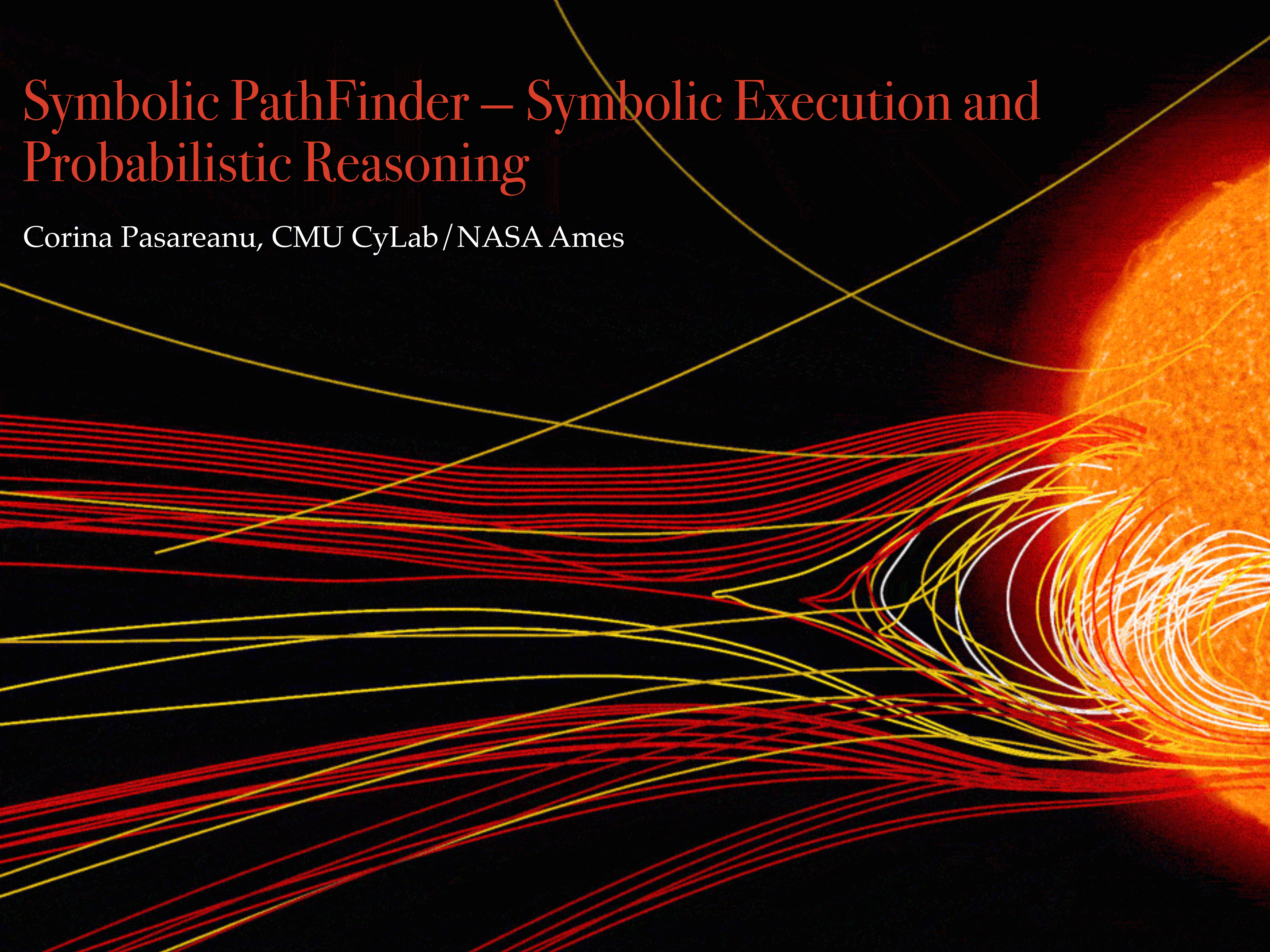
Symbolic PathFinder – Symbolic Execution and Probabilistic Reasoning

Corina Pasareanu, CMU CyLab/NASA Ames



Software Safety and Security

- Software systems become more pervasive and complex
- Increased need for techniques and tools that ensure safety and security of software systems
- Research interests:
  - developing automated verification techniques and
  - their application at all phases of software development
  - both theoretical foundations and practical tools






Approaches to finding errors

- Testing
  - Well accepted technique
  - May miss errors
- Model checking
  - Automatic, exhaustive
  - Scalability issues
- Static analysis
  - Automatic, scalable
  - Reported errors may be spurious

Symbolic Execution

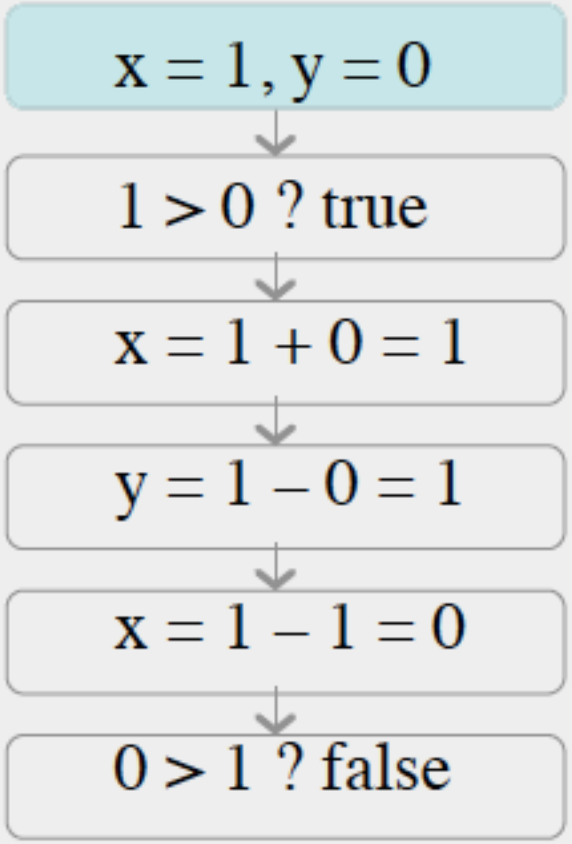
- Systematic program analysis technique — King [Comm. ACM 1976], Clarke [IEEE TSE 1976]
- Executes programs on symbolic inputs — represent multiple concrete inputs
- Path conditions — conditions on inputs following same program path
  - Check satisfiability – explore only feasible paths
  - Solve path conditions: obtain test inputs
- Bounded execution
- Many applications: test-case generation, error detection, …
- Many tools: SAGE, DART, KLEE, Pex, BitBlaze …
- Symbolic PathFinder

Example Concrete Execution

Code that swaps 2 integers

```
int x, y;
if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert false;
}
```

Concrete Execution Path

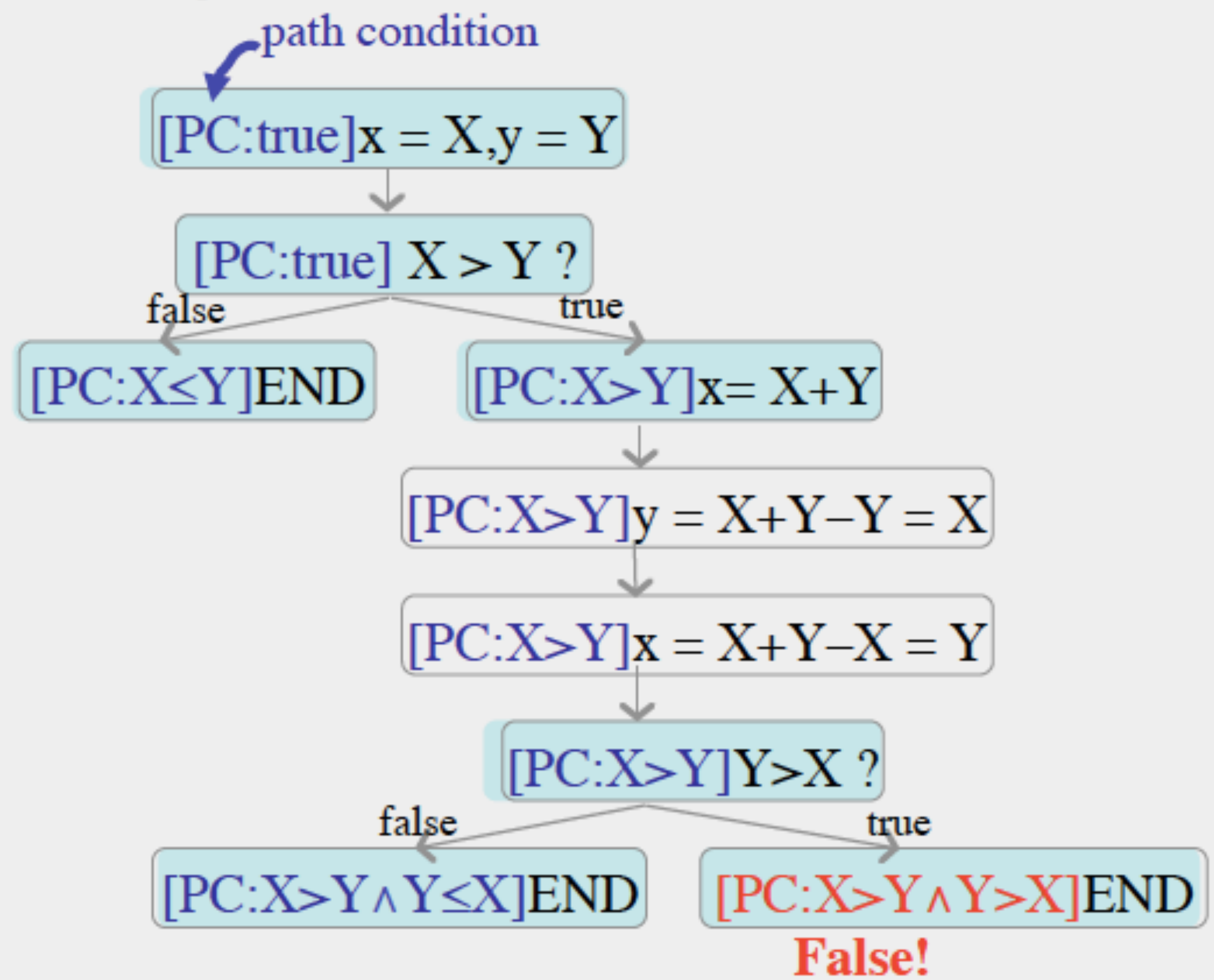


Example Symbolic Execution

Code that swaps 2 integers

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

Symbolic Execution Tree



Solve PCs: obtain test inputs

Another Example

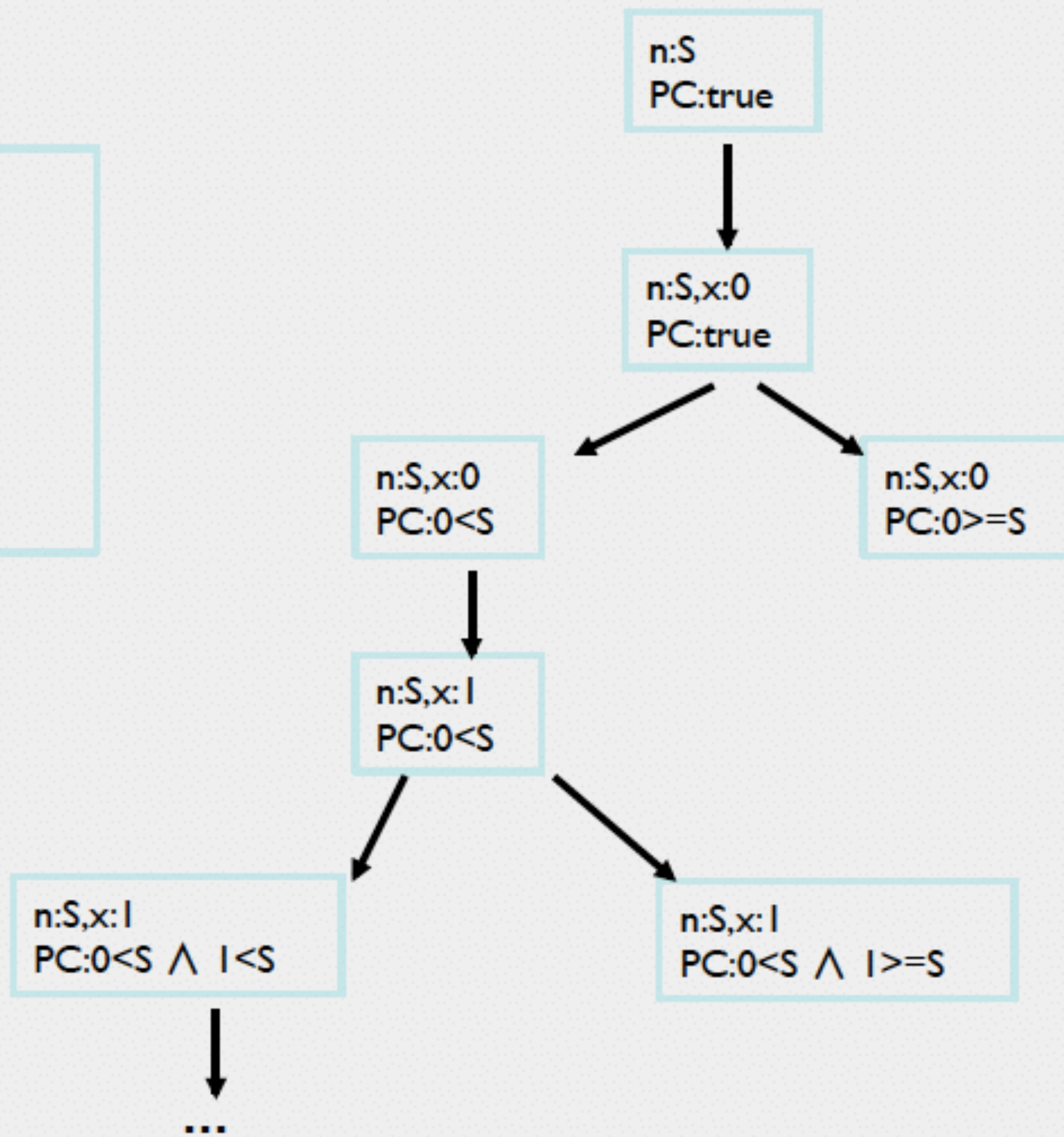
```
void test (int n) {
    int x=0;
    while (x<n)
        x=x+1;
}
```

Loops

example code

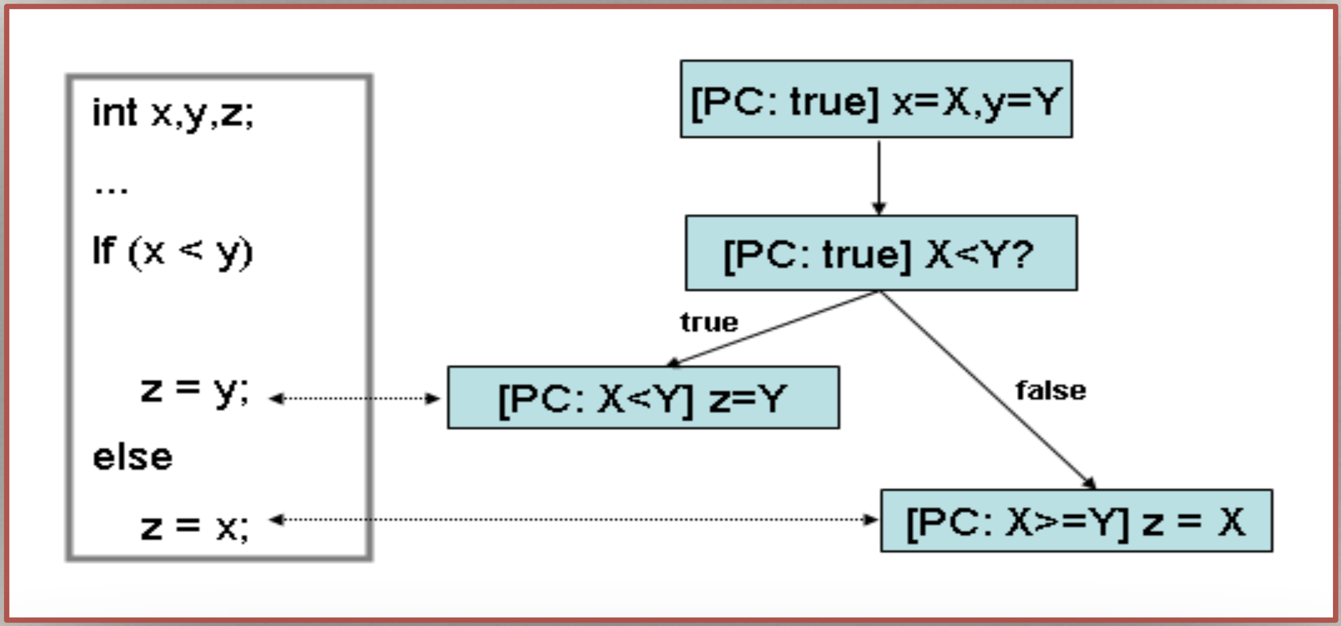
```
void test(int n) {
    int x = 0;
    while(x < n)
      x = x + 1;
}
```

infinite symbolic execution tree

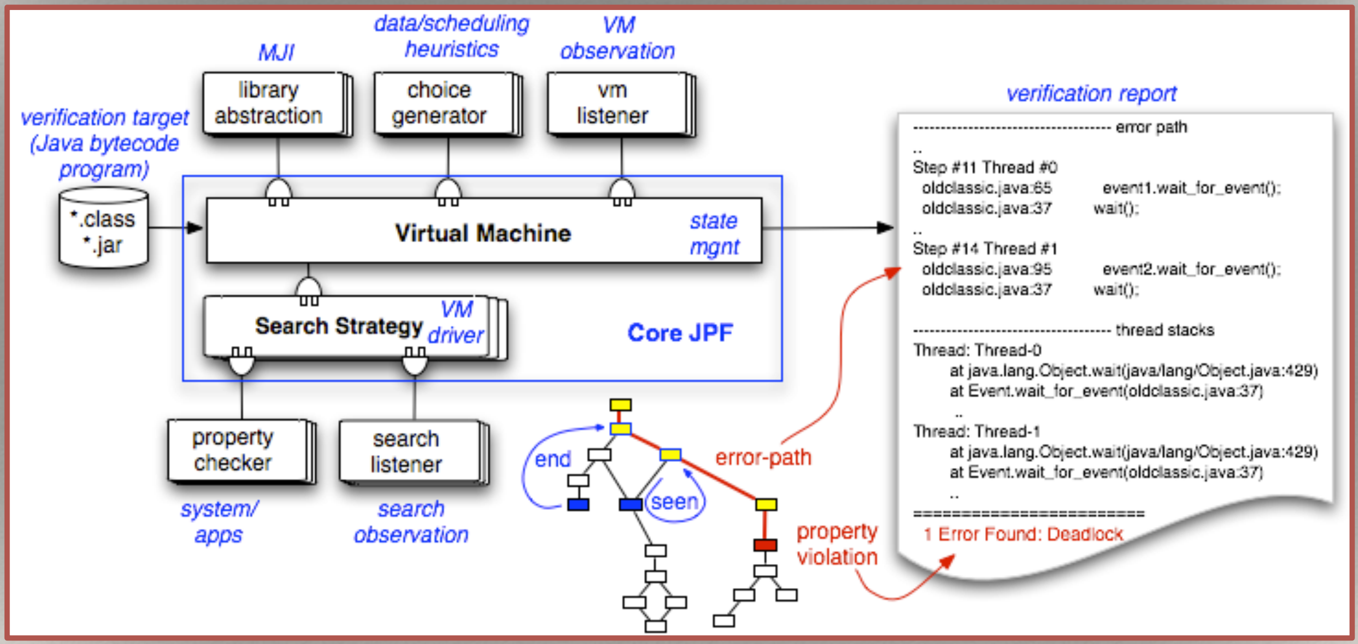


Symbolic PathFinder

- Symbolic execution tool for Java bytecode
- Lazy initialization for input data structures and arrays
- Handles multi-threading and string operations
- Supports quantitative reasoning
- Comes with library models
- Enables symbolic execution to start at “any point”
- Uses machine learning to infer “unit preconditions” based on concrete runs



Java PathFinder tool-set

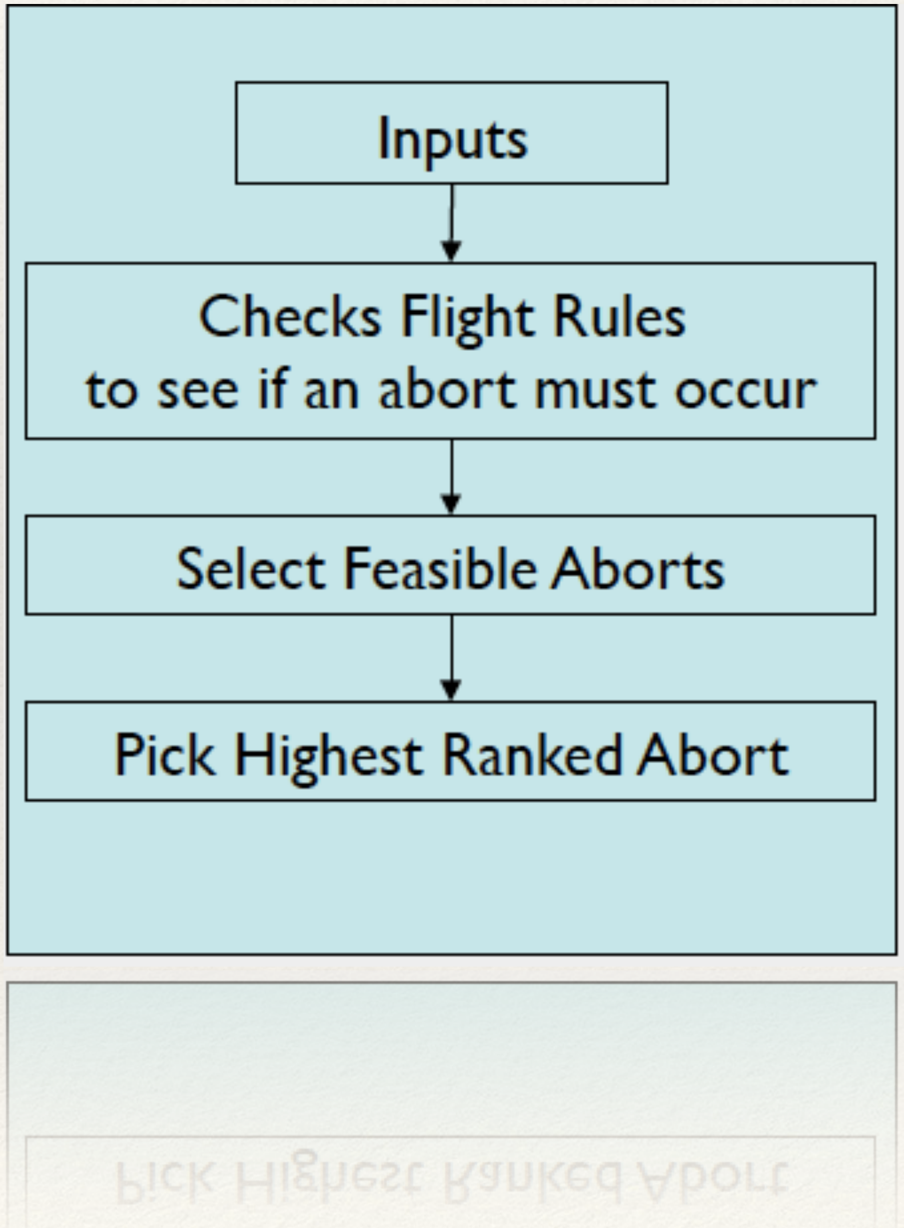


https://github.com/SymbolicPathFinder/jpf-symbc

Test Generation for NASA Applications

- NASA control software: onboard abort executive (OAE) [ISSTA'08]
  - manual testing: time consuming ~ 1 week
  - guided random testing could not obtain full coverage
  - SPF generated ~200 tests to obtain full coverage <1min
  - Flight rules covered 27/27
  - Aborts covered 7/7
  - Size of input: 27 values/test case
- Found major bug in new version

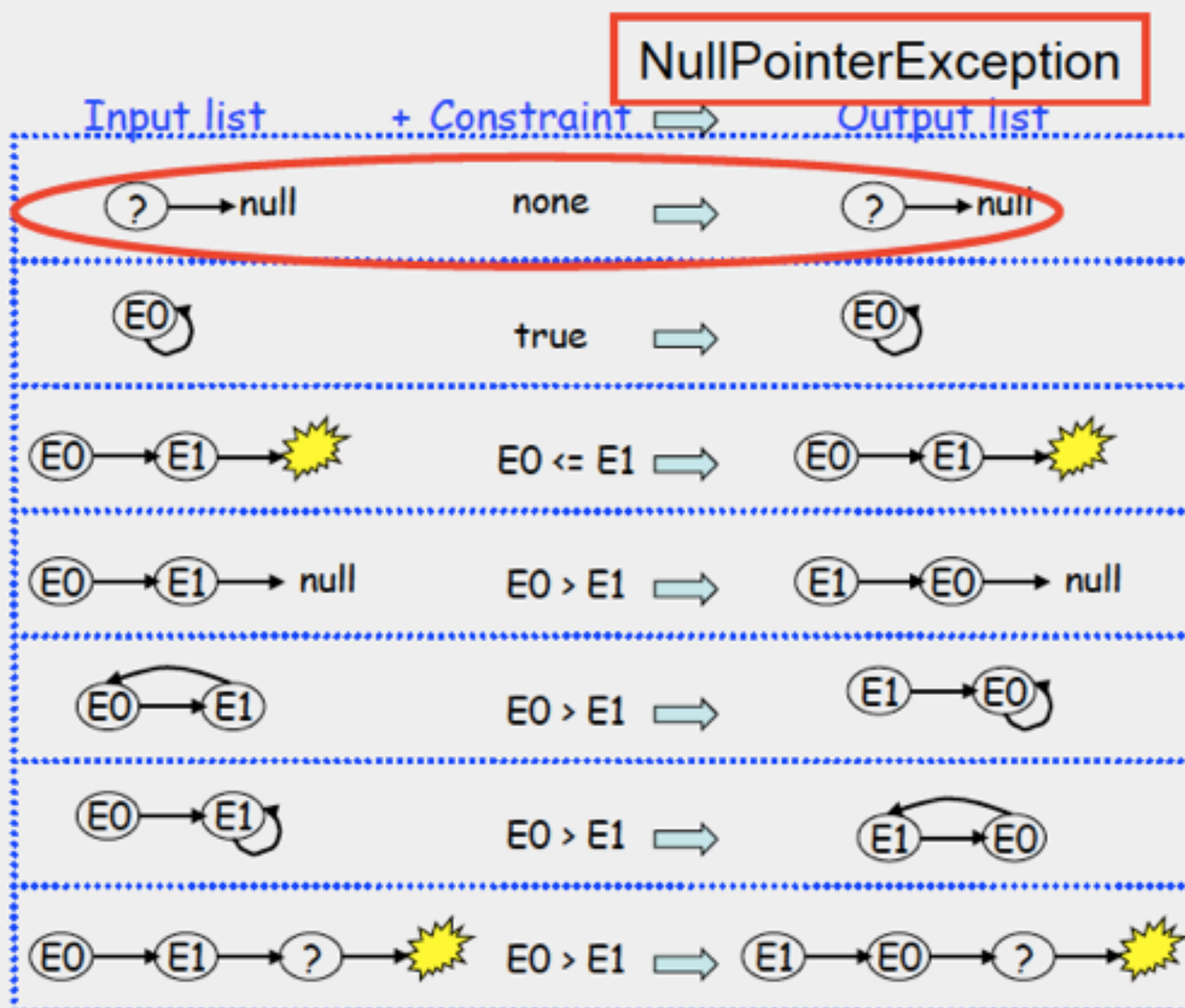
OAE structure



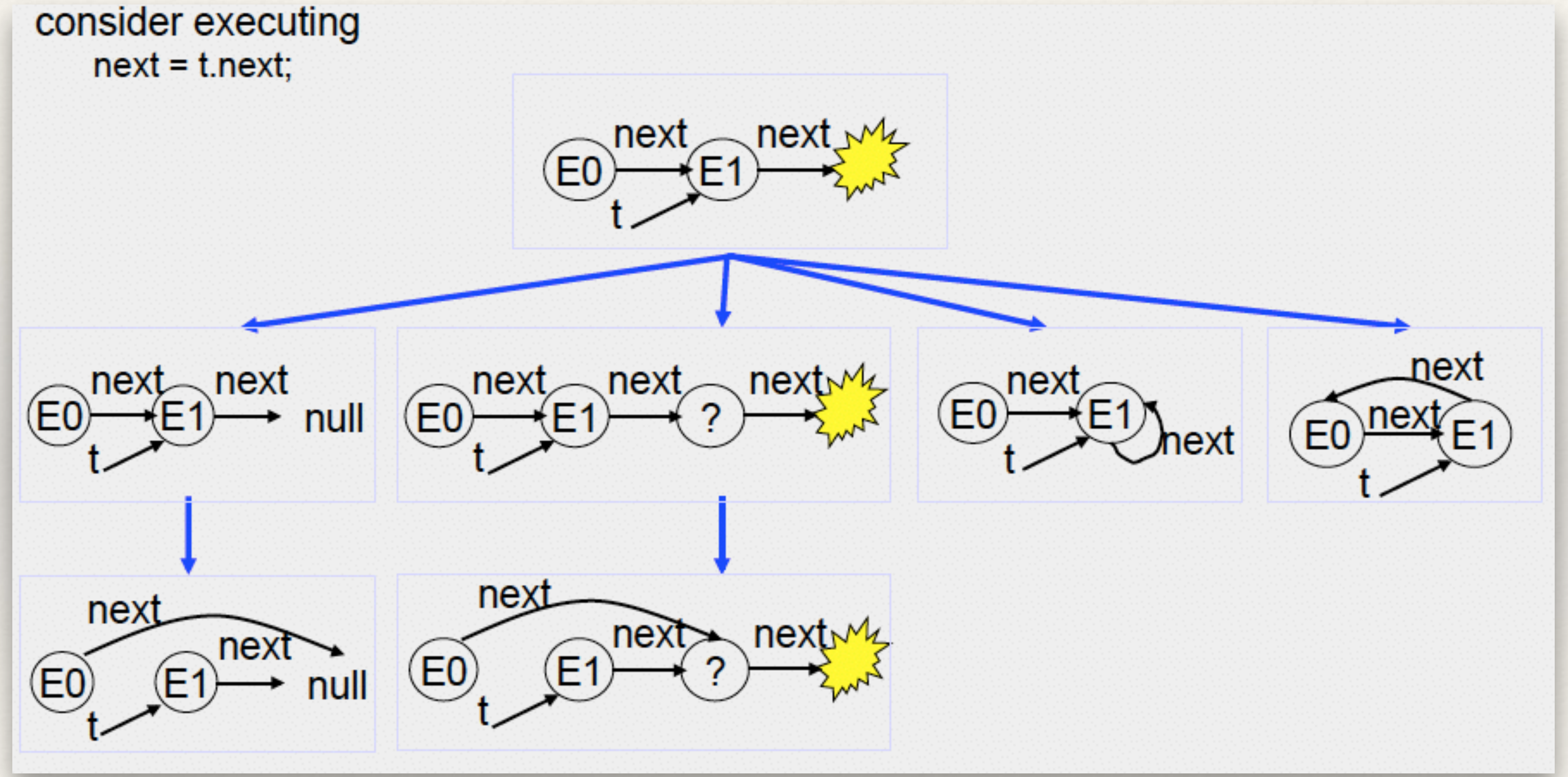
Handling Data Structures

- Lazy initialization [TACAS'03,ISSTA'04] — nondeterminism handles aliasing

```
class Node {
    int elem;
    Node next;

    Node swapNode() {
      if (next != null)
        if (elem > next.elem) {
          Node t = next;
          next = t.next;
          t.next = this;
          return t;
        }
      return this;
    }
}
```



Lazy Initialization



Dynamic Symbolic Execution/Concolic Testing

- collect symbolic constraints **during** concrete executions
- DART = Directed Automated Random Testing
- Concolic = Concrete/symbolic testing

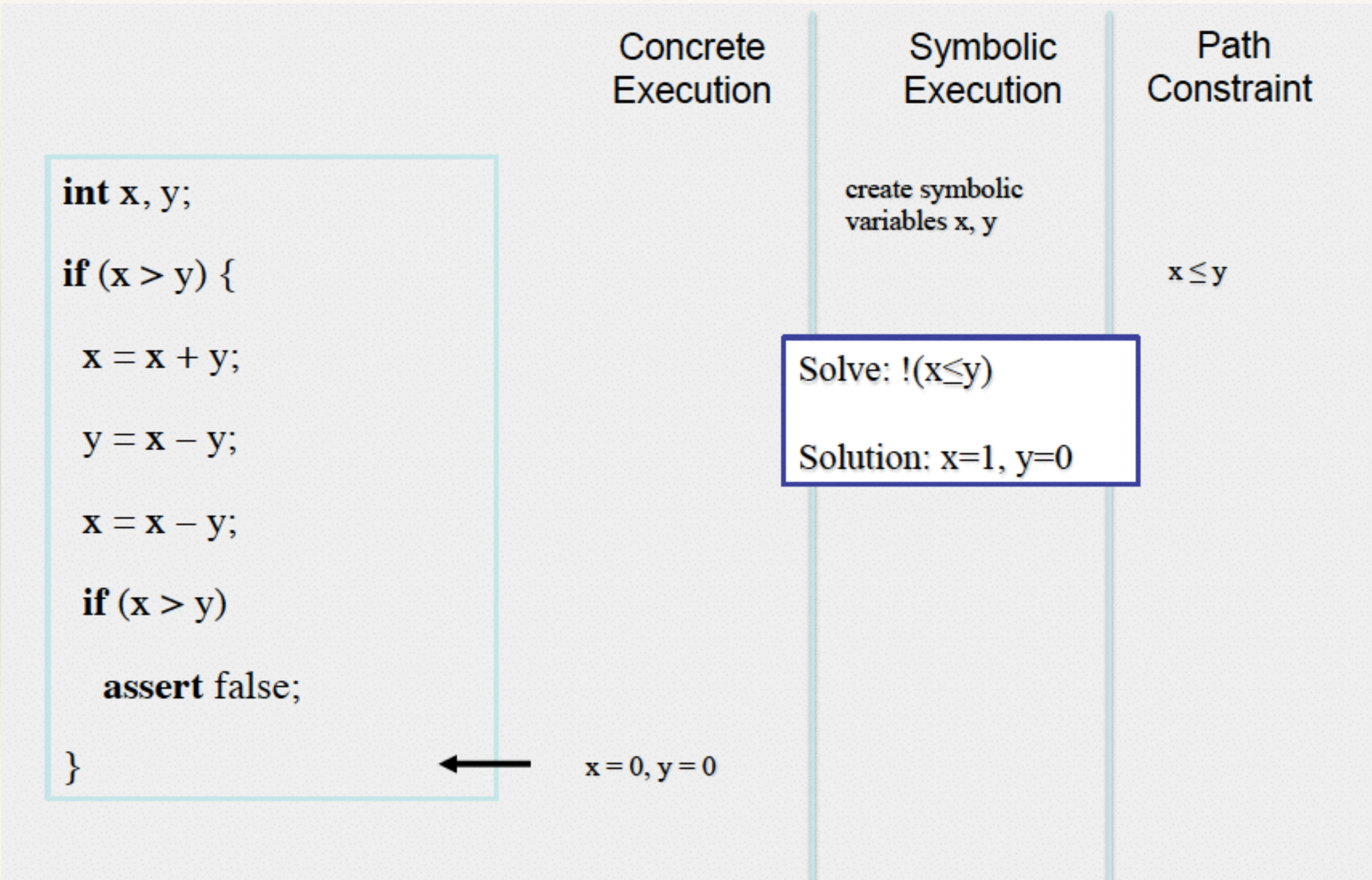
- P. Godefroid, K. Sen and many many others …
- very popular, simple to implement

Dynamic Symbolic Execution/Concolic Testing

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

Program	Concrete Execution	Symbolic Execution	Path Constraint
← (int x, y;)	x = 0, y = 0	create symbolic variables x, y	

Dynamic Symbolic Execution/Concolic Testing



Dynamic Symbolic Execution/Concolic Testing

```
int x, y;                 <--

if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert false;
}
```

Concrete Execution	Symbolic Execution	Path Constraint
x = 1, y = 0	create symbolic variables x, y	

Dynamic Symbolic Execution/Concolic Testing

```
int x, y;
if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert false;
}
```

Concrete Execution	Symbolic Execution	Path Constraint
x = 1, y = 0	create symbolic variables x, y	x > y

Dynamic Symbolic Execution/Concolic Testing

```
int x, y;
if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert false;
}
```

Code	Concrete Execution	Symbolic Execution	Path Constraint
int x, y;		create symbolic variables x, y	
if (x > y) {			x > y
x = x + y; ←	x = 1, y = 0	x = x+y	

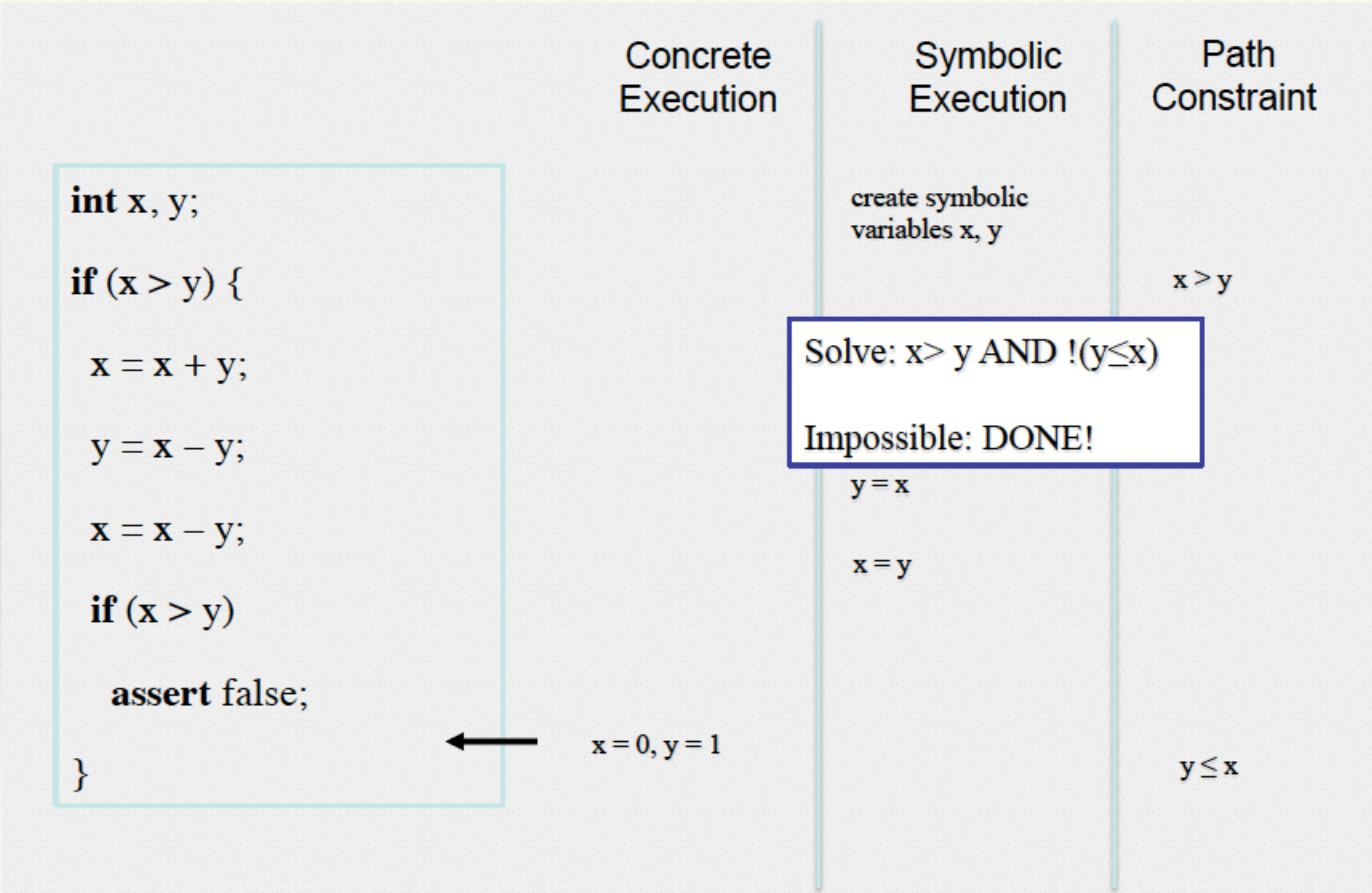
Dynamic Symbolic Execution/Concolic Testing

Code	Concrete Execution	Symbolic Execution	Path Constraint
`int x, y;`		create symbolic variables x, y	
`if (x > y) {`			x > y
`x = x + y;`		x = x+y	
`y = x – y;` ←	x = 1, y = 1	y = x	
`x = x – y;`			
`if (x > y)`			
`assert false;`			
`}`			

Dynamic Symbolic Execution/Concolic Testing

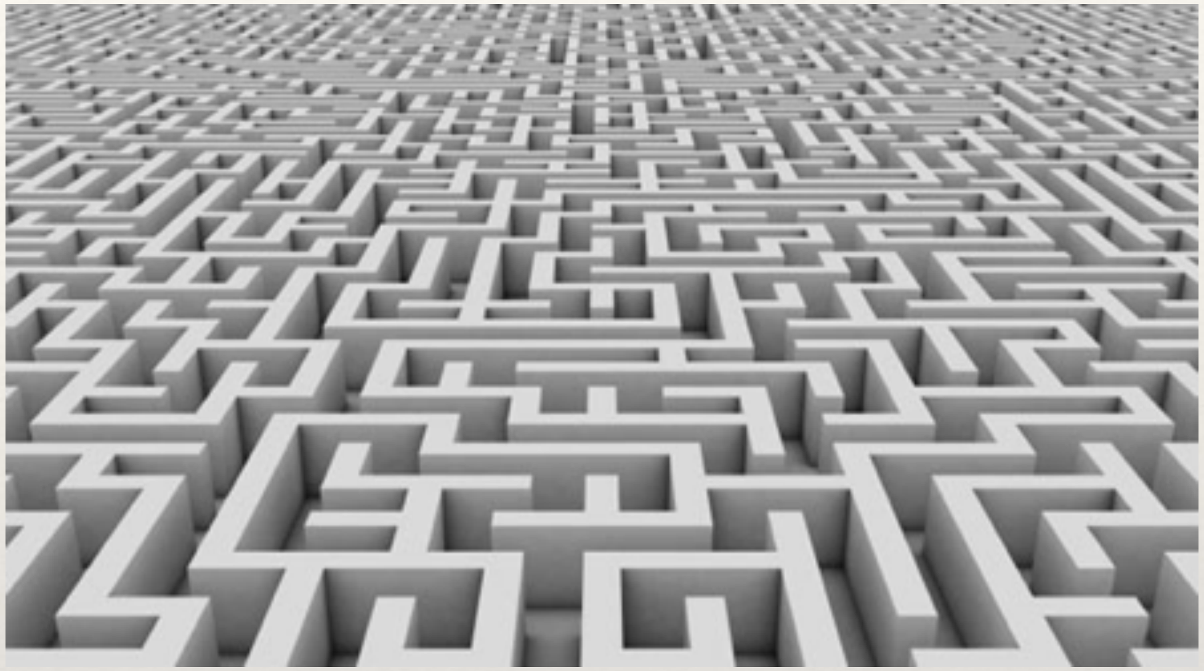
Code		Concrete Execution	Symbolic Execution	Path Constraint
int x, y;			create symbolic variables x, y	
if (x > y) {				x > y
x = x + y;				
y = x − y;			y = x	
x = x − y;	←	x = 0, y = 1	x = y	
if (x > y)				
assert false;				
}				

Dynamic Symbolic Execution/Concolic Testing



Complexity Analysis

- Problem
  - Estimate the worst-case complexity of programs
- Applications
  - Finding vulnerabilities related to denial-of-service attacks
  - Guiding compiler optimizations
  - Finding and fixing performance bottlenecks in software



DARPA STAC

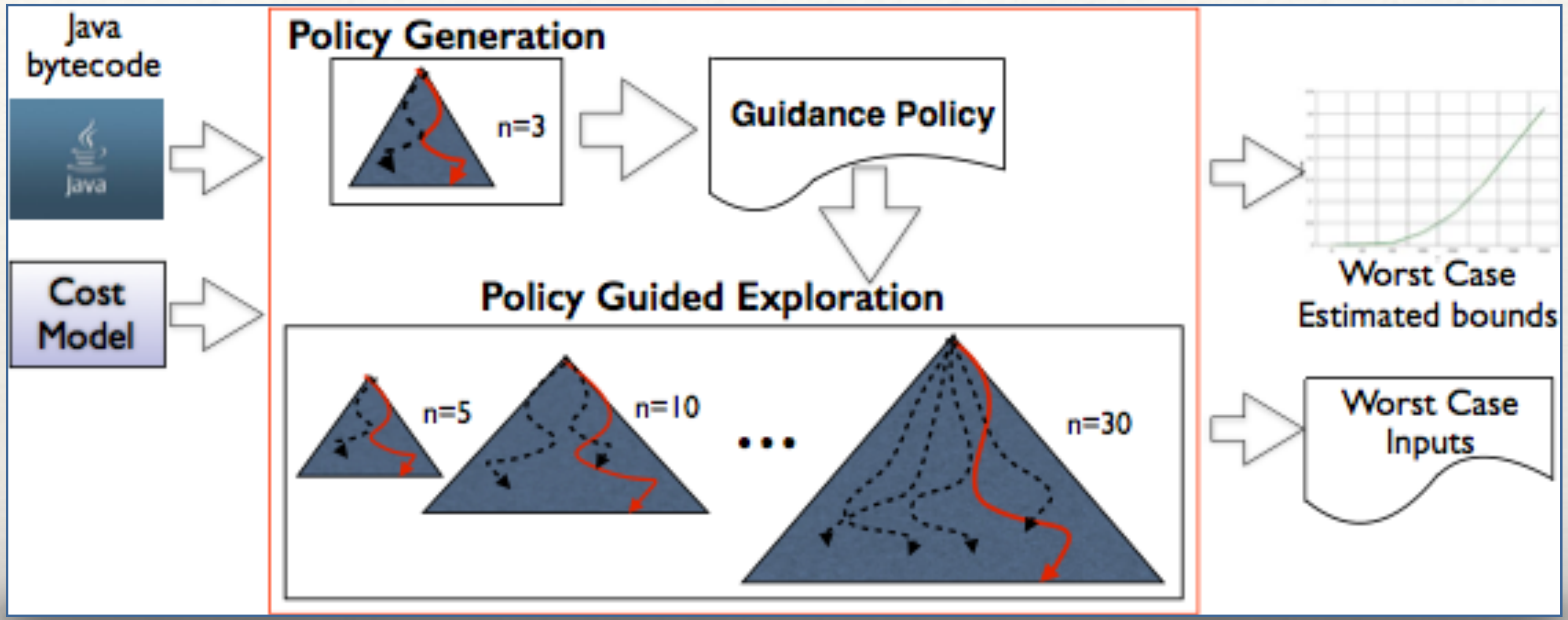
Symbolic Complexity Analysis

- Computes inputs that expose worst-case behavior
- Computes bounds on worst-case complexity
- Simple approach
  - Perform symbolic execution over the program — compute cost of each path
  - Return the path with largest cost
  - Has scalability issues
- Symbolic execution guided by path policies [ICST'17]
  - Encode choices along worst-case path
  - Intuition: worst-case behavior for small input can predict worst-case behavior for larger input

https://github.com/isstac/spf-wca

Guided Symbolic Execution

- Policy Generation
  - Exhaustive symbolic execution at small input size(s)
  - Compute path with largest cost
  - Build policy based on decisions taken along that path
- Policy Guided Execution
  - Symbolic execution for increasing input sizes
  - Explore only paths that conform with policy
  - For each input size compute path (and input) with largest cost



- Function fitting
  - Computes estimate of worst-case behavior as a function of input size
  - Gives lower bounds on worst-case complexity for any size

Guessed bounds can be proved using a resource analysis

Path Policies

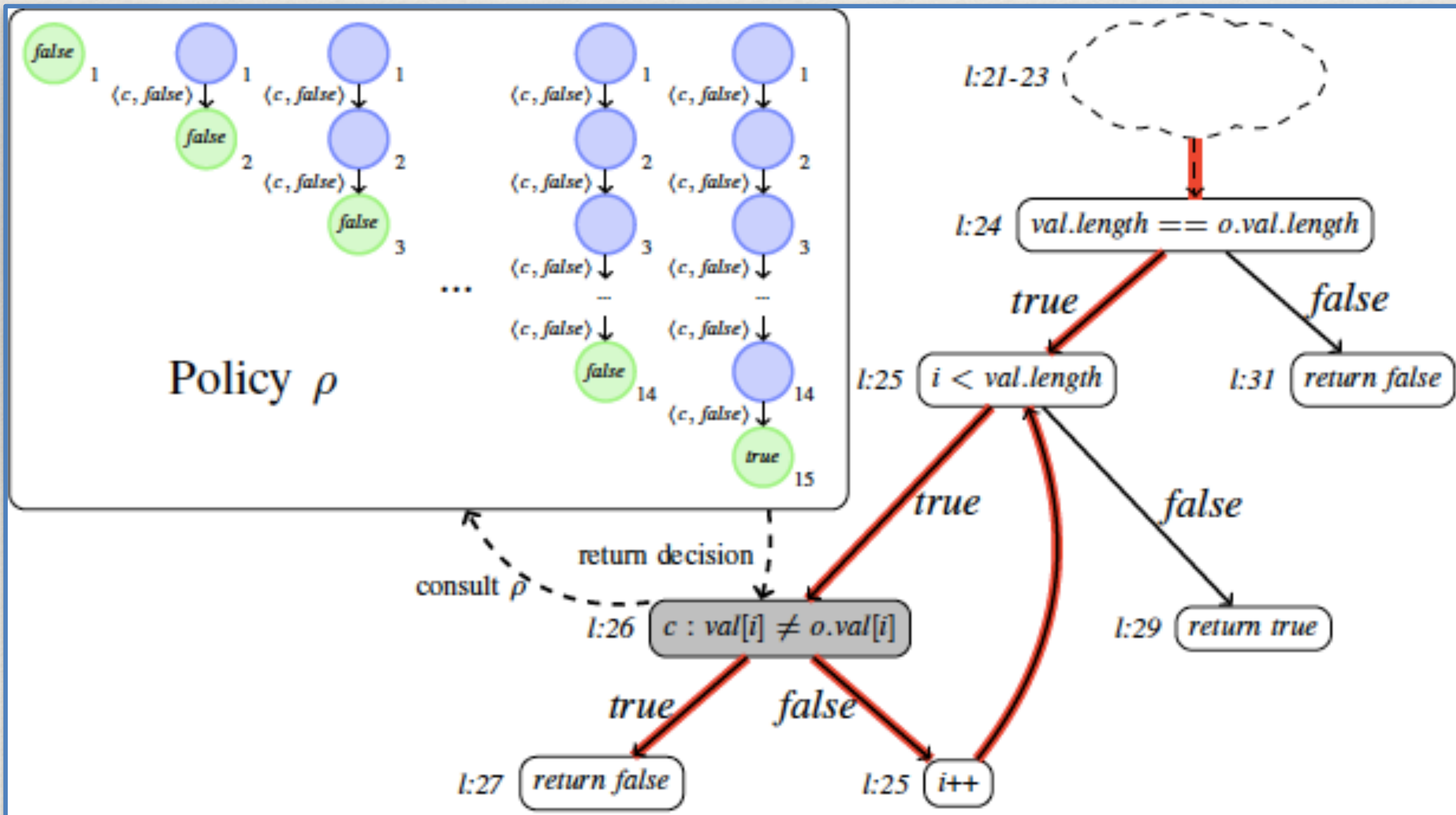
- Decide which branch to execute for the conditions in the program
  - Similar to e.g. [Burnim et al. ICSE'09, Zhang et al. ASE'11]
- **New**
  - History aware: take into account the history of choices made along a path to decide which branch to execute next
  - Context preserving: the decision for each condition depends on the history computed with respect to the enclosing method
- Symbolic execution, guided by policies, can reduce to exploring a single path regardless of input size
- Scales far beyond non-guided symbolic execution and outperforms previous techniques
- **Theoretical guarantee:** when policies are "unified", worst-case path policy is eventually found
  - Unification over policies obtained for successive small inputs
  - For each condition: take union over decisions specified by each policy

Example

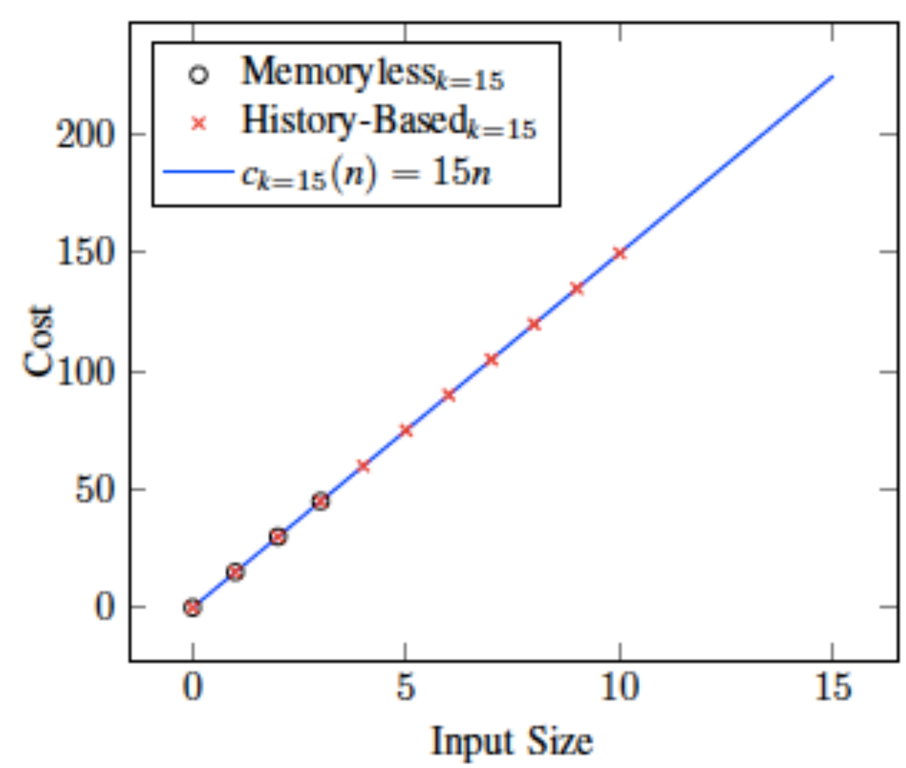
Hash collisions organized in a list

```
....
7 Entry findEntry(String o, ....) {
8     for(Entry e = l; e!=null; e=e.next) {
9         if (e.key.equals(o)) {
10            return e;
11        }
12    }
....
16    return null;
17 }
```

```
18 class String {
19 char[] value;
20 // ...
21  public boolean equals(Object oObj) {
22    // ...
23    String o = (String) oObj;
24    if (val.length == o.val.length) {
25       for(int i=0; i<val.length; i++) {
26         if (val[i]!=o.val[i])
27            return false;
28       }
29       return true;
30    }
31    return false;
32  }
33 }
```

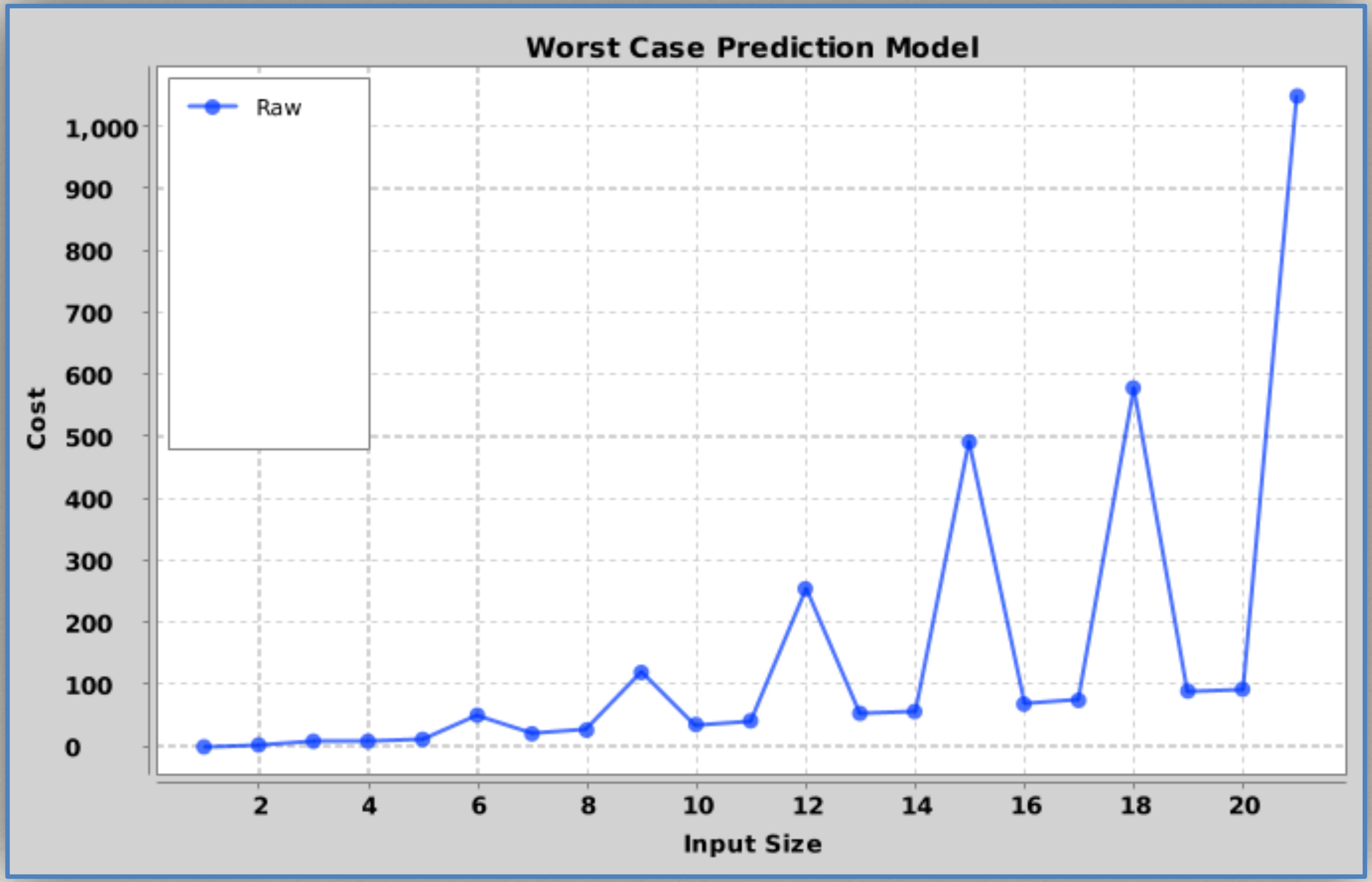


Regression analysis



Case Study: TextCruncher Sort

- Text processing application with various filters, e.g. *WordCount, NGramScore*
- Found vulnerability in sorting algorithm
- Triggered by files with 3 x n different words: 6000 words: 5 min; 6001 words: few secs.



From DARPA STAC

Vulnerability: exponential for lists of length n × 3

Probabilistic Reasoning

- Extension of symbolic execution with probabilistic reasoning [ICSE'13,PLDI'14]
  - Computes the probability of a target event, under an input distribution
- Model counting over symbolic constraints
  - Latte, Barvinok -- integer linear constraints, finite domain

Probabilistic Reasoning

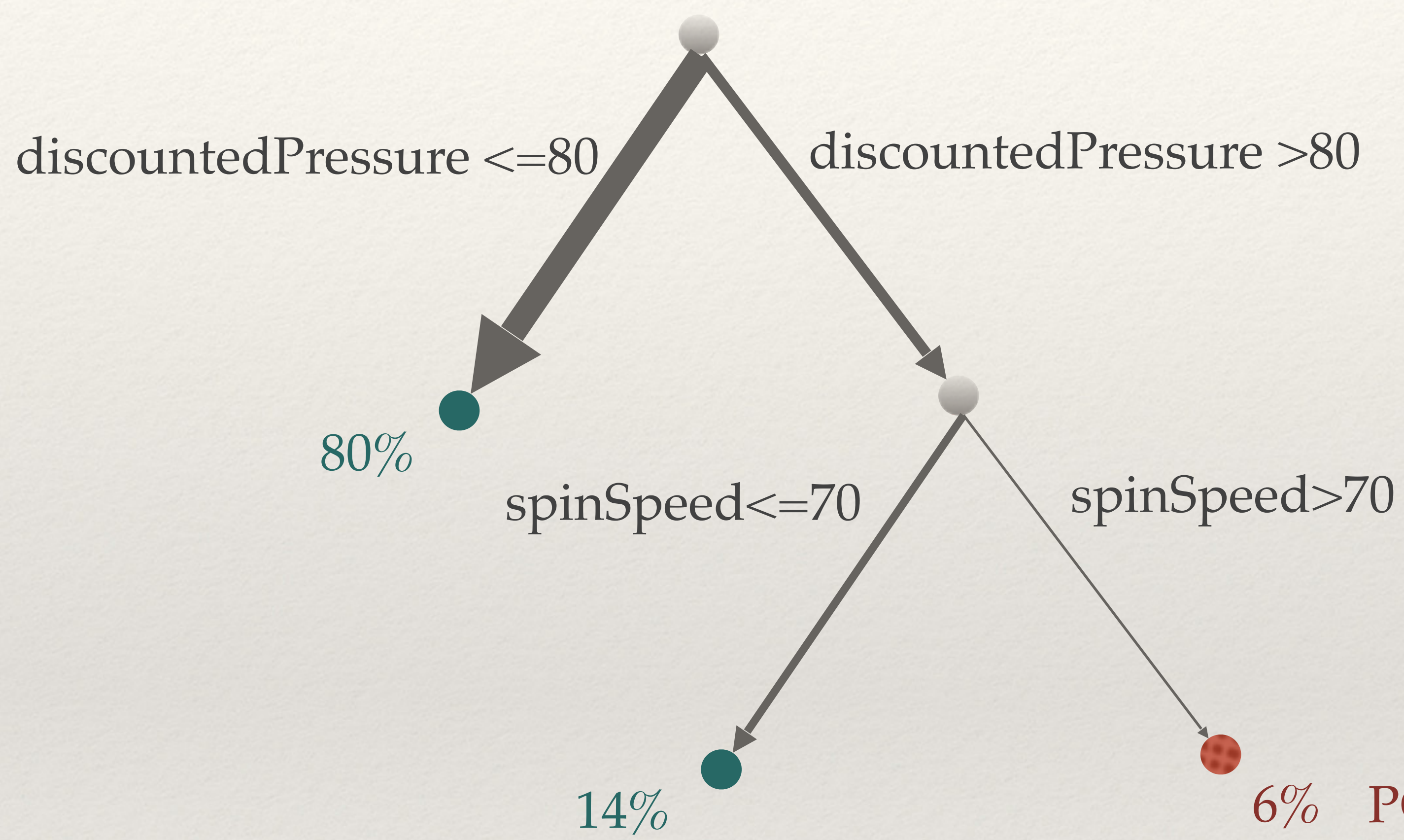
- E.g. assuming uniform distribution,
  - Compute path conditions that lead to target event
  - Count the number of input values that satisfy the corresponding path conditions
  - Divide it by the size of the input domain (#D)

Probability of event e (PC_i leads to e):

$$p(e) = \frac{1}{\#D} \sum \#PC_i$$

Example

input domain 100 x 100



PC: spinSpeed>70 & discountedPressure >80

Pr(Fail) = #(PC)/D

= #(spinSpeed>70 & discountedPressure >80)/D

= 30 x 20/10000 = 6%

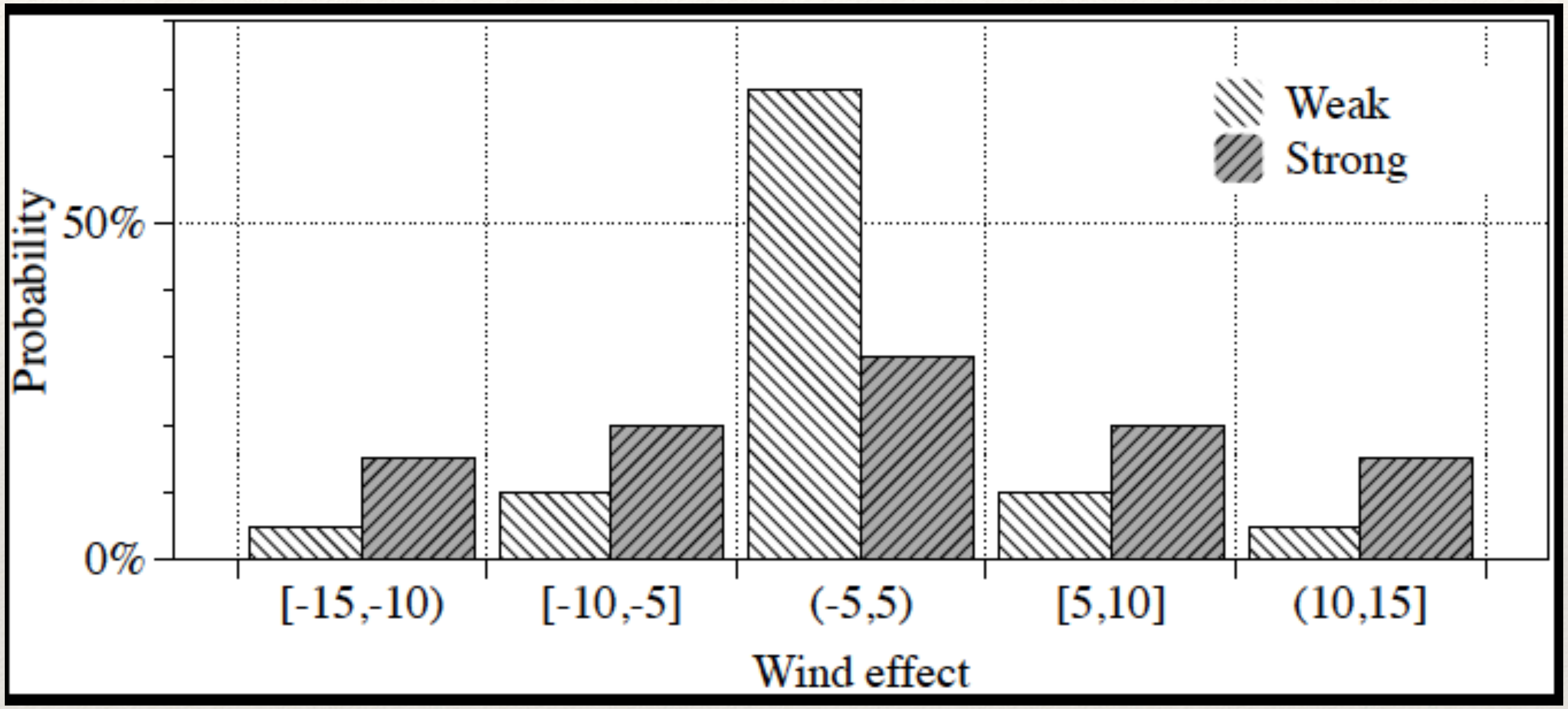
Software Reliability

- Probability of successful termination under stochastic environment assumptions

- Perform **bounded** symbolic execution: results in three sets of paths
  - Success (PCs): lead to successful termination
  - Fail (PCs): lead to failure
  - Grey (PCs): "don't know"
- For given usage profile UP: Pr(Fail | UP) = Pr(PCs | UP), e.g. for uniform UP:
  - Pr(Fail)= #(PC)/D= #(spinSpeed>70 & discountedPressure >80)/D= 30 x 20/10000=6 %.
- Pr(Success) and Pr(Grey) are computed similarly

- Pr(Fail)+Pr(Success)+Pr(Grey)=1
- Reliability = Pr(Success)
- **Confidence = 1 - Pr(Grey)** ("1" means that analysis is complete)

Usage Profiles



- Arbitrary UPs – handled through discretization
- UPs can be seen as "pre-conditions"
- Continuous input distributions [FSE'15]

Computing with usage profiles

- Usage profile: set of pairs <c_i, p_i>
- c_i — usage scenario, constraint on inputs
- p_i — probability that the input is in c_i

$$Rel = Pr^s(P) = \sum_i Pr(PC_i^s \mid UP) =$$

$$= \sum_i \sum_j Pr(PC_i^s \mid c_j) \cdot p_j = \sum_i \sum_j \frac{\sharp(PC_i^s \wedge c_j)}{\sharp(c_j)} \cdot p_j$$

Model Counting

- Latte, Barvinok -- integer linear constraints, finite domain — Polynomial in number of variables and constraints
  - Omega Lib used for algebraic simplifications
  - Optimizations: independence, caching
- Research on
  - model counting for data structures [SPIN'15],
  - strings [FSE'16] — ABC Solver (UC Santa Barbara)
  - non-linear constraints [NFM'17]

Model Counting for Data Structures

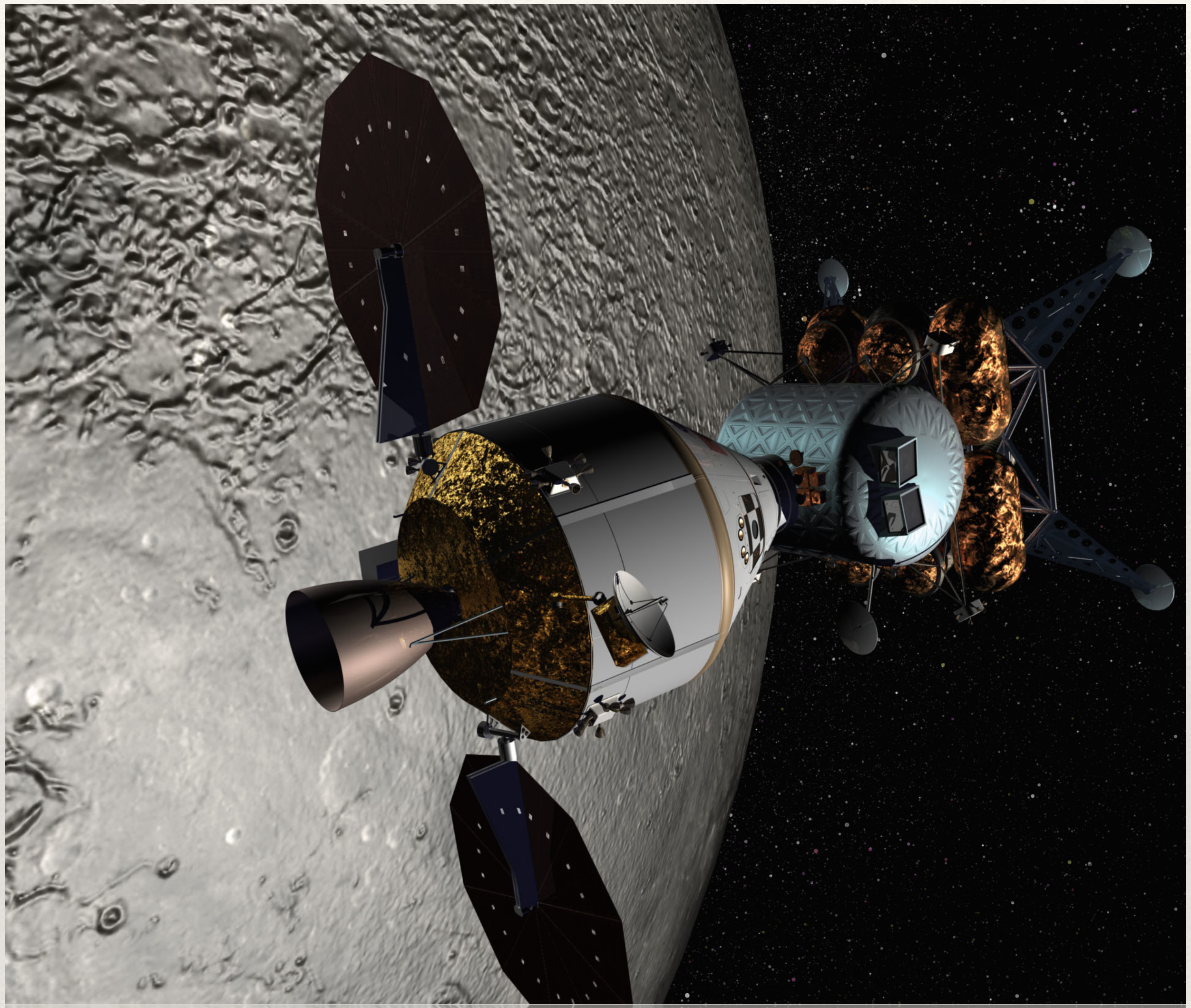
- SPF performs lazy initialization
- Computes Heap PC
- Explicit enumeration using Korat (MIT)
- Arbitrary complex predicates
  - E.g. “acyclic lists of integers with size smaller than the largest contained value”

Multi-threading

- Enumerate all possible schedules (using model checking, partial order reduction)
  - Compute best/worst “reliability”
  - Report best/worst schedule
  - Useful for debugging
- Tree-like schedules
  - Monte-Carlo sampling of symbolic paths
  - Reinforcement learning to iteratively compute schedules
  - Usage profiles summarize hundreds of hours of operation/simulation

Application: Onboard Abort Executive

- NASA control software
  - Mission aborts
  - 3754 paths, 36 input sensors
  - 30 usage scenarios
  - Execution time: 20.5 sec
  - Checking for “no aborts”
  - Rel > 0.9999999



Beyond Finite Domains

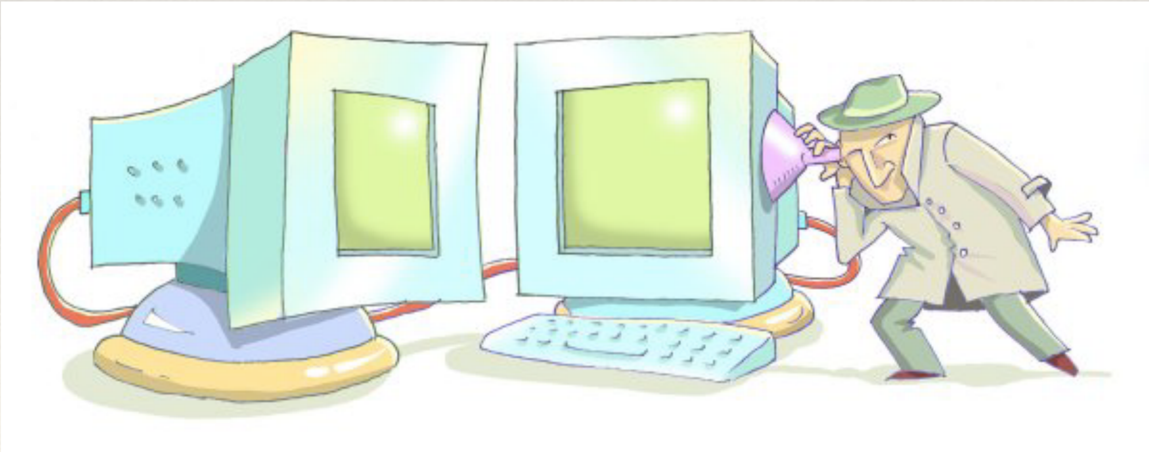
- Probabilistic symbolic execution
  - Arbitrary constraints
  - Continuous input distributions
  - Unbounded domains
  - “Iterative Distribution-Aware Sampling for Probabilistic Symbolic Execution” — Mateus Borges, Antonio Filieri, Marcelo D’Amorim, Corina S. Păsăreanu, ESEC/FSE 2015

Side-Channel Analysis

- Side-channel attacks
  - recover secret inputs to programs from non-functional characteristics of computations
  - time or power consumption, number of memory accesses or size of output files

```
boolean verifyPassword(byte [] input,
                       byte [] password) {
  for ( int i = 0; i < SIZE; i++) {
   if (password[ i ] != input[ i ])
     return false ;
   Thread.sleep(25L);
  }
  return true;
}
```

low
high



- An attack on “main” channel: exponential
- On “side channel”: linear

Side-Channel Analysis

- Non-interference — too strict
- Quantitative Information-Flow Analysis (QIF) to determine information leakage

- Perform symbolic execution (high and low symbolic)
- Collect all symbolic paths — each path leads to an observable

- Side channels produce a set of “observables” that partition the secret
- *Cost model* for observables: execution time, number of packets sent/received over network, etc.

$$\mathcal{O} = \{o_1, o_2, ...o_m\},$$

Quantifying Information Leakage

Channel Capacity

$$CC(P) = log_2(|\mathcal{O}|)$$

Shannon Entropy

$$\mathcal{H}(P) = -\sum_{i=1,m} p(o_i)\log_2(p(o_i))$$

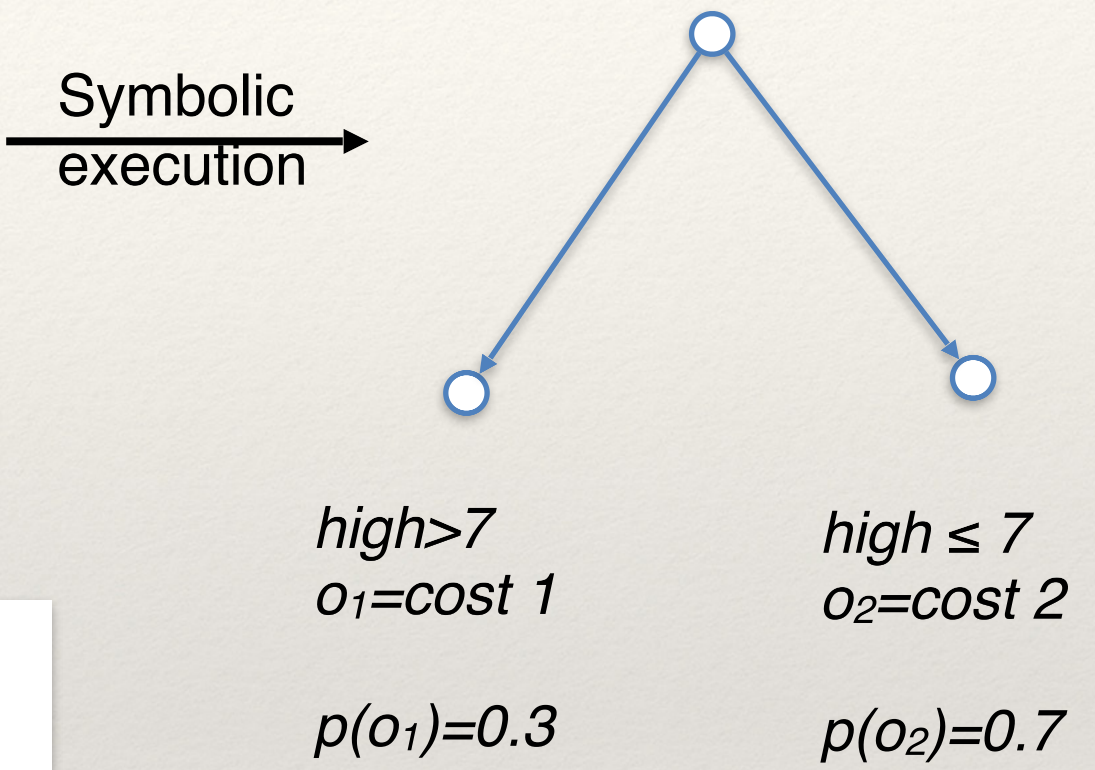
Computing Shannon Entropy

$$\mathcal{H}(P) = -\sum_{i=1,m} p(o_i)\log_2(p(o_i))$$

- Use symbolic execution and model counting [CSF'16,FSE'16,CSF'17]

Example

```
//“high” range: 1..10
if( high > 7 )
    ... cost = 1;
else
    ... cost = 2;
```



Channel capacity:
$\log_2(2)=1$ bit

Shannon Entropy:
$-0.3 \log_2(0.3) -0.7 \log_2(0.7)=$
$0.3 * 1.736966 + 0.7 * 0.514573=$
0.8812909 bits

Password Example

```
// 4-bit input and password; D=256
boolean verifyPassword(byte [] input,
                       byte [] password){
  for(int i = 0; i < SIZE; i++){
   if (password[i]!=input[i])
     return false ;
   Thread.sleep(25L);
  }
  return true;
}
```

```
// 4-bit input and password; D=256
boolean verifyPassword(byte [] input,
                       byte [] password){
  boolean matched=true;
  for(int i = 0; i < SIZE; i++){
   if (password[i]!=input[i])
     matched=false ;
   else
     matched=matched;
   Thread.sleep(25L);
  } return matched; }
```

Corrected!

- 5 paths
  - *h[0]!=l[0]* returns false: *128 values*
  - *h[0]=l[0] & h[1]!=l[1]* returns false: *64 values*
  - *h[0]=l[0] & h[1]=l[1] & h[2]!=l[2]* returns false: *32 values*
  - *h[0]=l[0] & h[1]=l[1] & h[2]=l[2] & h[3]!=l[3]* returns false: *16 values*
  - *h[0]=l[0] & h[1]=l[1] & h[2]=l[2] & h[3]=l[3]* returns true: *16 values*

Observable is time: *H=1.875*
Observable is output: *H=0.33729*

Maximizing Leakage

```
void example(int lo, int hi) {
 if(lo<0){
  if(hi<0) cost=1;
  else if(hi<5) cost=2;
  else cost=3;
 }
 else {
  if(hi>1) cost=4;
  else cost=5;
 }
}
```

- using symbolic low value over-approximates leakage
- example: 5 possible observables; **lo<0: 3 observables**, lo≥0: 2 observables

- Goal: find low input that maximizes number of observables (channel capacity)
- Shows most powerful "attack" in one step
- Shows most vulnerable program behavior

Maximizing Leakage using MaxSMT

```
void example(int lo, int hi) {
 if(lo<0){
  if(hi<0) cost=1;
  else if(hi<5) cost=2;
  else cost=3;
 }
 else {
  if(hi>1) cost=4;
  else cost=5;
 }
}
```

$C_1 :: (l < 0 \land h_1 < 0)$
$C_2 :: (l < 0 \land h_2 \geq 0 \land h_2 < 5)$
$C_3 :: (l < 0 \land h_3 \geq 5)$
$C_4 :: (l \geq 0 \land h_4 > 1)$
$C_5 :: (l \geq 0 \land h_5 \leq 1)$

MaxSMT solution: *Lo=-1* satisfies first 3 clauses

Leakage $\log_2(3)$=1.58 bits

- MaxSMT solving — generalization of SMT to optimization
  - given a set of weighted clauses
  - find solution that maximizes the sum of the weights of the satisfied clauses
- Assemble PCs that lead to same observable into "**clauses**" of weight "1"
- MaxSMT solution gives maximal assignment ⇒ largest number of observables
- Any other assignments lead to fewer observables

Multi-run Analysis

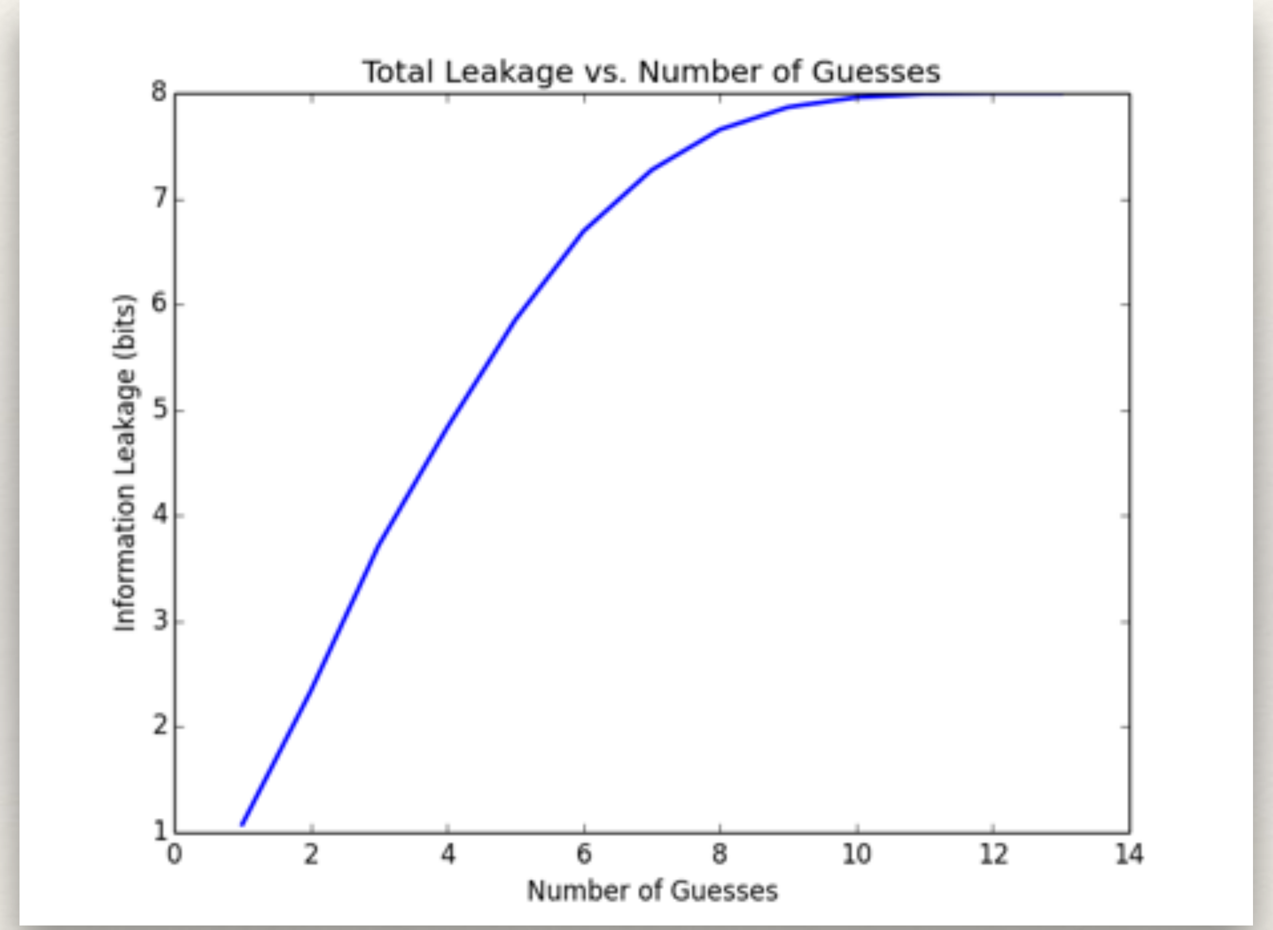
- The attacker learns the secret by observing multiple program runs
- Generalization to multiple-run side-channel analysis

$$P(h, l_1); P(h, l_2); ..P(h, l_k)$$

- An "observable" is a **sequence** of costs
- MaxSMT used to synthesize a sequence of public inputs that maximize leakage; non-adaptive attacks; greedy approach [CSF'16]
- Maximize Shannon leakage: parameterized model counting+ numerical optimization; adaptive attacks [CSF'17]
- Analysis of password examples and cryptographic functions
- Shown experimentally to perform better than previous approaches based on self composition or brute-force enumeration
- More work on side-channel analysis [ISSTA'18]

Results for Password Check

Results for 4 elements with 4 values (8 bits of information)



Timing Side Channel

Symbolic Execution and Fuzzing

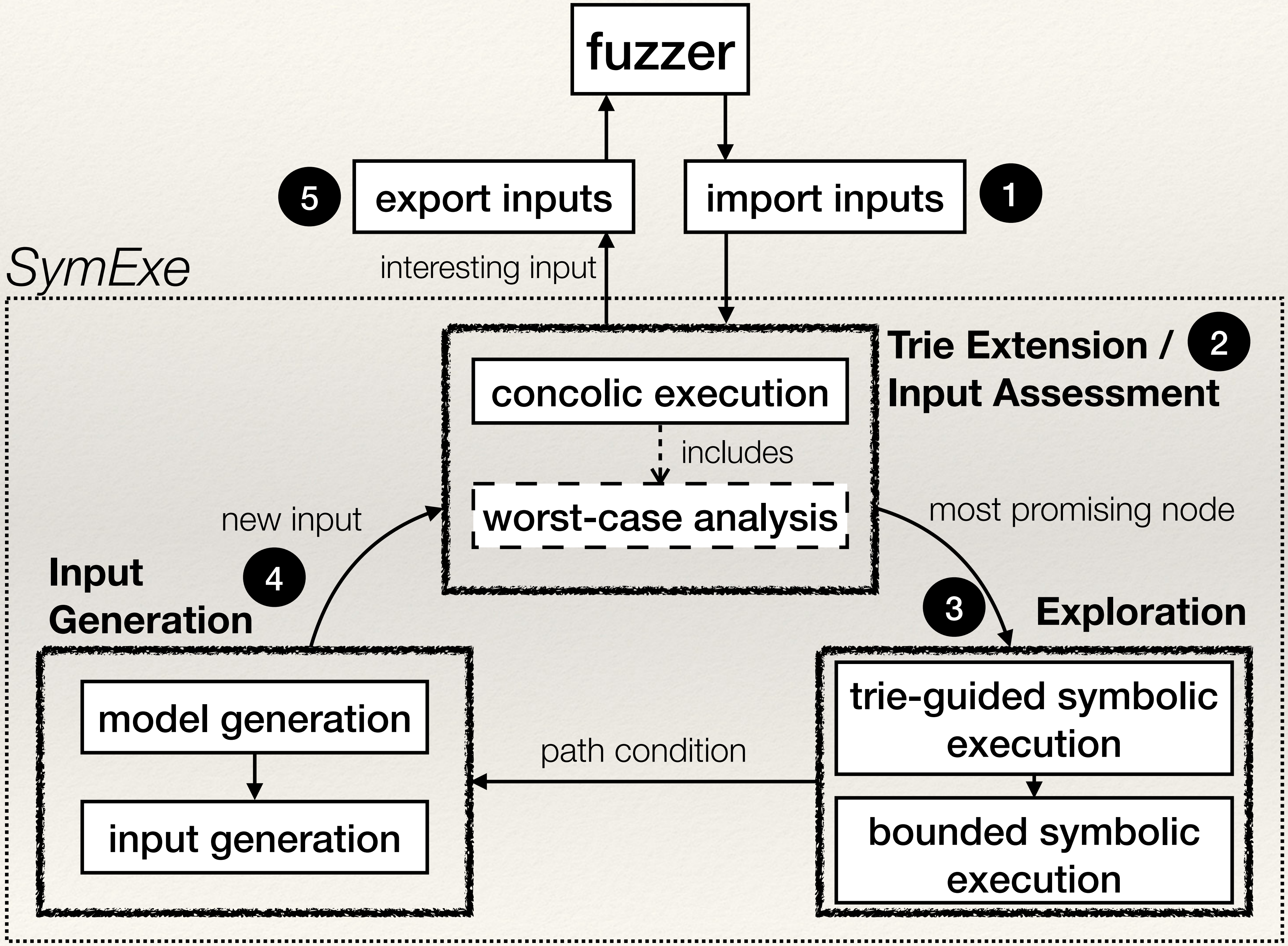
- Fuzzing: random testing with some fuzzing
  - Cheap
  - Not good at finding "deep paths" that depend on complicated constraints
- Symbolic execution
  - Expensive
  - Good at finding "deep paths"
- Better Together!

Symbolic Execution and Fuzzing

- Kelinci [CCS’17] — AFL-based fuzzing for Java
- Badger: Complexity Analysis with Fuzzing and Symbolic Execution [ISSTA’18]
- DifFuzz: differential fuzzing for side-channel analysis [ICSE’19]
- HyDiff: hybrid differential software analysis [ICSE’20]
- Fuzzing, Symbolic Execution, and Expert Guidance for Better Testing. [IEEE Software 2024]

Kelinci means rabbit in Indonesian, the language spoken on the Java island

Badger


fuzzer
5
export inputs
import inputs
1
SymExe
interesting input
Trie Extension /
2
Input Assessment
concolic execution
includes
worst-case analysis
new input
most promising node
Input
Generation
4
3
Exploration
model generation
input generation
path condition
trie-guided symbolic
execution
bounded symbolic
execution

Current and Future Work

- Neural network analysis —
  - NEUROSPF: A tool for the Symbolic Analysis of Neural Networks (ICSE'21, FoMLAS'21)
  - Probabilistic Analysis of Neural Networks (SEAMS'20, ISSRE '20)
  - NNRepair: Constraint-based Repair of Neural Network Classifiers (CAV'21)
- Using LLMs to *generalize* Symbolic PathFinder's results
- Side-channel analysis — new AWS small project

Thank you

Contact information: corina.s.pasareanu@nasa.gov