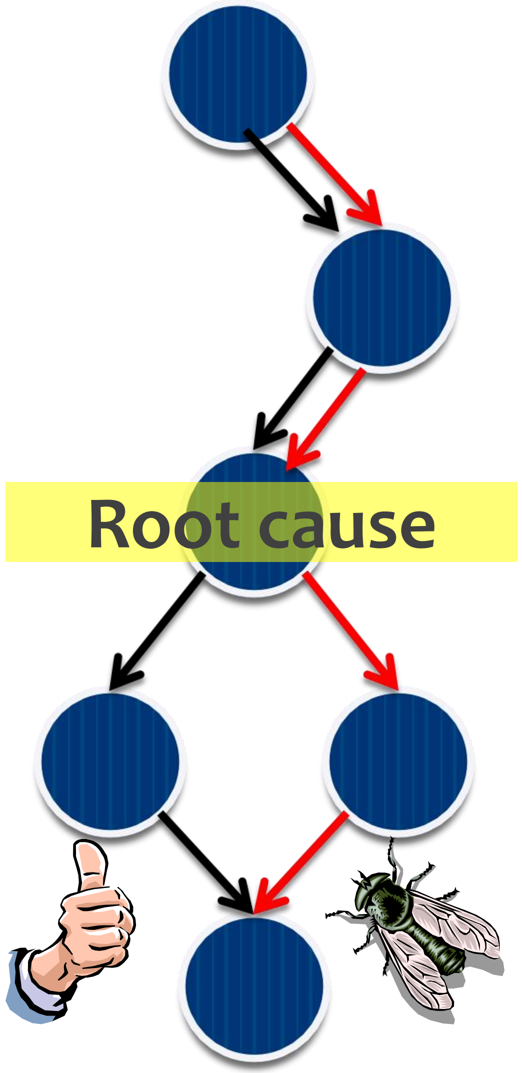


Trace Comparison?

Compare failing test with a **similar, successful** test.

Requirement: How do we find such an execution**?**

Question : why ignore the evolution?



Adapting trace comparison

Test Input *t*

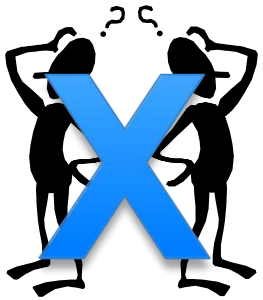
New Input *t'*

??

Old Stable Program *P*

New Buggy Program *P'*

Path σ for *t*

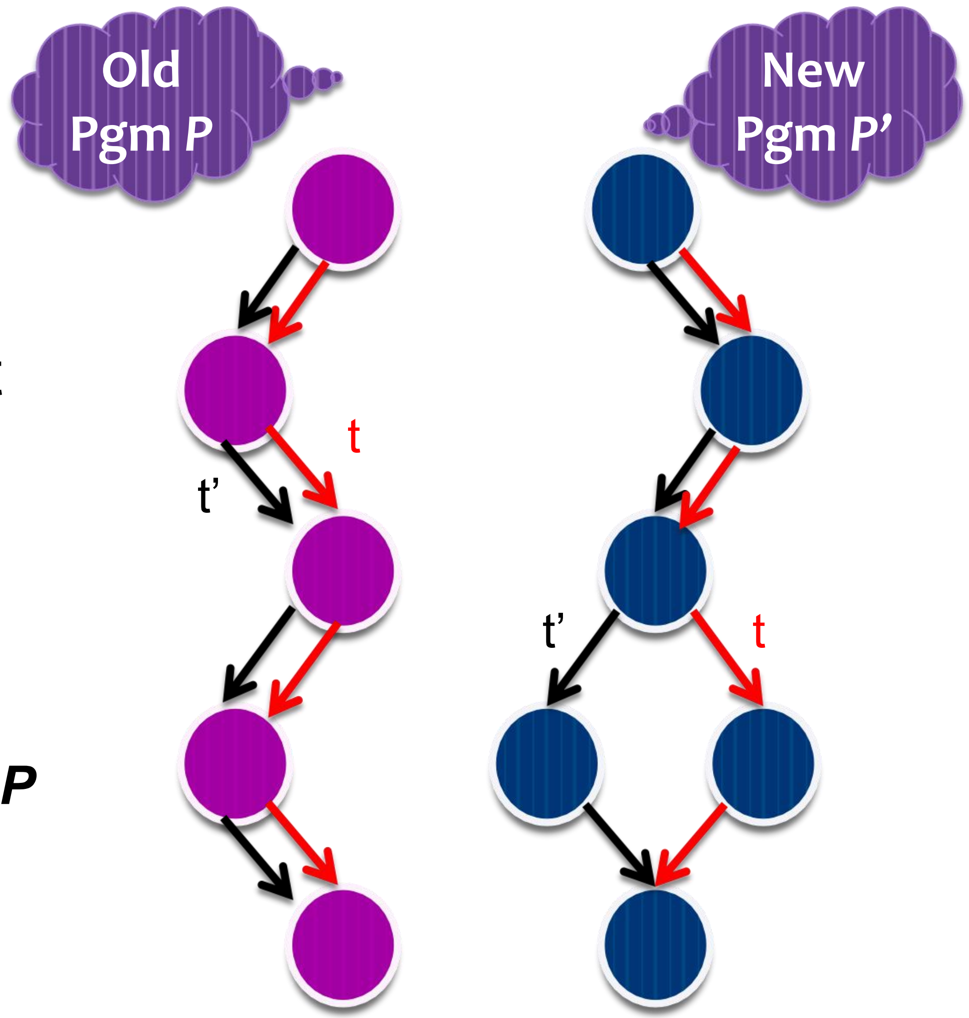


Path π for *t*

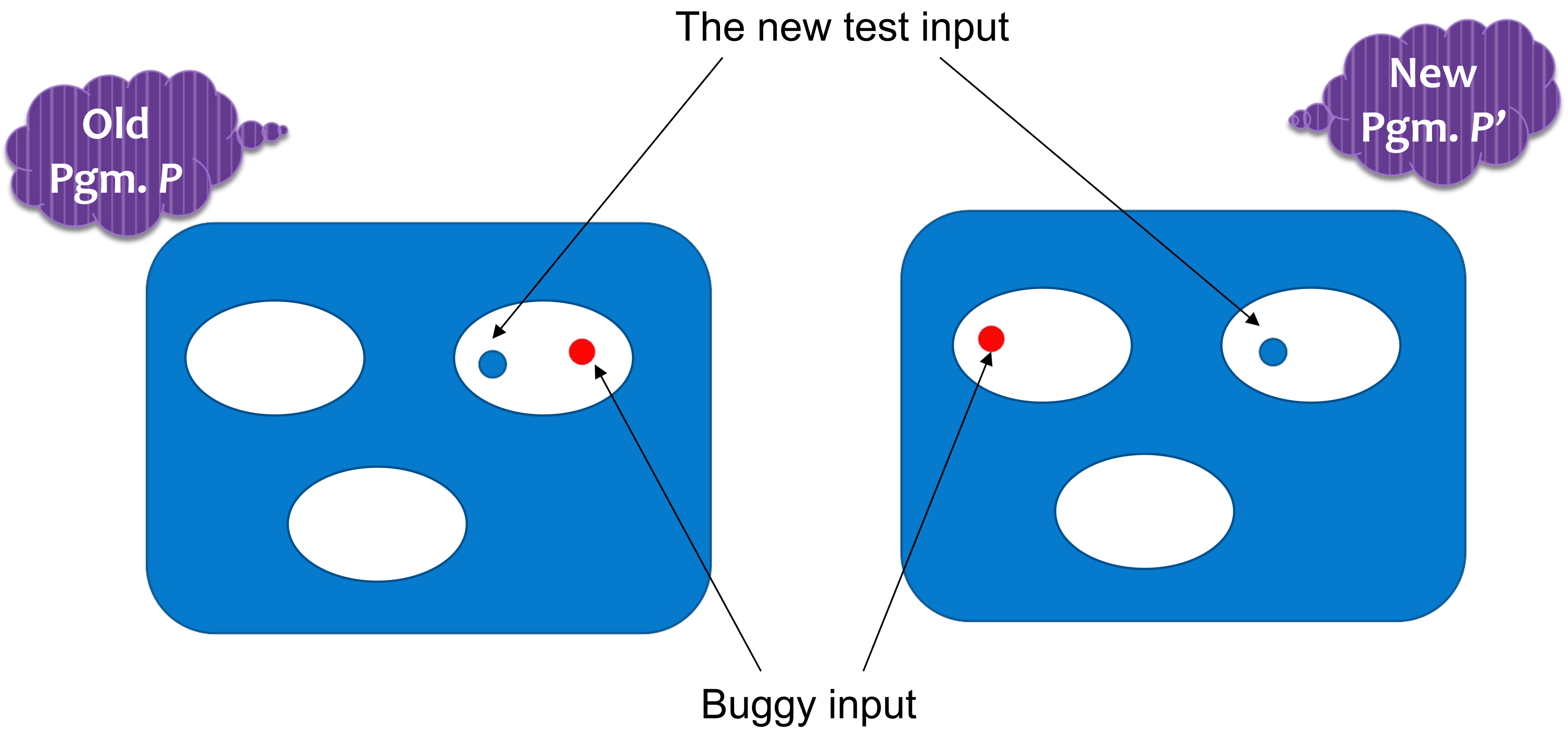
Directly Compare σ and π

How to obtain the new test?

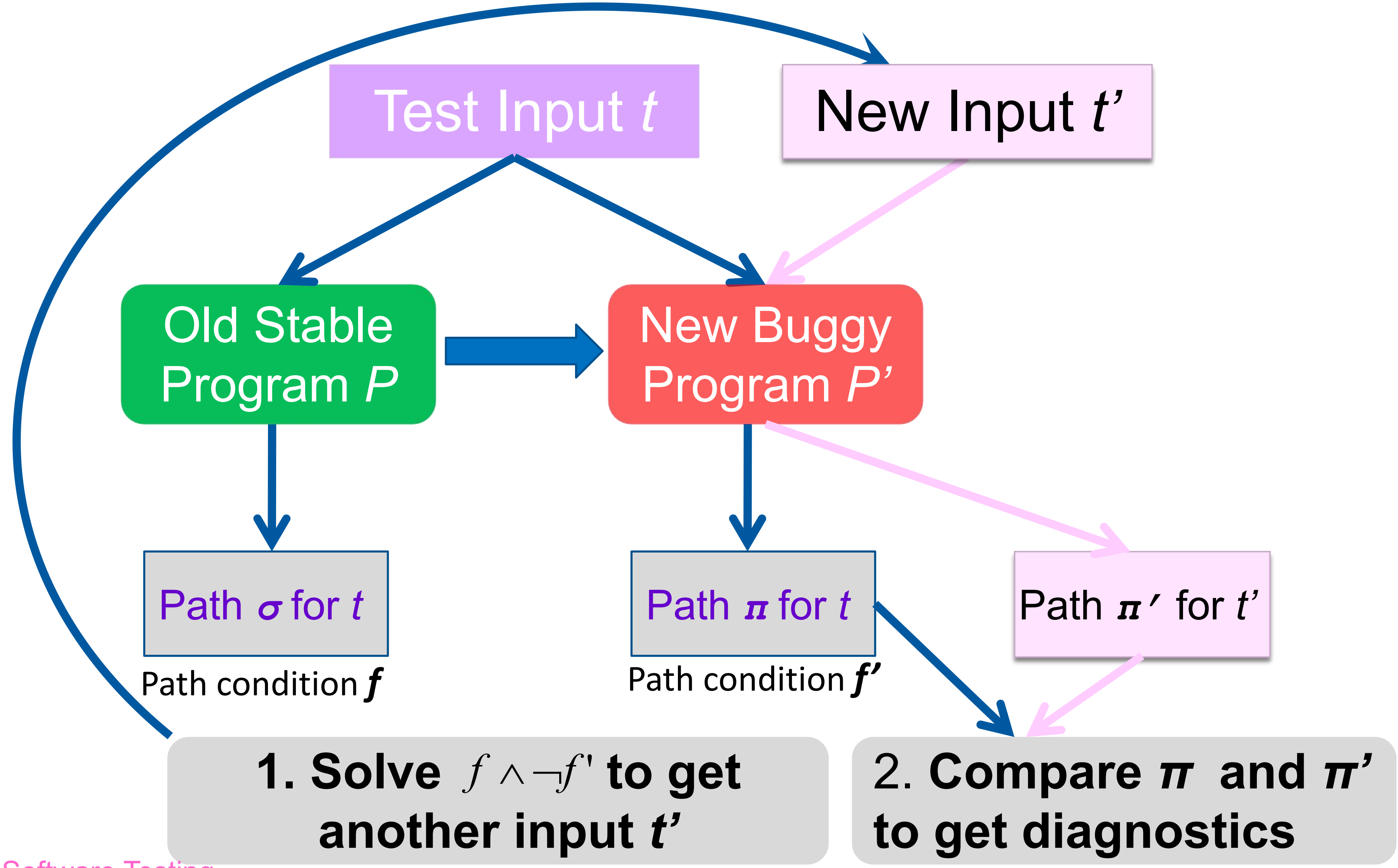
- We have:
  - Two versions of the program. (*P* and *P'*).
  - A test *t* that fails on *P'* but passes on *P*.
- Key requirement: Similarity
  - Test *t* and *t'* are **similar** if they induce
    - **same control flow path in *P*** but
    - **different paths in *P'*.**



How to obtain the new test?



DARWIN approach [FSE09]



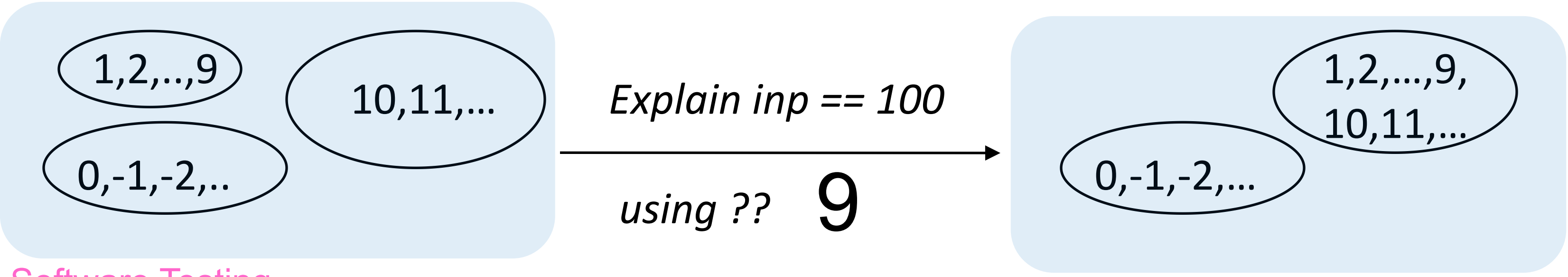
Simple Example

P

```c
int inp, outp;
scanf("%d", &inp);
if (inp >=1){
   outp = g(inp);
   if (inp>9){
      outp=g1(inp);
   }
} else{
  outp = h(inp);
}
printf("%d", outp);
```

P'

```c
int inp, outp;
scanf("%d", &inp);
if (inp >= 1){
   outp = g(inp);
   /* if (inp>9){
      outp=g1(inp);
   }  */
} else{
  outp = h(inp);
}
printf("%d", outp);
```



Simple Example: Exercise

P

```
int inp, outp;
scanf("%d", &inp);
if (inp >=1){
   outp = g(inp);
   if (inp>9){
      outp=g1(inp);
   }
} else{
  outp = h(inp);
}
printf("%d", outp);
```

P'

```
int inp, outp;
scanf("%d", &inp);
if (inp >= 1){
   outp = g(inp);
   /* if (inp>9){
      outp=g1(inp);
   }  */
} else{
  outp = h(inp);
}
printf("%d", outp);
```

- Write the Path Condition for P
- Write the Path Condition for P'

inp==100

```c
int inp, outp;
scanf("%d", &inp);
if (inp >=1){
    outp = g(inp);
    if (inp>9){
        outp=g1(inp);
    }
} else{
    outp = h(inp);
}
printf("%d", outp);
```

P

```c
int inp, outp;
scanf("%d", &inp);
if (inp >= 1){
    outp = g(inp);
    /* if (inp>9){
        outp=g1(inp);
    }   */
} else{
    outp = h(inp);
}
printf("%d", outp);
```

P'

Path condition f
`(inp >= 1)&& (inp>9)`

Path condition f'
`(inp >= 1)`

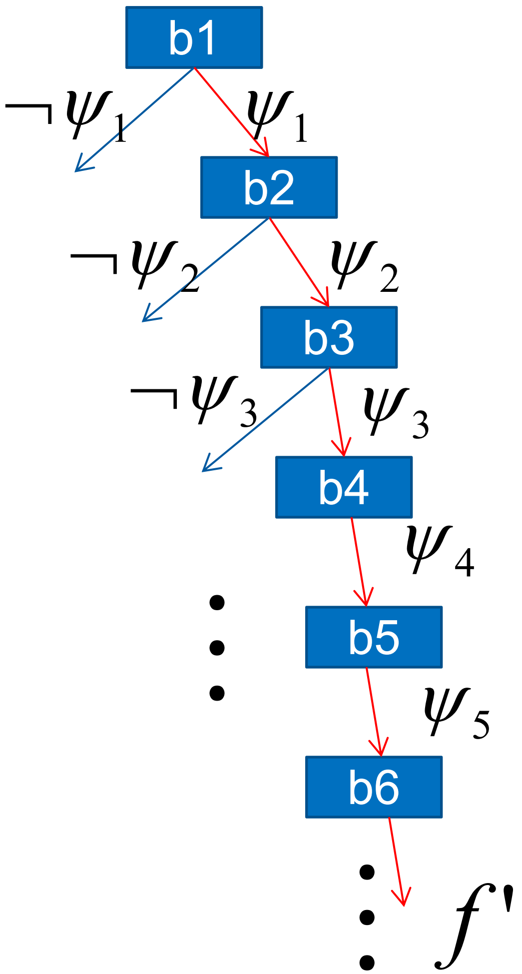
$$f \wedge \neg f' = (inp > 9)\,\&\&(inp < 1)$$

STP Solver → **No soln.**

$$f' \wedge \neg f = (inp >= 1)\,\&\&(inp <= 9)$$

STP Solver → **inp==9**

Generating new input



$$\text{Solve } f \wedge \neg f'$$

$$f' = (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m)$$

Check for satisfiability of

$$f \wedge \neg\psi_1$$

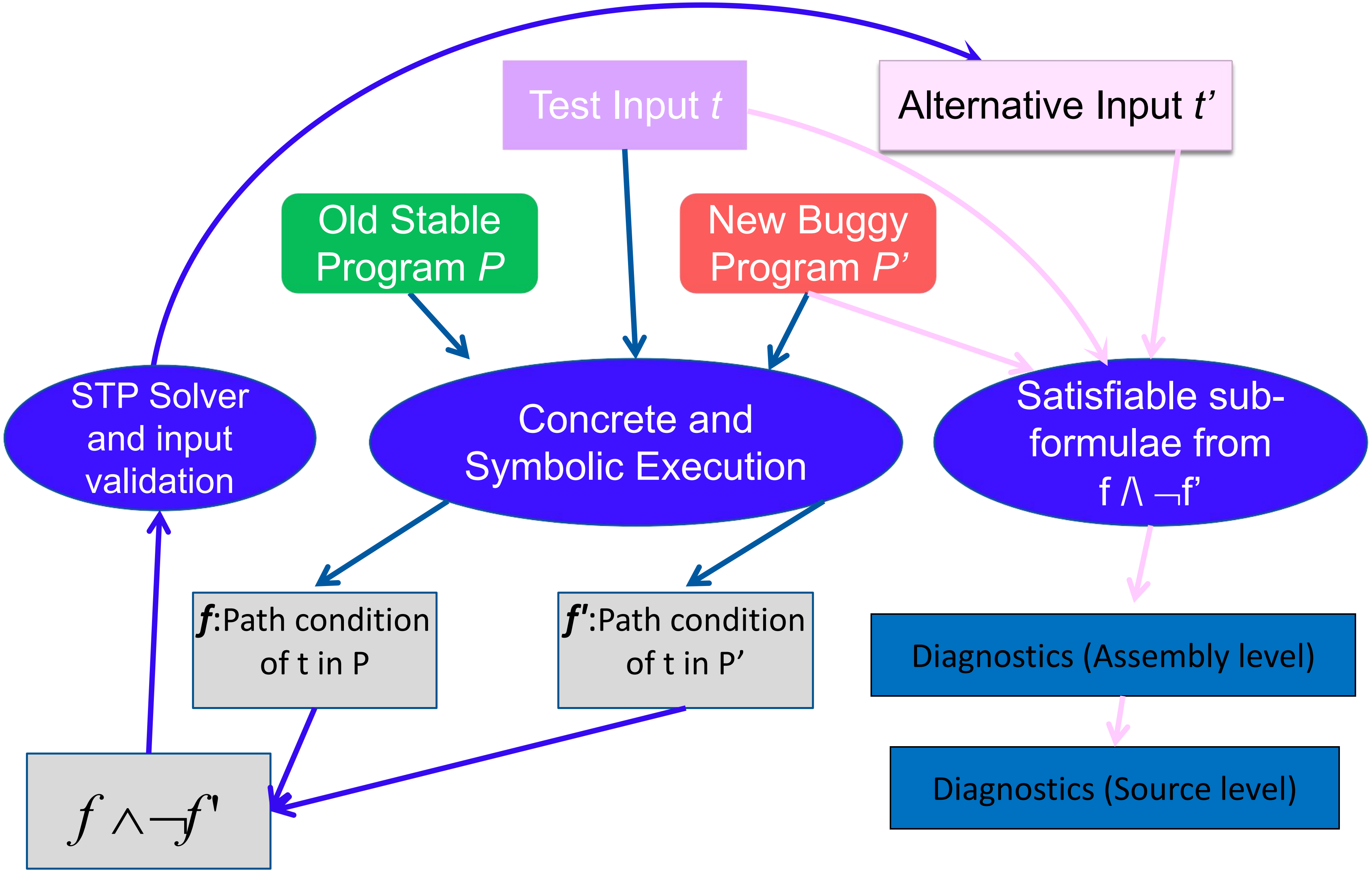
$$f \wedge \psi_1 \wedge \neg\psi_2$$

$$f \wedge \psi_1 \wedge \psi_2 \wedge \neg\psi_3$$

• • • • • •

At most m alternate inputs !!
Remove trace comparison altogether

DARWIN approach [FSE 09]



Specification discovery

- Given a reference implementation
  - Symbolic execution extracts specification of intended behaviour for a failed test t.
  - Can generate
    - Another input / class of inputs whose behaviour captures the intended behaviour of t [FSE09, FSE11]
    - Or, a direct semantic differencing of the behavior of t in two program version [FSE10]
  - Scalability + Precise diagnostics (~5min, ~5 LOC)
    - Program versions e.g. libPNG
    - Embedded SW – Busybox vs. Coreutils
    - Protocol impl. – e.g. Webservers – miniweb vs. Apache.

Retrospective

Debugging – some milestones

- Manual era: prints and breakpoints
- Statistical fault localization [e.g. Tarantula]
- Dynamic slicing [e.g. JSlice]
- Trace comparison and delta debugging
  - Look for workarounds – *how to avoid the error*?
- Symbolic techniques
  - Replace repeated experimentation with constraint solving.
  - Discover and (partially) infer intended semantics by symbolic analysis
- Future: repair (hints)

Constraint Solving - SAT

SAT: find an assignment to a set of Boolean variables that makes the Boolean formula true

Complexity: NP-Complete

Constraint Solving - SMT [2]

SMT (Satisfiability Modulo Theories) = SAT++

$$\sin(x)^3 = \cos(\log(y) \cdot x) \vee b \vee -x^2 \geq 2.3y$$

- An SMT formula is a Boolean combination of formulas over first-order theories
- Example of SMT theories include bit-vectors, arrays, integer and real arithmetic, strings, ...
- The satisfiability problem for these theories is typically hard in general (NP-complete, PSPACE-complete, ...)
- Program semantics are easily expressed over these theories
- Many software engineering problems can be easily reduced to the SAT problem over first-order theories

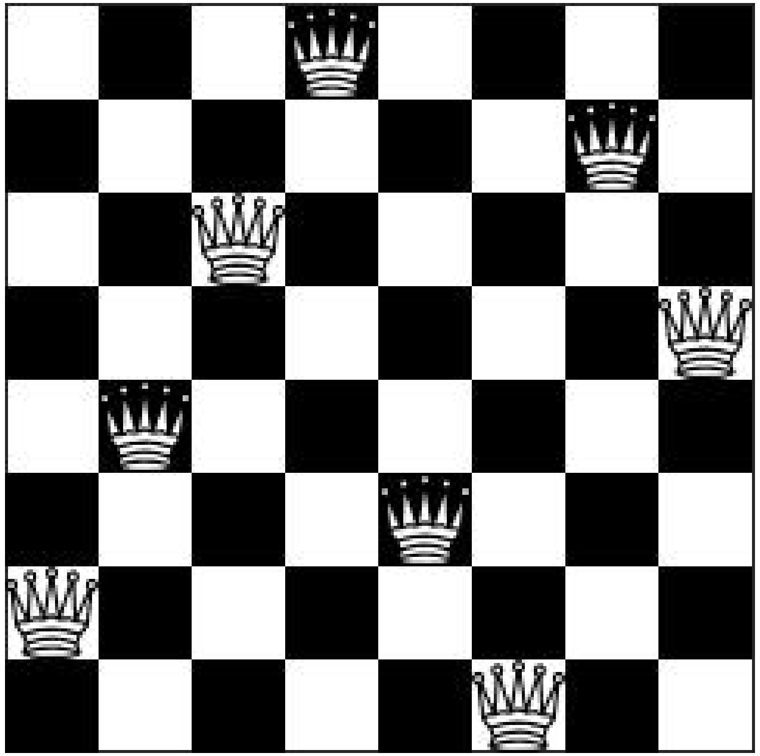
Constraint Solving - SMT

- The State of the Art: Handle linear integer constraints
- Challenges:
- Constraints that contain non-linear operands, e.g., sin(), cos()
- Float-point constraints: no theory support yet, convert to bit-vector computation
- String constraints: a = b.replace('x', 'y')
- Quantifies: ∃, ∀
- Disjunction

SAT, SMT and CSP solvers are used for solving problems involving constraints.

The term “constraint solver”, however, usually refers to a CSP solver.

The 8-queens problem



The Boolean satisfiability problem (SAT) is the problem of deciding whether there is a variable assignment that satisfies a given propositional formula.

SAT example

$$x_1 \lor x_2 \lor \neg x_4$$
$$\neg x_2 \lor \neg x_3$$

- x_i : a Boolean variable
- x_i, $\neg x_i$: a literal
- $\neg x_2 \lor \neg x_3$: a clause

The 8-queens problem

x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}
x_{21}	x_{22}	x_{23}	x_{24}	x_{25}	x_{26}	x_{27}	x_{28}
x_{31}	x_{32}	x_{33}	x_{34}	x_{35}	x_{36}	x_{37}	x_{38}
x_{41}	x_{42}	x_{43}	x_{44}	x_{45}	x_{46}	x_{47}	x_{48}
x_{51}	x_{52}	x_{53}	x_{54}	x_{55}	x_{56}	x_{57}	x_{58}
x_{61}	x_{62}	x_{63}	x_{64}	x_{65}	x_{66}	x_{67}	x_{68}
x_{71}	x_{72}	x_{73}	x_{74}	x_{75}	x_{76}	x_{77}	x_{78}
x_{81}	x_{82}	x_{83}	x_{84}	x_{85}	x_{86}	x_{87}	x_{88}

The 8-queens problem : SAT model

```
p cnf 64 744
1 2 3 4 5 6 7 8 0
-1 -2 0
-1 -3 0
-1 -4 0
-1 -5 0
-1 -6 0
-1 -7 0
-1 -8 0
-2 -3 0
-2 -4 0
-2 -5 0
-2 -6 0
-2 -7 0
-2 -8 0
-3 -4 0
-3 -5 0
-3 -6 ..
```

- Input expected in CNF
- Using DIMACS format
  - One clause per line delimited by 0
  - Variables encoded by integers, not variable encoded by negating integer

The Satisfiability Modulo Theories (SMT) is the problem of deciding whether there is a variable assignment that satisfies a given formula in first order logic with respect to a background theory.

Example background theories for SMT

- Equality with Uninterpreted Functions (e.g. $f(x) = y \wedge f(x) \neq y$ is UNSAT)
- Non-linear arithmetic (e.g. $x^2 + yz \leq 10$)
  :variables can be reals
- Arrays (e.g. $write(a, x, 3) = b$, $read(a, x) = b$)
- Bit vectors (e.g. $x[0:1] \neq y[0:1]$)

The 8-queens problem : SMT model

```
(set-logic QF_IDL)
(set-info :source |
Queens benchmarks generated by Hyondeuk Kim in SMT-LIB format.
|)
(set-info :smt-lib-version 2.0)
(set-info :category "crafted")
(set-info :status sat)
(declare-fun x0 () Int)
(declare-fun x1 () Int)
(declare-fun x2 () Int)
(declare-fun x3 () Int)
(declare-fun x4 () Int)
(declare-fun x5 () Int)
(declare-fun x6 () Int)
(declare-fun x7 () Int)
(declare-fun x8 () Int)
(assert (let ((?v_0 (- x0 x8)) (?v_1 (- x1 x8)) (?v_2 (- x2 x8)) (?v_3 (- x3 x8)) (?v_4 (- x4 x8)) (?
v_5 (- x5 x8)) (?v_6 (- x6 x8)) (?v_7 (- x7 x8)) (?v_8 (- x0 x1)) (?v_9 (- x0 x2)) (?v_10 (- x0 x3)) (?
v_11 (- x0 x4)) (?v_12 (- x0 x5)) (?v_13 (- x0 x6)) (?v_14 (- x0 x7)) (?v_15 (- x1 x2)) (?v_16 (- x1
x3)) (?v_17 (- x1 x4)) (?v_18 (- x1 x5)) (?v_19 (- x1 x6)) (?v_20 (- x1 x7)) (?v_21 (- x2 x3)) (?v_22
(- x2 x4)) (?v_23 (- x2 x5)) (?v_24 (- x2 x6)) (?v_25 (- x2 x7)) (?v_26 (- x3 x4)) (?v_27 (- x3 x5)) (?
v_28 (- x3 x6)) (?v_29 (- x3 x7)) (?v_30 (- x4 x5)) (?v_31 (- x4 x6)) (?v_32 (- x4 x7)) (?v_33 (- x5
x6)) (?v_34 (- x5 x7)) (?v_35 (- x6 x7))) (and (<= ?v_0 7) (>= ?v_0 0) (<= ?v_1 7) (>= ?v_1 0) (<= ?
v_2 7) (>= ?v_2 0) (<= ?v_3 7) (>= ?v_3 0) (<= ?v_4 7) (>= ?v_4 0) (<= ?v_5 7) (>= ?v_5 0) (<= ?v_6 7)
(>= ?v_6 0) (<= ?v_7 7) (>= ?v_7 0) (not (= x0 x1)) (not (= x0 x2)) (not (= x0 x3)) (not (= x0 x4))
(not (= x0 x5)) (not (= x0 x6)) (not (= x0 x7)) (not (= x1 x2)) (not (= x1 x3)) (not (= x1 x4)) (not
(= x1 x5)) (not (= x1 x6)) (not (= x1 x7)) (not (= x2 x3)) (not (= x2 x4)) (not (= x2 x5)) (not (= x2
x6)) (not (= x2 x7)) (not (= x3 x4)) (not (= x3 x5)) (not (= x3 x6)) (not (= x3 x7)) (not (= x4 x5))
(not (= x4 x6)) (not (= x4 x7)) (not (= x5 x6)) (not (= x5 x7)) (not (= x6 x7)) (not (= ?v_8 1)) (not
(= ?v_8 (- 1))) (not (= ?v_9 2)) (not (= ?v_9 (- 2))) (not (= ?v_10 3)) (not (= ?v_10 (- 3))) (not (= ?
v_11 4)) (not (= ?v_11 (- 4))) (not (= ?v_12 5)) (not (= ?v_12 (- 5))) (not (= ?v_13 6)) (not (= ?v_13
(- 6))) (not (= ?v_14 7)) (not (= ?v_14 (- 7))) (not (= ?v_15 1)) (not (= ?v_15 (- 1))) (not (= ?v_16
2)) (not (= ?v_16 (- 2))) (not (= ?v_17 3)) (not (= ?v_17 (- 3))) (not (= ?v_18 4)) (not (= ?v_18 (-
4))) (not (= ?v_19 5)) (not (= ?v_19 (- 5))) (not (= ?v_20 6)) (not (= ?v_20 (- 6))) (not (= ?v_21 1))
(not (= ?v_21 (- 1))) (not (= ?v_22 2)) (not (= ?v_22 (- 2))) (not (= ?v_23 3)) (not (= ?v_23 (- 3)))
(not (= ?v_24 4)) (not (= ?v_24 (- 4))) (not (= ?v_25 5)) (not (= ?v_25 (- 5))) (not (= ?v_26 1)) (not
(= ?v_26 (- 1))) (not (= ?v_27 2)) (not (= ?v_27 (- 2))) (not (= ?v_28 3)) (not (= ?v_28 (- 3))) (not
(= ?v_29 4)) (not (= ?v_29 (- 4))) (not (= ?v_30 1)) (not (= ?v_30 (- 1))) (not (= ?v_31 2)) (not (= ?
v_31 (- 2))) (not (= ?v_32 3)) (not (= ?v_32 (- 3))) (not (= ?v_33 1)) (not (= ?v_33 (- 1))) (not (= ?
v_34 2)) (not (= ?v_34 (- 2))) (not (= ?v_35 1)) (not (= ?v_35 (- 1)))))))
(check-sat)
(exit)
```

The Constraint Satisfaction Problem (CSP) is the problem of deciding whether there is a variable assignment that satisfies a given set of constraints.

The 8-queens problem : CSP model

ESSENCE' 1.0

given n : 8

letting queens_n be domain int(0..n – 1)

find queens : matrix indexed by [queens_n] of queens_n

such that

alldifferent(queens), forall i, j : queens_n .

(i > j) => ((queens[i] - i != queens[j] - j)

$\wedge$ (queens[i] + i != queens[j] + j))

SAT, SMT or CSP?

- SAT:

  + extremely efficient

  - problem with expressivity
- SMT:

  + better expressivity, incorporates domain-specific reasoning

  - some loss of efficiency
- CSP:

  + very expressive, uses domain-specific reasoning

  - some loss of efficiency

SAT, SMT or CSP?

- SAT:

+ extremely efficient
- problem with expressivity

- SMT:

+ better expressivity, incorporates domain-specific reasoning
- some loss of efficiency

- CSP:

+ very expressive, uses domain-specific reasoning
- some loss of efficiency

highly problem-dependent though..

DART: Directed Automated Random Testing

Koushik Sen
University of Illinois Urbana-Champaign

Joint work with Patrice Godefroid and Nils Klarlund

Software Testing

- Testing accounts for 50% of software development cost
- Software failure costs USA $60 billion annually
  - Improvement in software testing infrastructure can save one-third of this cost

"The economic impacts of inadequate infrastructure for software testing", NIST, May, 2002

- Currently, software testing is mostly done manually

Simple C code

```
int double(int x) {
    return 2 * x;
}
void test_me(int x, int y){
    int z = double(x);
    if(z==y){
            if(x != y+10){
                    printf("I am fine here");
            } else {
                    printf("I should not reach here");
                    abort();
            }
}
```

Automatic Extraction of Interface

- Automatically determine (code parsing)
  - inputs to the program
    - arguments to the entry function
  - variables: whose value depends on environment
    - external objects
  - function calls: return value depends on the environment
    - external function calls
- For simple C code
  - want to unit test the function test_me
  - int x and int y : passed as an argument to test_me forms the external environment

Generate Random Test Driver

- Generate a test driver automatically to simulate random environment of the extracted interface
  - most general environment
  - C – code
- Compile the program along with the test driver to create a closed executable.
- Run the executable several times to see if assertion violates

Random test-driver

Random Test Driver

```
main(){
    int tmp1 = randomInt();
    int tmp2 = randomInt();
    test_me(tmp1,tmp2);
}
```

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

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
    if(x != y+10){
        printf("I am fine here");
    } else {
        printf("I should not reach here");
        abort();
    }
}
```

Random test-driver

Random Test Driver

```
main(){
   int tmp1 = randomInt();
   int tmp2 = randomInt();
   test_me(tmp1,tmp2);
}
```

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

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
    if(x != y+10){
        printf("I am fine here");
    } else {
        printf("I should not reach here");
        abort();
    }
}
```

Probability of reaching abort() is extremely low

Limitations

- Hard to hit the assertion violated with random values of x and y
  - there is an extremely low probability of hitting assertion violation
- Can we do better?
  - Directed Automated Random Testing
    - White box assumption

DART Approach

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
           printf("I am fine here");
      } else {
           printf("I should not reach here");
           abort();
      }
}
```

DART Approach

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
t1=36	t1=m	

DART Approach

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
    if(x != y+10){
      printf("I am fine here");
    } else {
      printf("I should not reach here");
      abort();
    }
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints
t1=36, t2=-7		t1=m, t2=n	

DART Approach

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
t1=36, t2=-7	t1=m, t2=n	

DART Approach

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints
x=36, y=-7		x=m, y=n	

DART Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
           printf("I am fine here");
      } else {
           printf("I should not reach here");
           abort();
      }
}
```

x=36, y=-7, z=72

x=m, y=n, z=2m

DART Approach

Concrete Execution | Symbolic Execution

concrete state	symbolic state	constraints
x=36, y=-7, z=72	x=m, y=n, z=2m	2m != n

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```

DART Approach

Concrete Execution | Symbolic Execution

concrete state	symbolic state	constraints
		2m != n
x=36, y=-7, z=72	x=m, y=n, z=2m	

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
    if(x != y+10){
      printf("I am fine here");
    } else {
      printf("I should not reach here");
      abort();
    }
}
```

DART Approach

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y){
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

solve: 2m = n
m=1, n=2

2m != n

x=36, y=-7, z=72 | x=m, y=n, z=2m

DART Approach

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints
t1=1		t1=m	

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```

DART Approach

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints
t1=1, t2=2		t1=m, t2=n	

DART Approach

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
t1=1, t2=2	t1=m, t2=n	

DART Approach

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
    if(x != y+10){
      printf("I am fine here");
    } else {
      printf("I should not reach here");
      abort();
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
x=1, y=2	x=m, y=n	

DART Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```

x=1, y=2, z=2

x=m, y=n, z=2m

DART Approach

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
           printf("I am fine here");
      } else {
           printf("I should not reach here");
           abort();
      }
}
```

Concrete Execution: concrete state	Symbolic Execution: symbolic state	Symbolic Execution: constraints
x=1, y=2, z=2	x=m, y=n, z=2m	2m = n

DART Approach

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
    if(x != y+10){
      printf("I am fine here");
    } else {
      printf("I should not reach here");
      abort();
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
		2m = n
x=1, y=2, z=2	x=m, y=n, z=2m	m != n+10

DART Approach

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
		2m = n
		m != n+10
x=1, y=2, z=2	x=m, y=n, z=2m	

DART Approach

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```

Concrete Execution

Symbolic Execution

concrete state	symbolic state	constraints
		2m = n
		m != n+10
x=1, y=2, z=2	x=m, y=n, z=2m	

DART Approach

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
		2m = n
		m != n+10
x=1, y=2, z=2	x=m, y=n, z=2m	

solve: 2m = n and m=n+10
m= -10, n= -20

DART Approach

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
t1=-10	t1=m	

DART Approach

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
t1=-10, t2=-20	t1=m, t2=n	

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
           printf("I am fine here");
      } else {
           printf("I should not reach here");
           abort();
      }
}
```

DART Approach

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
    if(x != y+10){
      printf("I am fine here");
    } else {
      printf("I should not reach here");
      abort();
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
t1=-10, t2=-20	t1=m, t2=n	

DART Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```

x=-10, y=-20

x=m, y=n

DART Approach

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
x=-10, y=-20, z=-20	x=m, y=n, z=2m	

DART Approach

Concrete Execution | Symbolic Execution

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```

concrete state	symbolic state	constraints
x=-10, y=-20, z=-20	x=m, y=n, z=2m	2m = n

DART Approach

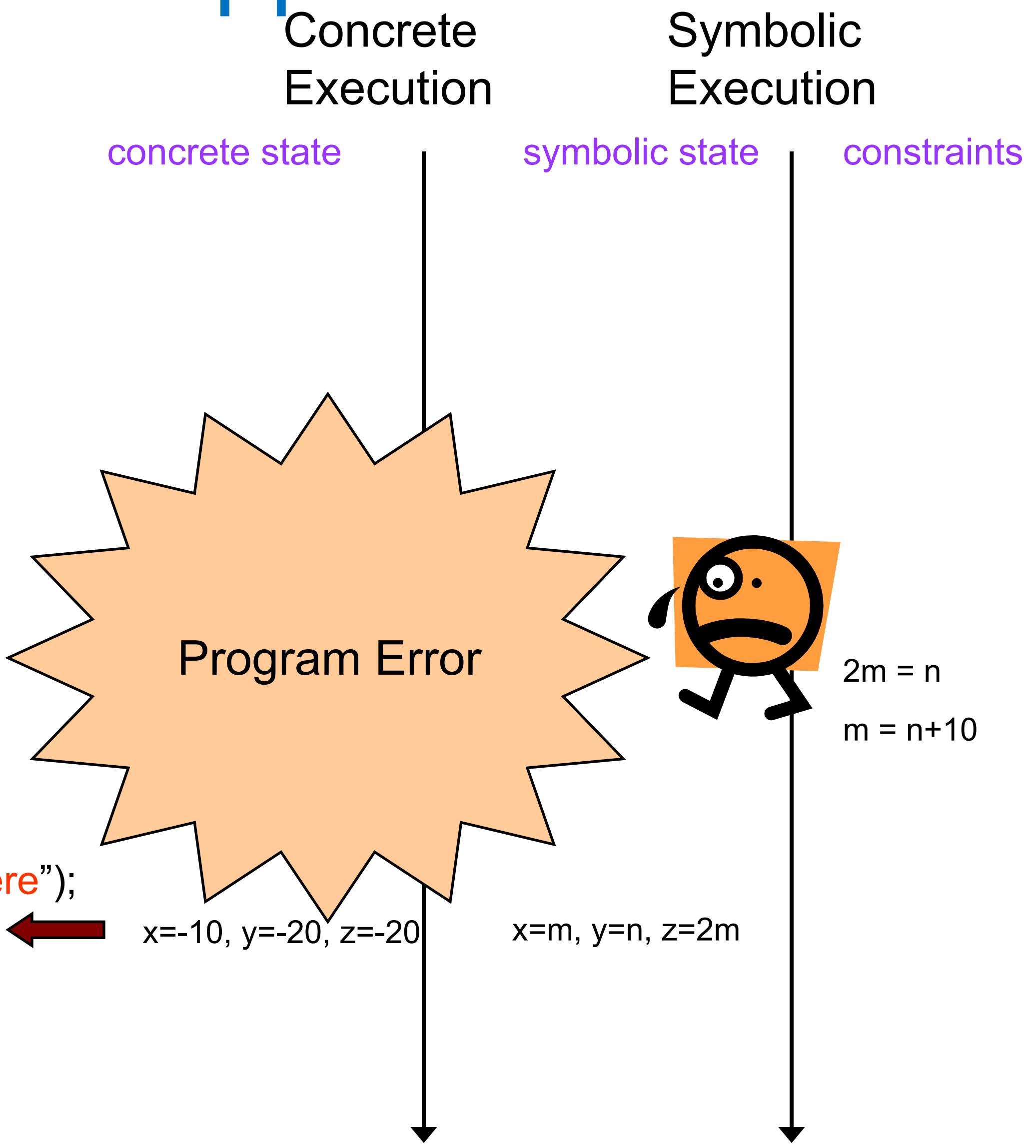
Concrete Execution | Symbolic Execution

concrete state	symbolic state	constraints
		2m = n
x=-10, y=-20, z=-20	x=m, y=n, z=2m	m = n+10

```
main(){
	int t1 = randomInt();
	int t2 = randomInt();
	test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
	if(x != y+10){
		printf("I am fine here");
	} else {
		printf("I should not reach here");
		abort();
	}
}
```

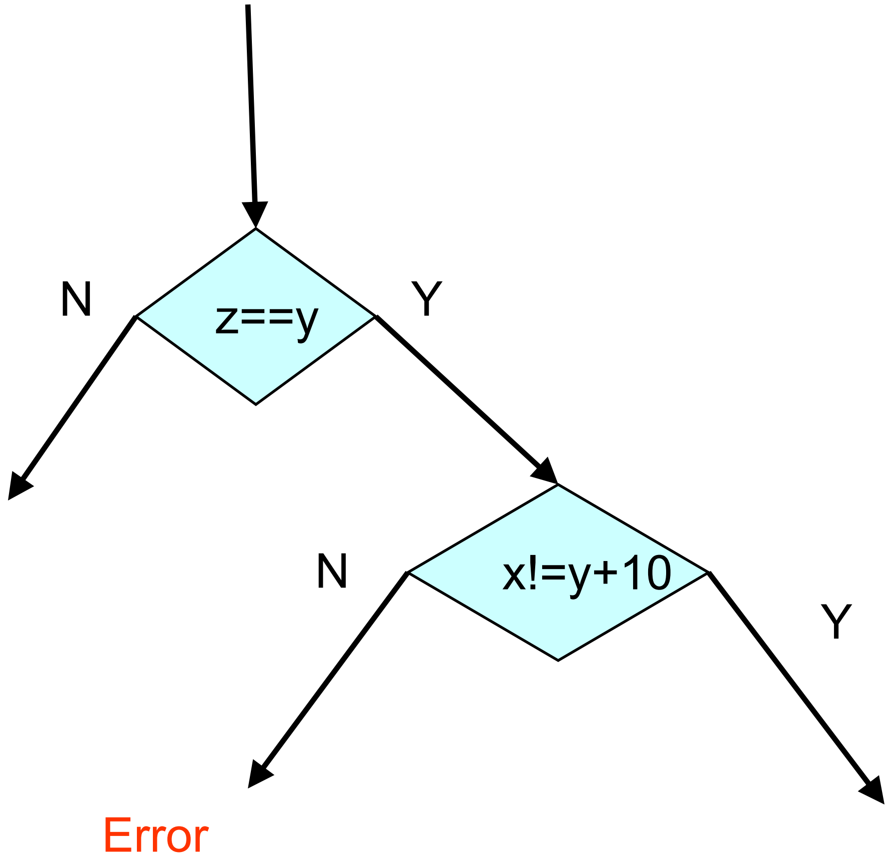
DART Approach

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          abort();
      }
}
```



DART Approach

```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if(z==y){
      if(x != y+10){
           printf("I am fine here");
      } else {
           printf("I should not reach here");
           abort();
      }
}
```



DART in a Nutshell

- Dynamically observe random execution and generate new test inputs to drive the next execution along an alternative path
  - do dynamic analysis on a random execution
  - collect symbolic constraints at branch points
  - negate one constraint at a branch point (say b)
  - call constraint solver to generate new test inputs
  - use the new test inputs for next execution to take alternative path at branch b
  - (Check that branch b is indeed taken next)

More details

- Instrument the C program to do both
  - Concrete Execution
    - Actual Execution
  - Symbolic Execution and Lightweight theorem proving (path constraint solving)
    - Dynamic symbolic analysis
    - Interacts with concrete execution
- Instrumentation also checks whether the next execution matches the last prediction.

Advantage of Dynamic Analysis over Static Analysis

```
struct foo { int i; char c; }

bar (struct foo *a) {
    if (a->c == 0) {
        *((char *)a + sizeof(int)) = 1;
        if (a->c != 0) {
            abort();
        }
    }
}
```

- Reasoning about dynamic data is easy
- Due to limitation of alias analysis "static analyzers" cannot determine that "a->c" has been rewritten
  - BLAST would infer that the program is safe
- DART finds the error
  - sound

Further advantages

```
1  foobar(int x, int y){
2    if (x*x*x > 0){
3       if (x>0 && y==10){
4           abort();
5       }
6    } else {
7       if (x>0 && y==20){
8           abort();
9       }
10   }
11 }
```

- static analysis based model-checkers would consider both branches
  - both abort() statements are reachable
  - false alarm
- Symbolic execution gets stuck at line number 2 (due to non-linear arithmetic)
- DART finds the only error

Discussion

- In comparison to existing testing tools, DART is
  - light-weight
  - dynamic analysis (compare with static analysis)
    - ensures no false alarms
  - concrete execution and symbolic execution run simultaneously
    - symbolic execution consults concrete execution whenever dynamic analysis becomes intractable
  - real tool that works on real C programs
    - completely automatic

Symbolic execution seems to be not so scalable, are there other approaches?

Fuzzing

Adapted from
http://www.cs.columbia.edu/~suman/secure_sw_devel/fuzzing.pptx

Suman Jana

*Acknowledgements: Dawn Song, Kostya Serebryany,
Peter Collingbourne

Techniques for bug finding

Automatic test case generation

Static analysis

Program verification

Fuzzing

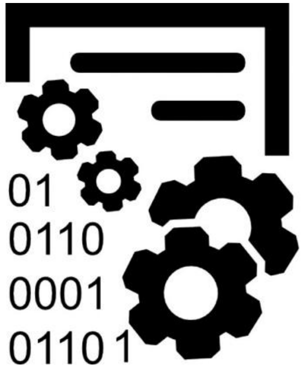
Dynamic symbolic execution

Lower coverage
Lower false positives
Higher false negatives

Higher coverage
Higher false positives
Lower false negatives

Blackbox fuzzing

Random input



Test program

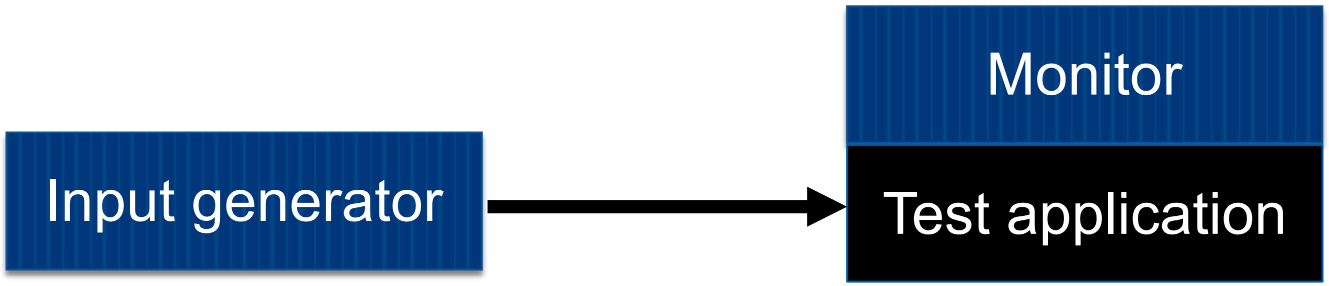
Miller et al. ‘89

Blackbox fuzzing

- Given a program simply feed random inputs and see whether it exhibits incorrect behavior (e.g., crashes)
- Advantage: easy, low programmer cost
- Disadvantage: inefficient
  - Inputs often require structures, random inputs are likely to be malformed
  - Inputs that trigger an incorrect behavior is a a very small fraction, probably of getting lucky is very low

Fuzzing

- Automatically generate test cases
- Many slightly anomalous test cases are input into a target
- Application is monitored for errors
- Inputs are generally either file based (.pdf, .png, .wav, etc.) or network based (http, SNMP, etc.)



Problem detection

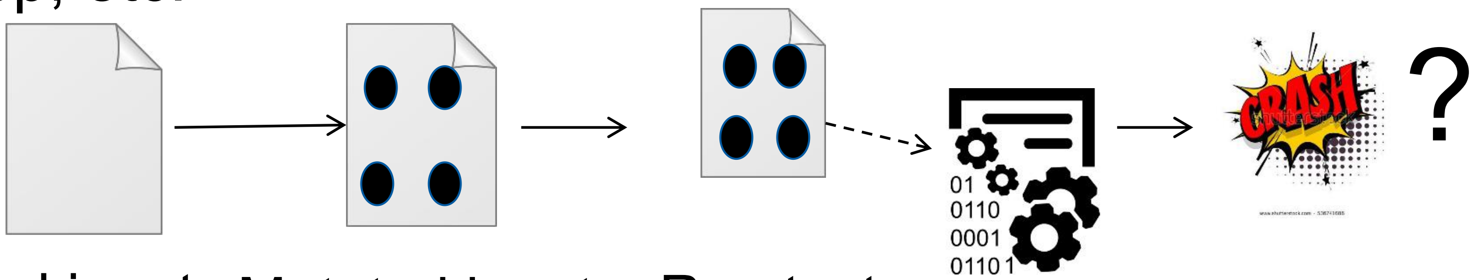
- See if program crashed
  - Type of crash can tell a lot (SEGV vs. assert fail)
- Run program under dynamic memory error detector (valgrind/purify/AddressSanitizer)
  - Catch more bugs, but more expensive per run.
- See if program locks up
- Roll your own dynamic checker e.g. valgrind skins

Regression vs. Fuzzing

	Regrssion	Fuzzing
Definition	Run program on many normal inputs, look for badness	Run program on many abnormal inputs, look for badness
Goals	Prevent normal users from encountering errors (e.g., assertion failures are bad)	Prevent attackers from encountering exploitable errors (e.g., assertion failures are often ok)

Enhancement 1:
Mutation-Based fuzzing

- Take a well-formed input, randomly perturb (flipping bit, etc.)
- Little or no knowledge of the structure of the inputs is assumed
- Anomalies are added to existing valid inputs
  - Anomalies may be completely random or follow some heuristics (e.g., remove NULL, shift character forward)
- Examples: ZZUF, Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.



Seed input Mutated input Run test program

Example: fuzzing a PDF viewer

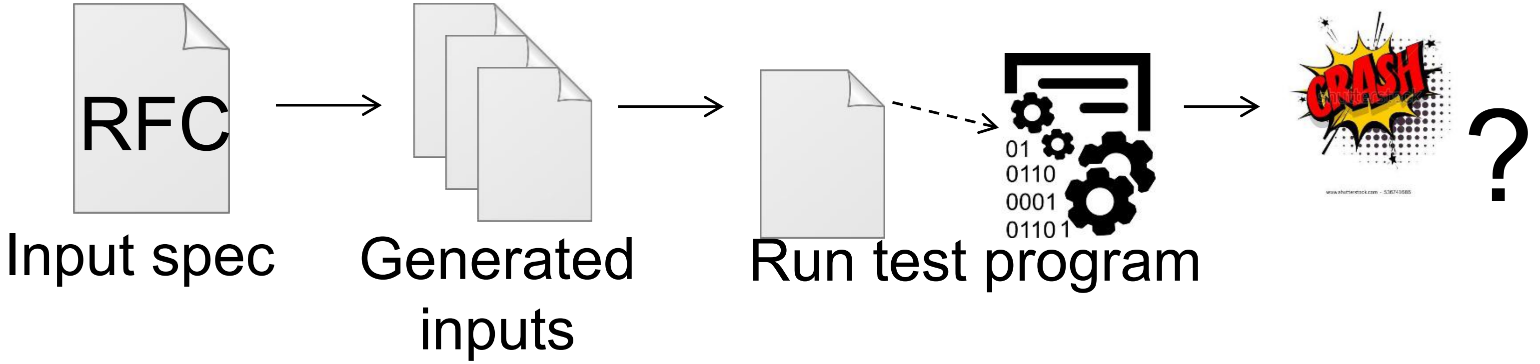
- Google for .pdf (about 1 billion results)
- Crawl pages to build a corpus
- Use fuzzing tool (or script)
  - Collect seed PDF files
  - Mutate that file
  - Feed it to the program
  - Record if it crashed (and input that crashed it)

Mutation-based fuzzing

- Super easy to setup and automate
- Little or no file format knowledge is required
- Limited by initial corpus
- May fail for protocols with checksums, those which depend on challenge

Enhancement II: Generation-Based Fuzzing

- Test cases are generated from some description of the input format: RFC, documentation, etc.
  - Using specified protocols/file format info
  - E.g., SPIKE by Immunity
- Anomalies are added to each possible spot in the inputs
- Knowledge of protocol should give better results than random fuzzing



Enhancement II: Generation-Based Fuzzing

```
//png.spk
//author: Charlie Miller

// Header - fixed.
s_binary("89504E470D0A1A0A");

// IHDRChunk
s_binary_block_size_word_bigendian("IHDR"); //size of data field
s_block_start("IHDRcrc");
        s_string("IHDR");  // type
        s_block_start("IHDR");
// The following becomes s_int_variable for variable stuff
// 1=BINARYBIGENDIAN, 3=ONEBYE
                s_push_int(0x1a, 1);     // Width
                s_push_int(0x14, 1);     // Height
                s_push_int(0x8, 3);      // Bit Depth - should be 1,2,4,8,16, base
                s_push_int(0x3, 3);      // ColorType - should be 0,2,3,4,6
                s_binary("00 00");       // Compression || Filter - shall be 00 00
                s_push_int(0x0, 3);      // Interlace - should be 0,1
        s_block_end("IHDR");
s_binary_block_crc_word_littleendian("IHDRcrc"); // crc of type and data
s_block_end("IHDRcrc");
...
```

Sample PNG spec

Mutation-based vs. Generation-based

- Mutation-based fuzzer
  - Pros: Easy to set up and automate, little to no knowledge of input format required
  - Cons: Limited by initial corpus, may fall for protocols with checksums and other hard checks
- Generation-based fuzzers
  - Pros: Completeness, can deal with complex dependncies (e.g, checksum)
  - Cons: writing generators is hard, performance depends on the quality of the spec

How much fuzzing is enough?

- Mutation-based-fuzzers may generate an infinite number of test cases. When has the fuzzer run long enough?
- Generation-based fuzzers may generate a finite number of test cases. What happens when they’re all run and no bugs are found?

Code coverage

- Some of the answers to these questions lie in *code coverage*
- Code coverage is a metric that can be used to determine how much code has been executed.
- Data can be obtained using a variety of profiling tools. e.g. gcov, lcov

Line coverage

- **Line/block coverage**: Measures how many lines of source code have been executed.
- For the code on the right, how many test cases (values of pair (a,b)) needed for full(100%) line coverage?

```
if( a > 2 )
    a = 2;
if( b >2 )
    b = 2;
```

Branch coverage

- Branch coverage: Measures how many branches in code have been taken (conditional jmps)
- For the code on the right, how many test cases needed for full branch coverage?

```
if( a > 2 )
    a = 2;
if( b >2 )
    b = 2;
```

Path coverage

- Path coverage: Measures how many paths have been taken
- For the code on the right, how many test cases needed for full path coverage?

```
if( a > 2 )
    a = 2;
if( b >2 )
    b = 2;
```

Benefits of Code coverage

- Can answer the following questions
  - How good is an initial file?
  - Am I getting stuck somewhere?

```
if (packet[0x10] < 7) { //hot path
} else { //cold path }
```

- How good is fuzzerX vs. fuzzerY
- Am I getting benefits by running multiple fuzzers?

Problems of code coverage

- For:

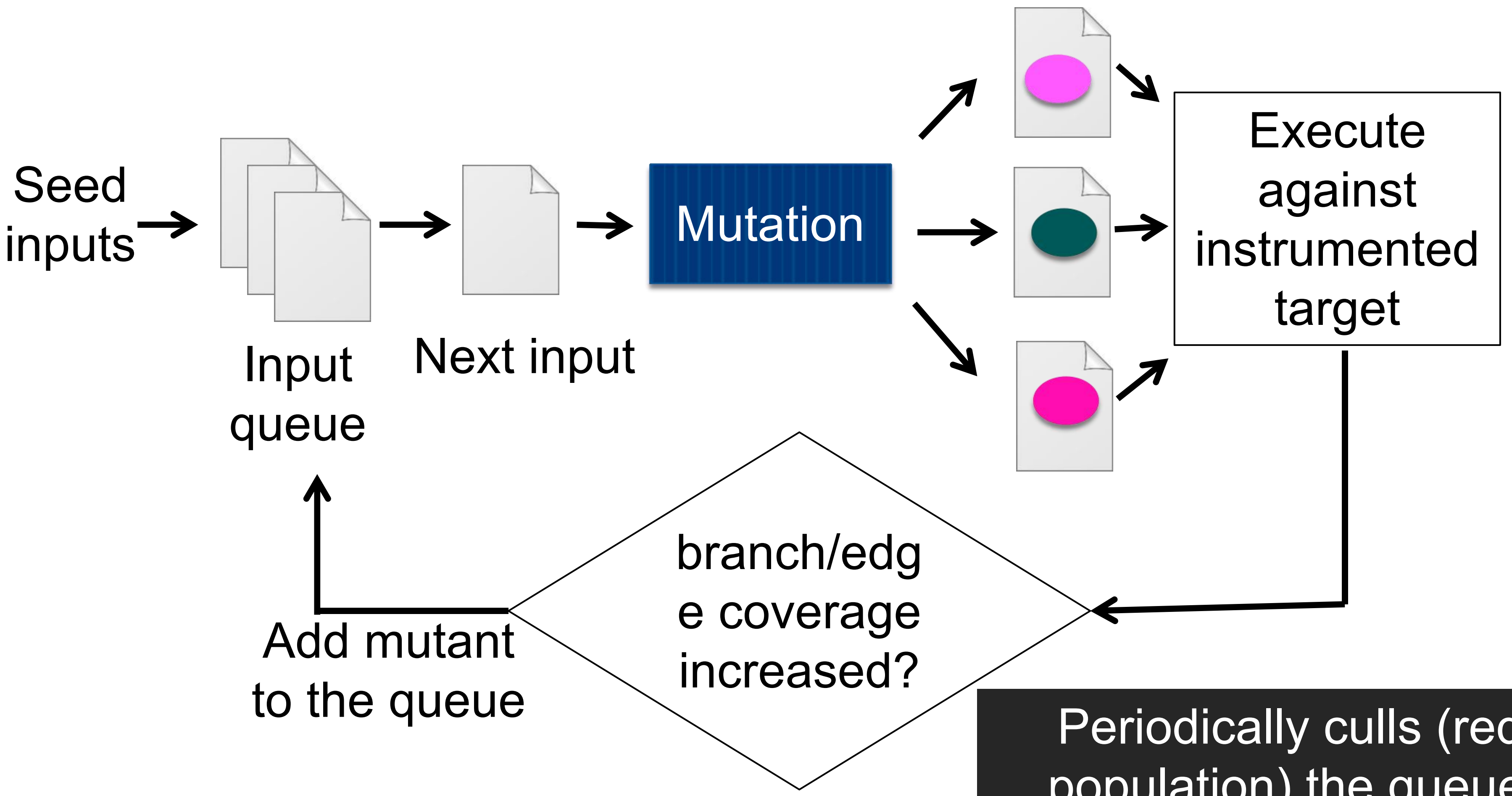
```
mySafeCopy(char *dst, char* src) {
    if(dst && src)
        strcpy(dst, src); }
```

- Does full line coverage guarantee finding the bug?
- Does full branch coverage guarantee finding the bug?

Enhancement III: Coverage-guided gray-box fuzzing

- Special type of mutation-based fuzzing
  - Run mutated inputs on instrumented program and measure code coverage
  - Search for mutants that result in coverage increase
  - Often use genetic algorithms, i.e., try random mutations on test corpus and only add mutants to the corpus if coverage increases
  - Examples: AFL, libfuzzer

American Fuzzy Lop (AFL)



Periodically culls (reduce the population) the queue without affecting total coverage