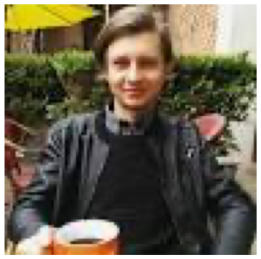
Software Complexity, Path Complexity, and Branch Selectivity

Tevfik Bultan

Verification Lab (VLab), Computer Science Department

University of California, Santa Barbara

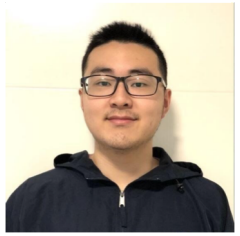
VLab Collaborators



Lucas Bang, Harvey Mudd

Seemanta Saha, Intel

Laboni Sarker, UCSB



Chaofan Shou, UC Berkeley

Abdulbaki Aydin, Meta

William Eiers
Stevens Institute of Technology

Mara Downing, UCSB

Ganesh Sankaran, Amazon

Tegan Brennan
Stevens Institute of Technology

MD Shafiuzzaman, UCSB

Albert Li, Oracle

Outline

- **Motivation**
- Software complexity
- Path complexity
- Branch selectivity
- Future directions

Software engineering is 56 years old!

- In 1968 a seminal NATO Conference was held in Garmisch, Germany

Attendees of the 1968 Garmisch conference

Margaret Hamilton
Director of Software Development for the NASA's Apollo mission

Purpose: to look for a solution to ***software crisis***

–50 distinguished computer scientists, programmers and industry leaders got together to ***look for a solution to the difficulties in building large software systems***

–Considered to be the birth of "***software engineering***" as a research area

What was software crisis?

Large software systems often:

- Do not provide the desired functionality
- Take too long to build
- Cost too much to build
- Require too much resources (time, space) to run
- Cannot evolve to meet changing needs

Software engineering as a remedy: *a systematic, disciplined, quantifiable approach to the production and maintenance of software.*

Software's chronic crisis

- A quarter century (1994) after the Garmisch conference, an article in Scientific American declared:

Software's Chronic Crisis

TRENDS IN COMPUTING by W. Wayt Gibbs, staff writer.
Copyright Scientific American; September 1994; Page 86
Despite 50 years of progress, the software industry remains years-perhaps decades-short of the mature engineering discipline needed to meet the demands of an information-age society

Software's chronic crisis

- Another quarter century later:

- This is a photo of the navigation system of my car
  - ***It crashes and reboots while I am driving!***

Software's chronic crisis

We are still looking for a:

- *systematic, disciplined, **quantifiable** approach to the production and maintenance of software*

which ensures:

- *safety, dependability, security, reliability, availability, usability, efficiency, scalability, and maintainability*

of software systems.

Outline

- Motivation
- **Software complexity**
- Path complexity
- Branch selectivity
- Future directions

Software Complexity

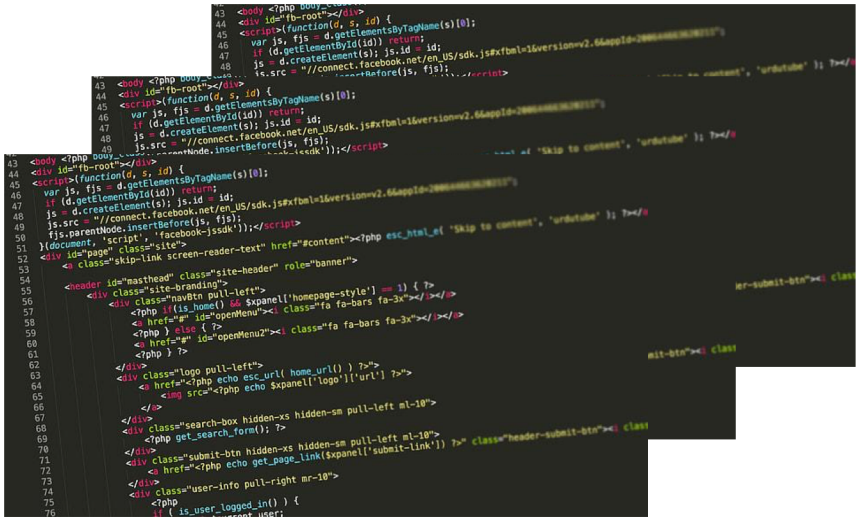
[Kearney et al. 1986]

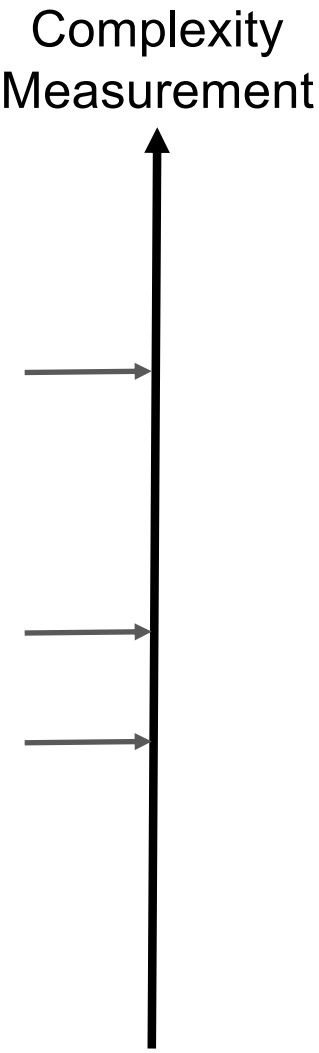
"In recent years much attention has been directed toward reducing software cost. To this end, researchers have attempted to find relationships between the characteristics of programs and the difficulty of performing programming tasks. ***The objective has been to develop measures of software complexity that can be used for cost projection, manpower allocation, and program and programmer evaluation.****"*

- **"Software complexity measurement" Joseph P. Kearney, Robert L. Sedlmeyer, William B. Thompson, Michael A. Gray, Michael A. Adler. Communications of the ACM, Volume 29, Issue 11, pp 1044–1050, 1986**

Software Complexity Measurement

Software





Software Complexity: What can we do with it?

Can we use software complexity measurements to predict/assess things like:

- bug-proneness
- defects
- testing effort
- verification effort
- program comprehension effort
- maintenance effort
- …

Note: This is not algorithmic complexity analysis

Software Complexity: Cyclomatic (McCabe) Complexity

Cyclomatic complexity:

$$M = E - N + 2P,$$

where

- E = the number of edges of the graph.
- N = the number of nodes of the graph.
- P = the number of connected components.

McCabe's 2008 presentation to the USA Department of Homeland Security:

Cyclomatic Complexity & Reliability Risk:

- **1 – 10 Simple procedure, little risk**
- **11- 20 More Complex, moderate risk**
- **21 – 50 Complex , high risk**
- **>50 Untestable, VERY HIGH RISK**

- **Wikipedia, "Cyclomatic Complexity"**
- **Thomas J. McCabe: "A Complexity Measure." IEEE Trans. Software Eng. 2(4): 308-320 (1976)**

Software Complexity: Halstead Complexity Measures

For a given problem, let:

- η_1 = the number of distinct operators
- η_2 = the number of distinct operands
- N_1 = the total number of operators
- N_2 = the total number of operands

From these numbers, several measures can be calculated:

- Program vocabulary: $\eta = \eta_1 + \eta_2$
- Program length: $N = N_1 + N_2$
- Calculated estimated program length: $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$
- Volume: $V = N \times \log_2 \eta$
- Difficulty : $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$
- Effort: $E = D \times V$

The difficulty measure is related to the difficulty of the program to write or understand,

The effort measure translates into actual coding time using the following relation,

- Time required to program: $T = \frac{E}{18}$ seconds

Halstead's delivered bugs (B) is an estimate for the number of errors in the implementation.

- Number of delivered bugs : $B = \frac{E^{\frac{2}{3}}}{3000}$ or, more recently, $B = \frac{V}{3000}$ is accepted.[1]

```
main()
{
  int a, b, c, avg;
  scanf("%d %d %d", &a, &b, &c);
  avg = (a+b+c)/3;
  printf("avg = %d", avg);
}
```

The distinct operators (η_1) are: `main` , `()` , `{}` , `int` , `scanf` , `&` , `=` , `+` , `/` , `printf` , `,` , `;`

The distinct operands (η_2) are: `a` , `b` , `c` , `avg` , `"%d %d %d"` , `3` , `"avg = %d"`

- $\eta_1 = 12, \eta_2 = 7, \eta = 19$
- $N_1 = 27, N_2 = 15, N = 42$
- Calculated Estimated Program Length: $\hat{N} = 12 \times log_2 12 + 7 \times log_2 7 = 62.67$
- Volume: $V = 42 \times log_2 19 = 178.4$
- Difficulty: $D = \frac{12}{2} \times \frac{15}{7} = 12.85$
- Effort: $E = 12.85 \times 178.4 = 2292.44$
- Time required to program: $T = \frac{2292.44}{18} = 127.357$ seconds
- Number of delivered bugs: $B = \frac{2292.44^{\frac{2}{3}}}{3000} = 0.05$

- **Wikipedia, “Halstead complexity measures”**
- **M. H. Halstead, “Elements of Software Science.” New York: Elsevier North-Holland, 1977.**

Cyclomatic (McCabe) complexity & Halstead complexity

- These complexity measures have been around ~50 years

- They seem oversimplified and coarse

Can we do better?

Outline

- Motivation
- Software complexity
- **Path complexity**
- Branch selectivity
- Future directions

Path Complexity

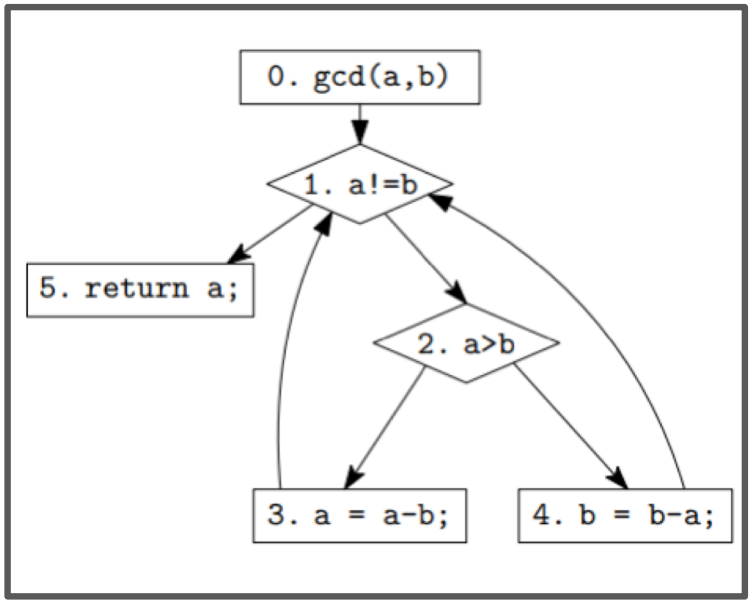
- Upper bound on number of paths from start to exit up to a given depth.
  - Depth is defined to be the length of path

Path Complexity

- Upper bound on number of paths from start to exit up to a given depth.
  - Depth is defined to be the length of path
- paths(up to depth 1) = 0

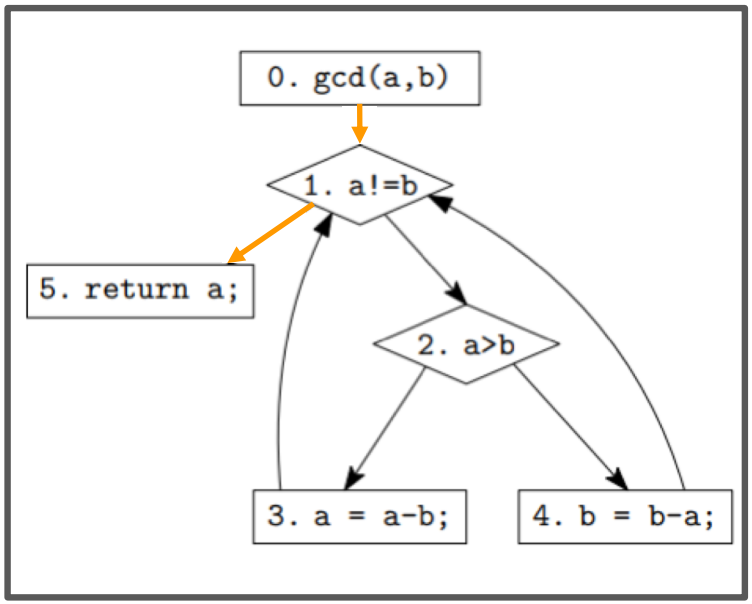
n	1
count(n)	0
path(n)	0

Control Flow Graph (CFG)



Path Complexity

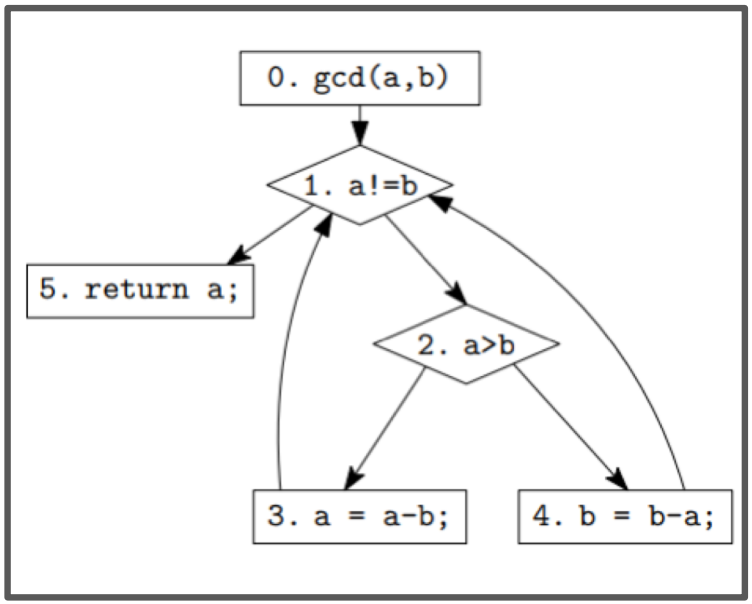
- Upper bound on number of paths from start to exit up to a given depth.
  - Depth is defined to be the length of path
- paths(up to depth 1) = 0
- paths(up to depth 2) = 1



n	1	2
count(n)	0	1
path(n)	0	1

Path Complexity

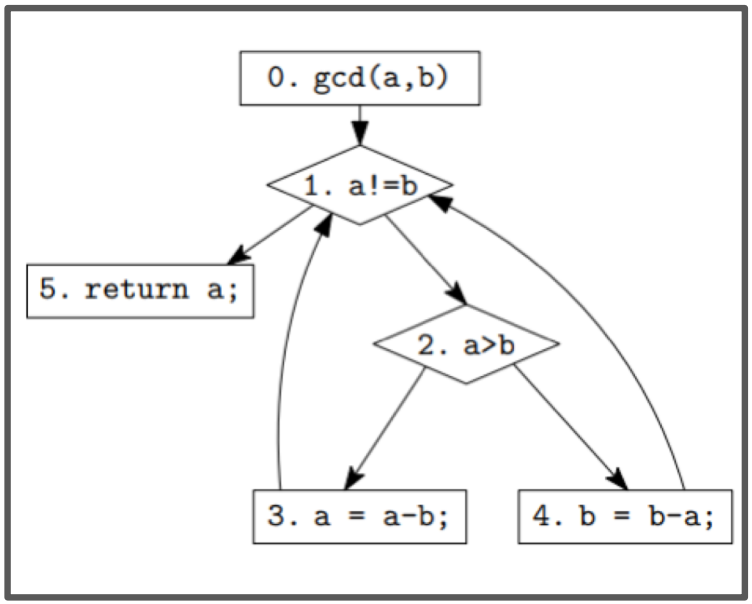
- Upper bound on number of paths from start to exit up to a given depth.
  - Depth is defined to be the length of path
- paths(up to depth 1) = 0
- paths(up to depth 2) = 1
- paths(up to depth 3) = 1



n	1	2	3
count(n)	0	1	0
path(n)	0	1	1

Path Complexity

- Upper bound on number of paths from start to exit up to a given depth.
  - Depth is defined to be the length of path
- paths(up to depth 1) = 0
- paths(up to depth 2) = 1
- paths(up to depth 3) = 1
- paths(up to depth 4) = 1

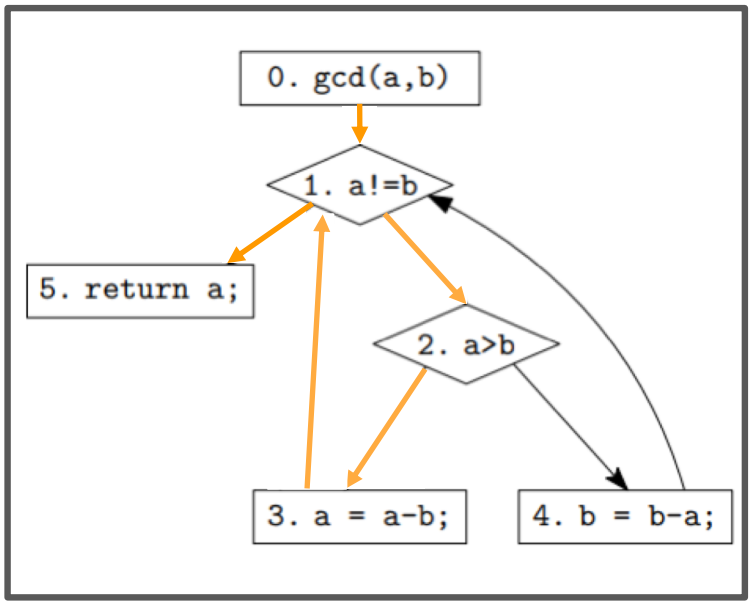


n	1	2	3	4
count(n)	0	1	0	0
path(n)	0	1	1	1

Path Complexity

- Upper bound on number of paths from start to exit up to a given depth.
  - Depth is defined to be the length of path
- paths(up to depth 1) = 0
- paths(up to depth 2) = 1
- paths(up to depth 3) = 1
- paths(up to depth 4) = 1
- paths(up to depth 5) = 3

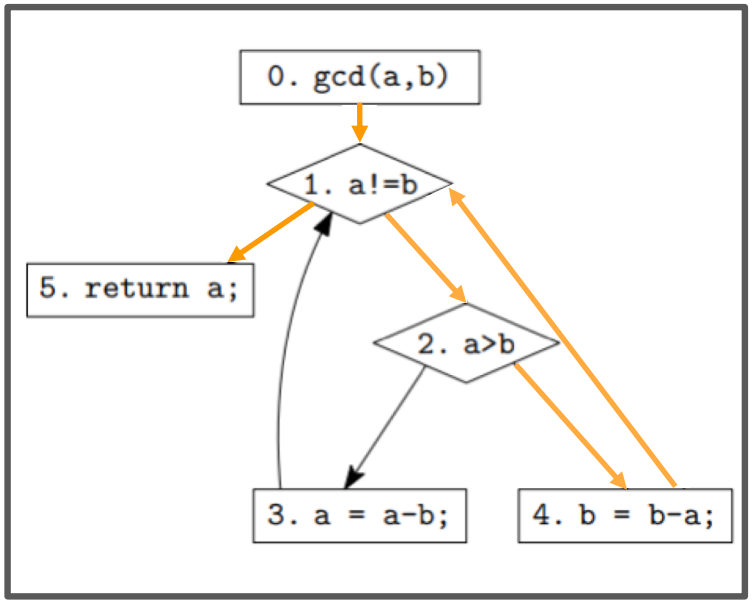
n	1	2	3	4	5
count(n)	0	1	0	0	2
path(n)	0	1	1	1	3



Path Complexity

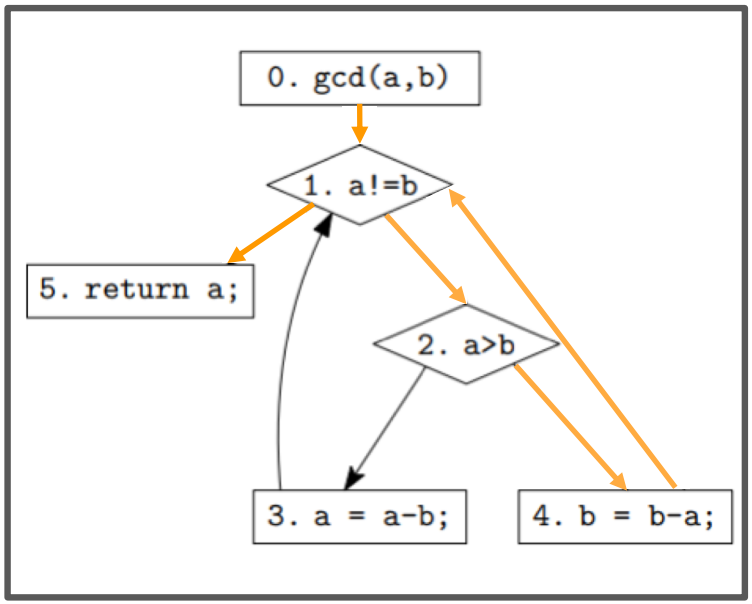
- Upper bound on number of paths from start to exit up to a given depth.
  - Depth is defined to be the length of path
- paths(up to depth 1) = 0
- paths(up to depth 2) = 1
- paths(up to depth 3) = 1
- paths(up to depth 4) = 1
- paths(up to depth 5) = 3

n	1	2	3	4	5
count(n)	0	1	0	0	2
path(n)	0	1	1	1	3



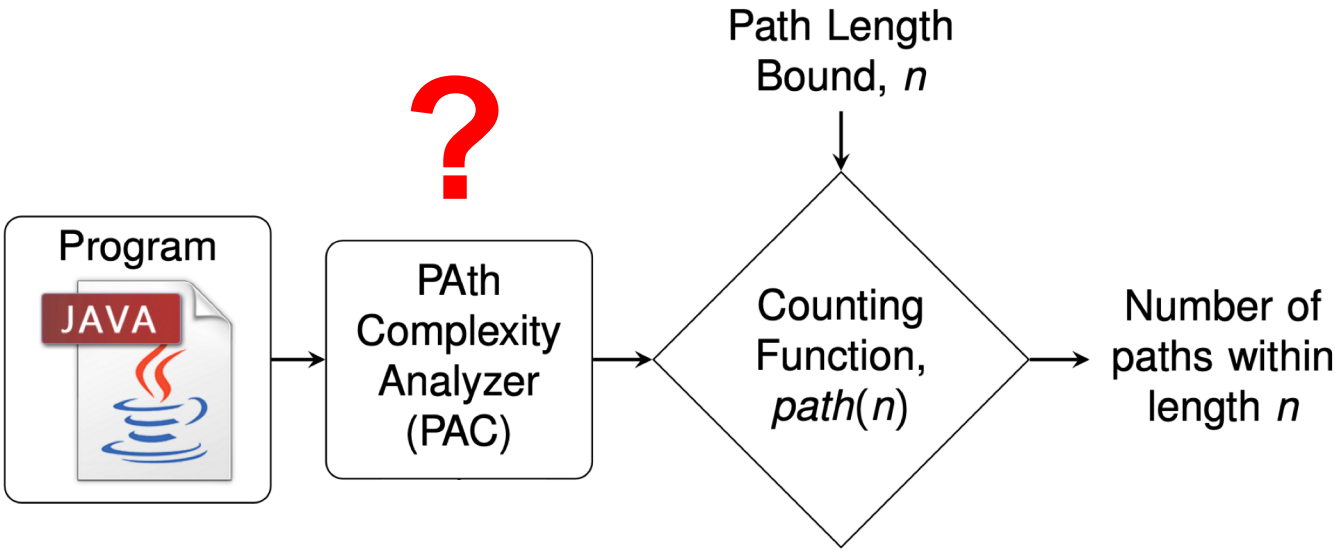
Path Complexity

- Upper bound on number of paths from start to exit up to a given depth.
  - Depth is defined to be the length of path
- paths(up to depth 1) = 0
- paths(up to depth 2) = 1
- paths(up to depth 3) = 1
- paths(up to depth 4) = 1
- paths(up to depth 5) = 3



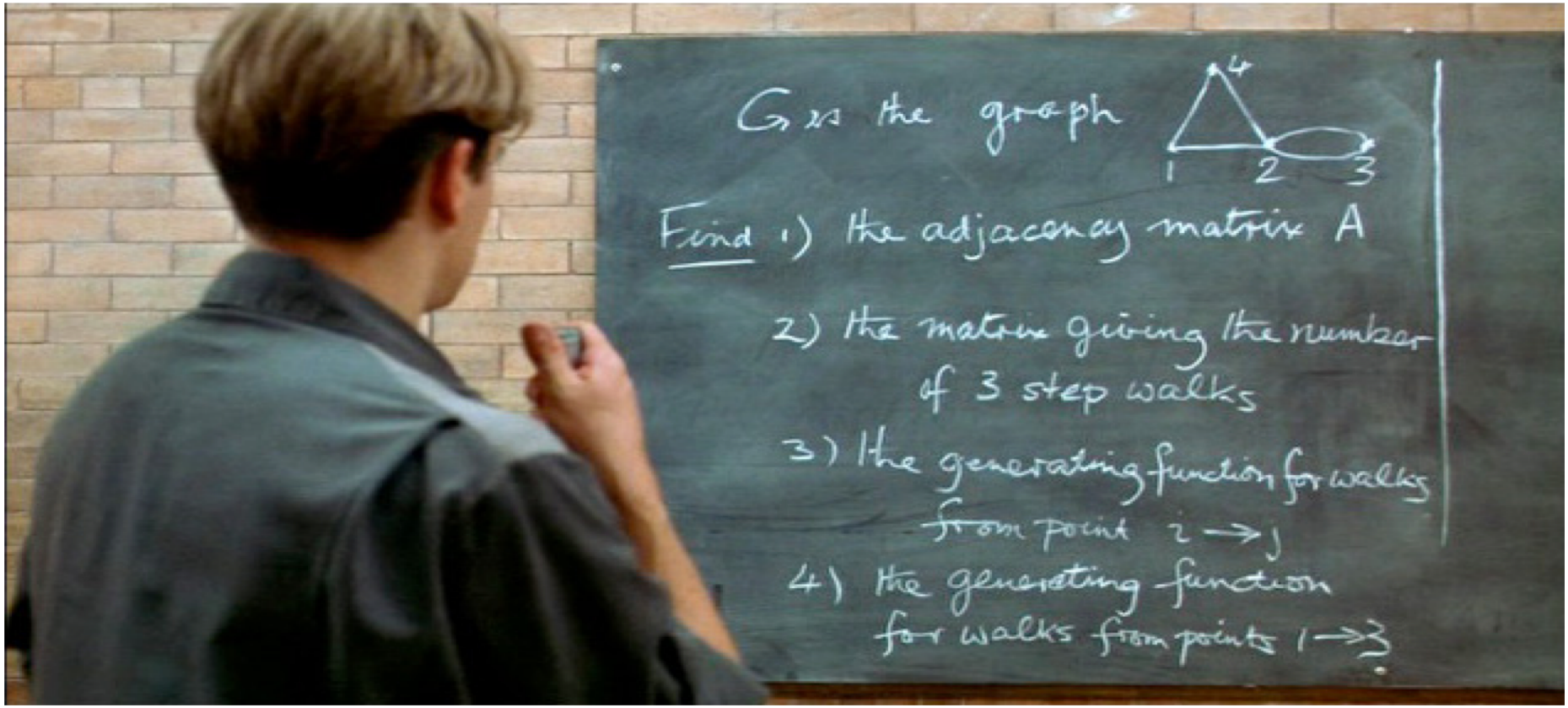
n	1	2	3	4	5	6	7	8	…
count(n)	0	1	0	0	2	0	0	4	…
path(n)	0	1	1	1	3	3	3	7	…

Path Complexity



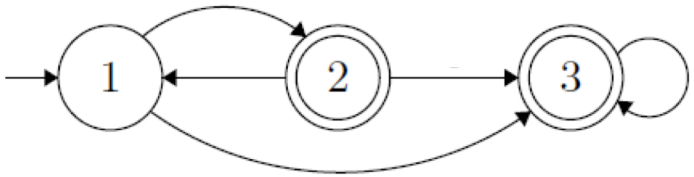
- Lucas Bang, Abdulbaki Aydin, Tevfik Bultan: Automatically computing path complexity of programs. ESEC/SIGSOFT FSE 2015: 61-72

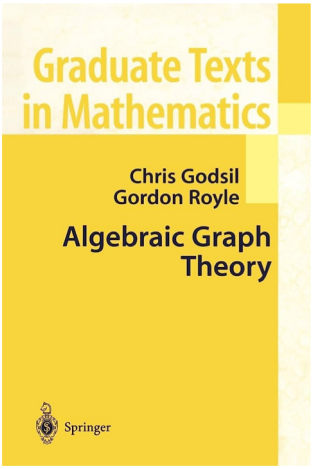
Can you count the paths Will Hunting?



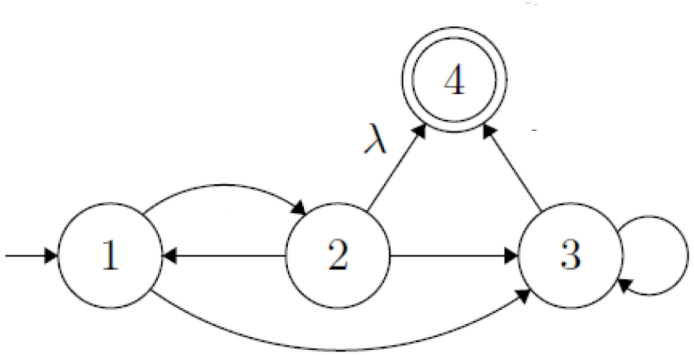
Path Counting

Control Flow Graph (CFG)





Path Counting via Matrix Exponentiation



T: adjacency matrix for the automaton

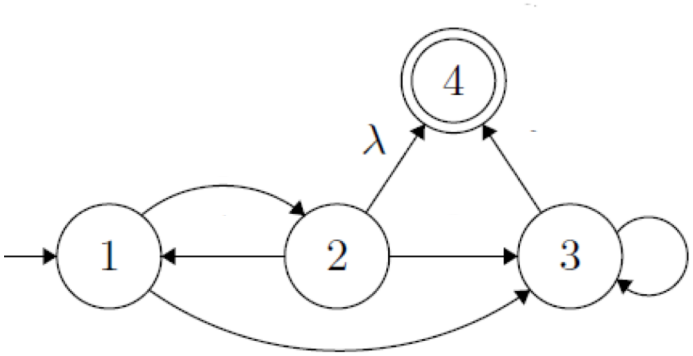
(i,j): number of edges from i to j

$$T = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, T^2 = \begin{bmatrix} 1 & 0 & 3 & 2 \\ 0 & 1 & 3 & 1 \\ 0 & 0 & 4 & 2 \\ 0 & 0 & 0 & 0 \end{bmatrix}, T^3 = \begin{bmatrix} 0 & 1 & 7 & 3 \\ 1 & 0 & 7 & 4 \\ 0 & 0 & 8 & 4 \\ 0 & 0 & 0 & 0 \end{bmatrix}, T^4 = \begin{bmatrix} 0 & 1 & 15 & 8 \\ 1 & 0 & 15 & 7 \\ 0 & 0 & 16 & 8 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$f(0) = 0 \qquad f(1) = 2 \qquad f(2) = 3 \qquad f(3) = 8$$

Counting Paths via Generating Functions

- We can compute a generating function, $g(z)$, using the adjacency matrix



$$T = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$g(z) = (-1)^n \frac{\det(I - zT : n+1, 1)}{z \times \det(I - zT)} = \frac{2z - z^2}{1 - 2z - z^2 + 2z^3}$$

Counting Paths via Generating Functions

$$g(z) = \frac{2z - z^2}{1 - 2z - z^2 + 2z^3}$$

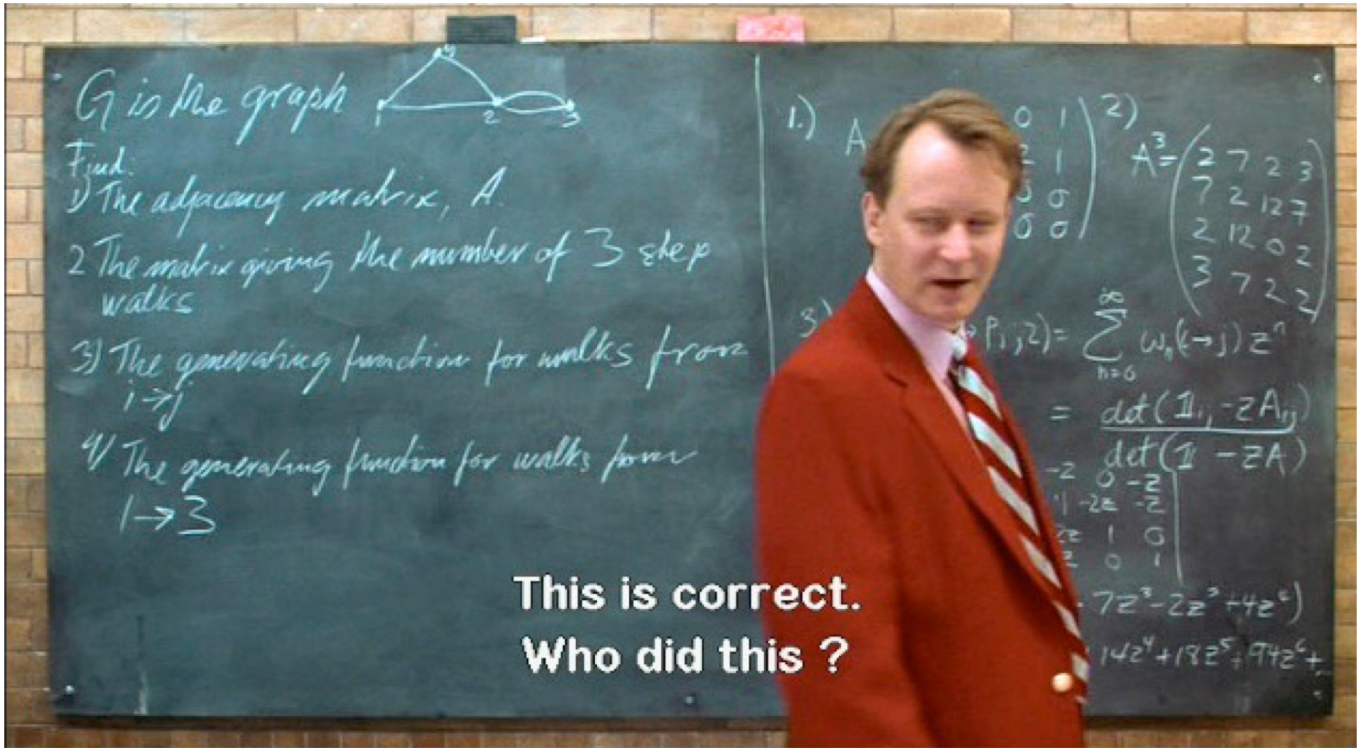
- Each $f(i)$ can be computed by Taylor expansion of $g(z)$

$$g(z) = \frac{g(0)}{0!}z^0 + \frac{g^{(1)}(0)}{1!}z^1 + \frac{g^{(2)}(0)}{2!}z^2 + \cdots + \frac{g^{(n)}(0)}{n!}z^n + \cdots$$

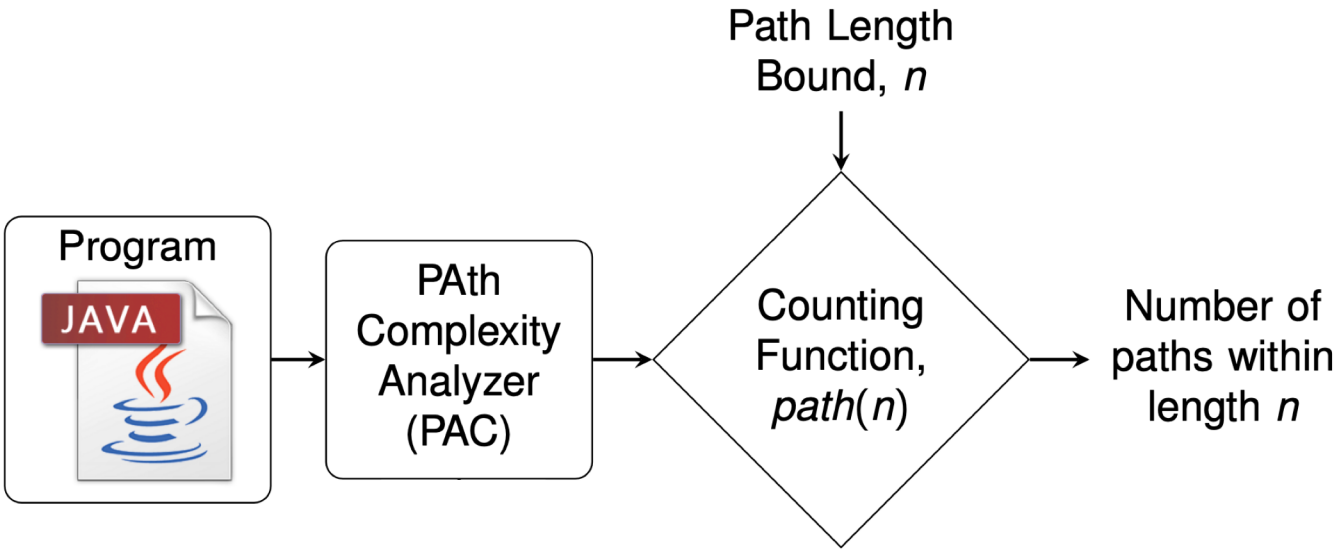
$$g(z) = 0z^0 + 2z^1 + 3z^2 + 8z^3 + 15z^4 + \cdots$$

$$g(z) = f(0)z^0 + f(1)z^1 + f(2)z^2 + f(3)z^3 + f(4)z^4 + \cdots$$

Good job Will Hunting!

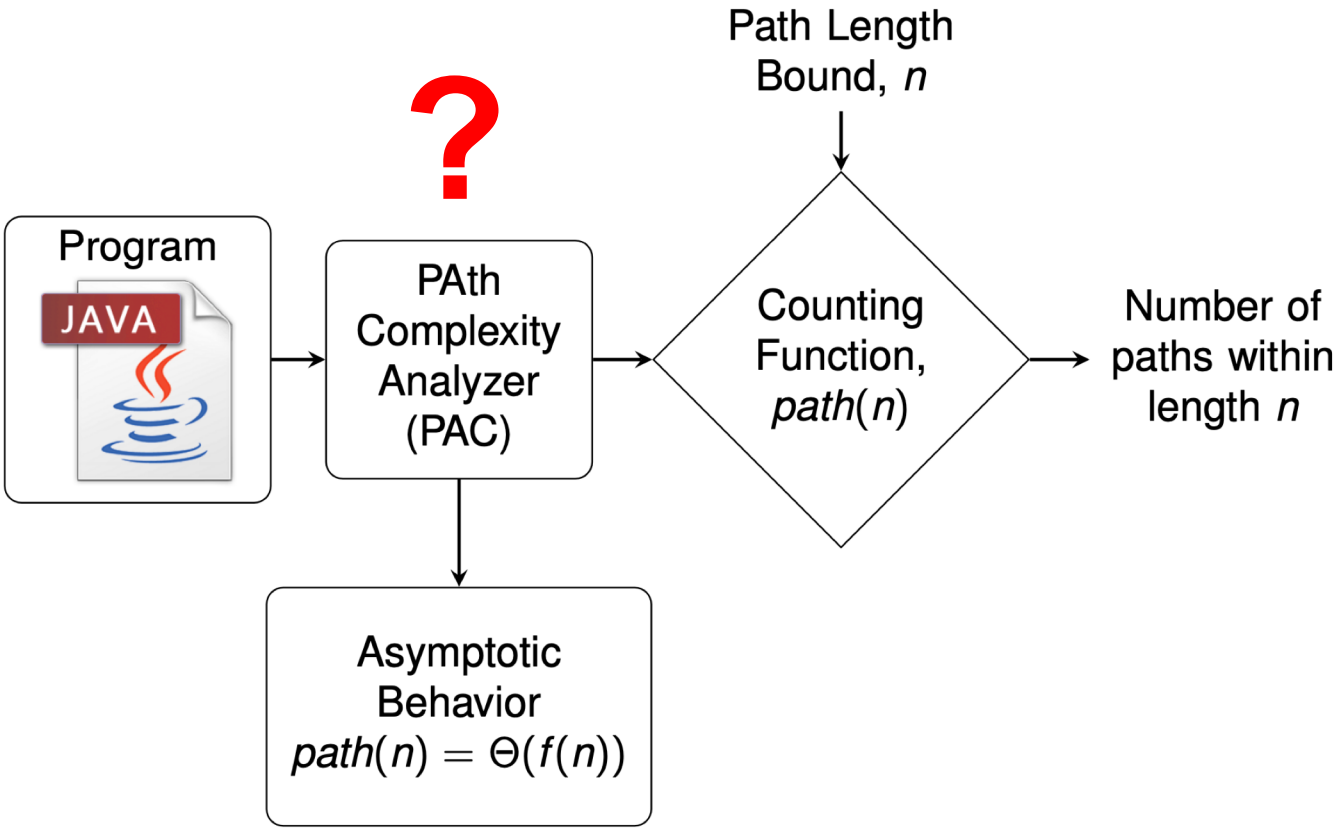


Path Complexity



- **Lucas Bang, Abdulbaki Aydin, Tevfik Bultan: Automatically computing path complexity of programs. ESEC/SIGSOFT FSE 2015: 61-72**

Path Complexity



- **Lucas Bang, Abdulbaki Aydin, Tevfik Bultan: Automatically computing path complexity of programs. ESEC/SIGSOFT FSE 2015: 61-72**

Path Complexity & Asymptotic Path Complexity

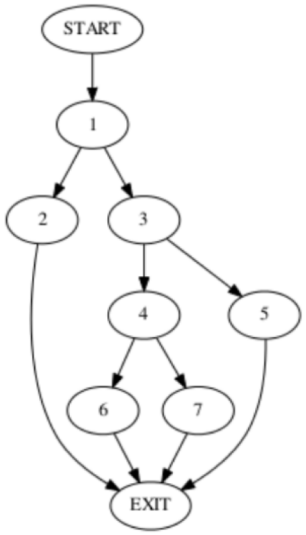
- A closed-form solution for the path counting function can be computed from the generating function
- It involves
  - finding the roots of the denominator of the generating function
  - taking linearly independent combination of exponentiated roots
  - solving for the coefficients

Asymptotic path complexity:

- extract the highest order term using asymptotic analysis

Examples from Java SDK

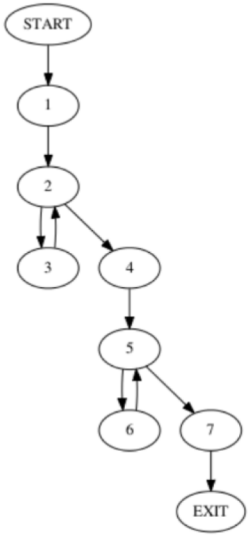
```
private static void rangeCheck(int length,
  int fromIndex, int toIndex) {

  if (fromIndex > toIndex) {
    throw new IllegalArgumentException(
    "fromIndex(" + fromIndex + ") >
     toIndex(" + toIndex + ")");
  }
  if (fromIndex < 0) {
    throw new ArrayIndexOutOfBoundsException(fromIndex);
  }
  if (toIndex > length) {
    throw new ArrayIndexOutOfBoundsException(toIndex);
  }
}
```



- Path Complexity: 4
- Asymptotic: $\Theta(1)$
- Complexity Class: Constant

Examples from Java SDK

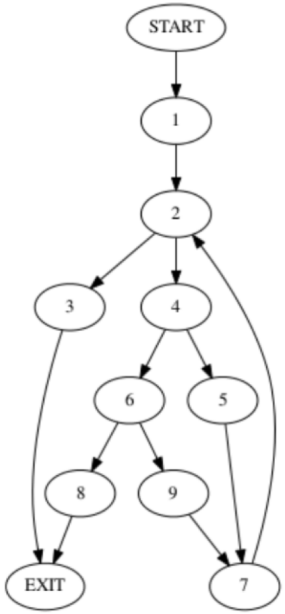
```
public Matcher reset() {
  first = -1;
  last = 0;
  oldLast = -1;
  for(int i=0; i<groups.length; i++)
    groups[i] = -1;
  for(int i=0; i<locals.length; i++)
    locals[i] = -1;
  lastAppendPosition = 0;
  from = 0;
  to = getTextLength();
  return this;
}
```



- Path Complexity: $0.12n^2 + 1.25n + 3$
- Asymptotic: $\Theta(n^2)$
- Complexity Class: Polynomial

Examples from Java SDK

```
private static int binarySearch0(long[] a,
  int fromIndex, int toIndex, long key) {

  int low = fromIndex;
  int high = toIndex - 1;
  while (low <= high) {
    int mid = (low + high) >>> 1;
    long midVal = a[mid];
    if (midVal < key)
      low = mid + 1;
    else if (midVal > key)
      high = mid - 1;
    else
      return mid; // key found
  }
  return -(low + 1); // key not found.
}
```



- Path Complexity: $(6.86)(1.17)^n + (0.22)(1.1)^n + (0.13)(0.84)^n + 2$
- Asymptotic: $\Theta(1.17^n)$
- Complexity Class: Exponential

Comparison with other Complexity Measures

Cyclomatic Complexity: corresponds to the maximum number of linearly independent paths in the CFG

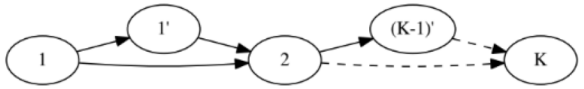
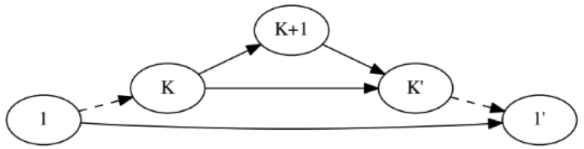
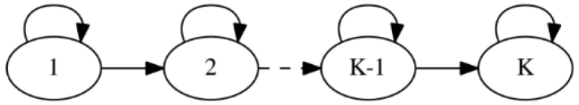
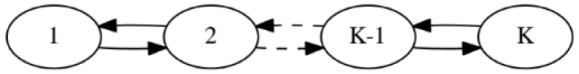
NPATH Complexity: the number of acyclic paths in the CFG

Path Complexity

Comparison with other Complexity Measures

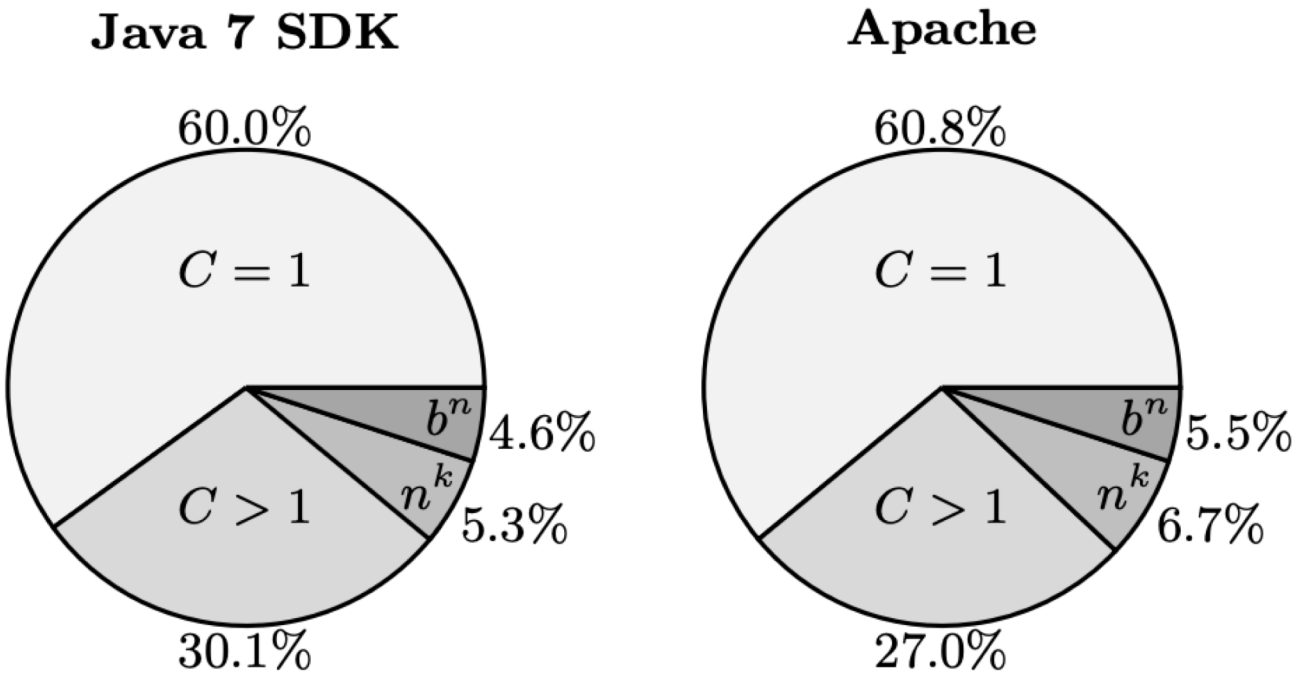
Method	Cyclomatic Complexity	NPATH Complexity	Path Complexity	Asymptotic Complexity
rangeCheck()	4	4	4	$\Theta(1)$
reset()	3	4	$0.12n^2 + 1.25n + 3$	$\Theta(n^2)$
binarySearch0()	4	4	$(6.86)1.17^n + (0.22)1.1^n$ $+(0.13)(0.84)^n + 2$	$\Theta(1.17^n)$

Comparison with other Complexity Measures

Pattern	Control Flow Graph	Cyclomatic Complexity	NPATH Complexity	Asymptotic Complexity
K If-Else in sequence		$K+1$	2^K	2^K
K If-Else nested		$K+1$	$K+1$	$K+1$
K Loops in sequence		$K+1$	2^K	$\Theta(n^K)$
K Loops nested		$K+1$	$K+1$	$\Theta(b^n)$

Asymptotic Path Complexity

Asymptotic path complexity of methods in Java SDK and Apache libraries



Path Complexity for Recursive Functions

Path complexity analysis we have discussed so far is intra-procedural

It only counts the paths within one method

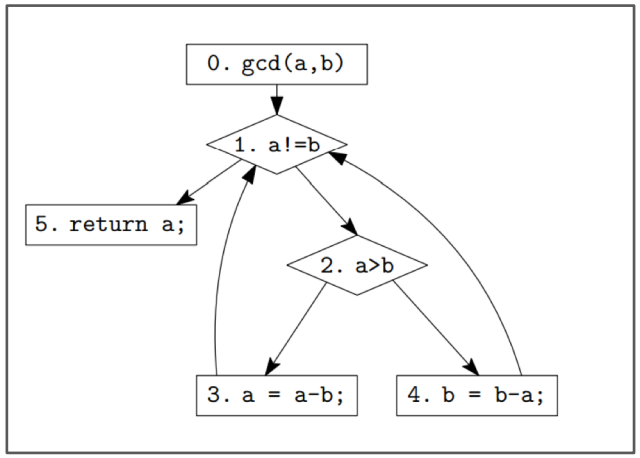
How about procedure calls, recursion?

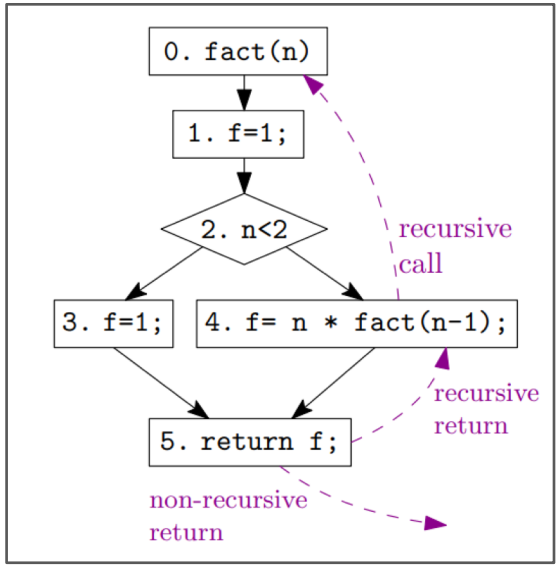
Can we count paths in recursive functions?

Eli Pregerson, Shaheen Cullen-Baratloo, David Chen, Duy Lam, Max Szostak, Lucas Bang: "Formalizing Path Explosion for Recursive Functions via Asymptotic Path Complexity." FormaliSE 2023: 76-85

Path Complexity for Recursive Functions

- Control flow graphs support a *regular language* of paths (DFA)
  - Do not match function call to function return
- Paths in recursive functions are *not regular*
  - Need a **stack** to match function calls and returns (PDA)
- *Context free grammars* work



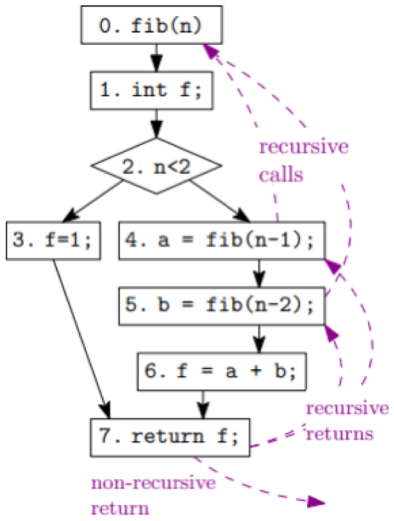


Path Complexity for Recursive Functions

Program

```
0. int fib(int n) {
1.   int f;
2.   if(n < 2)
3.     f = 1;
4.   else {int a = fib(n-1);
5.         int b = fib(n-2);
6.         f = a + b;}
7.   return f;}
```

Inter-procedural control flow graph



Context free grammar

T → 0A
A → 1B
B → 2C
C → 3D|4E
D → 7
E → T5F
F → T6G
G → 7

Path Complexity for Recursive Functions

Context
free grammar

$T \rightarrow 0A$
$A \rightarrow 1B$
$B \rightarrow 2C$
$C \rightarrow 3D|4E$
$D \rightarrow 7$
$E \rightarrow T5F$
$F \rightarrow T6G$
$G \rightarrow 7$

Replace terminal characters with z

$T = zA$
$A = zB$
$B = zC$
$C = zD+zE$
$D = z$
$E = zTF$
$F = zTG$
$G = z$

Eliminate variables to get the generating function

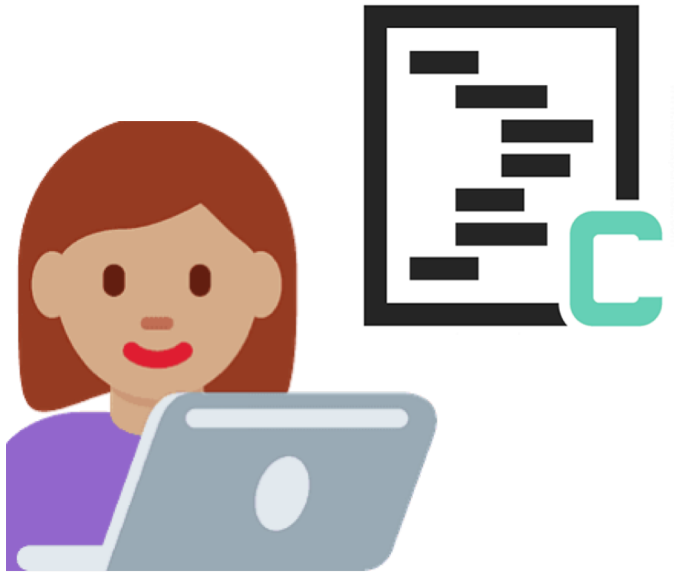
$z^5 + z^7T^2 - T = 0$

Asymptotic Path Complexity

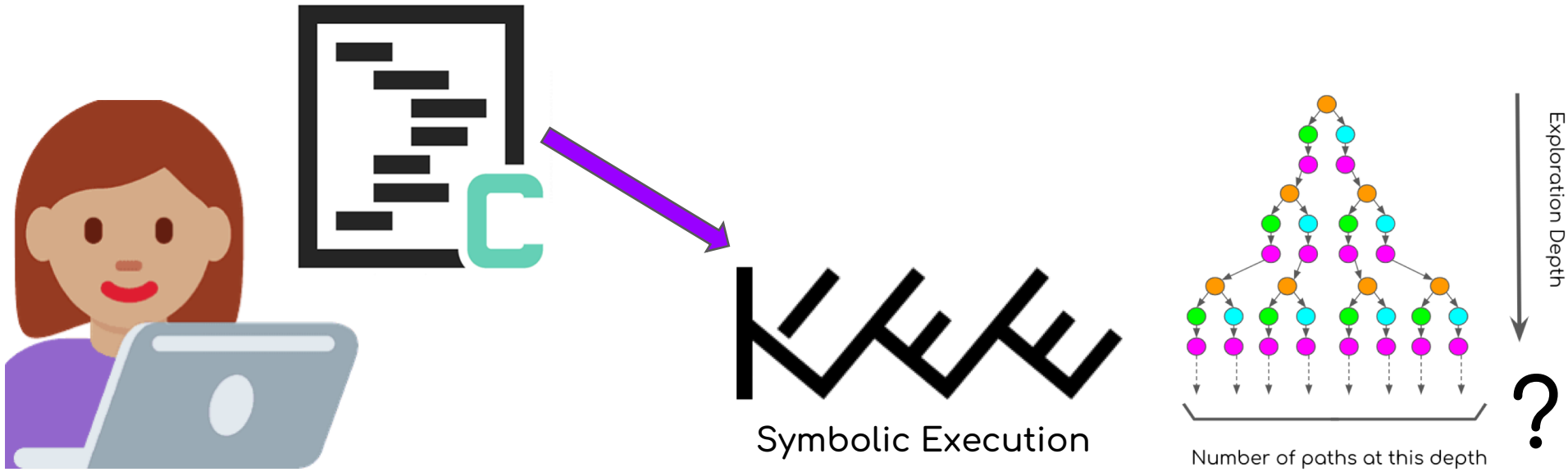
$4^{n/12} = 1.12^n$

Asymptotic Path Complexity Predicts the Severity of Symbolic Execution Path Explosion

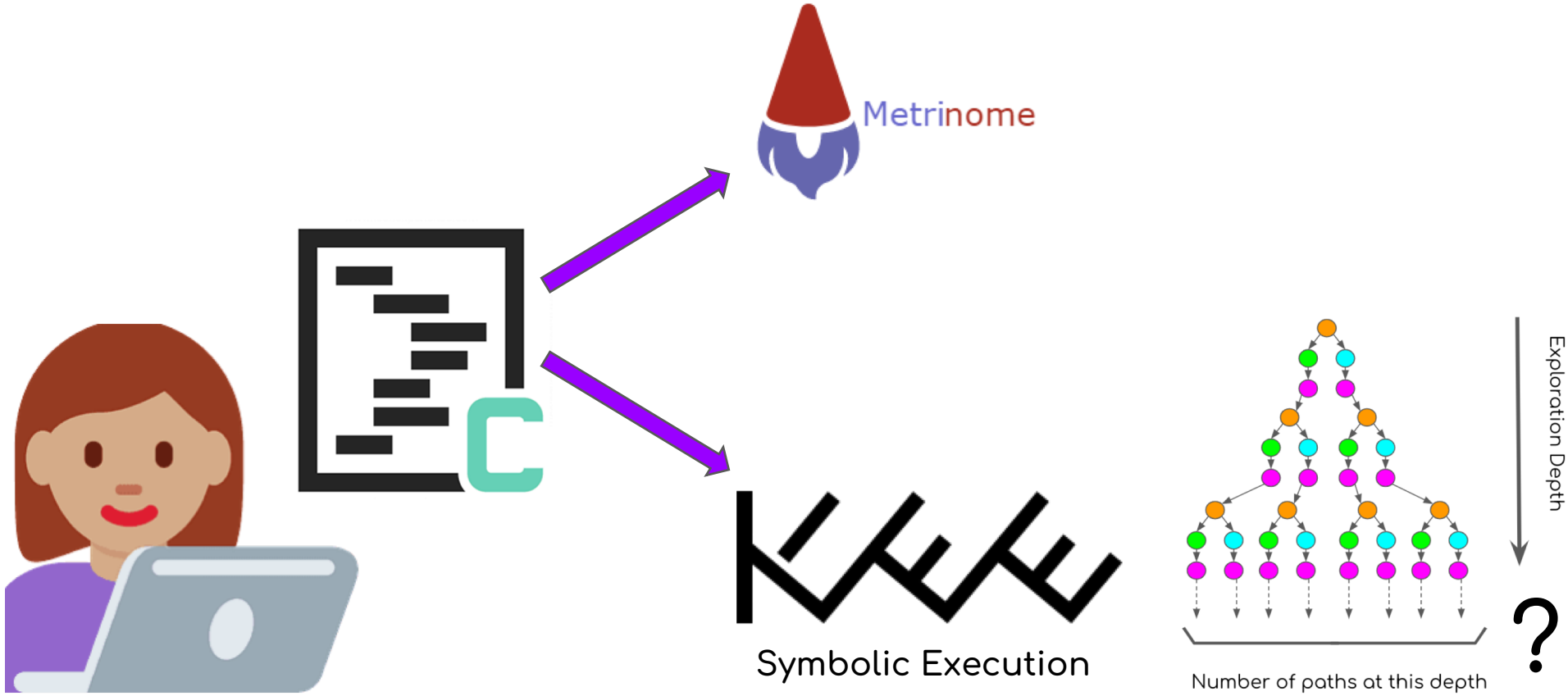
Gabriel Bessler, Josh Cordova, Shaheen Cullen-Baratloo, Sofiane Dissem, Emily Lu, Sofia Devin, Ibrahim Abugharah, Lucas Bang: Metrinome: Path Complexity Predicts Symbolic Execution Path Explosion. ICSE (Companion Volume) 2021: 29-32



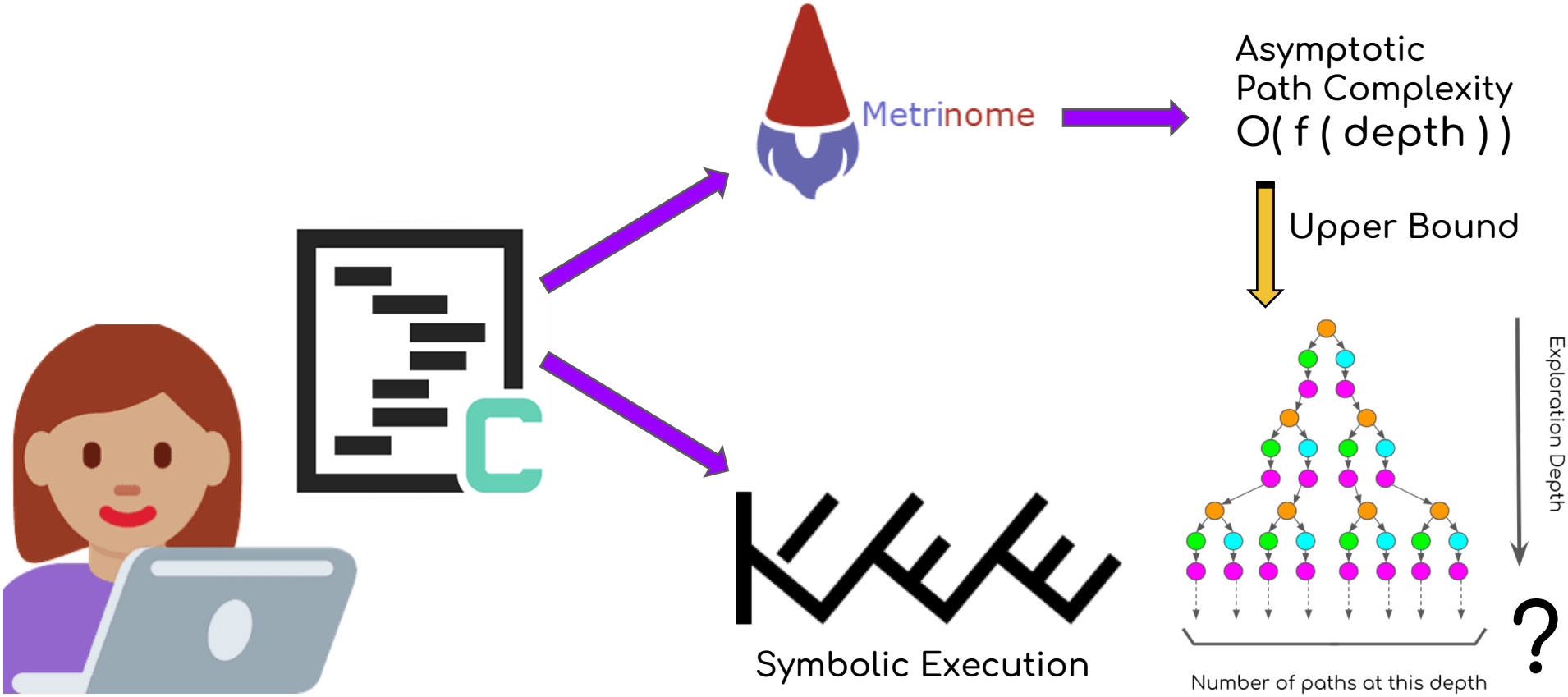
Asymptotic Path Complexity Predicts the Severity of Symbolic Execution Path Explosion



Asymptotic Path Complexity Predicts the Severity of Symbolic Execution Path Explosion



Asymptotic Path Complexity Predicts the Severity of Symbolic Execution Path Explosion



KLEE's path count correlates with path complexity

Code Comprehension: An fMRI Study

[Peitek, Norman, et al.]

- Compared existing complexity metrics against fMRI data
  - Participants have to look at a function and figure out what it does
  - Collect brain (de)activation data and human-based difficulty ratings
  - See how they correlate with existing metrics focused on code properties
    - LOC, McCabe's, Halstead, Dependency Degree

Peitek, Norman, et al. "Program Comprehension and Code Complexity Metrics: An fMRI Study" ICSE, 2021

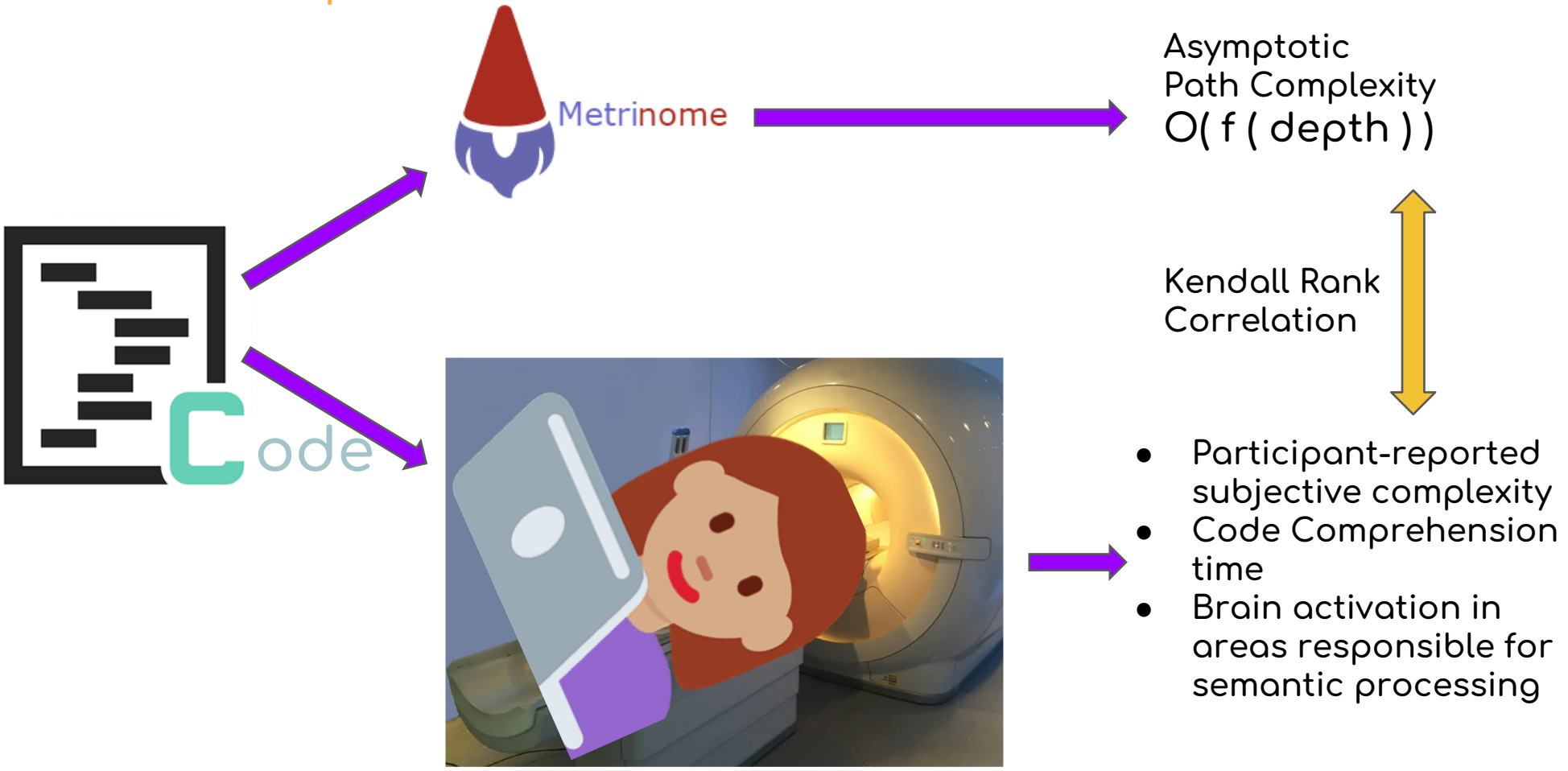
Results

- Overall, all analyzed metrics, except McCabe, are useful predictors of code comprehension difficulty

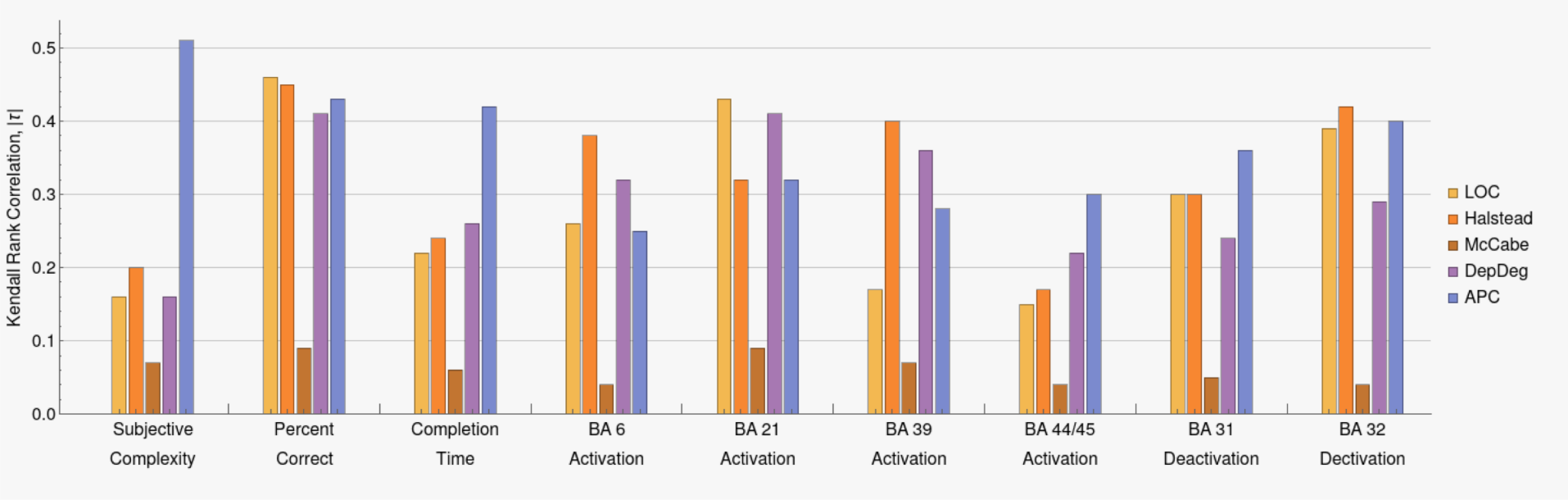
- Each metric performed better on different Brodmann Areas and code-comprehension indicators

Asymptotic Path Complexity Correlates with Code Comprehension

Sofiane Dissem, Eli Pregerson, Adi Bhargava, Josh Cordova, Lucas Bang: Path Complexity Correlates with Source Code Comprehension Effort Indicators. ICPC 2023: 266-274



Kendall Rank Correlations by Comprehension Effort Indicator



Asymptotic path complexity (APC) has a higher correlation than any other metric for several code comprehension difficulty indicators

Software complexity, verifiability, understandability

[Bessler et al.] Path complexity → verifiability via symbolic execution

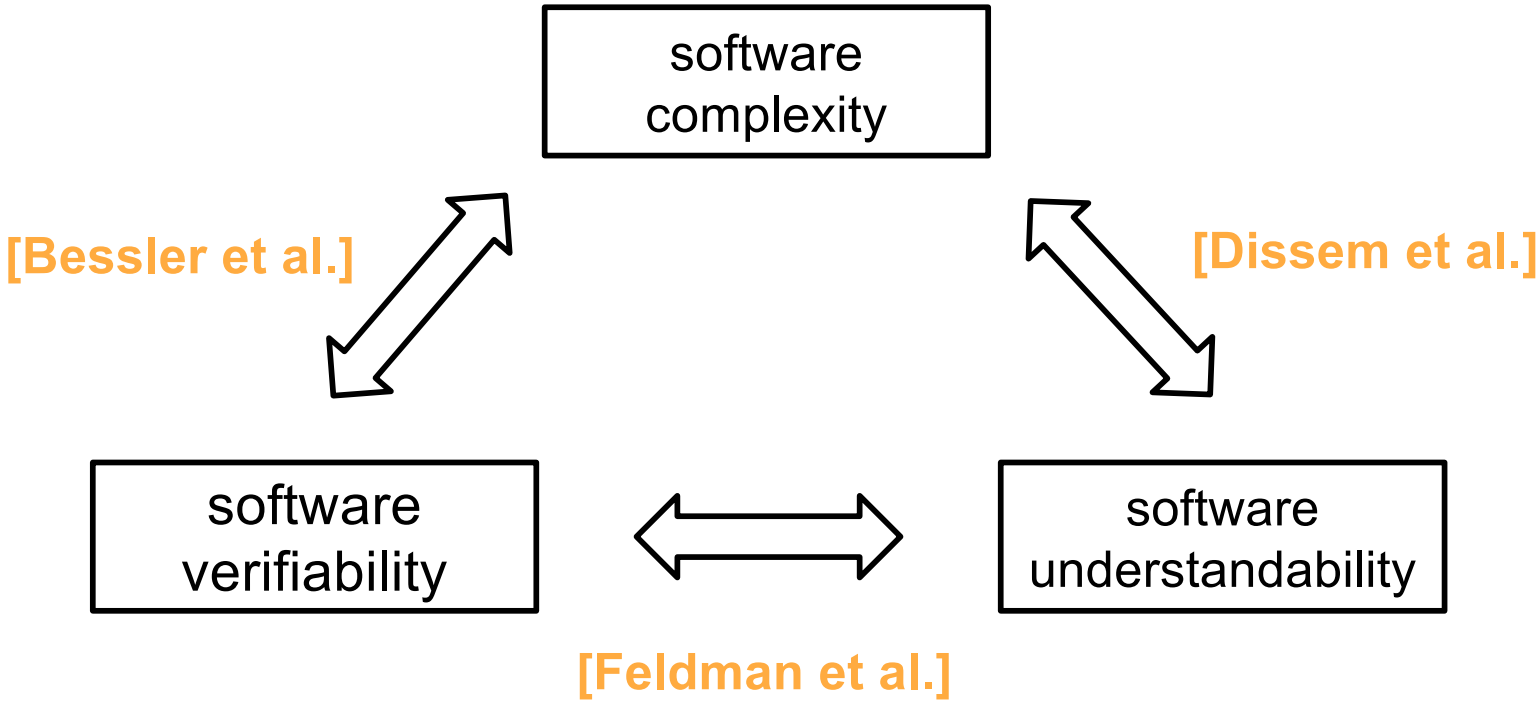
[Dissem et al.] Path complexity → code understandability

[Feldman et al.] Code verifiability → code understandability

"Our empirical study on the correlation between tool-based verifiability and human-based metrics of code understanding suggests there is a connection between whether a tool can verify a code snippet and how easy it is for a human to understand"

Kobi Feldman, Martin Kellogg, Oscar Chaparro: "On the Relationship between Code Verifiability and Understandability." ESEC/SIGSOFT FSE 2023: 211-223

Software complexity, verifiability, understandability



Outline

- Motivation
- Software complexity
- Path complexity
- **Branch selectivity**
- Future directions

What about infeasible paths?

Just looking at paths in the control flow graph (like in path complexity) does not give us the full picture about a program's behavior

- Not all paths are feasible

For some paths, it may be easy to check that they are feasible

- Randomly pick input values and see which paths are executed

But, for some paths it may not be easy to find a value that triggers the path

```
 1 public class Main {
 2   public static void main ( String [] args ) {
 3     int arg = Verifier.nondetInt ();
 4     if ( arg < 0 )
 5       return ;
 6     int x = arg / 5;
 7     int y = arg / 5;
 8     Main inst = new Main ();
 9     inst.test(x, y);
10  }
11  public void test ( int x, int z) {
12     System.out.println("Testing ExSymExe7");
13     int y = 3;
14     z = x - y - 4;
15     if ( z != 0 )
16       System.out.println ("branch FOO1");
17     else {
18       System.out.println ("branch FOO2");
19       assert false;
20     }
21     if ( y != 0 )
22       System.out.println("branch BOO1");
23     else
24       System.out.println("branch BOO2");
25   }
26 }
```

Is the assertion statement hard to reach for a random testing tool?

What is the likelihood of reaching the assertion statement if we run a random testing tool?

Identify Hard to Reach Statements

Program + Target Statement

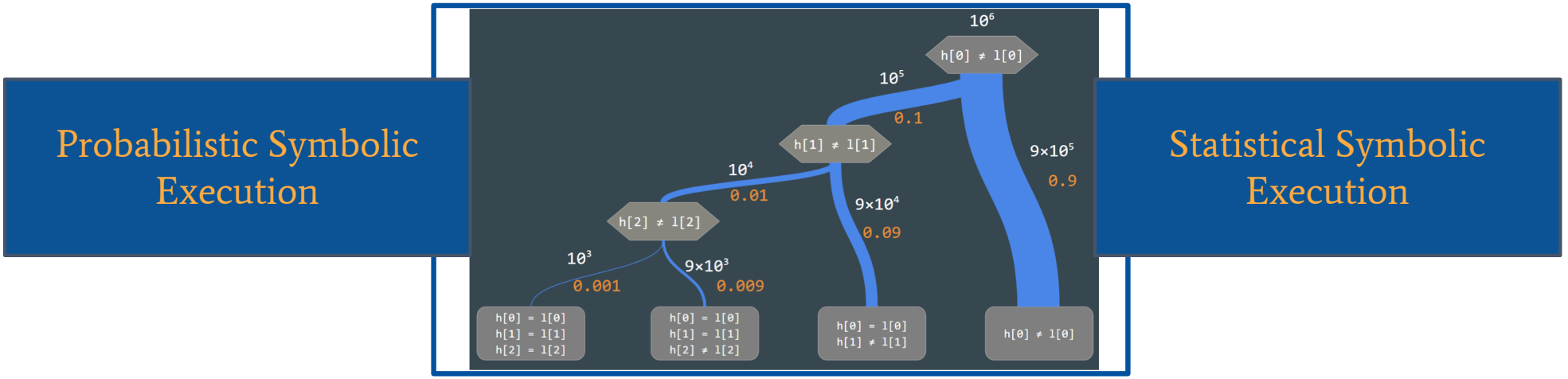
Easy to Reach Statement

Target Statement

Hard to Reach Statement

Probabilistic Reachability Analysis

Program + Target Statement



Easy to Reach Statement

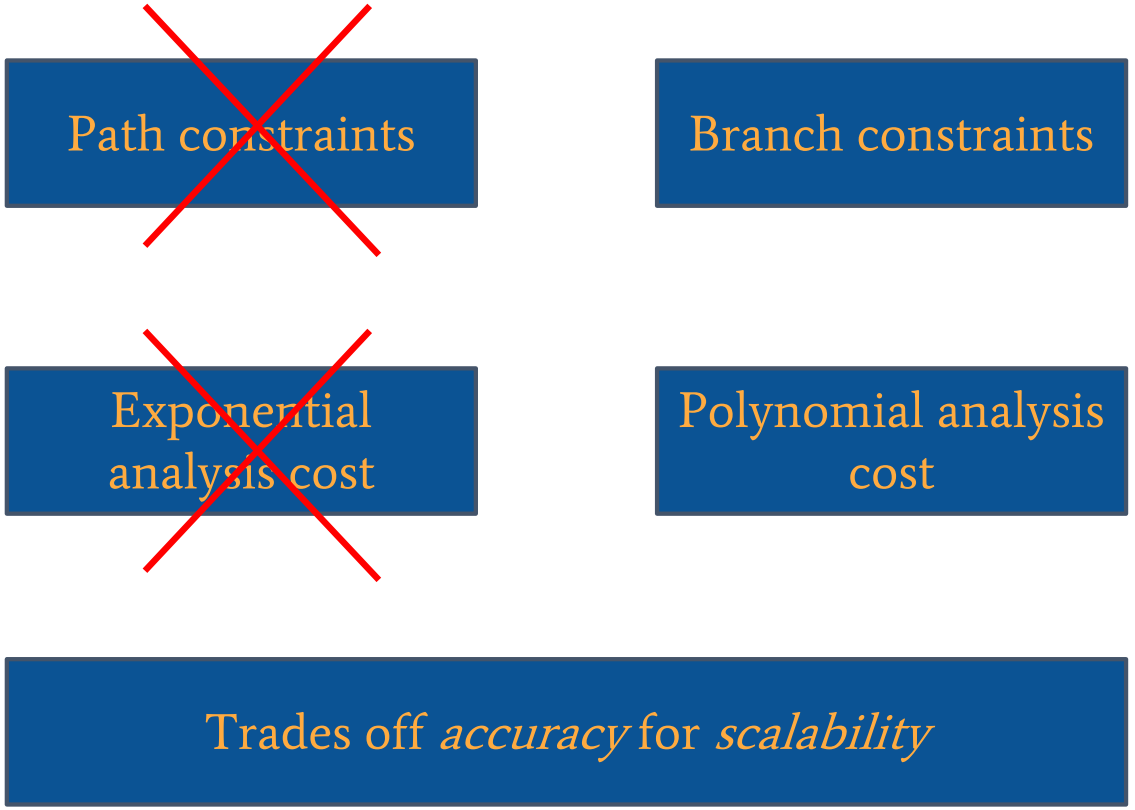
Target Statement

Hard to Reach Statement

Symbolic Execution based Techniques: Issues

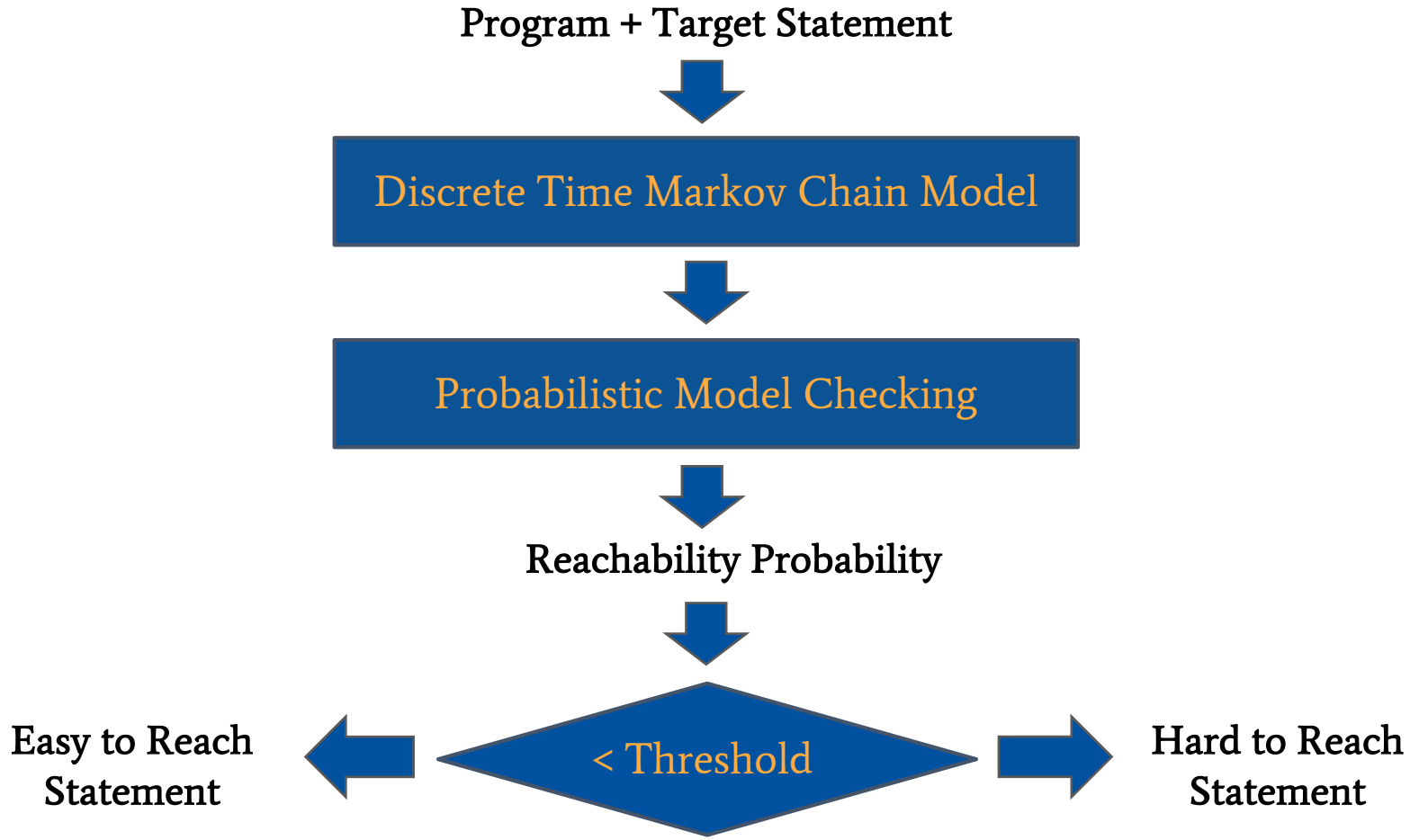
- Exponential increase in the number of program paths
- Linear increase in path constraint size ⇒ exponential increase in cost of symbolic execution
- Complexity increases further with model counting

PReach


Path constraints
Branch constraints
Exponential analysis cost
Polynomial analysis cost
Trades off *accuracy* for *scalability*

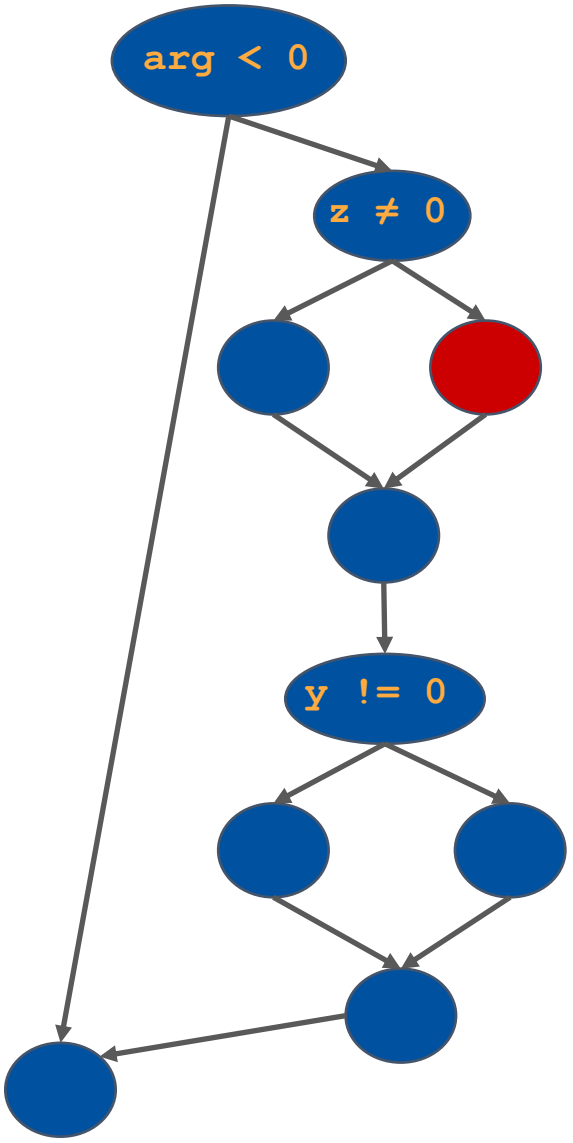
PReach

Seemanta Saha, Mara Downing, Tegan Brennan, Tevfik Bultan: "PREACH: A Heuristic for Probabilistic Reachability to Identify Hard to Reach Statements." ICSE 2022: 1706-1717

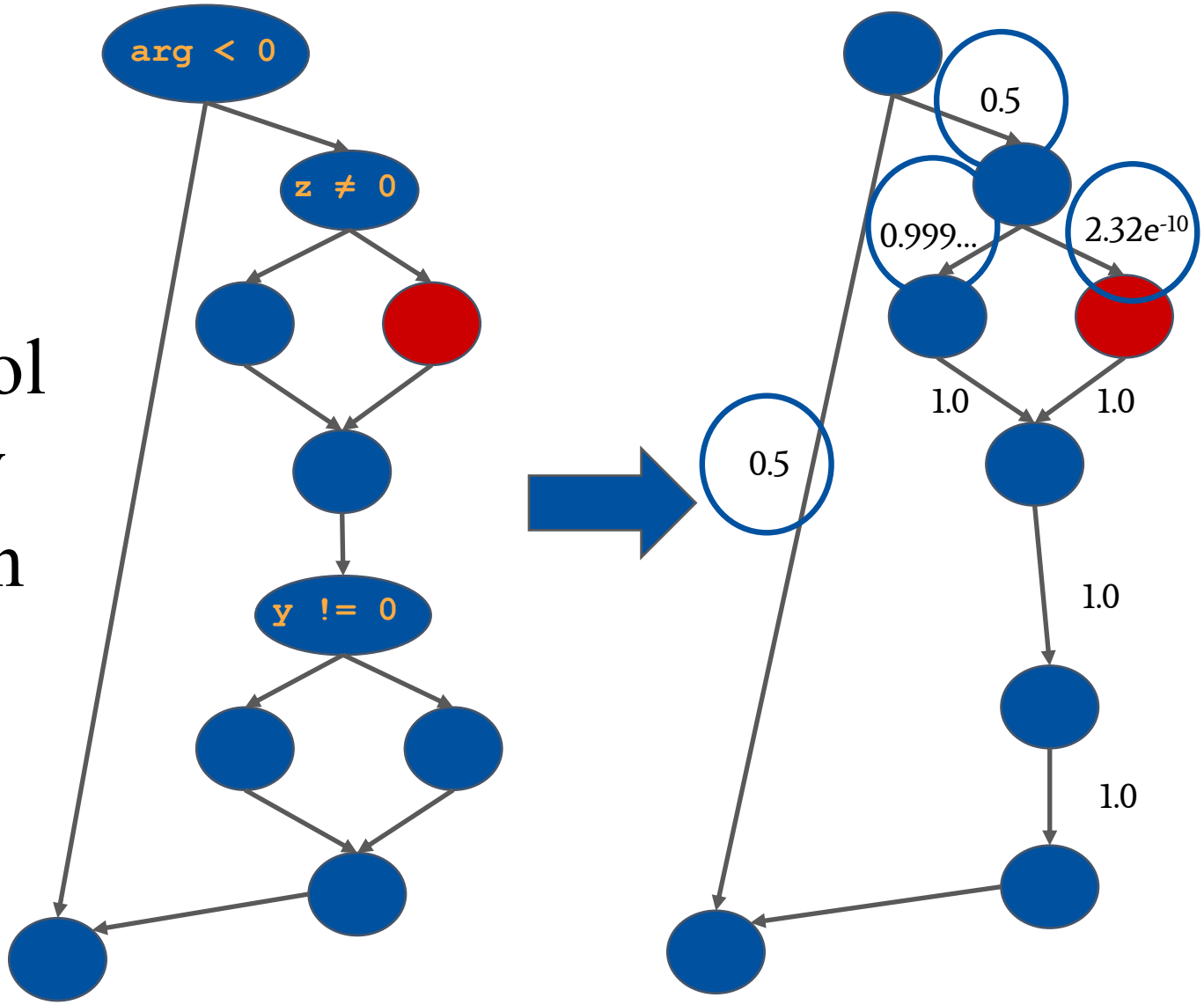


```
 1 public class Main {
 2   public static void main ( String [] args ) {
 3     int arg = Verifier.nondetInt ();
 4     if ( arg < 0 )
 5       return ;
 6     int x = arg / 5;
 7     int y = arg / 5;
 8     Main inst = new Main ();
 9     inst.test(x, y);
10  }
11  public void test ( int x, int z)
12     System.out.println("Testing F
13     int y = 3;
14     z = x - y - 4;
15     if ( z != 0 )
16       System.out.println ("branch FOO1");
17     else {
18       System.out.println ("branch FOO2");
19       assert false;
20     }
21     if ( y != 0 )
22       System.out.println("branch BOO1");
23     else
24       System.out.println("branch BOO2");
25   }
26 }
```

Control Flow Graph



Control Flow Graph



Discrete Time Markov Chain

Branch Selectivity

32-bit signed integer, 2^{32} possible values

```
1 public int test (int input) {
2    if (input == 0)
3        assert false;
4    else if (input > 0)
5        return 1;
6    else
7        return -1;
8 }
```

Selectivity of this branch is **$1/2^{32}$**

input == 0

Branch Condition (T_b)

(declare-fun input() Int)
(assert (= input 0))
(check-sat)

Branch Constraint

Branch Model Count (|Tb|)= 1

Model Counting

Branch Count ($|T_b|$) = 1
Domain Size ($|D_b|$) = 2^{32}
Selectivity, S(b) = $1/2^{32}$

Model Counting

The classic (Boolean) SAT problem: is formula ϕ satisfiable?

$$\phi = (x \vee y) \wedge (\neg x \vee z) \wedge (z \vee w) \wedge x \wedge (y \vee v)$$

ϕ is satisfied by setting $(x, y, z, w, v) = (T, F, T, F, T)$

- A satisfying assignment is called a model for ϕ

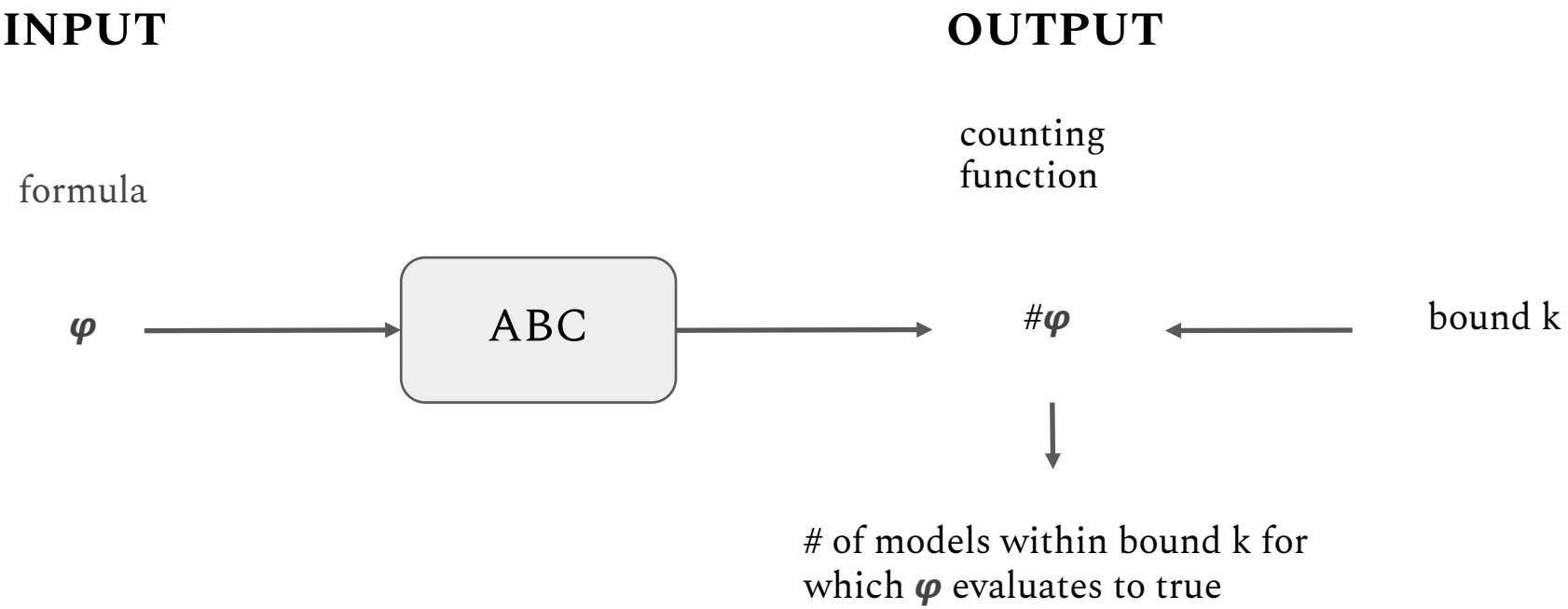
The model counting problem:

how many models are there for ϕ?

ABC: Model counting constraint solver

Abdulbaki Aydin, Lucas Bang, Tevfik Bultan: "Automata-Based Model Counting for String Constraints." CAV (1) 2015: 255-272

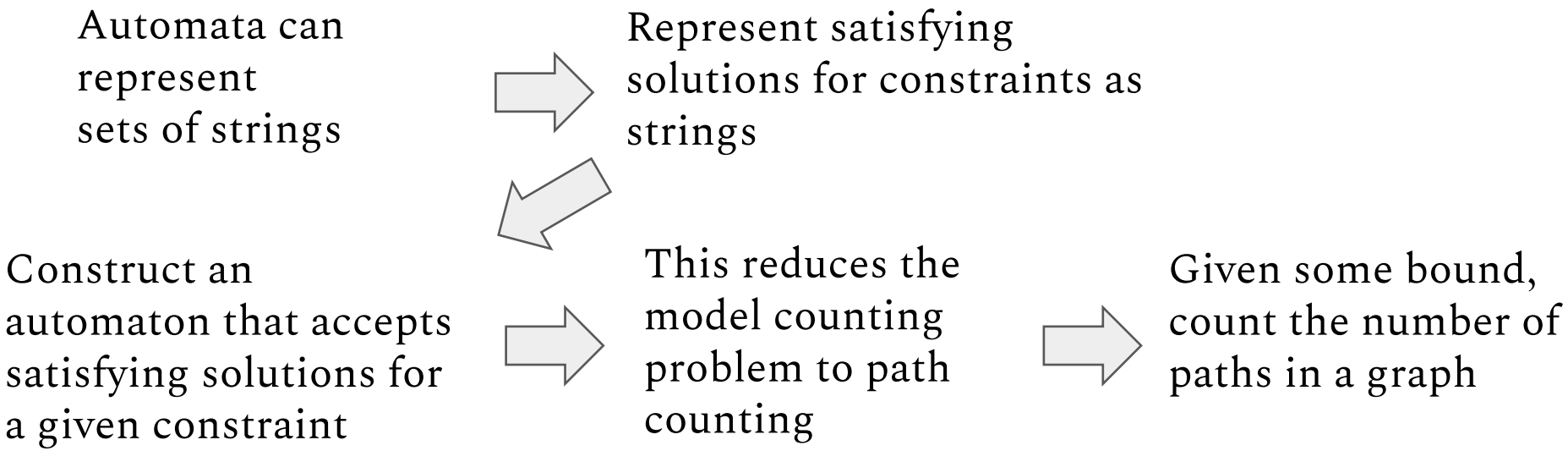
Abdulbaki Aydin, William Eiers, Lucas Bang, Tegan Brennan, Miroslav Gavrilov, Tevfik Bultan, Fang Yu: "Parameterized model counting for string and numeric constraints." ESEC/SIGSOFT FSE 2018: 400-410



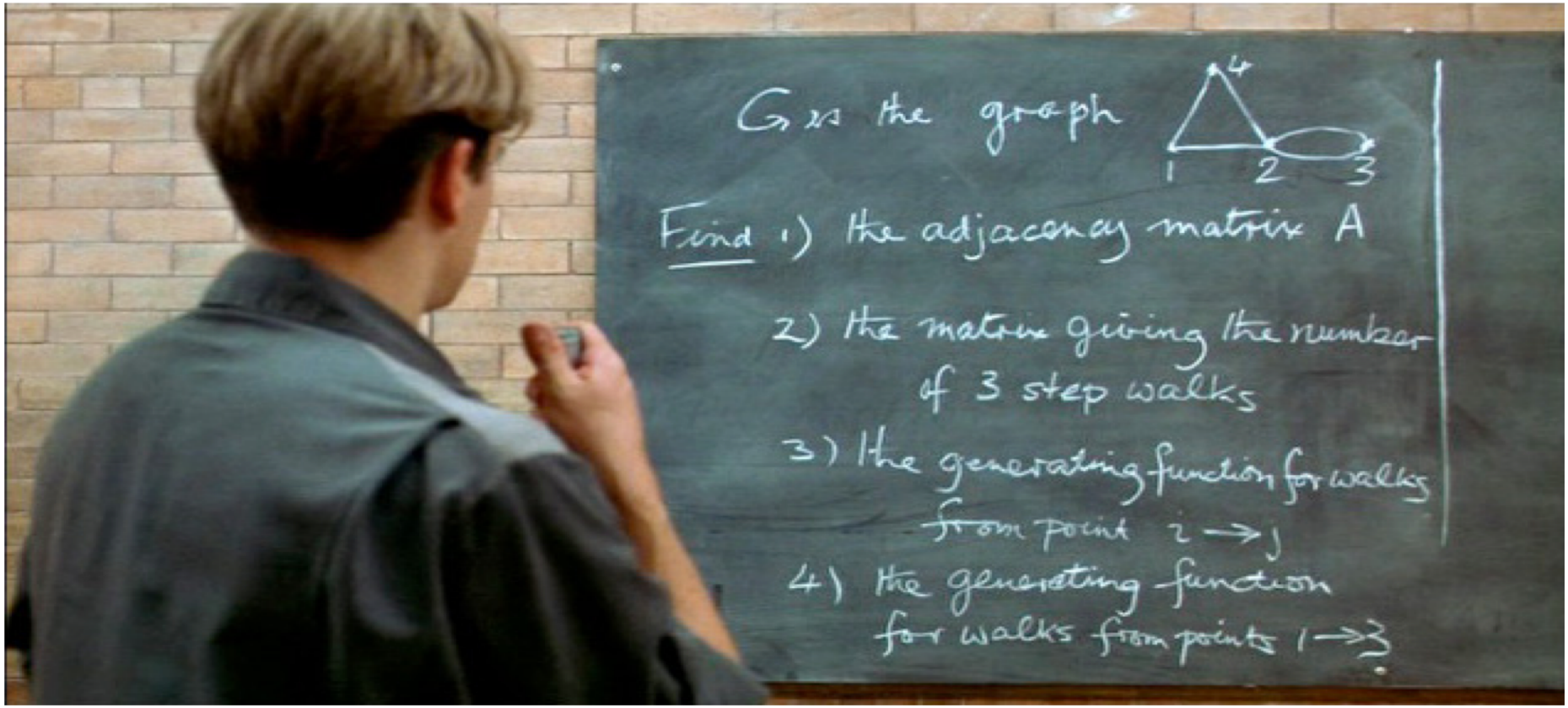
ABC in a nutshell

Automata-based constraint solving

Basic idea:



Again, path counting!

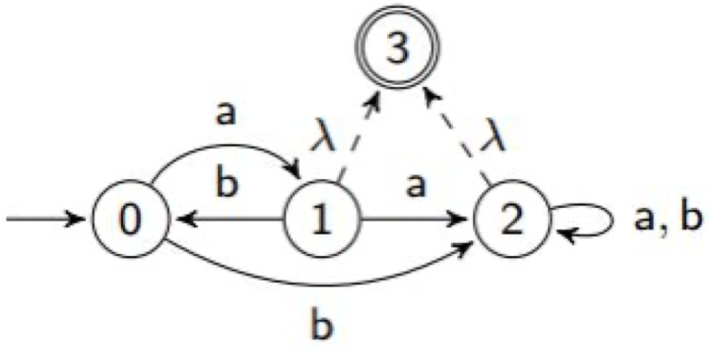


Automata-based model counting

Automata can represent sets of strings

- Represent satisfying solutions for constraints as strings

  `¬match(v, (ab)*)`

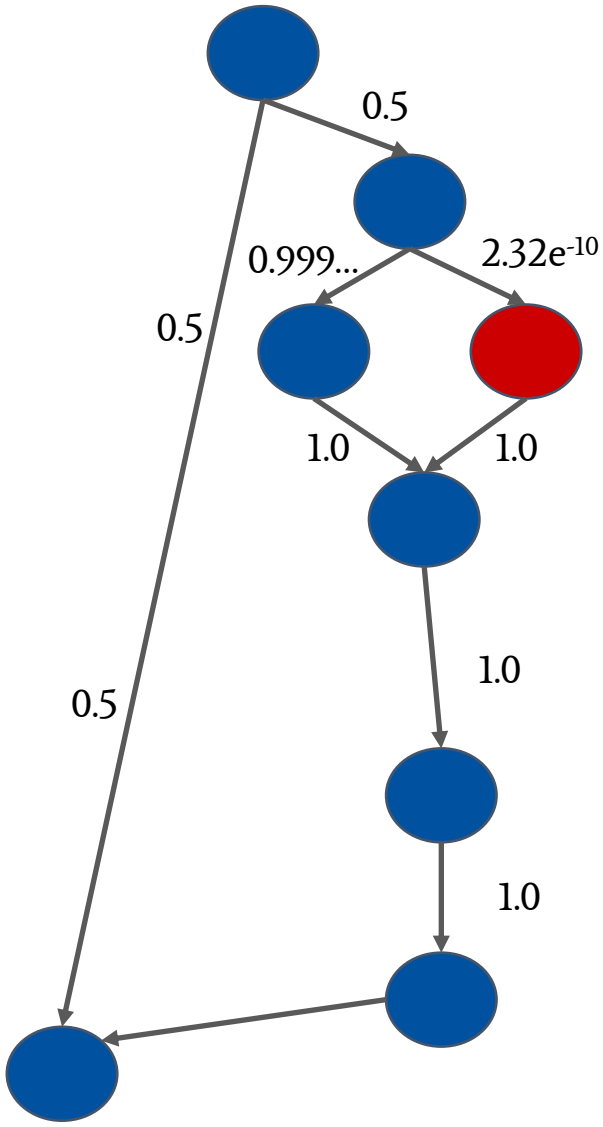


This reduces the model counting problem to path counting

- Given some bound, count the number of paths in a graph

$$T = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, T^2 = \begin{bmatrix} 1 & 0 & 1 & 2 \\ 0 & 1 & 3 & 1 \\ 0 & 0 & 4 & 2 \\ 0 & 0 & 0 & 0 \end{bmatrix}, T^3 = \begin{bmatrix} 0 & 1 & 7 & 3 \\ 1 & 0 & 7 & 4 \\ 0 & 0 & 8 & 4 \\ 0 & 0 & 0 & 0 \end{bmatrix}, T^4 = \begin{bmatrix} 0 & 1 & 15 & 8 \\ 1 & 0 & 15 & 7 \\ 0 & 0 & 16 & 8 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$f(0) = 0 \qquad f(1) = 2 \qquad f(2) = 3 \qquad f(3) = 8$$




```
dtmc
module test
s : [0..7] init 0;
[] s = 0 -> 0.5 : (s' = 1) + 0.5 :
(s' = 2);
[] s = 2 -> 0.99999999976 : (s' = 3)
+ 2.32e-10 : (s' = 4);
[] s = 3 -> 1.0 : (s' = 5);
[] s = 4 -> 1.0 : (s' = 5);
[] s = 5 -> 1.0 : (s' = 6);
[] s = 6 -> 1.0 : (s' = 7);
[] s = 7 -> 1.0 : (s' = 1);
[] s = 1 -> 1.0 : (s' = 1);
endmodule
```

DTMC Model in PRISM

What is the probability that the target node is reached eventually?

```
P=? [F s = 4]
```

PCTL Query

PReach: Tools and Benchmarks

- Branch selectivity → **PReach**
- Refined branch selectivity
  - using **interval analysis** (box domain) → **PReach-I**
  - using **relational analysis** (polyhedra domain) → **PReach-P**

Assessment on SV-Comp Benchmarks

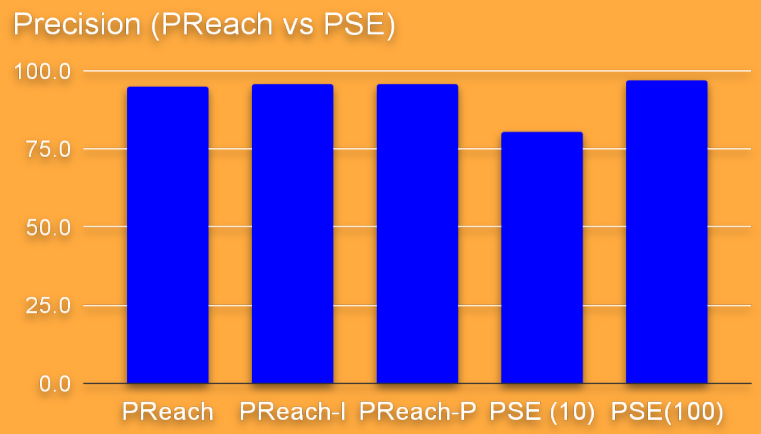
How successful is Preach in identifying hard to reach statements?

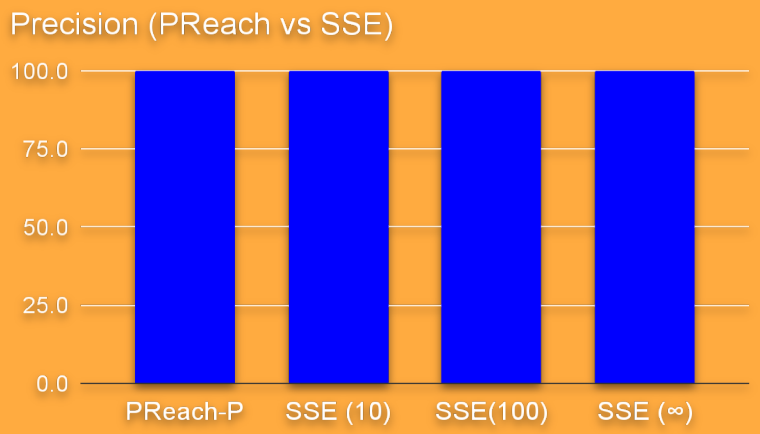
Accuracy: 95.8

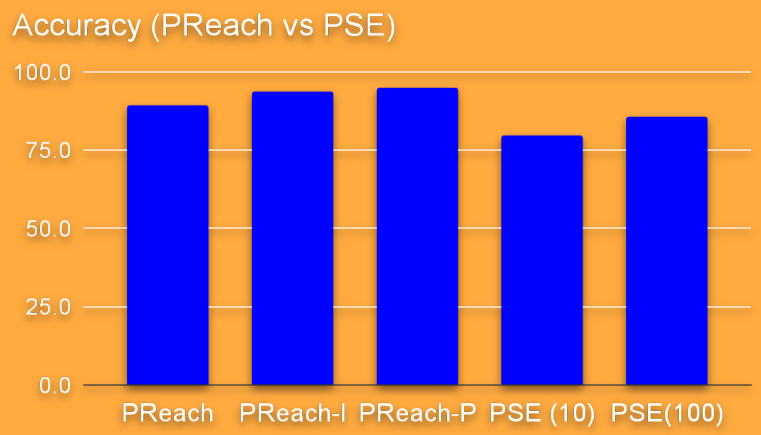
Precision: 95.1

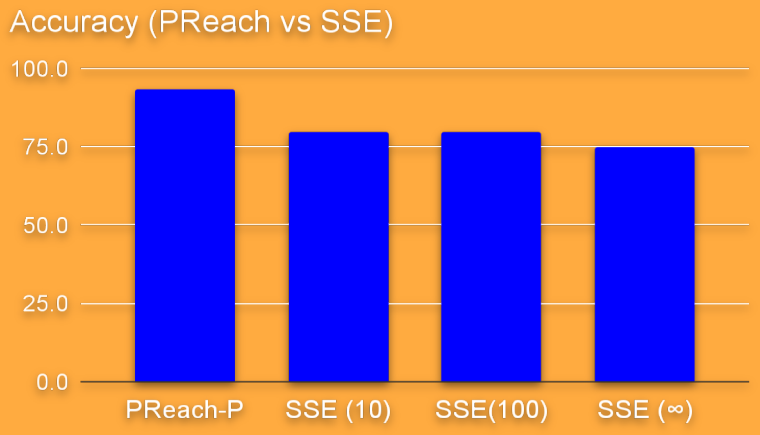
Recall: 90.2

PReach vs. PSE and SSE:

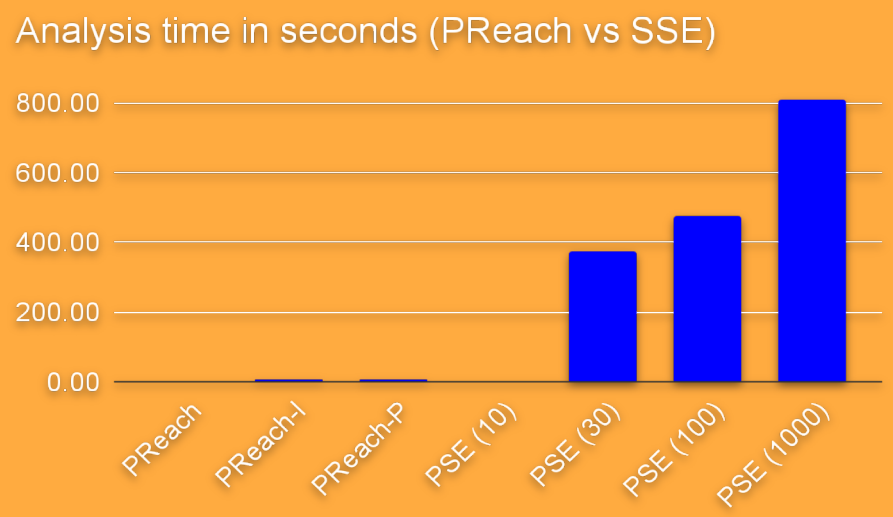


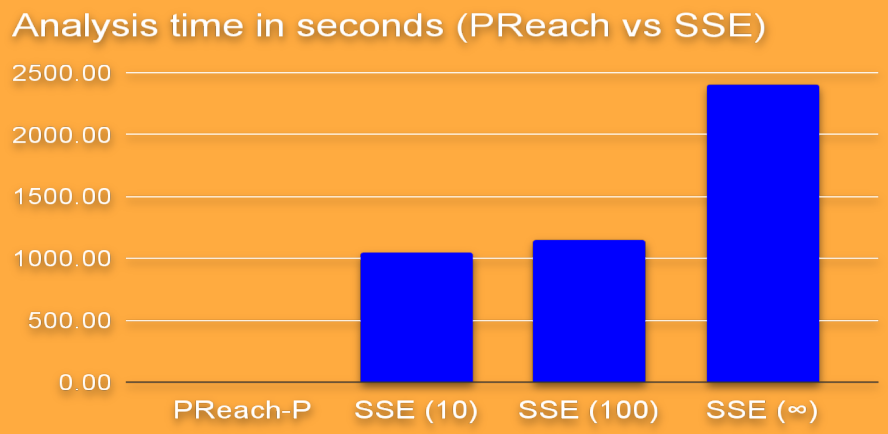






PReach vs. PSE and SSE:
PReach is orders of magnitude faster



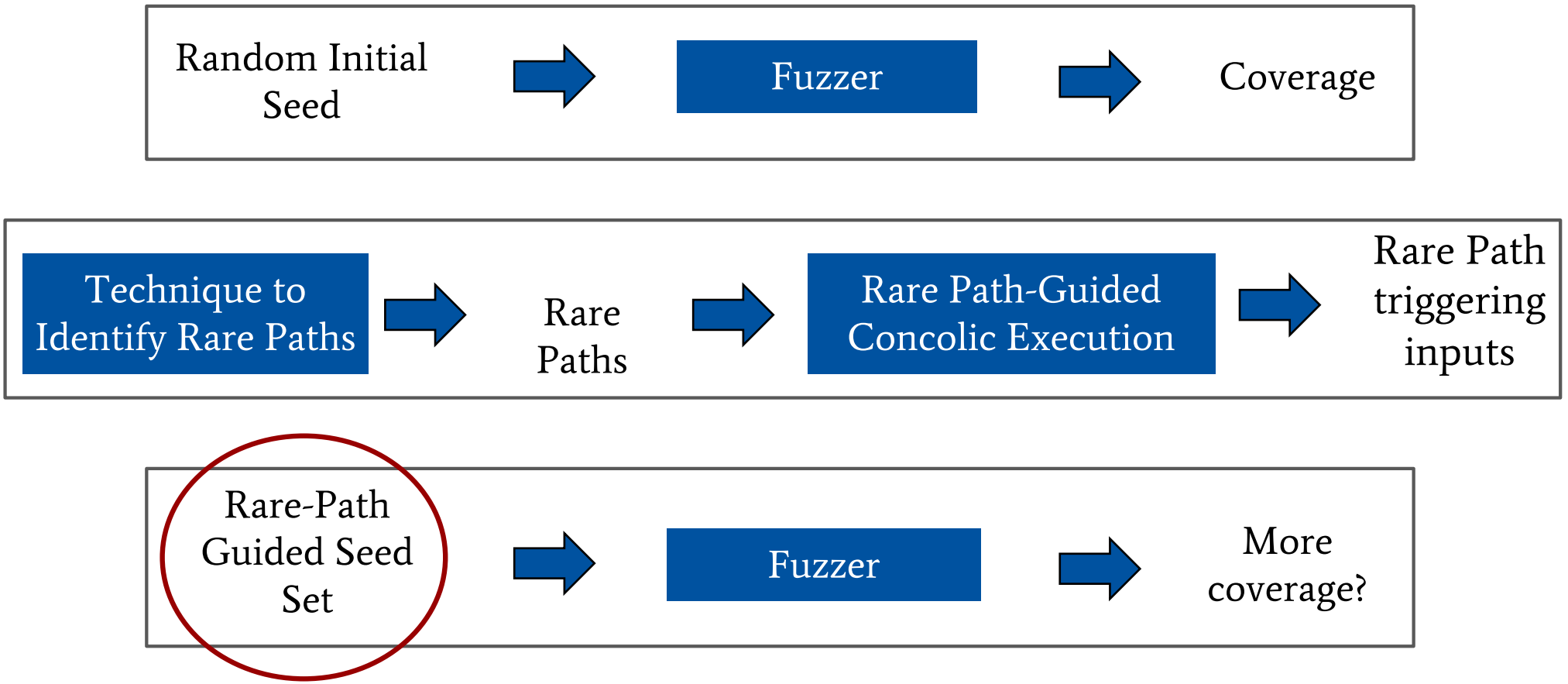


Can we use branch selectivity to improve verifiability?

- We showed that branch selectivity can help us identify hard to reach statements

- Can we use branch selectivity and probabilistic analysis to improve performance of fuzzing tools?

Rare-Path Guided Fuzzing



Seemanta Saha, Laboni Sarker, Md Shafiuzzaman, Chaofan Shou, Albert Li, Ganesh Sankaran, Tevfik Bultan: "Rare Path Guided Fuzzing." ISSTA 2023: 1295-1306

Identify Rare Program Paths

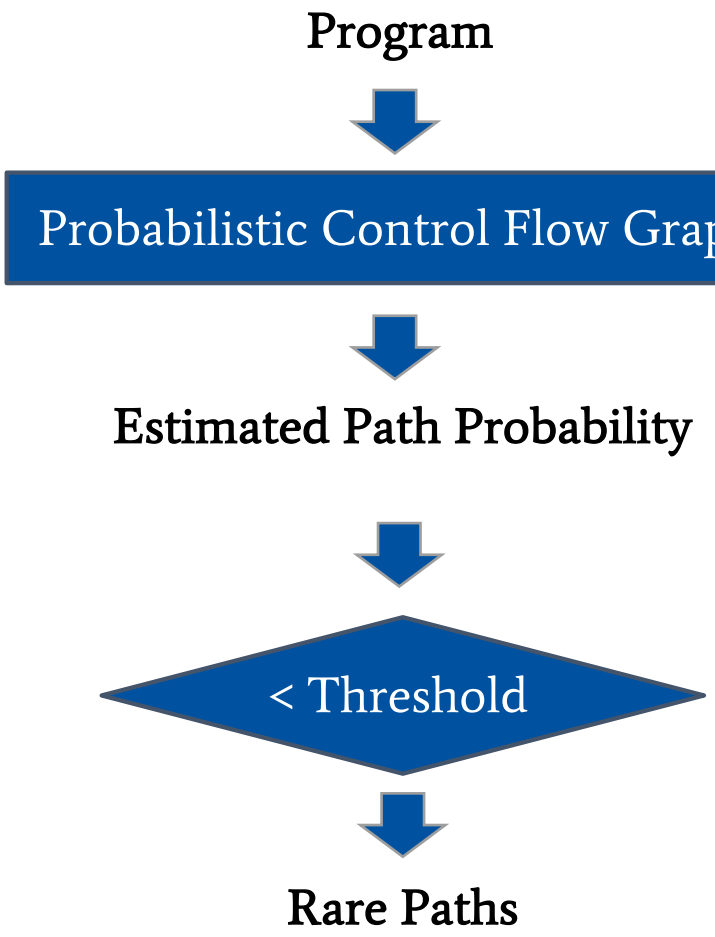
UCSB

Rare Path : A program path which is unlikely to be exercised if we run fuzzer with a uniformly chosen random input.

To identify rare paths:

- Use Branch Selectivity
- Same approach we used earlier for identifying hard to reach statements

Heuristic: Identify Rare Program Paths



Branch Selectivity

32-bit signed integer, 2^{32} possible values

```
1 public int test (int input) {
2     if (input == 0)
3         assert false;
4     else if (input > 0)
5         return 1;
6     else
7         return -1;
8 }
```

Selectivity of this branch is $1/2^{32}$

Rare-Path Guided Fuzzing: Motivation

UCSB

```
char *CUR;
#define CMP3( s, c1, c2, c3 ) \
  ( ((unsigned char *) s)[ 0 ] == c1 && \
    ((unsigned char *) s)[ 1 ] == c2 && \
    ((unsigned char *) s)[ 2 ] == c3 )
int main(int argc, char **argv) {
  CUR = argv[1];
  if (CMP3(CUR, 'D', 'O', 'C')) {
    CUR = CUR + 3;
    parse_cmt();
    if(parse_att())
      /* go deeper */
  }
  return 0;
}
void parse_cmt() {
  if(*CUR == '<' || CUR == '>')
    CUR++;
}
int parse_att() {
  if (CMP3(CUR, 'A', 'T', 'T'))
    return 1;
  return 0;
}
```

DOC

SKIPS '<' or '>'

ATT

Inputs:
DOCATT
DOC<ATT
DOC>ATT

Fuzzer with random seed **cannot generate sequences "DOC" and "ATT" within 1 hour**

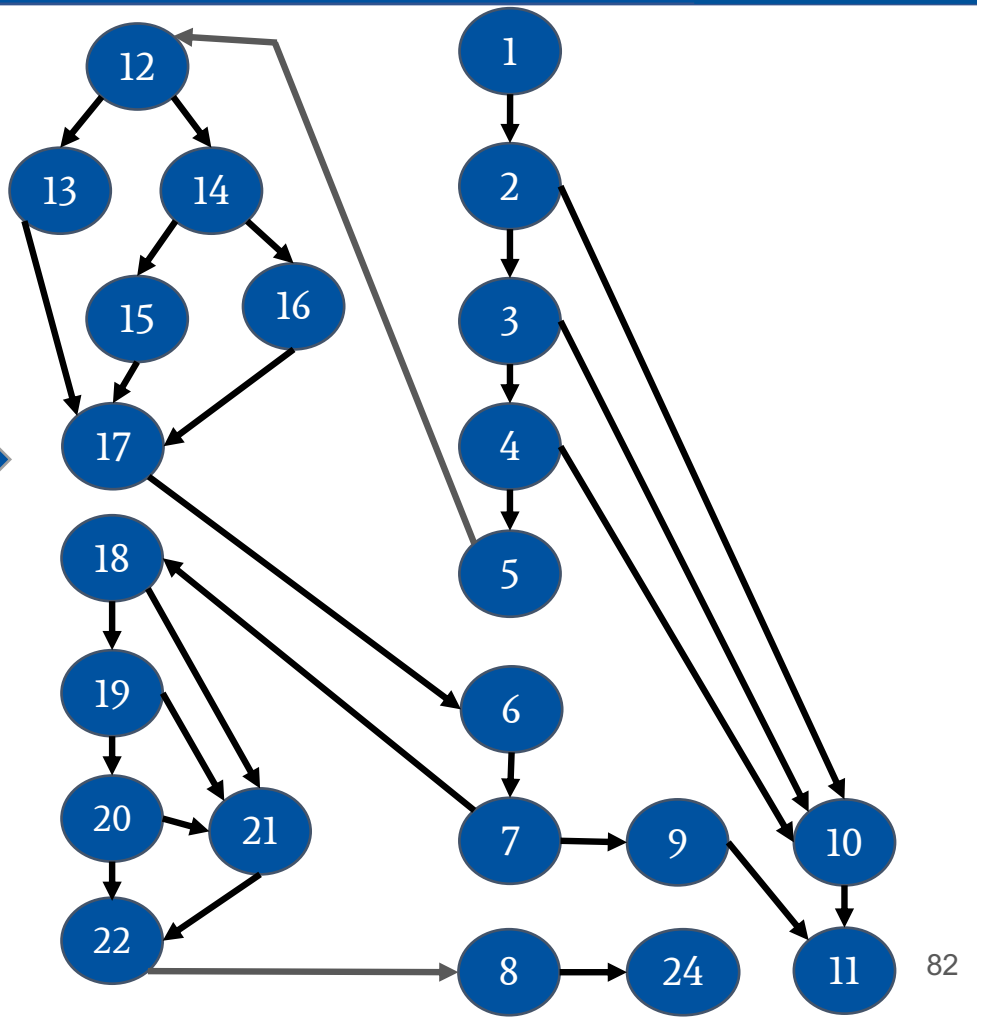
Our rare path analysis **can generate seed "DOC<ATT"** within 1 minute and fuzzer can explore deeper functionalities immediately

Heuristic: Identify Rare Program Paths

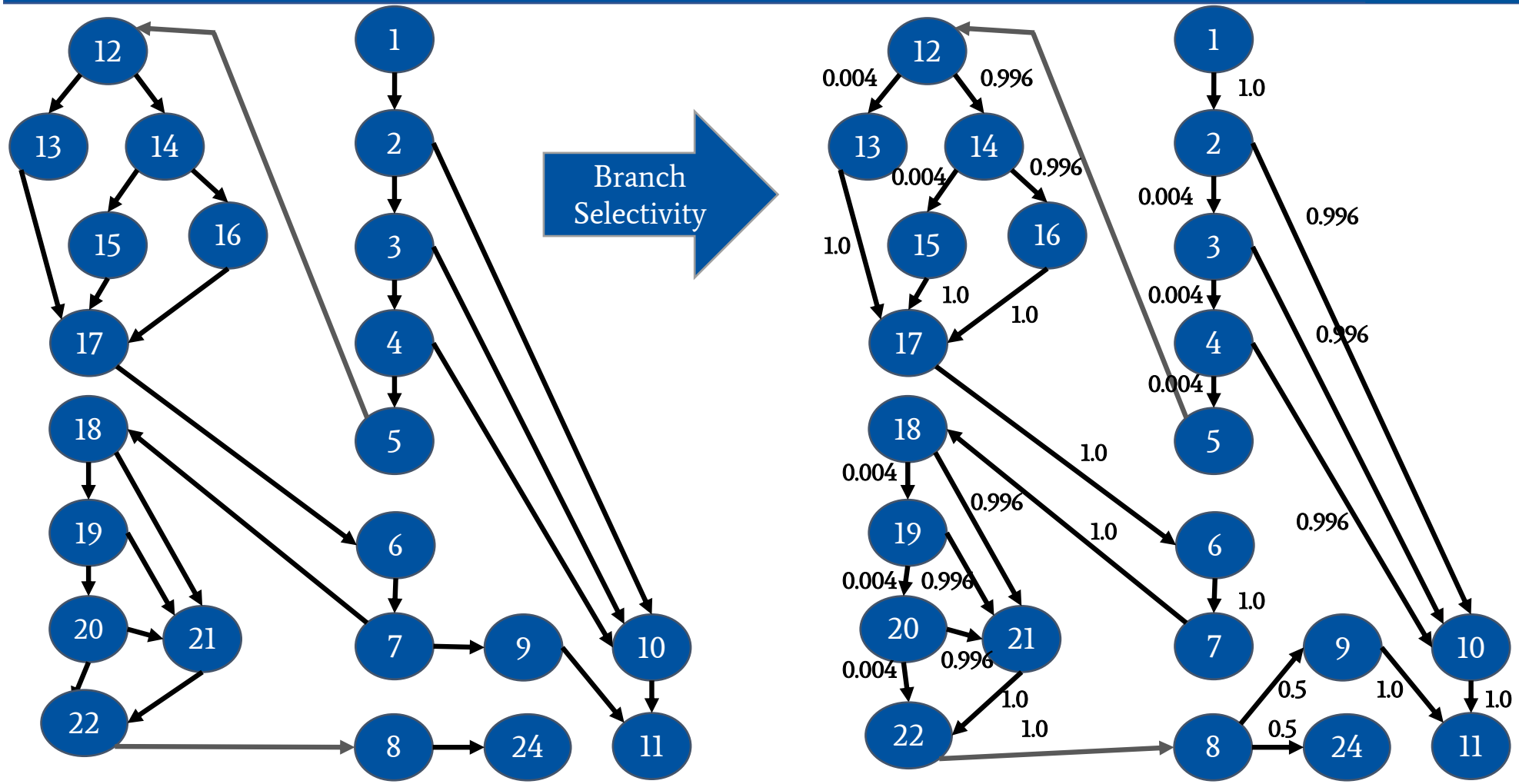
UCSB

```
char *CUR;
#define CMP3( s, c1, c2, c3 ) \
  ( ((unsigned char *) s)[ 0 ] == c1 && \
    ((unsigned char *) s)[ 1 ] == c2 && \
    ((unsigned char *) s)[ 2 ] == c3 )
int main(int argc, char **argv) {
  CUR = argv[1];
  if (CMP3(CUR, 'D', 'O', 'C')) {
    CUR = CUR + 3;
    parse_cmt();
    if(parse_att())
      /* go deeper */
  }
  return 0;
}
void parse_cmt() {
  if(*CUR == '<' || CUR == '>')
    CUR++;
}
int parse_att() {
  if (CMP3(CUR, 'A', 'T', 'T'))
    return 1;
  return 0;
}
```

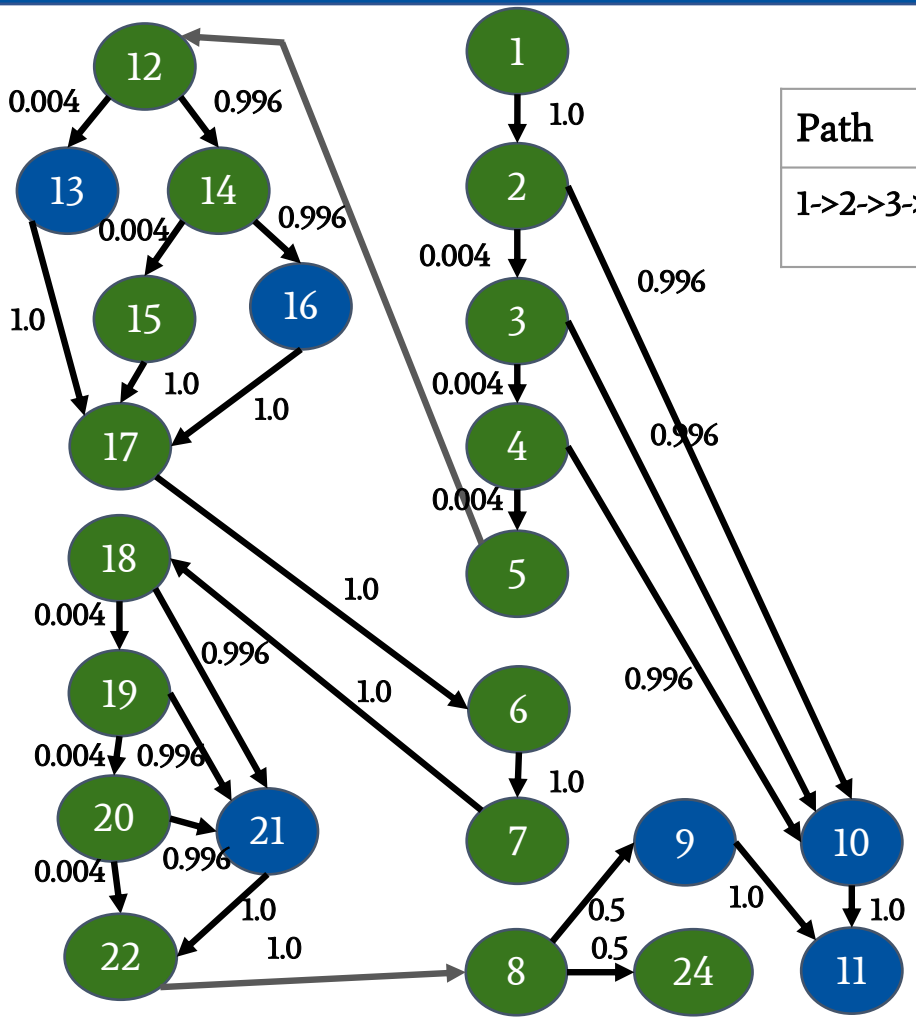
Control Flow Graph



Probabilistic CFG



Path Probability Estimation

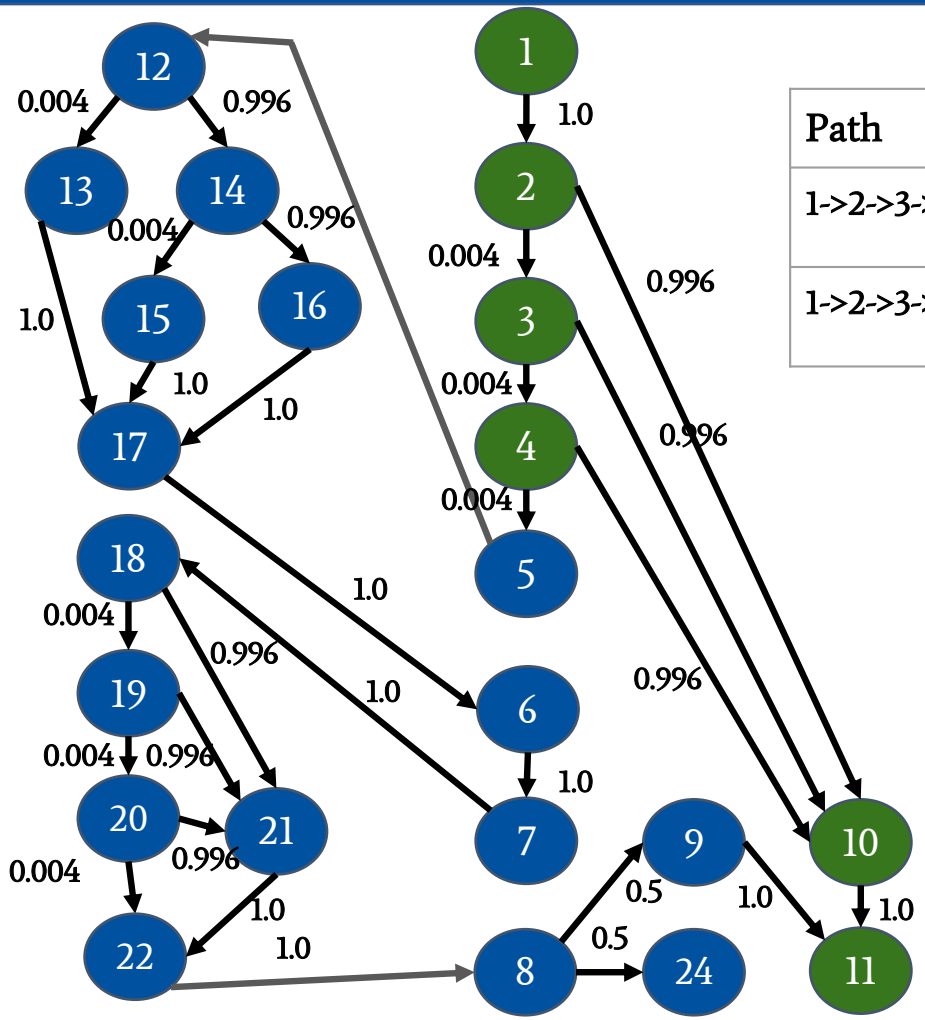


Path	Probability
1->2->3->4->5->12->14->15->17->6->7->18->19->20->22->8->24	8.16×10^{-18}

Multiply edge probabilities to compute path probability

Nodes taken by the path

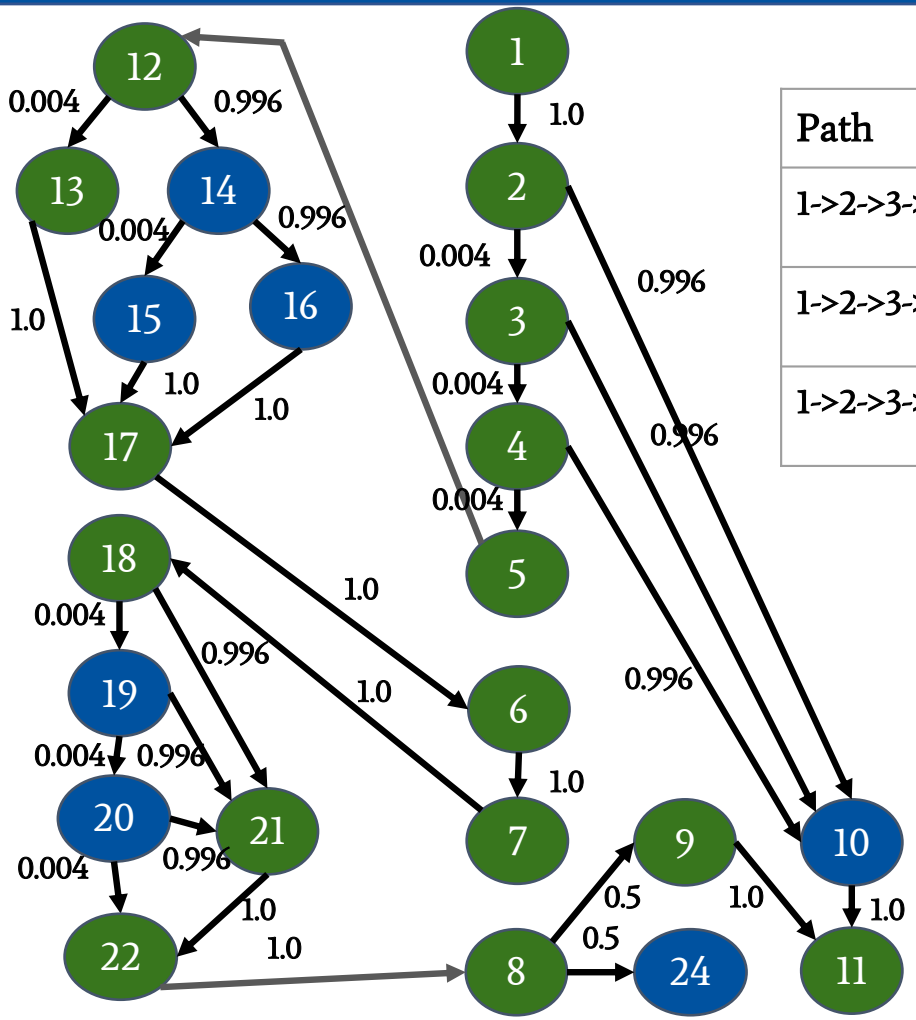
Path Probability Estimation



Path	Probability
1->2->3->4->5->12->14->15->17->6->7->18->19->20->22->8->24	8.16×10^{-18}
1->2->3->4->10->11	1.59×10^{-5}

Nodes taken by the path

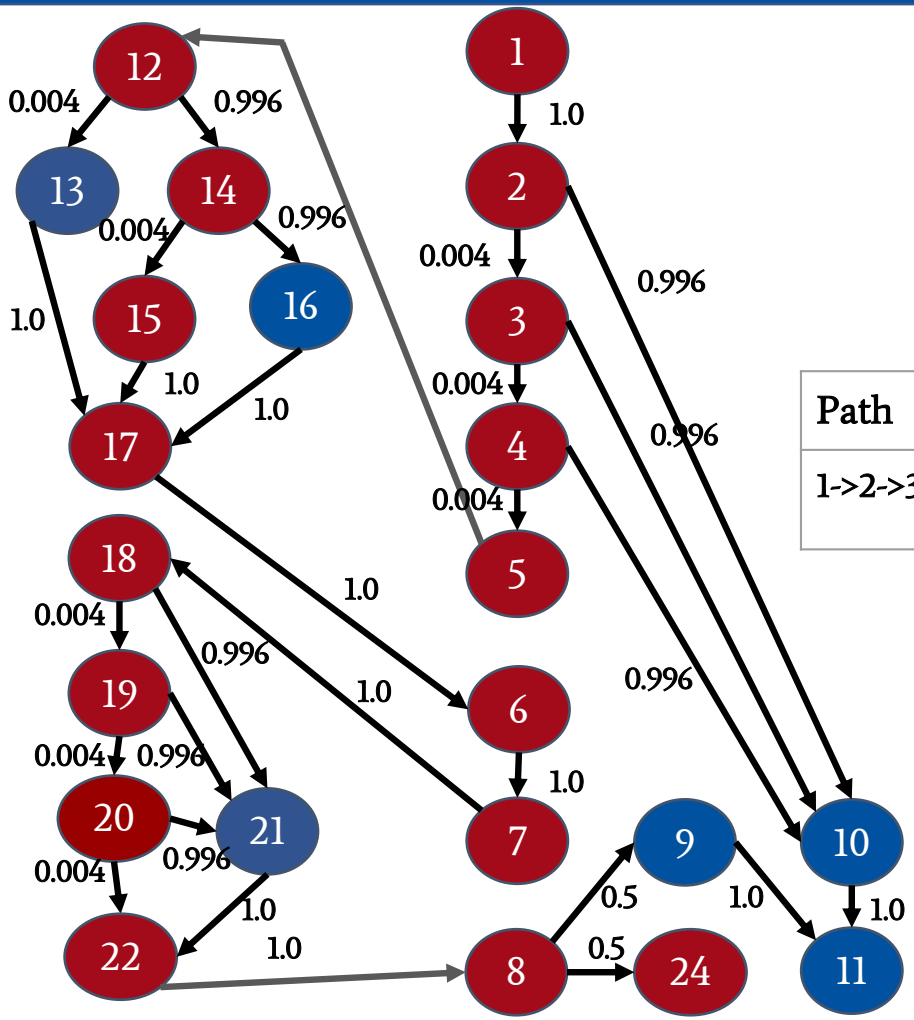
Path Probability Estimation



Path	Probability
1->2->3->4->5->12->14->15->17->6->7->18->19->20->22->8->24	8.16×10^{-18}
1->2->3->4->10->11	1.59×10^{-5}
1->2->3->4->5->12->13->17->6->18->21->22->8->9->11	1.27×10^{-10}

Nodes taken by the path

Rare Path

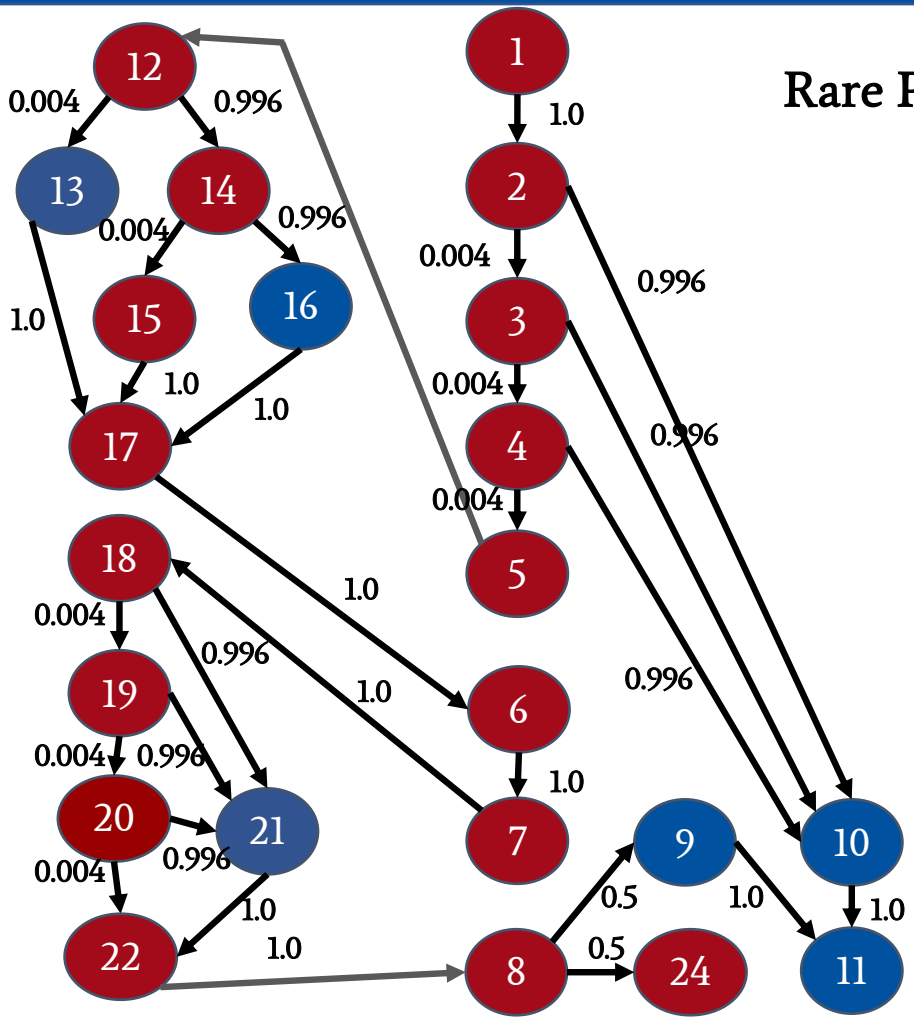


Estimated path probability for a path less than a threshold

Path	Probability
1->2->3->4->5->12->14->15->17->6->7->18->19->20->22->8->24	8.16×10^{-18}

Nodes taken by the rare path

Path-Guided Concolic Execution



Rare Path

1->2->3->4->5->12->14->15->17->6->7->18->19->20->22->8->24

Constraint Solving

DOC<ATT

Nodes taken by the rare path

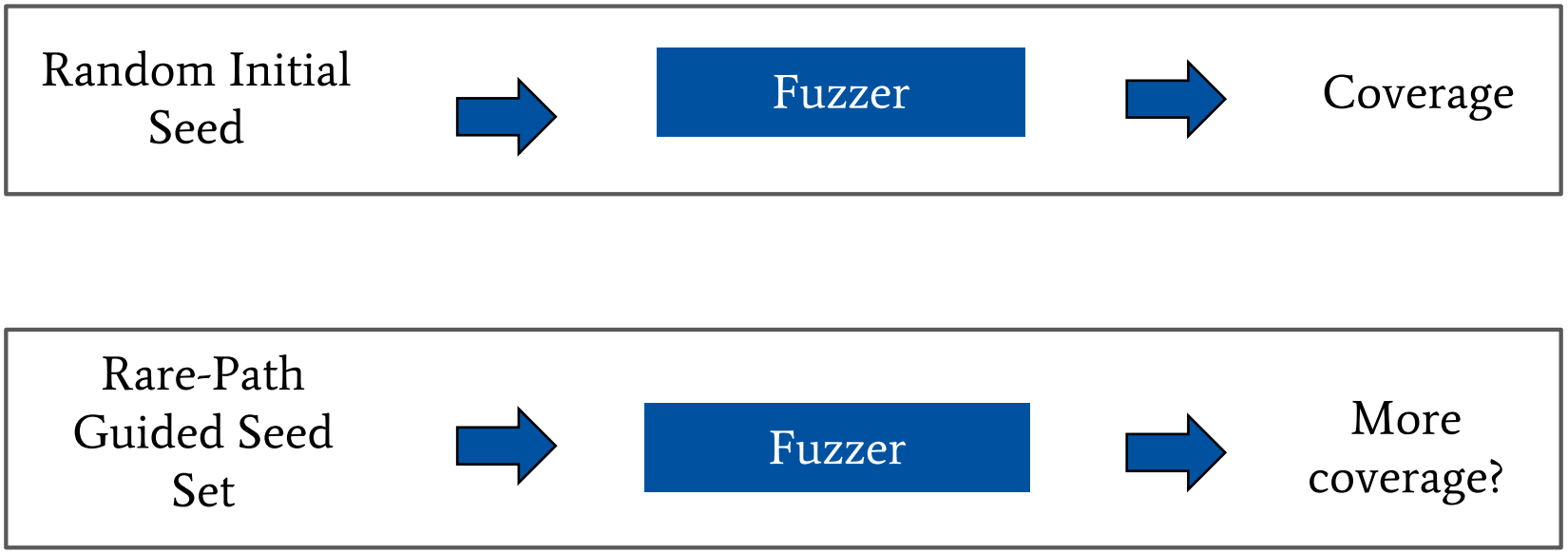
Experimental Setup and Benchmarks

UCSB

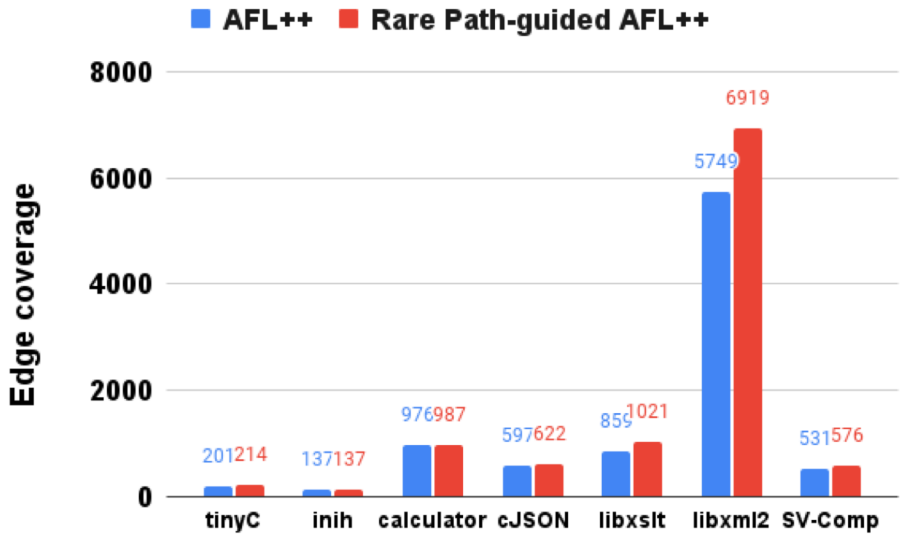
- We ran a fuzzer with a random seed for 24 hours
- We used maximum of 25% time for rare seed generation (6 hours) and ran the fuzzer for the remaining 75% time (18 hours) for fuzzing

Benchmarks	Lines of Code
tinyC	190
inih	243
calculator	1312
cJSON	3845
libxslt	33371
libxml2	186116
SV-Comp (seq-mthreded)	1016 (avg.)

Experimental Evaluation



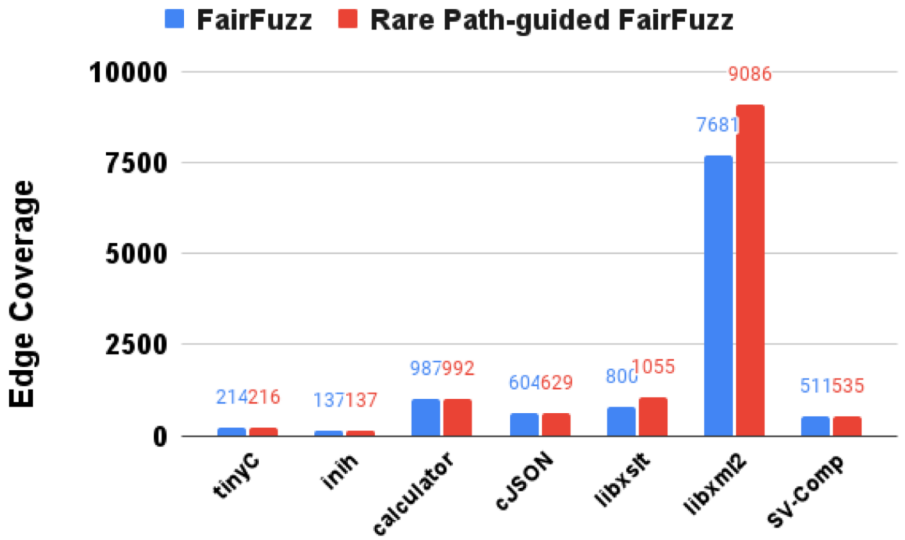
Coverage Improvement



% coverage improvement	6.47%	0.00%	1.33%	4.19%	18.86%	20.35%	8.47%

Coverage Improvement over AFL++

(Higher is better)

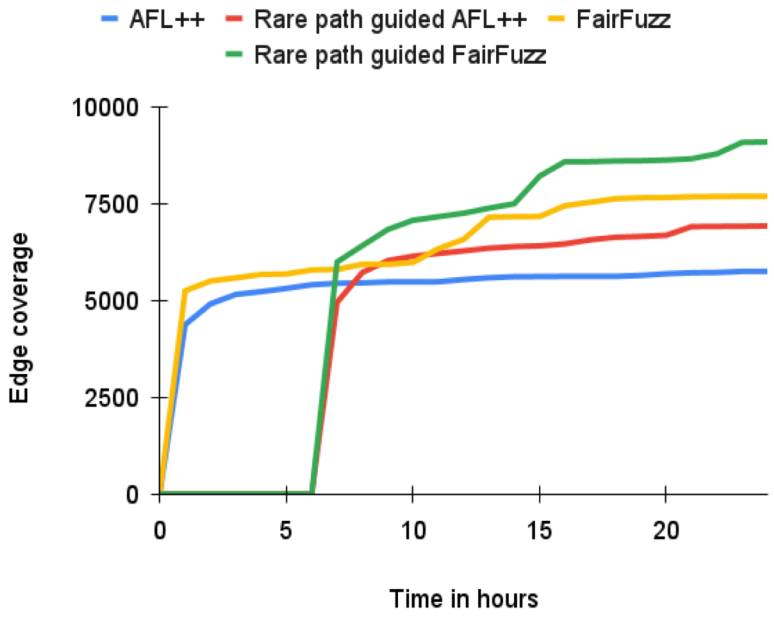


0.94%	0.00%	0.51%	4.14%	31.86%	18.29%	4.7%

Coverage Improvement over FairFuzz

(Higher is better)

Coverage Improvement



libxml2

Coverage Improvement over AFL++ and FairFuzz

(Higher is better)

Outline

- Motivation
- Software complexity
- Path complexity
- Branch selectivity
- **Conclusions and Future directions**

Conclusions

Recent results indicate that

- Path complexity can be computed in a scalable manner
- There are effective heuristics for probabilistic reachability analysis and rare path analysis
- Software complexity, verifiability, understandability are related
- Refined software complexity analysis can be used to predict performance of different types of testing and verification techniques

Future directions

- Developing a complexity metric that combines path complexity & branch selectivity
- Developing heuristics/guidance for verification/testing tools using path complexity & branch selectivity
- Using ML techniques to assess if path complexity and branch selectivity are useful features

THE END