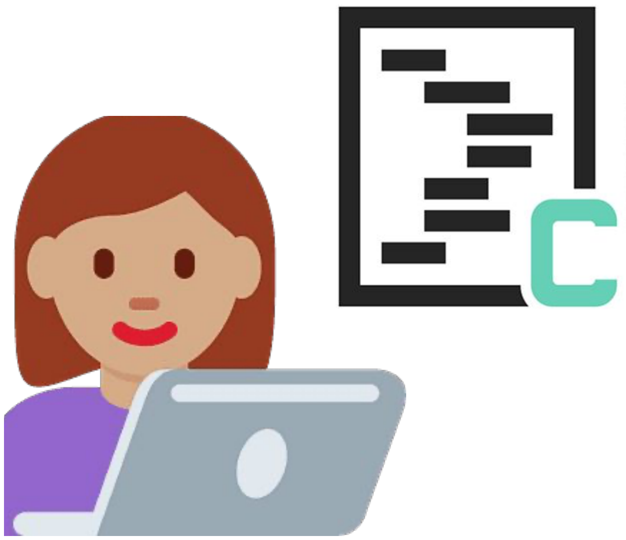
Formalizing Path Explosion for Interprocedural Functions via Asymptotic Path Complexity

Ana Beatriz Studart, Mira Kaniyur, Yuki Yang, Sangeon Park, Lucas Bang

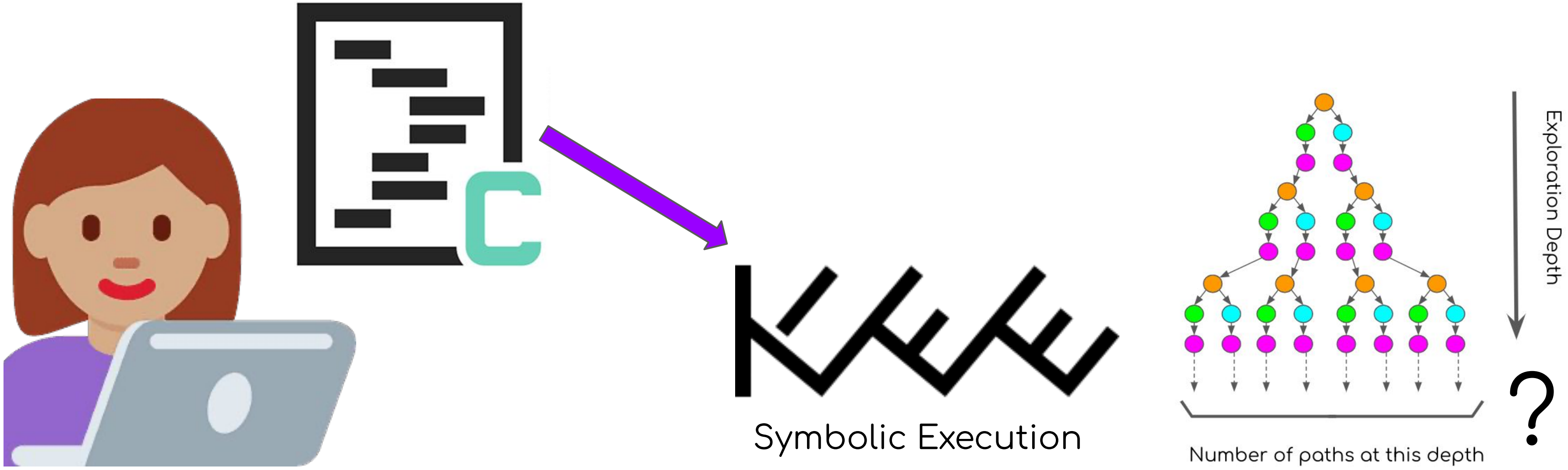
Computer Science Department
Harvey Mudd College
Claremont, California, USA

Motivation

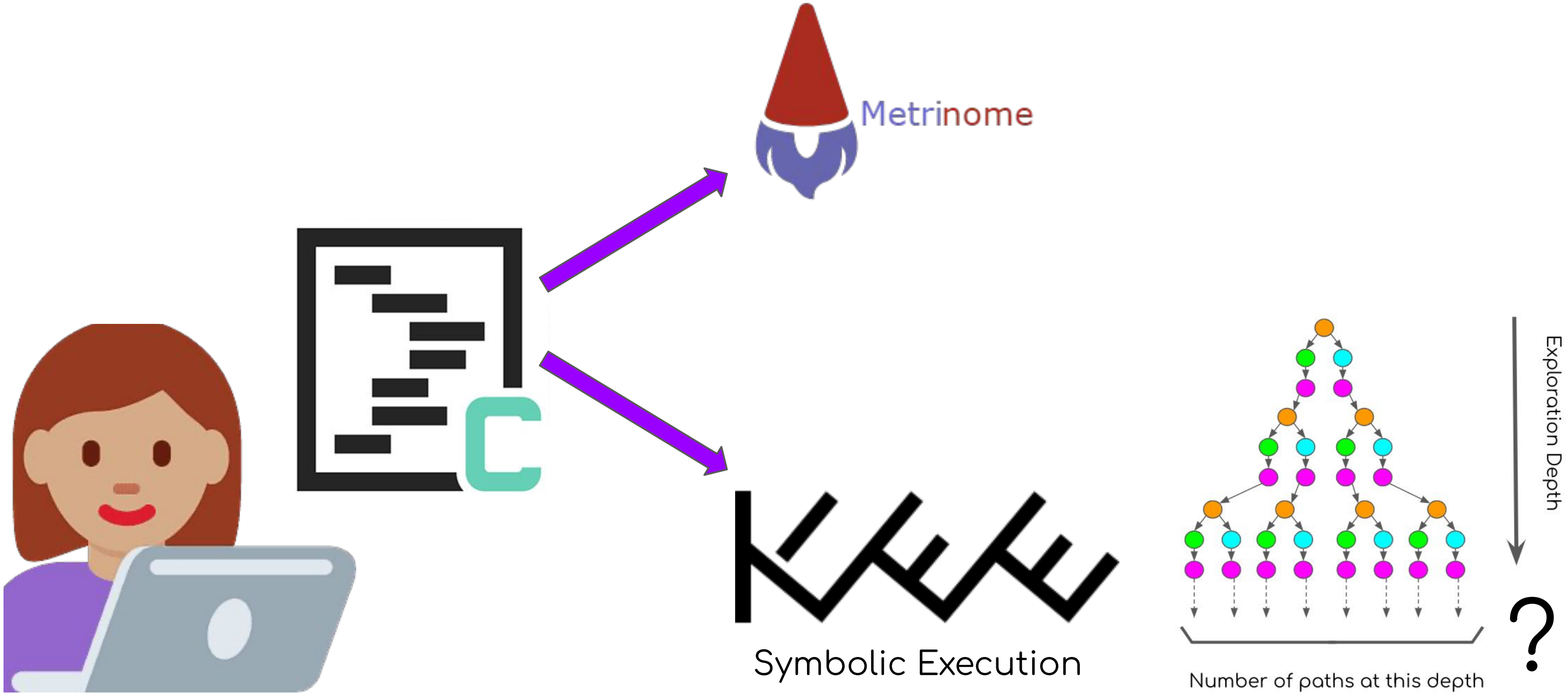
Asymptotic Path Complexity Predicts the Severity of Symbolic Execution Path Explosion



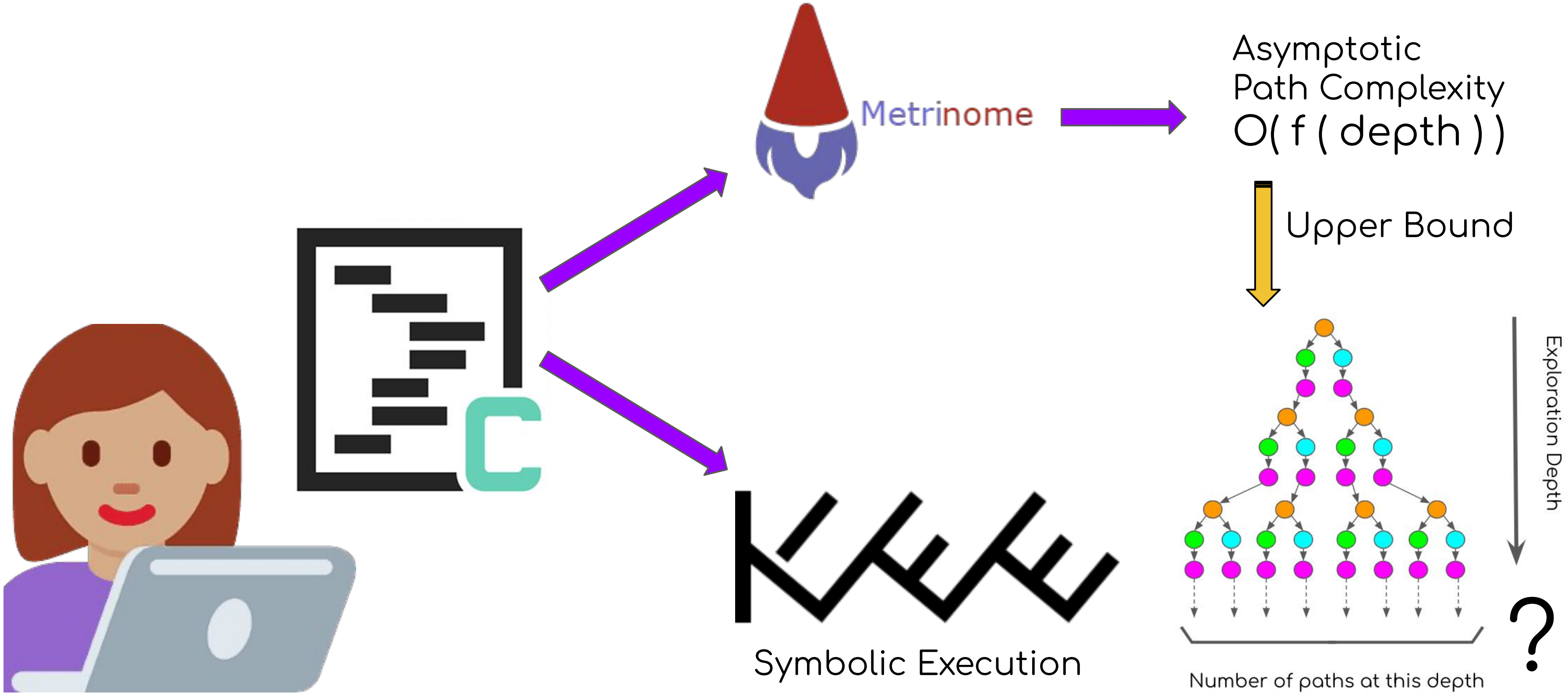
Asymptotic Path Complexity Predicts the Severity of Symbolic Execution Path Explosion



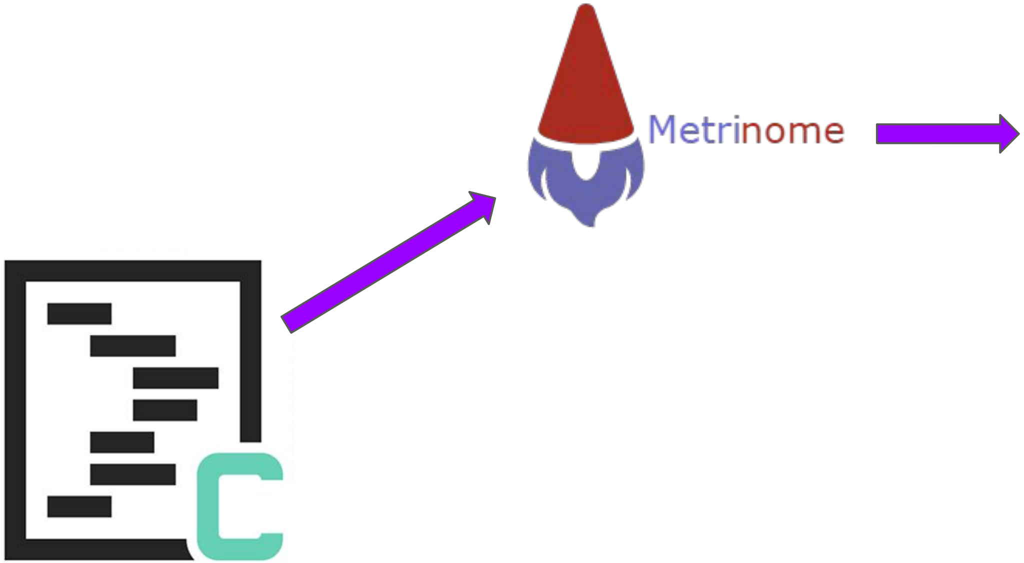
Asymptotic Path Complexity Predicts the Severity of Symbolic Execution Path Explosion



Asymptotic Path Complexity Predicts the Severity of Symbolic Execution Path Explosion

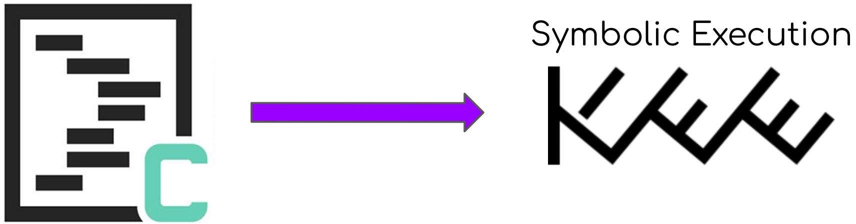


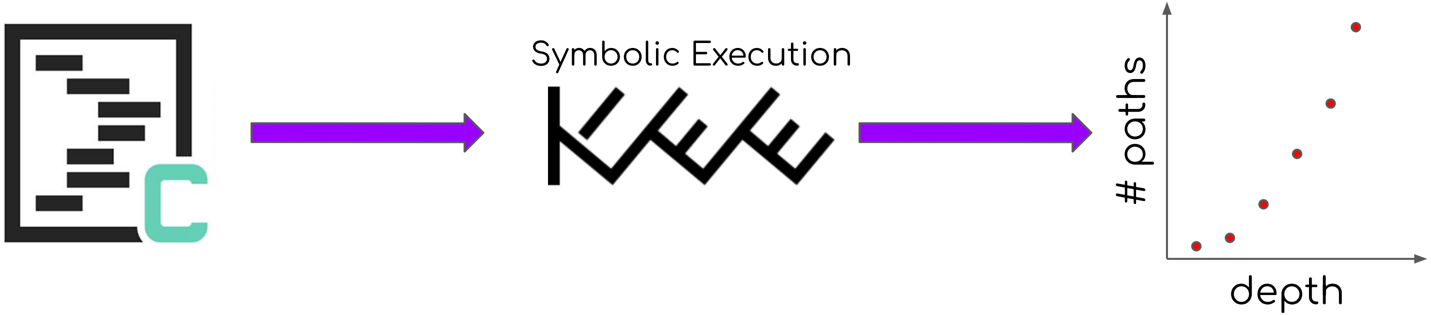
Asymptotic Path Complexity Predicts the Severity of Symbolic Execution Path Explosion

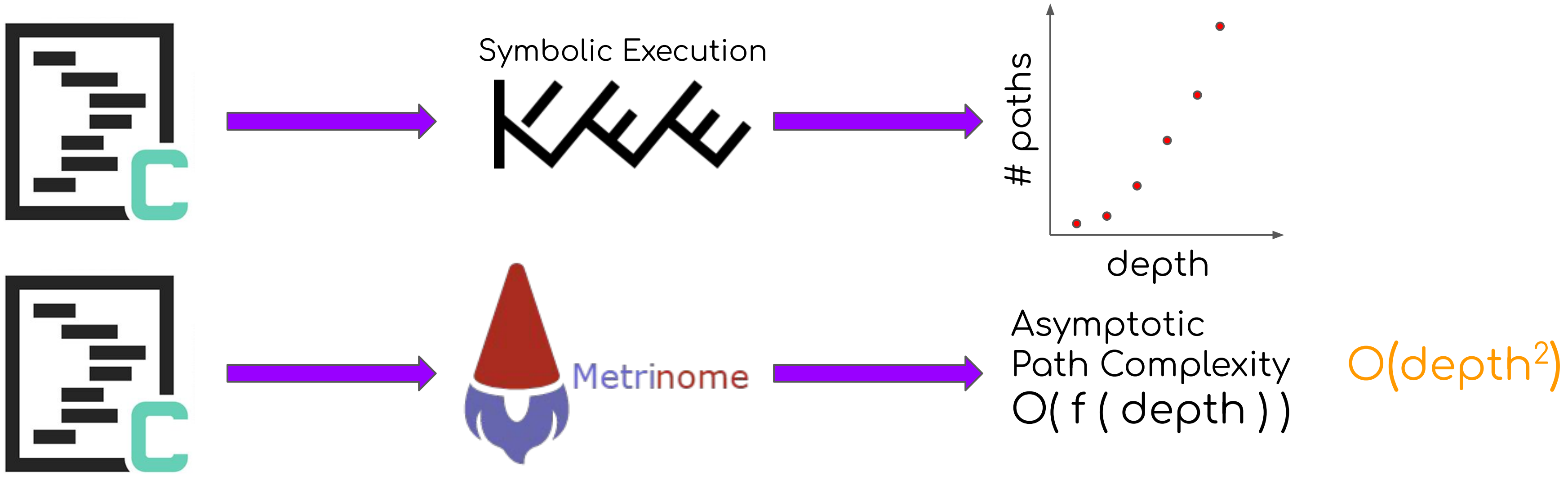


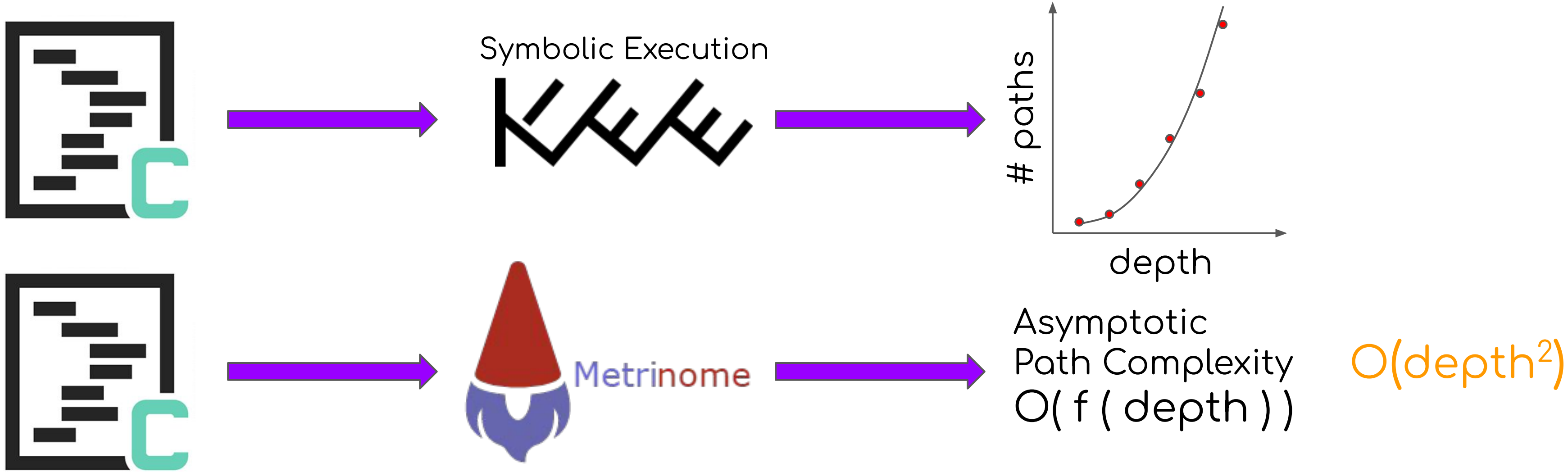
Asymptotic
Path Complexity
O(f (depth))

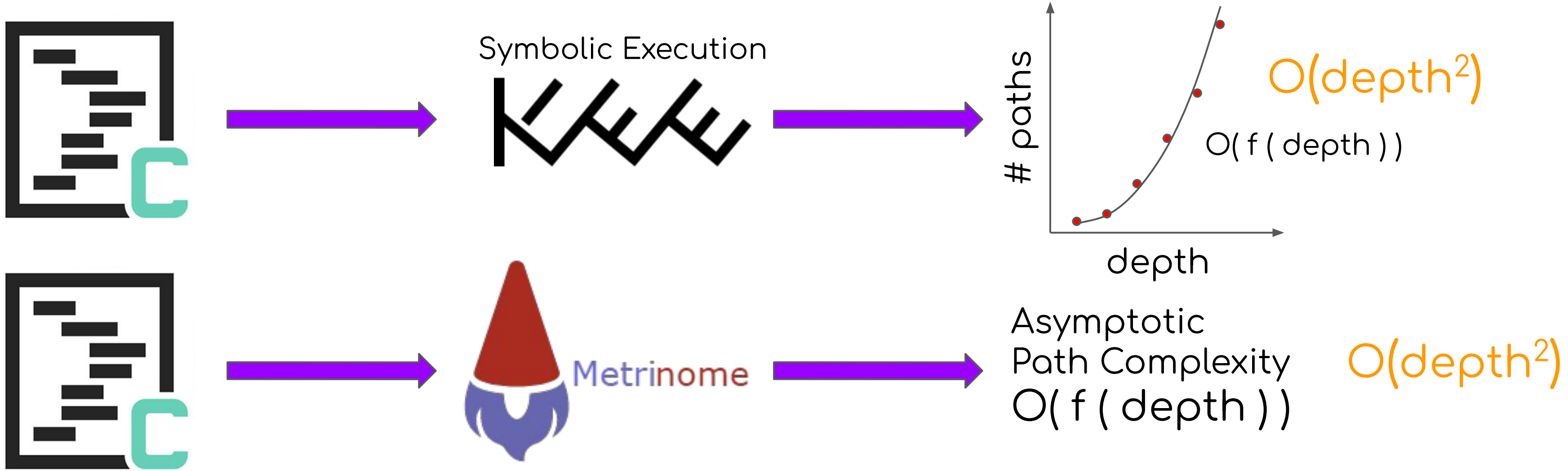
Background

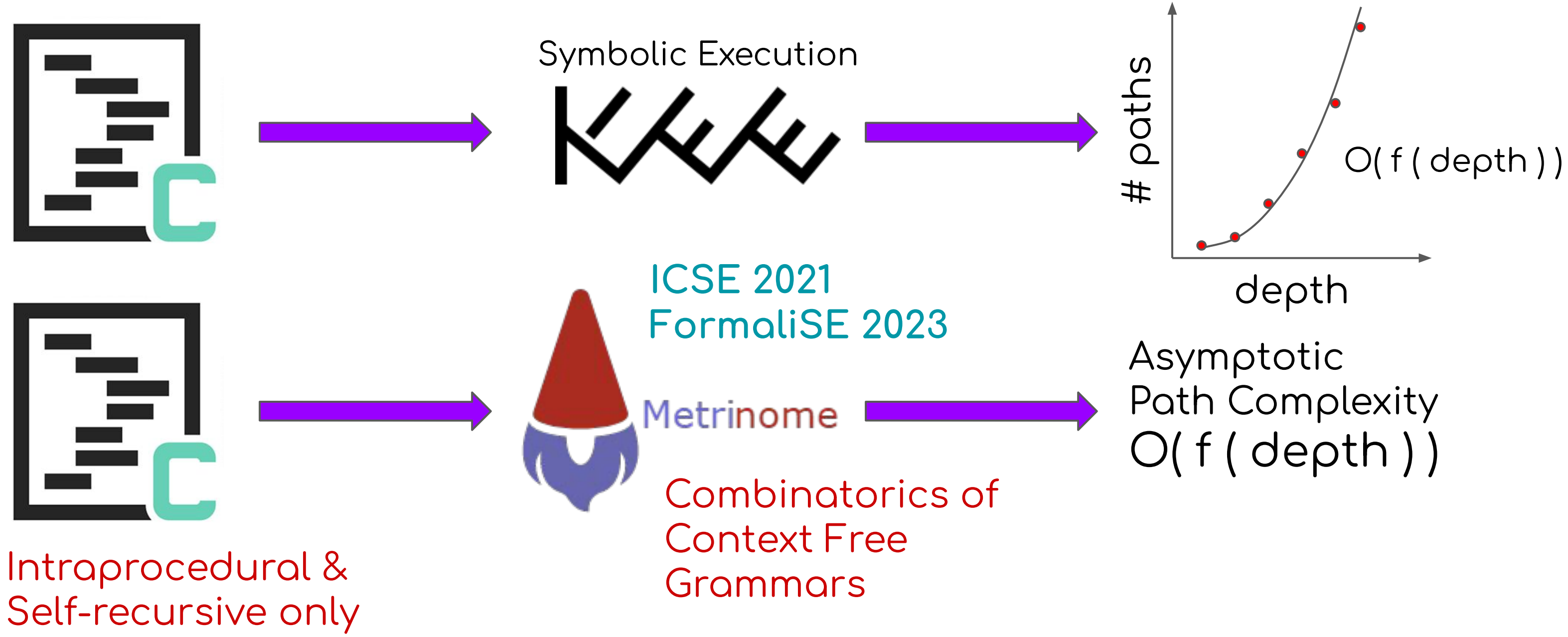

Symbolic Execution

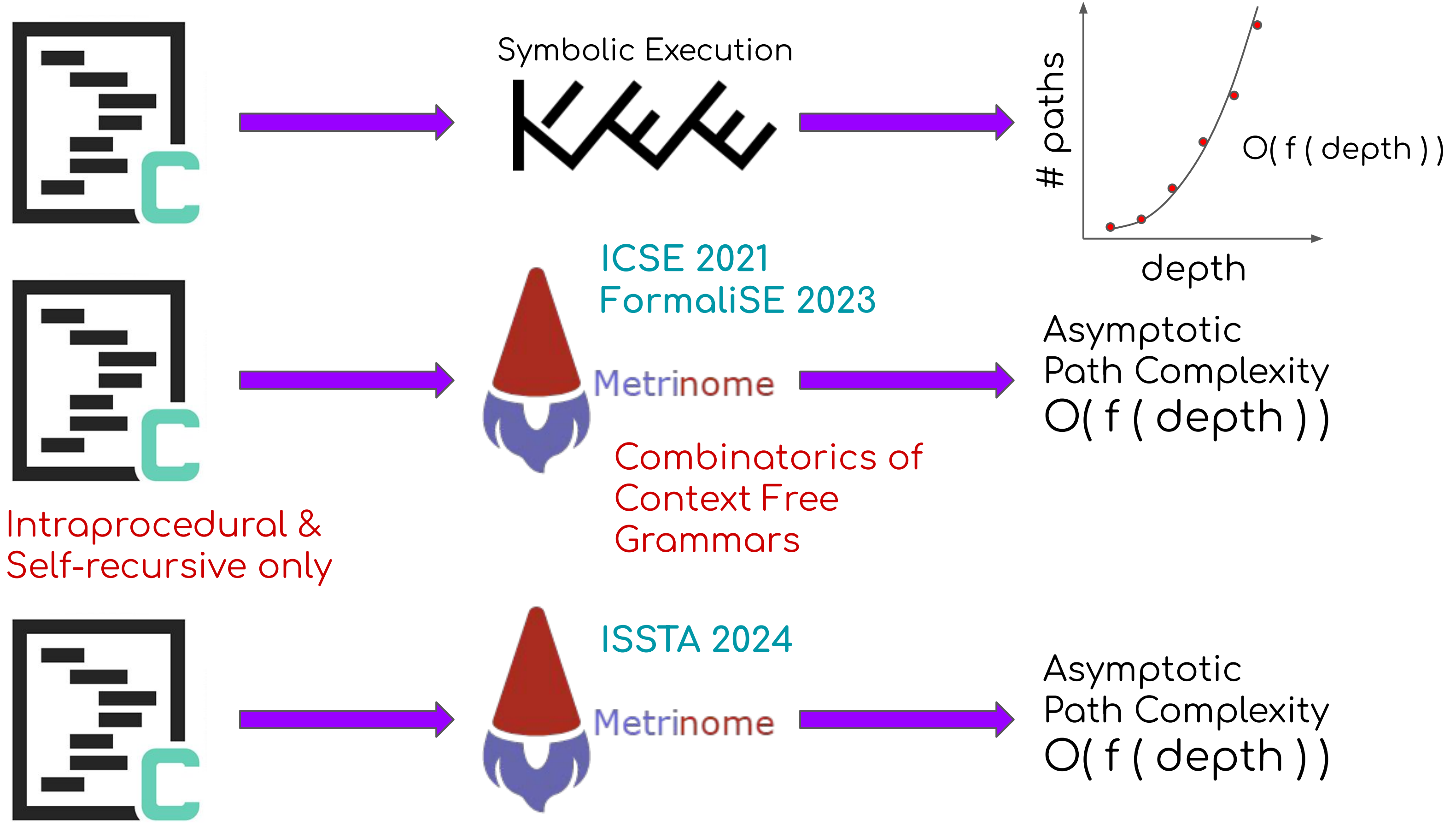

Symbolic Execution
paths
depth

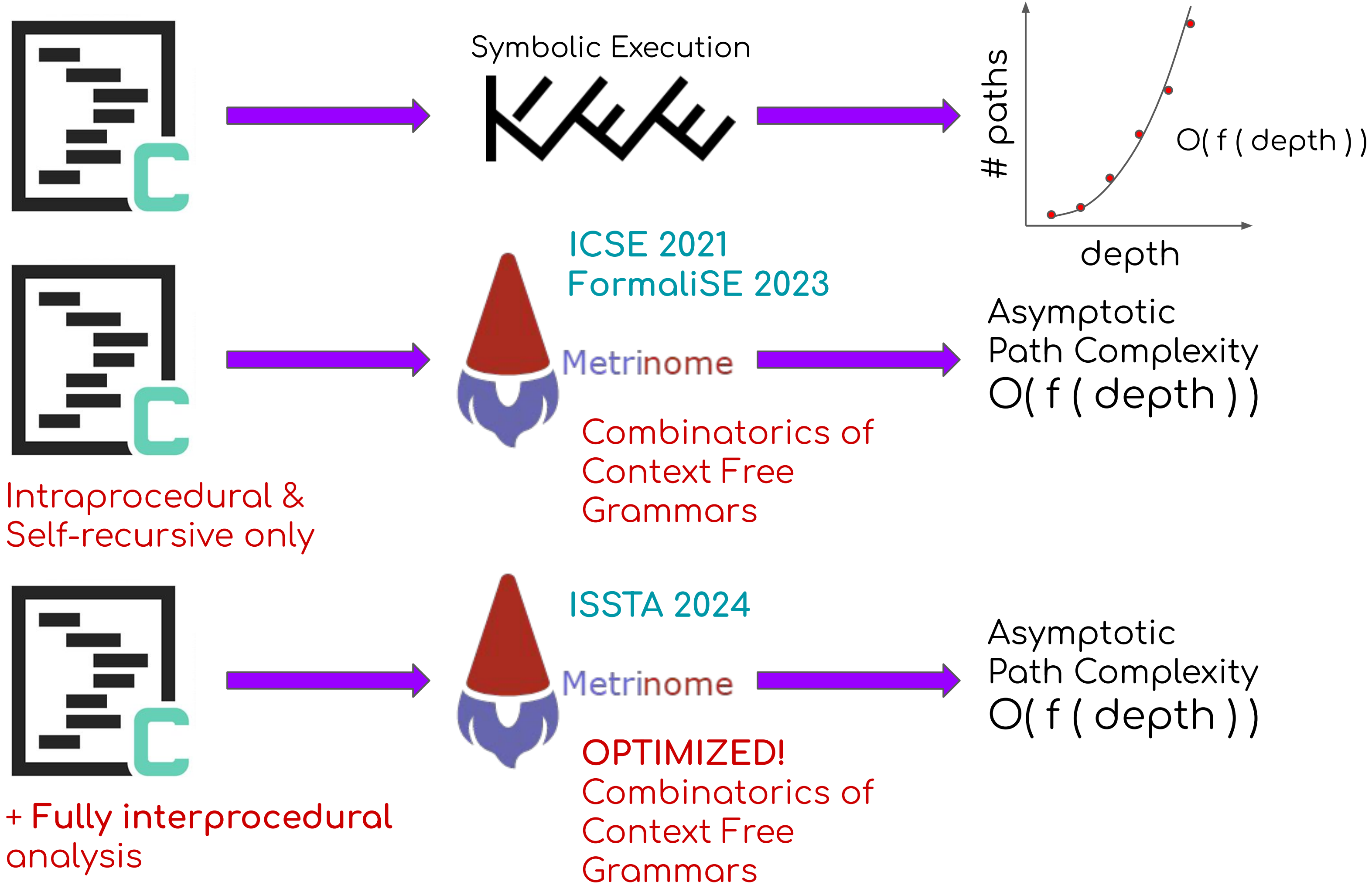

Symbolic Execution
paths
depth
Metrinome
Asymptotic
Path Complexity
O(f (depth))
O(depth2)


Symbolic Execution
KLEE
paths
depth
Metrinome
Asymptotic
Path Complexity
O(f (depth))
O(depth2)


Symbolic Execution
paths
depth
O(f (depth))
O(depth²)
Metrinome
Asymptotic
Path Complexity
O(f (depth))
O(depth²)


Symbolic Execution
paths
depth
O(f (depth))
ICSE 2021
FormaliSE 2023
Metrinome
Asymptotic
Path Complexity
O(f (depth))
Combinatorics of
Context Free
Grammars
Intraprocedural &
Self-recursive only


Symbolic Execution
# paths
O(f (depth))
depth
ICSE 2021
FormaliSE 2023
Metrinome
Asymptotic
Path Complexity
O(f (depth))
Combinatorics of
Context Free
Grammars
Intraprocedural &
Self-recursive only
ISSTA 2024
Metrinome
Asymptotic
Path Complexity
O(f (depth))



Symbolic Execution
KLEE
# paths
O(f (depth))
depth
ICSE 2021
FormaliSE 2023
Metrinome
Asymptotic
Path Complexity
O(f (depth))
Combinatorics of
Context Free
Grammars
Intraprocedural &
Self-recursive only
ISSTA 2024
Metrinome
Asymptotic
Path Complexity
O(f (depth))
OPTIMIZED!
Combinatorics of
Context Free
Grammars
+ Fully interprocedural
analysis

APC-IP (Interprocedural Asymptotic Path Complexity)

APC-IP subsumes earlier APC work

- produces the same results on the simpler benchmarks

APC-IP extends earlier APC work

- handles fully interprocedural code, unlike previous work

APC-IP outperforms earlier APC work

- much faster when it matters

APC-IP predicts symbolic execution explosion rate

- upper bound on execution paths explored by KLEE

What is Path Complexity?

Path Complexity

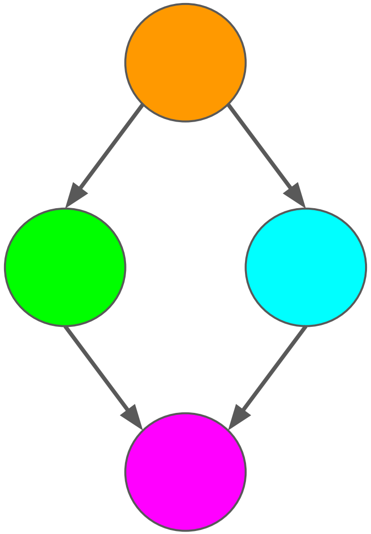
Asymptotic upper bound on the

number of paths in control flow graph from start to exit

parameterized by execution depth.

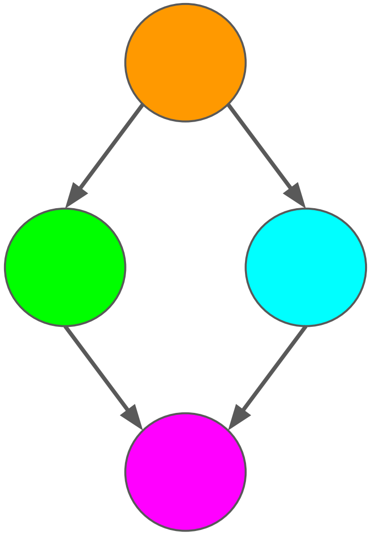
Path Complexity Quantifies Path Explosion

Control Flow Graph (CFG)



Path Complexity Quantifies Path Explosion

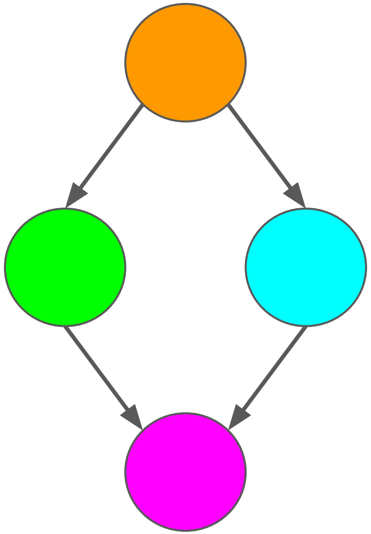
Control Flow Graph (CFG)



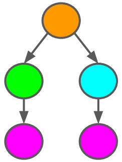
Symbolic Execution Tree

Path Complexity Quantifies Path Explosion

Control Flow Graph (CFG)

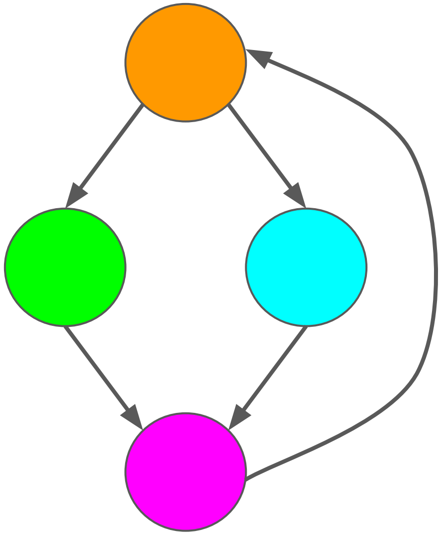


Symbolic Execution Tree

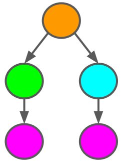


Path Complexity Quantifies Path Explosion

Control Flow Graph (CFG)

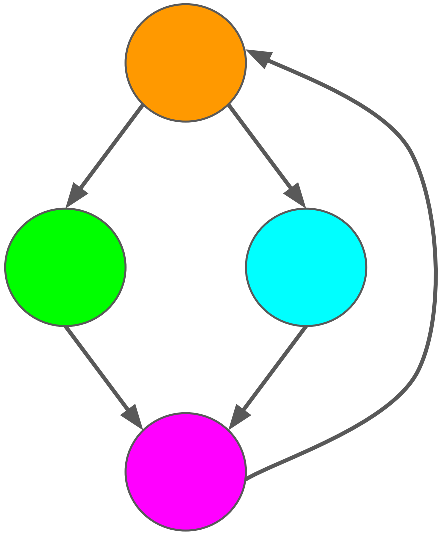


Symbolic Execution Tree

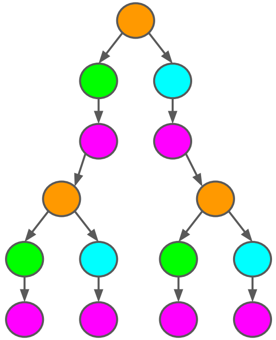


Path Complexity Quantifies Path Explosion

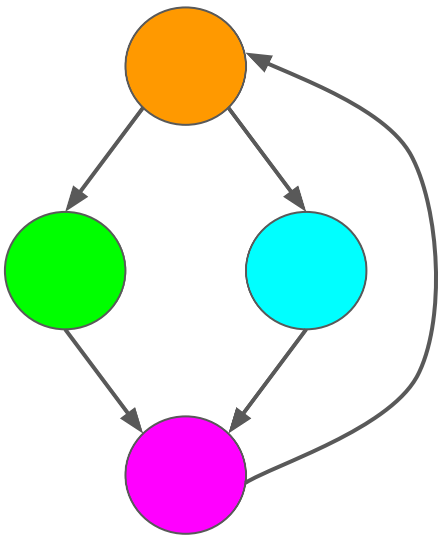
Control Flow Graph (CFG)

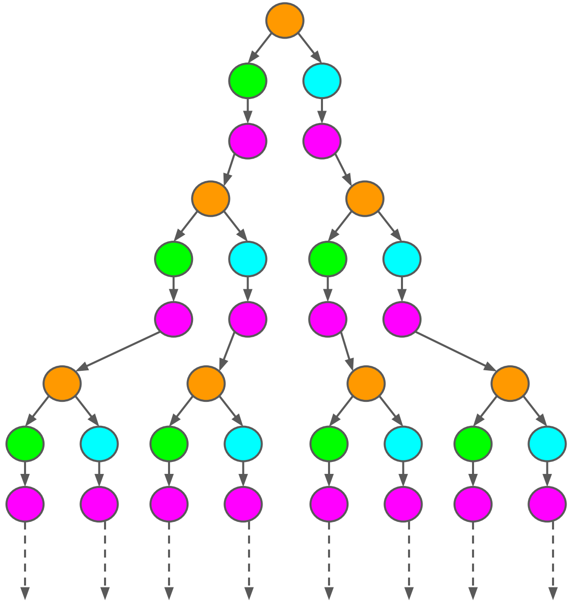


Symbolic Execution Tree



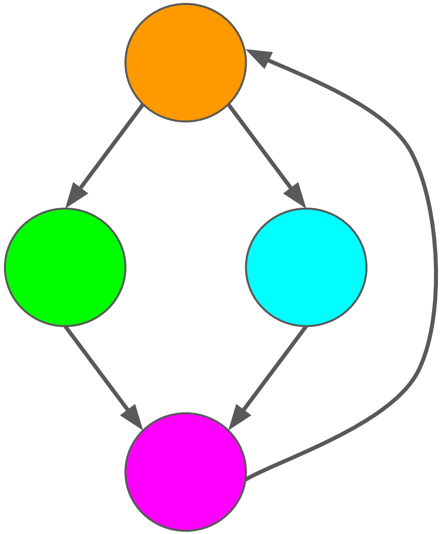
Path Complexity Quantifies Path Explosion



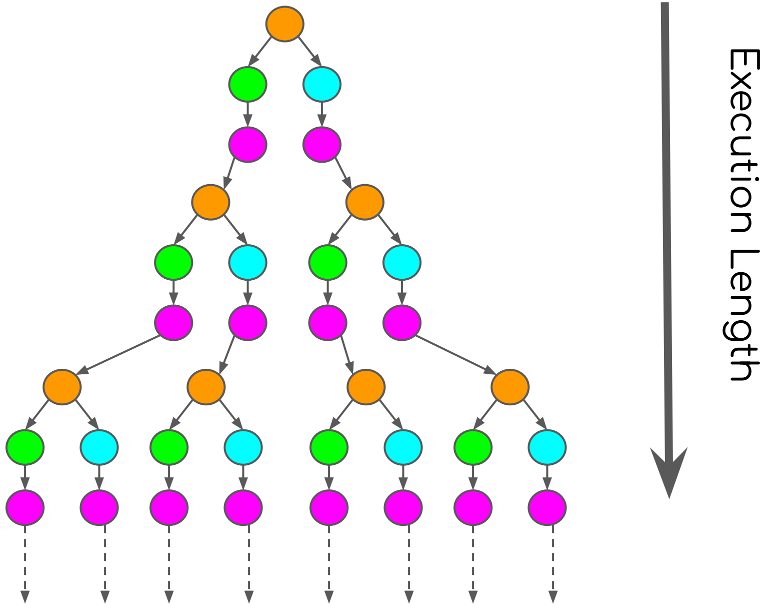


Path Complexity Quantifies Path Explosion

Control Flow Graph (CFG)

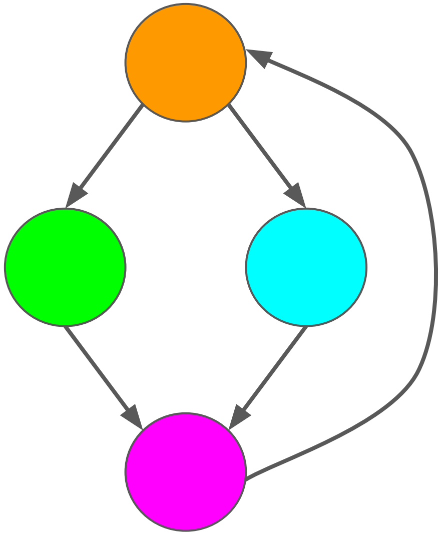


Symbolic Execution Tree

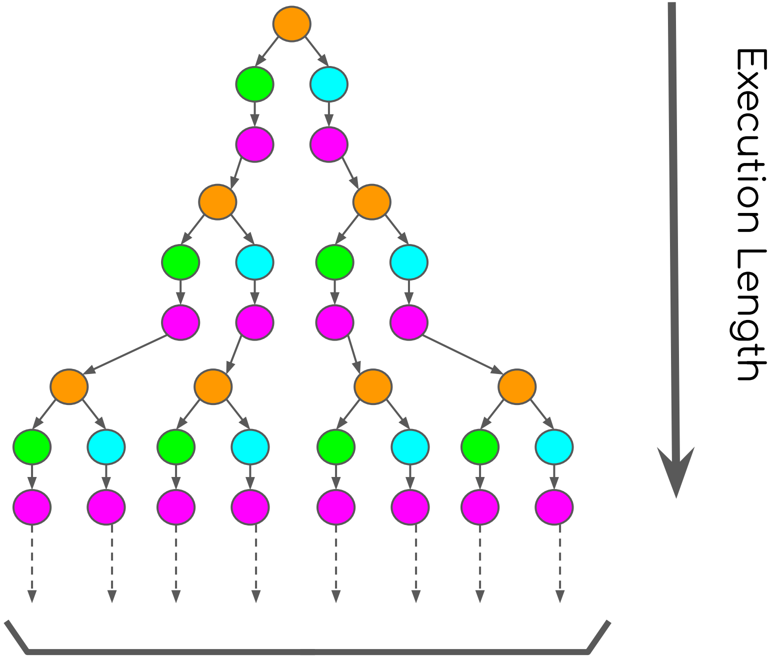


Path Complexity Quantifies Path Explosion

Control Flow Graph (CFG)



Symbolic Execution Tree



Number of paths at this length

$$2^{(length\ +\ 1)/3} = O(1.26^{length})$$

APC with Recursive Functions

APC-R

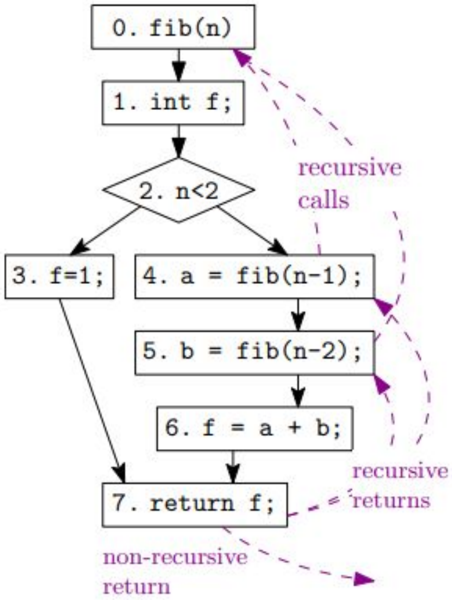
(FormaliSE 2023)

APC-R: Code → Control Flow Graph → Grammar

```
int fib(int n){
    int f;
    if (n < 2)
    f = 1;
    else {int a = fib(n - 1);
          int b = fib(n - 2);
          f = a + b; }
    return f; }
```

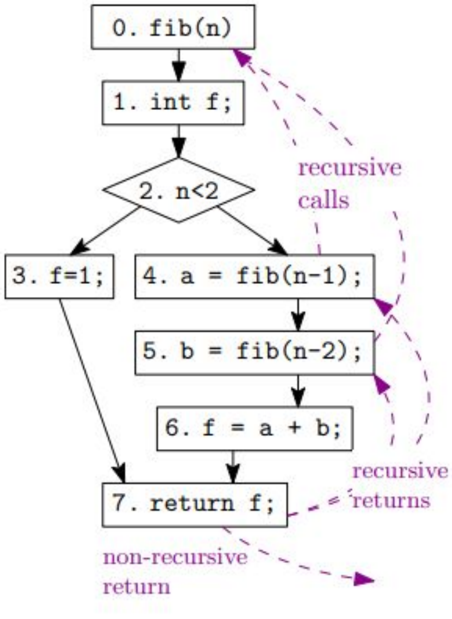
APC-R: Code → Control Flow Graph → Grammar

```
int fib(int n){
    int f;
    if (n < 2)
    f = 1;
    else {int a = fib(n - 1);
          int b = fib(n - 2);
          f = a + b; }
    return f; }
```



APC-R: Code → Control Flow Graph → Grammar

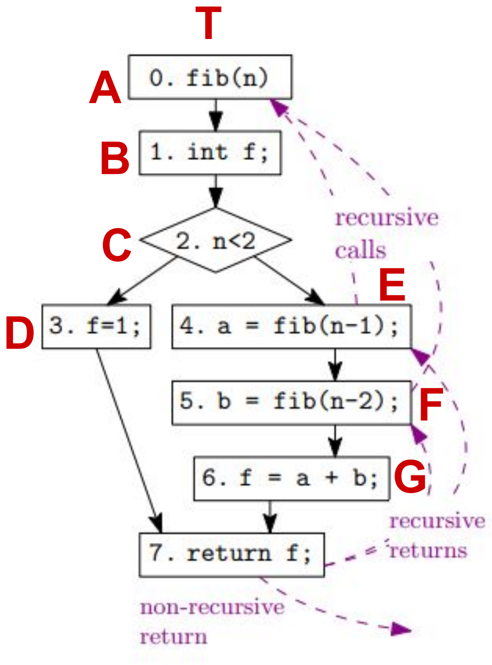
```
0. int fib(int n){
1.     int f;
2.     if (n < 2)
3.     f = 1;
4.     else {int a = fib(n - 1);
5.           int b = fib(n - 2);
6.           f = a + b; }
7.     return f; }
```



Terminals are node numbers
0, 1, 2, 3, 4, 5, 6, 7

APC-R: Code → Control Flow Graph → Grammar

```
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; }
```

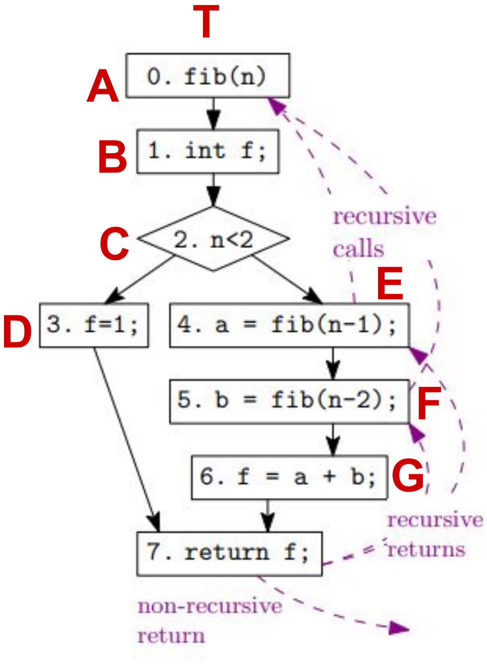


Terminals are node numbers
0, 1, 2, 3, 4, 5, 6, 7

Variables represent “all possible paths following from that node”
T, A, B, C, D, E, F, G

APC-R: Code → Control Flow Graph → Grammar

```
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; }
```



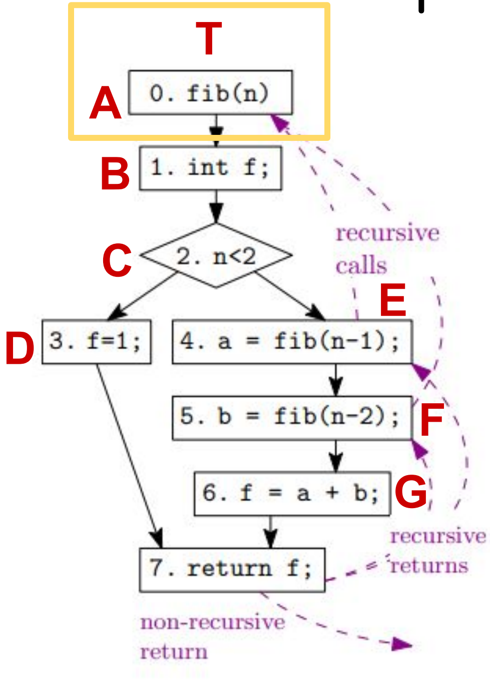
T → 0A
A → 1B
B → 2C
C → 3D | 4E
D → 7
E → T5F
F → T6G
G → 7

Terminals are node numbers
0, 1, 2, 3, 4, 5, 6, 7

Variables represent “all possible paths following from that node”
T, A, B, C, D, E, F, G

APC-R: Code → Control Flow Graph → Grammar

```
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; }
```



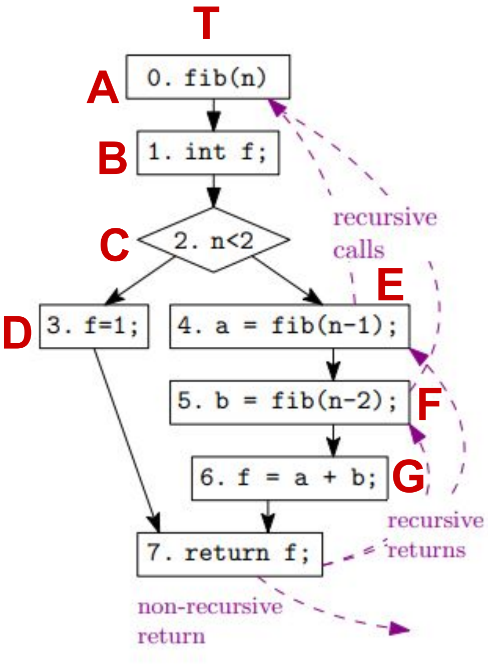
T → 0A
A → 1B
B → 2C
C → 3D | 4E
D → 7
E → T5F
F → T6G
G → 7

Terminals are node numbers
0, 1, 2, 3, 4, 5, 6, 7

Variables represent "all possible paths following from that node"
T, A, B, C, D, E, F, G

APC-R: Code → Control Flow Graph → Grammar

```
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; }
```



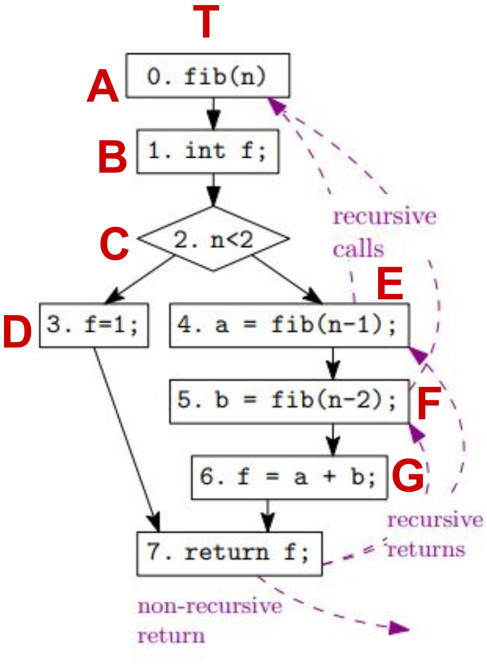
T → 0A
A → 1B
B → 2C
C → 3D | 4E
D → 7
E → T5F
F → T6G
G → 7

Terminals are node numbers
0, 1, 2, 3, 4, 5, 6, 7

Variables represent "all possible paths following from that node"
T, A, B, C, D, E, F, G

APC-R: Code → Control Flow Graph → Grammar

```
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; }
```



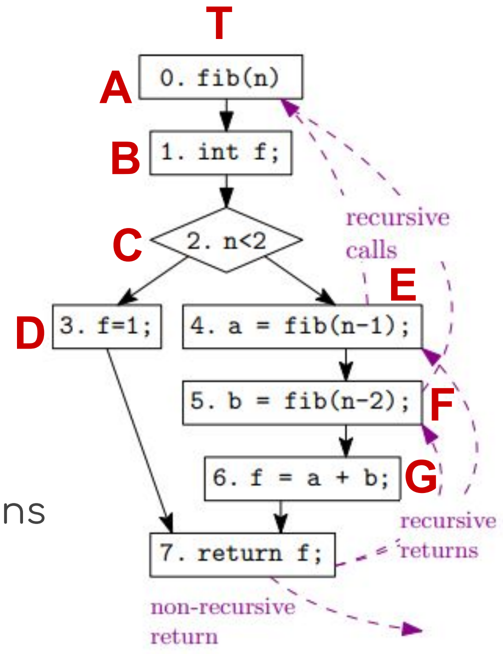
T → 0A
A → 1B
B → 2C
C → 3D | 4E
D → 7
E → T5F
F → T6G
G → 7

Terminals are node numbers
0, 1, 2, 3, 4, 5, 6, 7

Variables represent "all possible paths following from that node"
T, A, B, C, D, E, F, G

APC-R: Code → Control Flow Graph → Grammar

```
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; }
```

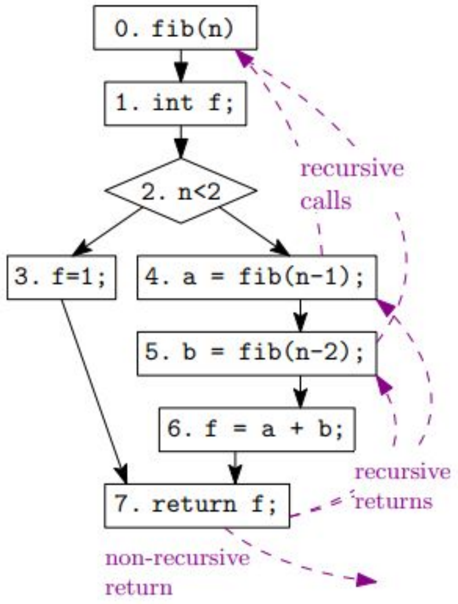


T → 0A
A → 1B
B → 2C
C → 3D | 4E
D → 7
E → T5F
F → T6G
G → 7

Strings From Grammar ≅ Code Executions

01237
0124012375012376 7
01240124012375012376750123767
01240123750124012375012376767
...

Path Complexity from Grammar



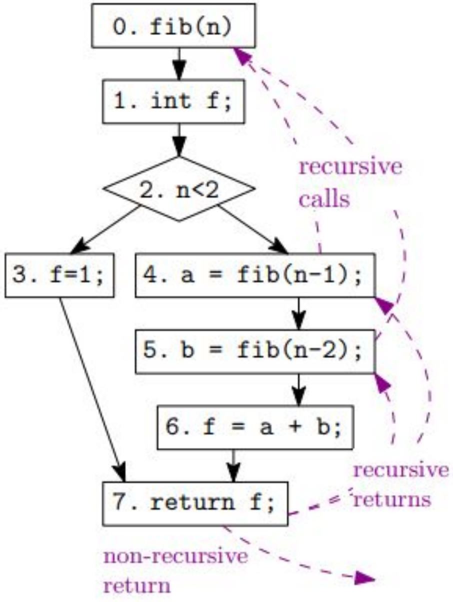
Control Flow Graph

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

Grammar

System of Equations

Path Complexity from Grammar



Control Flow Graph

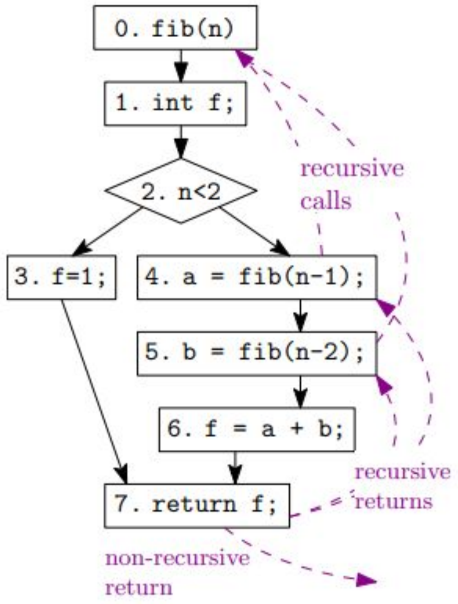
```
T → 0A
A → 1B
B → 2C
C → 3D | 4E
D → 7
E → T5F
F → T6G
G → 7
```

Grammar

Chomsky–Schützenberger theorem ⟹

System of Equations

Path Complexity from Grammar



Control Flow Graph

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

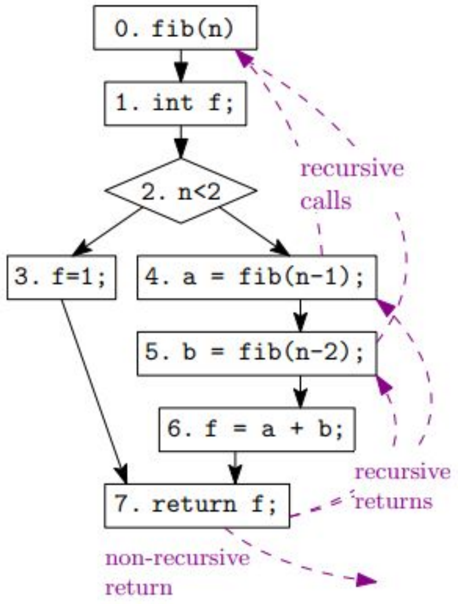
Grammar

Chomsky–Schützenberger theorem ⟹

→	↦	=
t	↦	z
|	↦	+

System of Equations

Path Complexity from Grammar



Control Flow Graph

$$
\begin{aligned}
T &\to 0A \\
A &\to 1B \\
B &\to 2C \\
C &\to 3D \mid 4E \\
D &\to 7 \\
E &\to T5F \\
F &\to T6G \\
G &\to 7
\end{aligned}
$$

Grammar

Chomsky–Schützenberger theorem

$\to \ \mapsto\ =$

$t \ \mapsto\ z$

$| \ \mapsto\ +$

$$
\begin{aligned}
T &= Az \\
A &= Bz \\
B &= Cz \\
C &= Dz + Ez \\
D &= z \\
E &= TFz \\
F &= TGz \\
G &= z
\end{aligned}
$$

System of Equations

Path Complexity from Grammar

```
T = Az
A = Bz
B = Cz
C = Dz + Ez
D = z
E = TFz
F = TGz
G = z
```

Path Complexity from Grammar

```
T = Az
A = Bz
B = Cz
C = Dz + Ez
D = z
E = TFz
F = TGz
G = z
```

Eliminate Variables to Isolate T

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

Path Complexity from Grammar

```
T = Az
A = Bz
B = Cz
C = Dz + Ez
D = z
E = TFz
F = TGz
G = z
```

Eliminate Variables to Isolate T

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

Compute Discriminant

$$1 - 4z^{12} = 0$$

$$\mathrm{Disc}(p) = \frac{(-1)^{n(n-1)/2}}{a_n} \mathrm{Res}\left(p, \frac{d}{dx}p\right)$$

$$\mathrm{Res}(p,q) = \begin{vmatrix} a_0 & 0 & \cdots & 0 & b_0 & 0 & \cdots & 0 \\ a_1 & a_0 & \cdots & 0 & b_1 & b_0 & \cdots & 0 \\ a_2 & a_1 & \ddots & 0 & b_2 & b_1 & \ddots & 0 \\ \vdots & \vdots & \ddots & a_0 & \vdots & \vdots & \ddots & b_0 \\ a_d & a_{d-1} & \cdots & \vdots & b_e & b_{e-1} & \cdots & \vdots \\ 0 & a_d & \ddots & \vdots & 0 & b_e & \ddots & \vdots \\ \vdots & \vdots & \ddots & a_{d-1} & \vdots & \vdots & \ddots & b_{e-1} \\ 0 & 0 & \cdots & a_d & 0 & 0 & \cdots & b_e \end{vmatrix}$$

Path Complexity from Grammar

```
T = Az
A = Bz
B = Cz
C = Dz + Ez
D = z
E = TFz
F = TGz
G = z
```

Eliminate Variables to Isolate T

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

Compute Discriminant

$1 - 4z^{12} = 0$

Compute Roots

$4^{1/12}e^{k \pi i / 6}$ for $k = 1 \ldots 12$

$$\mathrm{Disc}(p) = \frac{(-1)^{n(n-1)/2}}{a_n}\mathrm{Res}\left(p, \frac{d}{dx}p\right)$$

$$\mathrm{Res}(p,q) = \begin{vmatrix} a_0 & 0 & \cdots & 0 & b_0 & 0 & \cdots & 0 \\ a_1 & a_0 & \cdots & 0 & b_1 & b_0 & \cdots & 0 \\ a_2 & a_1 & \ddots & 0 & b_2 & b_1 & \ddots & 0 \\ \vdots & \vdots & \ddots & a_0 & \vdots & \vdots & \ddots & b_0 \\ a_d & a_{d-1} & \cdots & \vdots & b_e & b_{e-1} & \cdots & \vdots \\ 0 & a_d & \ddots & \vdots & 0 & b_e & \ddots & \vdots \\ \vdots & \vdots & \ddots & a_{d-1} & \vdots & \vdots & \ddots & b_{e-1} \\ 0 & 0 & \cdots & a_d & 0 & 0 & \cdots & b_e \end{vmatrix}$$

Path Complexity from Grammar

```
T = Az
A = Bz
B = Cz
C = Dz + Ez
D = z
E = TFz
F = TGz
G = z
```

Eliminate Variables to Isolate T

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

Compute Discriminant

$1 - 4z^{12} = 0$

Compute Roots

$4^{1/12}e^{k \pi i / 6}$ for $k = 1 \ldots 12$

Asymptotic Path Complexity

APC-R = $4^{n/12} = 1.12^n$

$$\mathrm{Disc}(p) = \frac{(-1)^{n(n-1)/2}}{a_n} \mathrm{Res}\left(p, \frac{d}{dx}p\right)$$

$$\mathrm{Res}(p,q) = \begin{vmatrix} a_0 & 0 & \cdots & 0 & b_0 & 0 & \cdots & 0 \\ a_1 & a_0 & \cdots & 0 & b_1 & b_0 & \cdots & 0 \\ a_2 & a_1 & \ddots & 0 & b_2 & b_1 & \ddots & 0 \\ \vdots & \vdots & \ddots & a_0 & \vdots & \vdots & \ddots & b_0 \\ a_d & a_{d-1} & \cdots & \vdots & b_e & b_{e-1} & \cdots & \vdots \\ 0 & a_d & \ddots & \vdots & 0 & b_e & \ddots & \vdots \\ \vdots & \vdots & \ddots & a_{d-1} & \vdots & \vdots & \ddots & b_{e-1} \\ 0 & 0 & \cdots & a_d & 0 & 0 & \cdots & b_e \end{vmatrix}$$

$$f(n) = \sum_{i=1}^{D} \sum_{j=0}^{m_i - 1} c_{i,j} n^j \left(\frac{1}{|r_i|}\right)^n$$

Naive Interprocedural APC

NAPC-IP

Idea! Use APC-R for Interprocedural Code!

Ideally, we could just apply the same principles from APC-R directly to interprocedural code.

We call this approach Naive Interprocedural Asymptotic Path Complexity

NAPC-IP

Naive

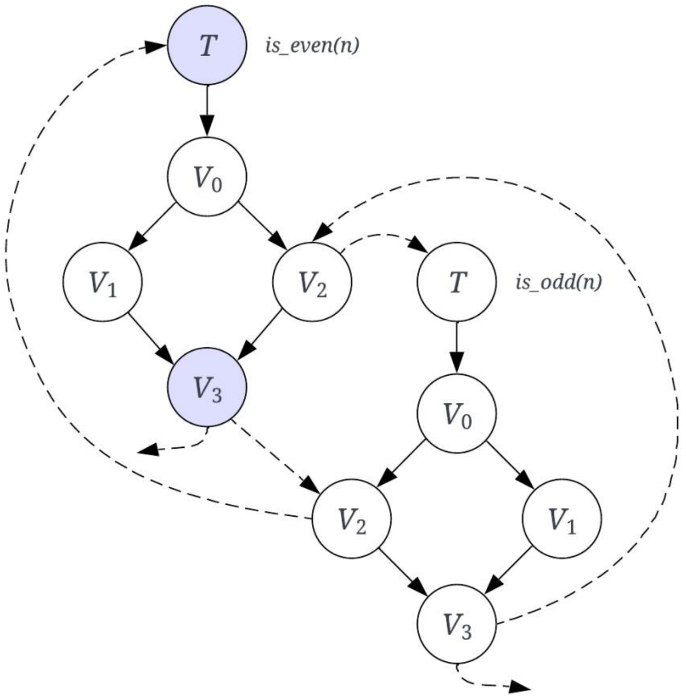
Asymptotic Path Complexity

Inter-Procedural

NAPC-IP intuition

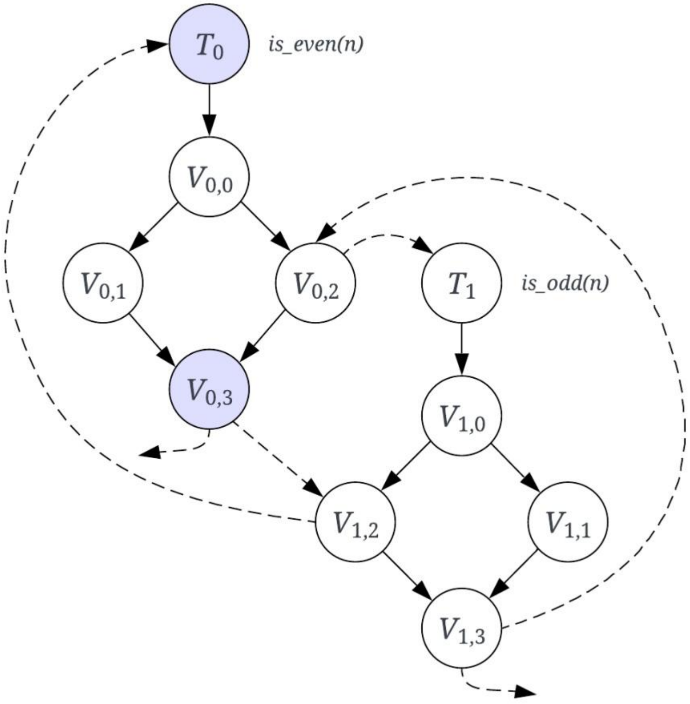
```
bool is_even(int n){
    if (n == 0) return true;
    else return is_odd(n - 1);
}


bool is_odd(int n){
    if (n == 0) return false;
    else return is_even(n - 1);
}
```



NAPC-IP intuition

- Need to distinguish between functions
- Add an extra subscript to each node label
- Treat interprocedural calls just like recursive calls
- Now solve for T_0



NAPC-IP intuition

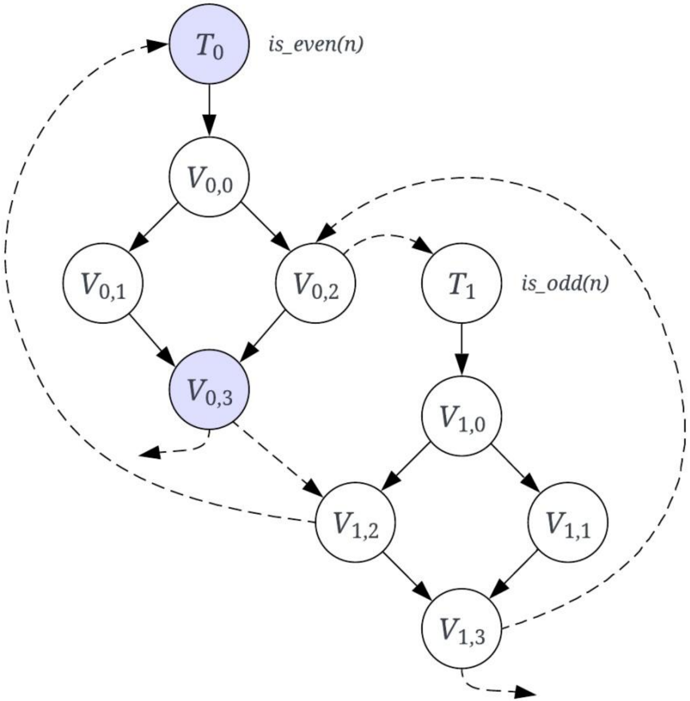
- Apply APC-R algorithms

System 0 (is_even)

$T_0 = V_{0,0}x$
$V_{0,0} = V_{0,1}x + V_{0,2}x$
$V_{0,1} = V_{0,3}x$
$V_{0,2} = T_1 V_{0,3}x$
$V_{0,3} = 1$

System 1 (is_odd)

$T_1 = V_{1,0}x$
$V_{1,0} = V_{1,1}x + V_{1,2}x$
$V_{1,1} = V_{1,3}x$
$V_{1,2} = T_0 V_{1,3}x$
$V_{1,3} = 1$



NAPC-IP intuition

- Apply APC-R algorithms

System 0 (is_even)

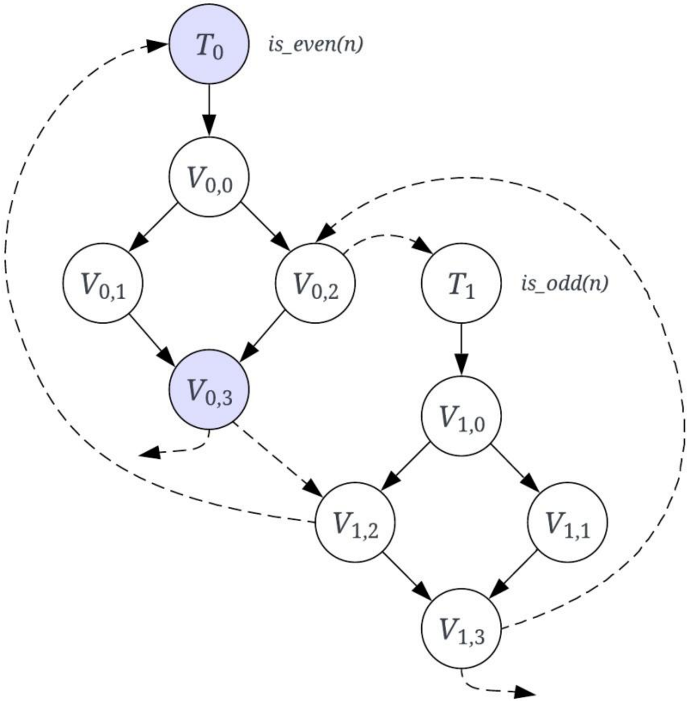
$T_0 = V_{0,0}x$
$V_{0,0} = V_{0,1}x + V_{0,2}x$
$V_{0,1} = V_{0,3}x$
$V_{0,2} = T_1 V_{0,3}x$
$V_{0,3} = 1$

System 1 (is_odd)

$T_1 = V_{1,0}x$
$V_{1,0} = V_{1,1}x + V_{1,2}x$
$V_{1,1} = V_{1,3}x$
$V_{1,2} = T_0 V_{1,3}x$
$V_{1,3} = 1$

- Use APC-R solving techniques

  APC for `is_even` is O(n/3)



NAPC-IP intuition

- Apply APC-R algorithms

System 0 (is_even)

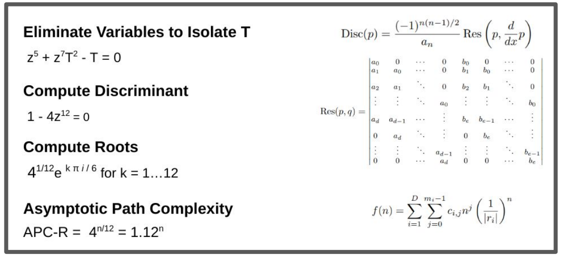
$$T_0 = V_{0,0}x$$
$$V_{0,0} = V_{0,1}x + V_{0,2}x$$
$$V_{0,1} = V_{0,3}x$$
$$V_{0,2} = T_1V_{0,3}x$$
$$V_{0,3} = 1$$

System 1 (is_odd)

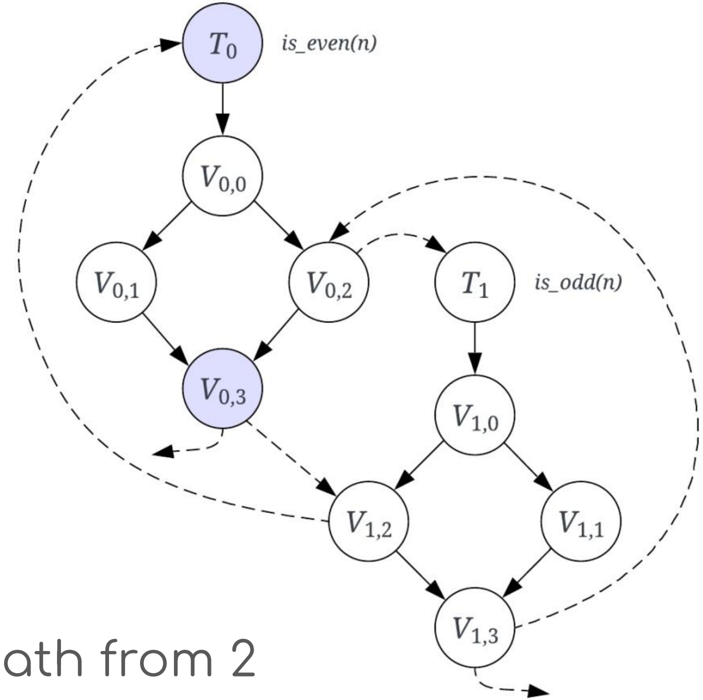
$$T_1 = V_{1,0}x$$
$$V_{1,0} = V_{1,1}x + V_{1,2}x$$
$$V_{1,1} = V_{1,3}x$$
$$V_{1,2} = T_0V_{1,3}x$$
$$V_{1,3} = 1$$

- Use APC-R solving techniques

APC for `is_even` is O(n/3)



Same math from 2 minutes ago



It works! But there is a problem!

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

Eliminate Variables to Isolate T

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

Compute Discriminant

$1 - 4z^{12} = 0$

Compute Roots

$4^{1/12}e^{k\pi i/6}$ for $k = 1\ldots12$

Asymptotic Path Complexity

$\mathrm{RAPC} = 4^{n/12} = 1.12^n$

$$\mathrm{Disc}(p) = \frac{(-1)^{n(n-1)/2}}{a_n}\mathrm{Res}\left(p, \frac{d}{dx}p\right)$$

$$\mathrm{Res}(p,q) = \begin{vmatrix} a_0 & 0 & \cdots & 0 & b_0 & 0 & \cdots & 0 \\ a_1 & a_0 & \cdots & 0 & b_1 & b_0 & \cdots & 0 \\ a_2 & a_1 & \ddots & 0 & b_2 & b_1 & \ddots & 0 \\ \vdots & \vdots & \ddots & a_0 & \vdots & \vdots & \ddots & b_0 \\ a_d & a_{d-1} & \cdots & \vdots & b_e & b_{e-1} & \cdots & \vdots \\ 0 & a_d & \ddots & \vdots & 0 & b_e & \ddots & \vdots \\ \vdots & \vdots & \ddots & a_{d-1} & \vdots & \vdots & \ddots & b_{e-1} \\ 0 & 0 & \cdots & a_d & 0 & 0 & \cdots & b_e \end{vmatrix}$$

$$f(n) = \sum_{i=1}^{D}\sum_{j=0}^{m_i-1} c_{i,j}n^j\left(\frac{1}{|r_i|}\right)^n$$

If there are even only a few interprocedural and recursive calls, system of equations is too large, METRINOME explodes and runs out of time and memory

Optimizations

APC-IP

Optimization 1: Better Combinatorial Analysis

Before

Compute APC using

directly straightforward analytic combinatorics

$$f(n) = \sum_{i=1}^{D} \sum_{j=0}^{m_i-1} c_{i,j} n^j \left(\frac{1}{|r_i|} \right)^n$$

Slow!

Optimization 1: Better Combinatorial Analysis

Before

Compute APC using

directly straightforward analytic combinatorics

$$f(n) = \sum_{i=1}^{D} \sum_{j=0}^{m_i - 1} c_{i,j} n^j \left(\frac{1}{|r_i|} \right)^n$$

Slow!

After

Compute APC using

Generalized Expansion Theorem for Rational Generating Functions

GETRGF

Fast!

General Expansion Theorem for Rational Generating Functions (GETRGF)

If $g(x) = P(x)/Q(x)$, *where* $Q(x) = q_0(1-\rho_1 x)^{d_1}(1-\rho_2 x)^{d_2}\ldots(1-\rho_t x)^{d_t}$ *and the numbers* $(\rho_1, \rho_2, \ldots, \rho_t)$ *are distinct, and if* $P(x)$ *is a polynomial of degree less than* $d_1 + d_2 + \ldots + d_t$, *then* $[x^n]g(x) = f_1(n)\rho_1^n + f_2(n)\rho_2^n + \ldots + f_t(n)\rho_t^n$, *where each* $f_k(n)$ *is a polynomial of degree* $d_k - 1$ *with leading coefficient*

$$a_k = \frac{P(1/\rho_k)}{(d_k-1)!q_0\prod_{j\neq k}(1-\rho_j/\rho_k)^{d_j}}.$$

Intuition: we only need to compute a small number of coefficients to determine the highest order term for APC.

Optimization 1: Better Combinatorial Analysis

Full details in the paper!

Interprocedural Path Complexity Analysis

Mira Kaniyur, Harvey Mudd College, Claremont, USA, mkaniyur@hmc.edu
Ana Cavalcante-Studart, Harvey Mudd College, Claremont, USA, astudart@hmc.edu
Yihan Yang, Harvey Mudd College, Claremont, USA, yukyang@hmc.edu
Sangeon Park, Harvey Mudd College, Claremont, USA, sangpark@hmc.edu
David Chen, Harvey Mudd College, Claremont, USA, davidchen@hmc.edu
Duy Lam, Harvey Mudd College, Claremont, USA, tlam@hmc.edu
Lucas Bang, Harvey Mudd College, Claremont, USA, bang@cs.hmc.edu

ABSTRACT

Software testing techniques like symbolic execution face significant challenges with path explosion. Asymptotic Path Complexity (APC) quantifies this path explosion complexity, but existing APC methods do not work for interprocedural functions in general. Our new algorithm, APC-IP, efficiently computes APC for a wider range of functions, including interprocedural ones, improving over previous methods in both speed and scope. We implement APC-IP atop the existing software Metrinome, and test it against a benchmark of C functions, comparing it to existing and baseline approaches as well as comparing it to the path explosion of the symbolic execution engine Klee. The results show that APC-IP not only aligns with previous APC values but also excels in performance, scalability, and handling complex source code. It also provides a complexity prediction of the number of paths explored by Klee, extending the APC metric's applicability and surpassing previous implementations.

CCS CONCEPTS

• Software and its engineering → Software performance; Software verification and validation; • Mathematics of computing → Generating functions.

KEYWORDS

Code Complexity, Path Explosion, Testing Complexity

ACM Reference Format:

Mira Kaniyur, Ana Cavalcante-Studart, Yihan Yang, Sangeon Park, David Chen, Duy Lam, and Lucas Bang. 2024. Interprocedural Path Complexity Analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3652118

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09.
https://doi.org/10.1145/3650212.3652118

1 INTRODUCTION

Software testing and verification techniques rely on program path coverage to increase confidence in software correctness. However, for automated software testing approaches like symbolic execution, path explosion is a well-known bottleneck in code analysis [1, 5, 14]. Asymptotic path complexity (APC) is a metric that formalizes the degree of path explosion, thus quantifying the computational difficulty of achieving path coverage. Previous work [2] has demonstrated that asymptotic path complexity is a more accurate and refined metric to measure code complexity than other common complexity metrics such as cyclomatic [16] or NPATH [17] complexity. Further, it has been shown that asymptotic path complexity (APC) is useful in the context of automated software testing [3, 19], providing an upper bound on the growth rate of paths explored by a popular symbolic execution software such as Klee [6]. In earlier works, approaches to computing APC only handled intraprocedural analysis, including functions that make no recursive calls or make only self-recursive calls [3, 19]. In this paper, we extend and optimize the asymptotic path complexity (APC) metric to measure the complexity of interprocedural programs. We give a new APC algorithm, APC-IP, able to compute path complexity for interprocedural functions, which subsumes prior approaches and is significantly more scalable. Our APC-IP is an algorithm that computes the asymptotic path complexity of intraprocedural and interprocedural code. APC-IP thus provides a way to quickly predict the difficulty of automatic test generation for intraprocedural and interprocedural code.

Contributions. We claim the following research contributions.

APC-IP Formalization. Extension of the theory and algorithms established for APC to account for interprocedural functions.

Optimization Over All Previous APC Approaches. Replacing theoretical steps in previous algorithms to improve performance for both interprocedural and intraprocedural code.

APC-IP Implementation. Implementing APC-IP atop Metrinome, an existing APC analysis tool.

APC-IP Experimental Validation. Verification that APC-IP gives an accurate APC for both intraprocedural and interprocedural, and is the fastest option to process complex source codes. APC-IP is a predictor of path explosion in symbolic execution experiments.

Algorithm 7 GETRGF $(g(x))$

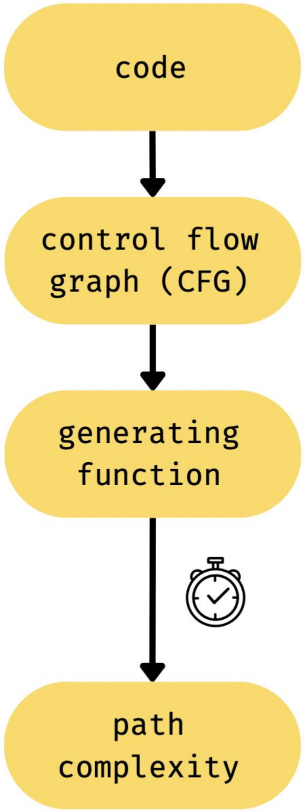
LET
$$g(x) = \frac{P(x)}{Q(x)}.$$
if $(deg(P(x)) \geq deg(Q(x)))$ **then**
    $P(x) \leftarrow R(x)$ **where** ▷ Make $deg(P(x)) < deg(Q(x))$
        $R(x) \leftarrow$ remainder of $P(x)/Q(x)$
$r \leftarrow$ ROOTS$(Q(x))$
$\rho \leftarrow$ inverses of the roots in r
$\rho_{\text{MAX}} \leftarrow$ maximum magnitude of all the ρ
$m \leftarrow$ maximum multiplicity among the ρ such that $|\rho_i| = \rho_{\text{MAX}}$
$\rho_k \leftarrow$ any ρ with magnitude ρ_{MAX} and multiplicity m
$q_0 \leftarrow$ constant term for $Q(x)$
for each ρ_k **do** ▷ Compute with GETRGF Theorem
$$a_k = \frac{P(1/\rho_k)}{(m-1)!q_0 \prod_{j \neq k}(1 - \rho_j/\rho_k)^m}.$$
$c \leftarrow \sum_{\rho_k} a_k$
$APC \leftarrow c \cdot n^{m-1} \rho_{\text{MAX}}^n$
return APC ▷ Return asymptotic path complexity

Optimization 1: Replacing theoretical steps

Before

We computed path complexity with **Taylor expansions** of the generating function yielding the number of paths .

$$path(n) = \sum_{i=0}^{D} \sum_{j=0}^{m_i-1} c_{i,j} n^j \left(\frac{1}{r_i}\right)^n$$



Optimization 1: Replacing theoretical steps

Before

We computed path complexity with **Taylor expansions** of the generating function yielding the number of paths .

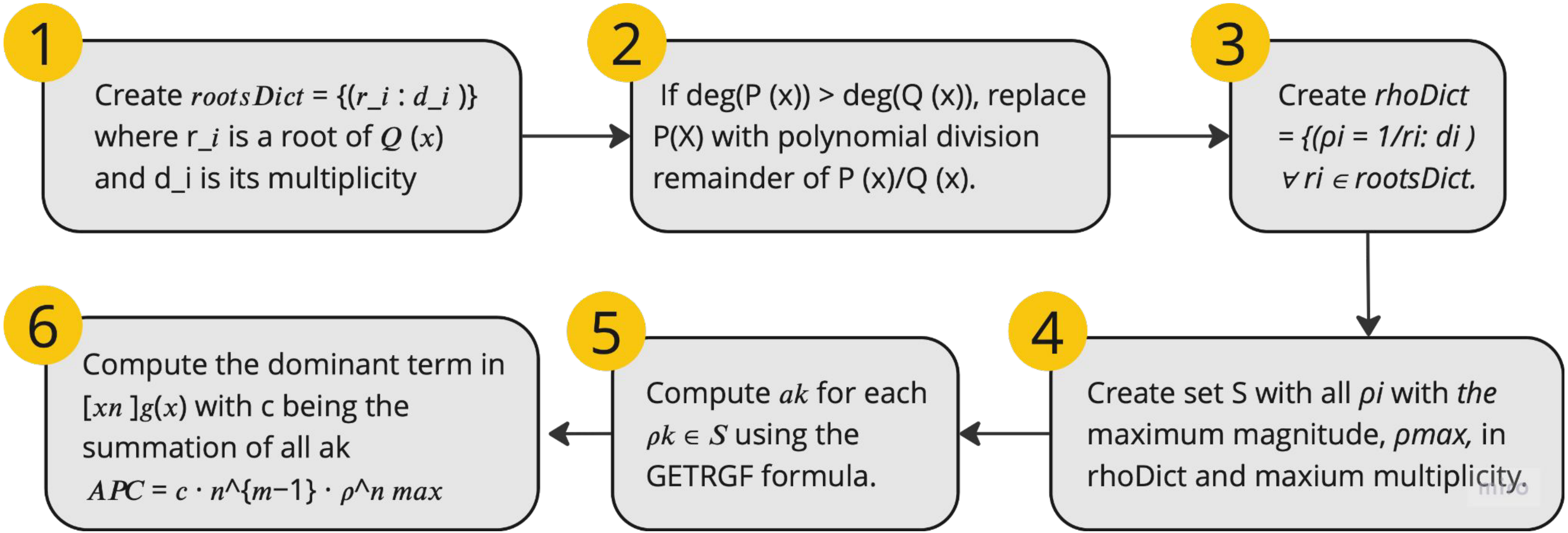
$$path(n) = \sum_{i=0}^{D} \sum_{j=0}^{m_i - 1} c_{i,j} n^j \left(\frac{1}{r_i}\right)^n$$

After

New method is bounded by the roots of $Q(x)$.

GETRGF

Optimization 1: Replacing theoretical steps



Optimization 1: Replacing theoretical steps

Before

$$path\ complexity = c \cdot n^{m-1} \cdot |\rho|^n$$

$\boldsymbol{\rho}$ is the inverse of the root of $Q(x)$ with the smallest magnitude

$\boldsymbol{m}$ is the maximum multiplicity of the roots among those with minimum magnitude

$\boldsymbol{c}$ is the sum of the $\boldsymbol{a_k}$, shown at right

After

$$a_k = \frac{P(1/\rho_k)}{(m-1)!q_0 \prod_{j \neq k}(1 - \rho_j/\rho_k)^m}$$

$\boldsymbol{q_0}$ is the constant term of $Q(x)$

$\boldsymbol{\rho_k}$ are the inverses of the roots with the same magnitude and multiplicity as $\boldsymbol{\rho}$

$\boldsymbol{\rho_j}$ are all the distinct roots

Optimization 2: Careful "Chunking" of Systems of Equations

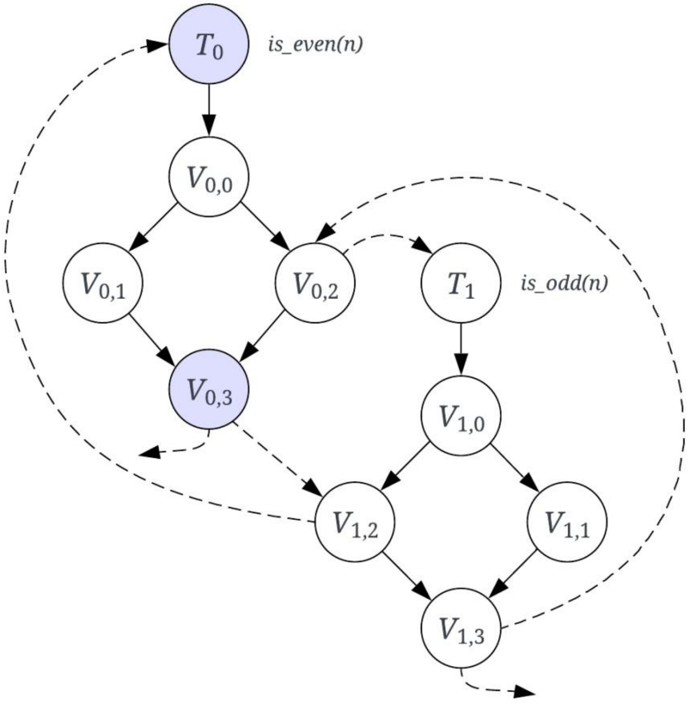
- APC-R treats this as one big system

System 0 (is_even)

$T_0 = V_{0,0}x$
$V_{0,0} = V_{0,1}x + V_{0,2}x$
$V_{0,1} = V_{0,3}x$
$V_{0,2} = T_1 V_{0,3}x$
$V_{0,3} = 1$

System 1 (is_odd)

$T_1 = V_{1,0}x$
$V_{1,0} = V_{1,1}x + V_{1,2}x$
$V_{1,1} = V_{1,3}x$
$V_{1,2} = T_0 V_{1,3}x$
$V_{1,3} = 1$



Optimization 2: Careful "Chunking" of Systems of Equations

- APC-R treats this as one big system

System 0 (is_even)

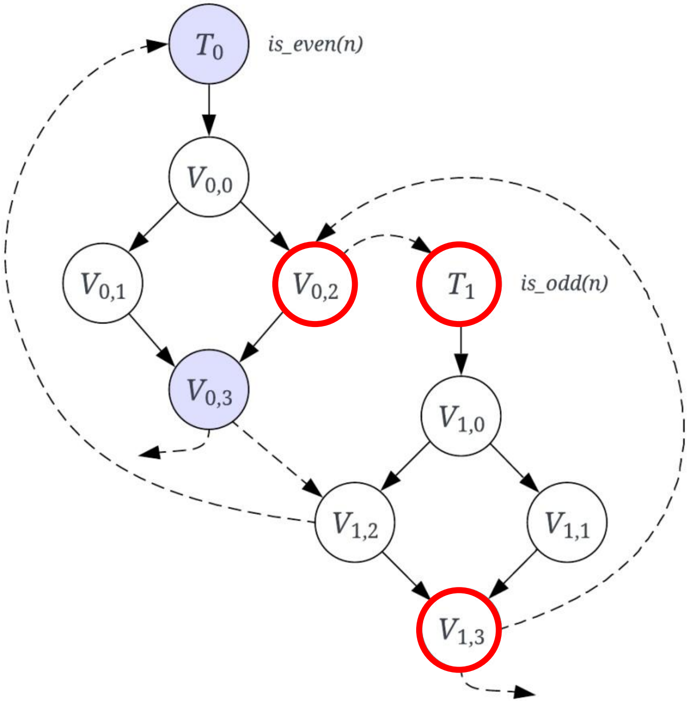
$$T_0 = V_{0,0}x$$
$$V_{0,0} = V_{0,1}x + V_{0,2}x$$
$$V_{0,1} = V_{0,3}x$$
$$V_{0,2} = T_1 V_{0,3}x$$
$$V_{0,3} = 1$$

System 1 (is_odd)

$$T_1 = V_{1,0}x$$
$$V_{1,0} = V_{1,1}x + V_{1,2}x$$
$$V_{1,1} = V_{1,3}x$$
$$V_{1,2} = T_0 V_{1,3}x$$
$$V_{1,3} = 1$$

APC-IP will instead

- reduce each sub-system as much as possible
- Solve while carefully respecting coupling variables (e.g. $V_{1,3}$, $V_{0,2}$, T_1)



Optimization 2: Careful “Chunking” of Systems of Equations

Full details in the paper!

Interprocedural Path Complexity Analysis

Mira Kaniyur, Harvey Mudd College, Claremont, USA, mkaniyur@hmc.edu

Ana Cavalcante-Studart, Harvey Mudd College, Claremont, USA, astudart@hmc.edu

Yihan Yang, Harvey Mudd College, Claremont, USA, yukyang@hmc.edu

Sangeon Park, Harvey Mudd College, Claremont, USA, sangpark@hmc.edu

David Chen, Harvey Mudd College, Claremont, USA, davidchen@hmc.edu

Duy Lam, Harvey Mudd College, Claremont, USA, tlam@hmc.edu

Lucas Bang, Harvey Mudd College, Claremont, USA, bang@cs.hmc.edu

ABSTRACT

Software testing techniques like symbolic execution face significant challenges with path explosion. Asymptotic Path Complexity (APC) quantifies this path explosion complexity, but existing APC methods do not work for interprocedural functions in general. Our new algorithm, APC-IP, efficiently computes APC for a wider range of functions, including interprocedural ones, improving over previous methods in both speed and scope. We implement APC-IP atop the existing software Metrinome, and test it against a benchmark of C functions, comparing it to existing and baseline approaches as well as comparing it to the path explosion of the symbolic execution engine Klee. The results show that APC-IP not only aligns with previous APC values but also excels in performance, scalability, and handling complex source code. It also provides a complexity prediction of the number of paths explored by Klee, extending the APC metric's applicability and surpassing previous implementations.

CCS CONCEPTS

• **Software and its engineering → Software performance**; **Software verification and validation**; • **Mathematics of computing → Generating functions**.

KEYWORDS

Code Complexity, Path Explosion, Testing Complexity

ACM Reference Format:

Mira Kaniyur, Ana Cavalcante-Studart, Yihan Yang, Sangeon Park, David Chen, Duy Lam, and Lucas Bang. 2024. Interprocedural Path Complexity Analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3652118

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09.
https://doi.org/10.1145/3650212.3652118

1 INTRODUCTION

Software testing and verification techniques rely on program path coverage to increase confidence in software correctness. However, for automated software testing approaches like symbolic execution, path explosion is a well-known bottleneck in code analysis [1, 5, 14]. Asymptotic path complexity (APC) is a metric that formalizes the degree of path explosion, thus quantifying the computational difficulty of achieving path coverage. Previous work [2] has demonstrated that asymptotic path complexity is a more accurate and refined metric to measure code complexity than other common complexity metrics such as cyclomatic [16] or NPATH [17] complexity. Further, it has been shown that asymptotic path complexity (APC) is useful in the context of automated software testing [3, 19], providing an upper bound on the growth rate of paths explored by a popular symbolic execution software such as Klee [6]. In earlier works, approaches to computing APC only handled intraprocedural analysis, including functions that make no recursive calls or make only self-recursive calls [3, 19]. In this paper, we extend and optimize the asymptotic path complexity (APC) metric to measure the complexity of interprocedural programs. We give a new APC algorithm, APC-IP, able to compute path complexity for interprocedural functions, which subsumes prior approaches and is significantly more scalable. Our APC-IP is an algorithm that computes the asymptotic path complexity of intraprocedural and interprocedural code. APC-IP thus provides a way to quickly predict the difficulty of automatic test generation for intraprocedural and interprocedural code.

Contributions. We claim the following research contributions.

APC-IP Formalization. Extension of the theory and algorithms established for APC to account for interprocedural functions.

Optimization Over All Previous APC Approaches. Replacing theoretical steps in previous algorithms to improve performance for both interprocedural and intraprocedural code.

APC-IP Implementation. Implementing APC-IP atop Metrinome, an existing APC analysis tool.

APC-IP Experimental Validation. Verification that APC-IP gives an accurate APC for both intraprocedural and interprocedural, and is the fastest option to process complex source codes. APC-IP is a predictor of path explosion in symbolic execution experiments.

Algorithm 5 Eliminate-Optimized (Systems S, Vars V)

$S = S_0, S_1, \ldots, S_n$

$T = \{\}$

for each $i \in len(S)$ **do** ▷ Solve each system for T_i

  d ← substitution dictionary for eliminating

  d = $\{\{V_k$: all eqns containing $V_k\} \forall V_k \in S_i\}$

  T ← add Partial-Eliminate(S_i, V_i, d)

d = $\{T_i$: {all eqns ∈ T containing $T_i\}\}$

$v = \{T_0, T_1, \ldots, T_n\}$ ▷ Variables for eliminating Ts

return Partial-Eliminate(T, v, d) ▷ Solve Ts for T_0

Algorithm 6 Partial-Eliminate (sys s, vars v, dict d)

if $len(s) = 1$ **then**

  return $s[0]$ ▷ Return $T_i = A$

var = $v[-1]$ ← var to eliminate

eqn = $s[-1]$ ← eqn of form var = A

sub ← right side of eqn, equal to var

if var ∈ sub **then** ▷ Must solve for var

  for each eq ∈ d[var] s.t. eq in bounds **do**

    sub = $solve$(eq, var)

    if len(sub) = 1 **then** ▷ Unique solution for var

      break

for each eq ∈ d[var] **do**

  eq ← substitute var with sub

  d ← update dict d after substitution

return Partial-Eliminate($s[: -1], v[: -1], d$)

Takeaways for APC-IP

- More advanced combinatorial analysis of grammar using GETRGF
- More sophisticated **solving for systems of coupled equations**
- **Same** path complexity **results** as APC-R, but **faster**!

Experiments

Experimental Overview

- Benchmark Functions
- APC (recursive, interprocedural, naive interprocedural)
- Overall result
- APC and KLEE

Benchmark Functions

Benchmark Functions

- 76 well known C algorithms found in https://github.com/TheAlgorithms/C repository
    - Contains non-recursive, recursive, and interprocedural functions
    - Combination of straight line code, nested conditions, loops
- 3 running examples

Benchmark Functions

- 76 well known C algorithms found in https://github.com/TheAlgorithms/C repository
  - Contains non-recursive, recursive, and interprocedural functions
  - Combination of straight line code, nested conditions, loops
- 3 running examples

79 functions

APC

- **APC-R**: able to perform non recursive and recursive APC analysis on single functions
- **NAPC-IP**: APC-R with minimal modification (relabel variables) to handle interprocedural code
- **APC-IP**: fully interprocedural and optimized APC

APC: Result

- For 42 non-interprocedural functions, APC-R = NAPC-IP = APC-IP
- For 37 interprocedural functions, NAPC-IP = APC-IP
- But APC-IP is FASTER!
  - Majority run < 1 seconds
  - Most run <5 seconds
  - 3 outliers ~100 seconds

Overall result: APC-R vs APC-IP

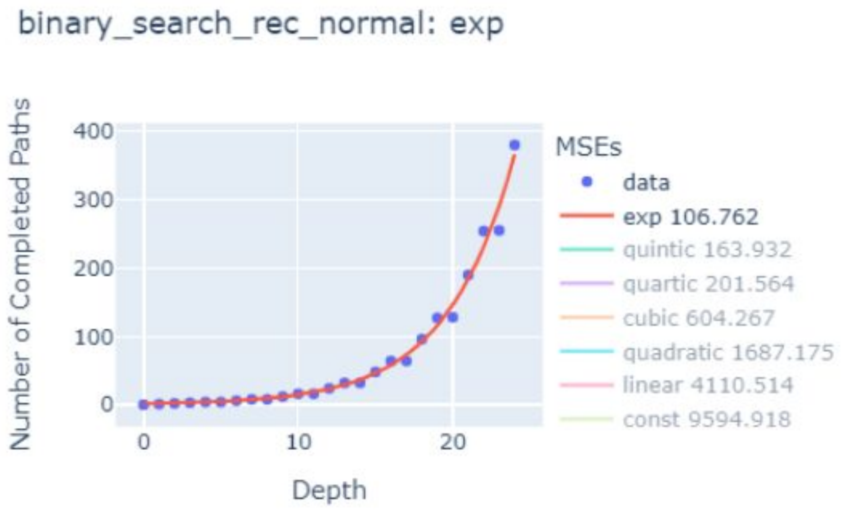
- 42 non-interprocedural code
- APC-IP is faster in 32 cases, in 5 cases APC-IP is more than 100 times faster than APC-R
  - EX: bubble sort: ***>200s → 0.63s***.
- When APC-IP is slower than APC-R, both are less than 1 second

Overall result: NAPC-IP vs APC-IP

- 79 functions
- APC-IP is faster in 65 cases, in 16 cases APC-IP is 100 times faster than NAPC-IP
  - EX: Heap sort: ***>6000s → 4.1s***
- When APC-IP is slower, it is still <1 second

KLEE Data: Curve Fitting

- Number of paths explored for increasing depth



Results: APC and KLEE

- Ran KLEE on 57/79 functions.
- APC-IP successfully predicts the upper bound on KLEE's path explosion, even for interprocedural functions!
- Detailed data in the PAPER.

Table 4: APC and KLEE data on C files showing APC-IP and best fit curve for KLEE path explosion.

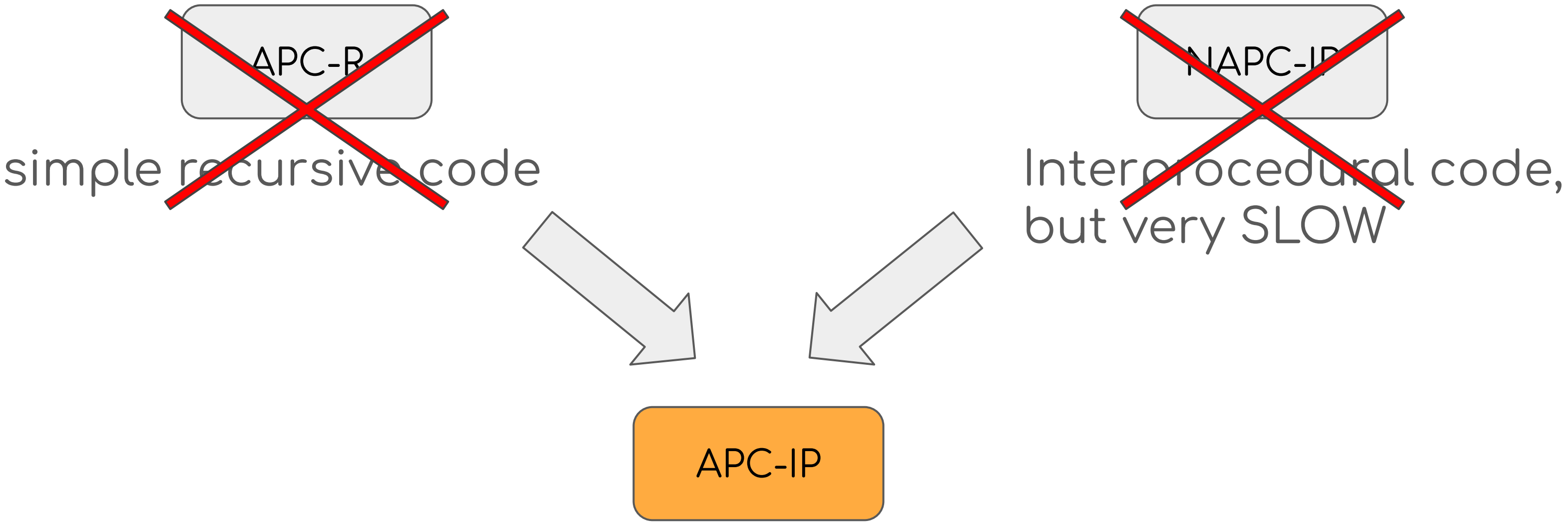
		APC		KLEE		
Index	**Function**	**APC-IP**	**APC-IP Time(s)**	**Best Fit**	**APC-IP**	**KLEE Time(s)**
1	Even-Odd §	$n/3$	0.144	n	yes	24.95
2	GCD †	$n/3$	0.223	n	yes	2.58
3	Floyd Alg. §	$0.125 * n^2$	0.635	$9.58 * 1.06^n$	no, but close	34.06
4	Catalan §	n^3	0.272	n	upper bound	215.5
5	Fib. Search	$2.33 * 1.22^n$	0.88	$5.71 * 1.28^n$	yes	63.69
6	Bead Sort	$0.37 * 1.30^n$	4.91	$1.90 * 1.54^n$	yes	23.31
7	Fib. (R) †	1.34^n	0.123	n	upper bound	1682.06

§ represents that the source code is interprocedural.
† This version of the function is implemented recursively.

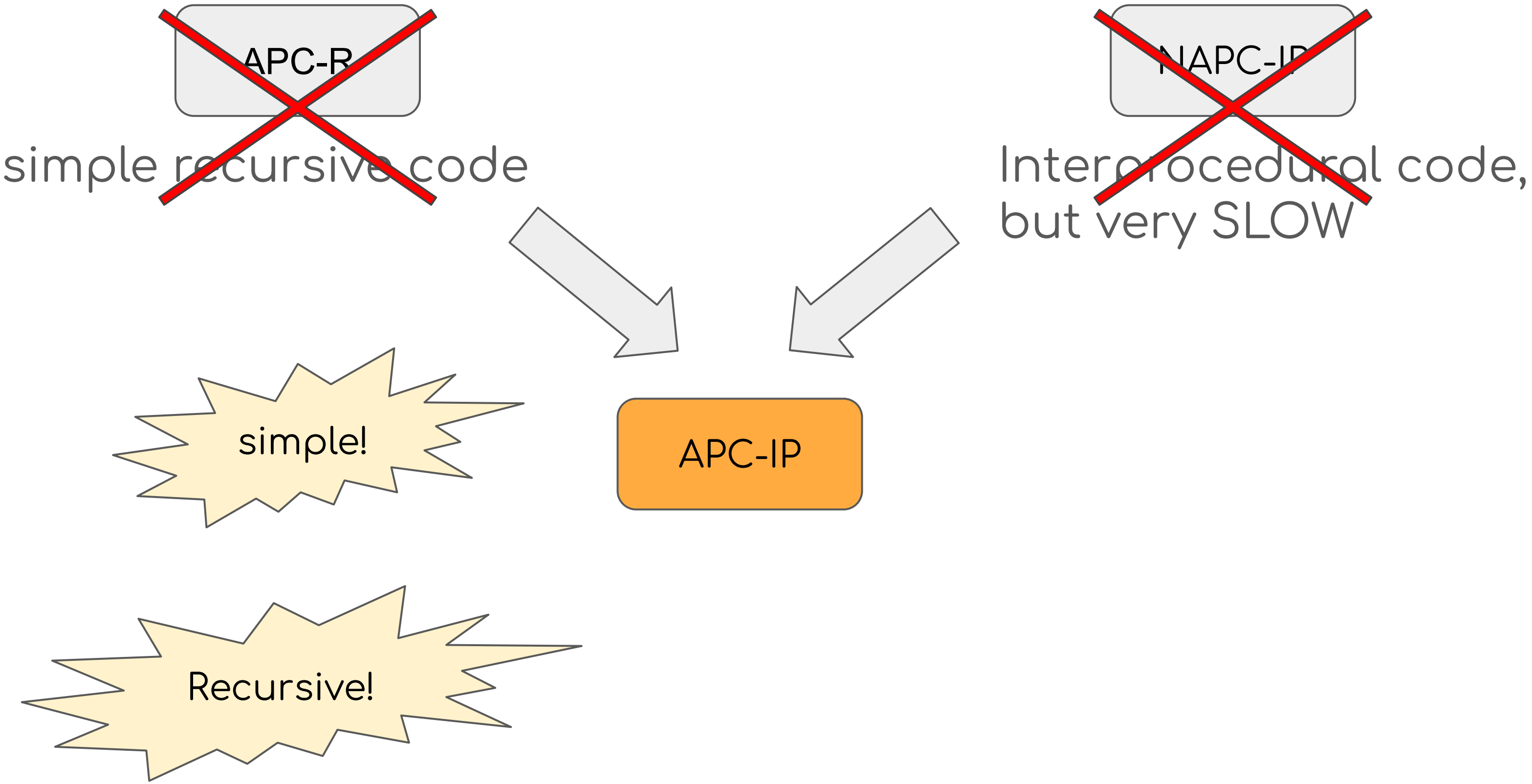
Results: APC and KLEE

- Ran KLEE on 57 functions.
- 46 functions, APC-IP is in the same complexity class as KLEE's best fit line
  - KLEE bound exploration by branch count, while APC-IP is by edge count in CFG
- 53 APC-IP bound KLEE best fit line
- 3 cases we don't have enough data for the best fit line
- 1 case where KLEE is exponential but APC-IP is quadratic
  - Suspect this is due to overfitting

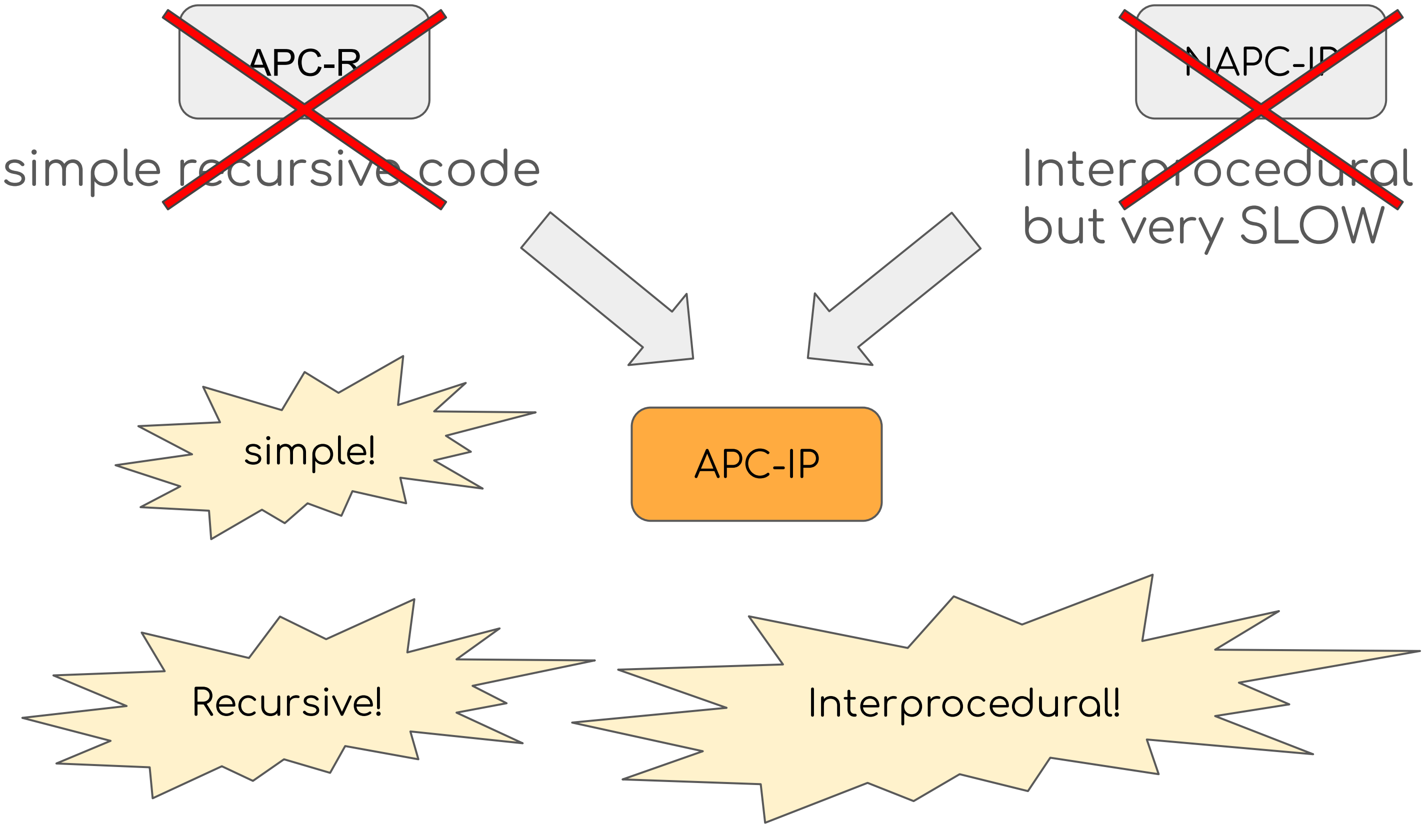
Conclusion


APC-R
simple recursive code
NAPC-IP
Interprocedural code,
but very SLOW
APC-IP

Conclusion

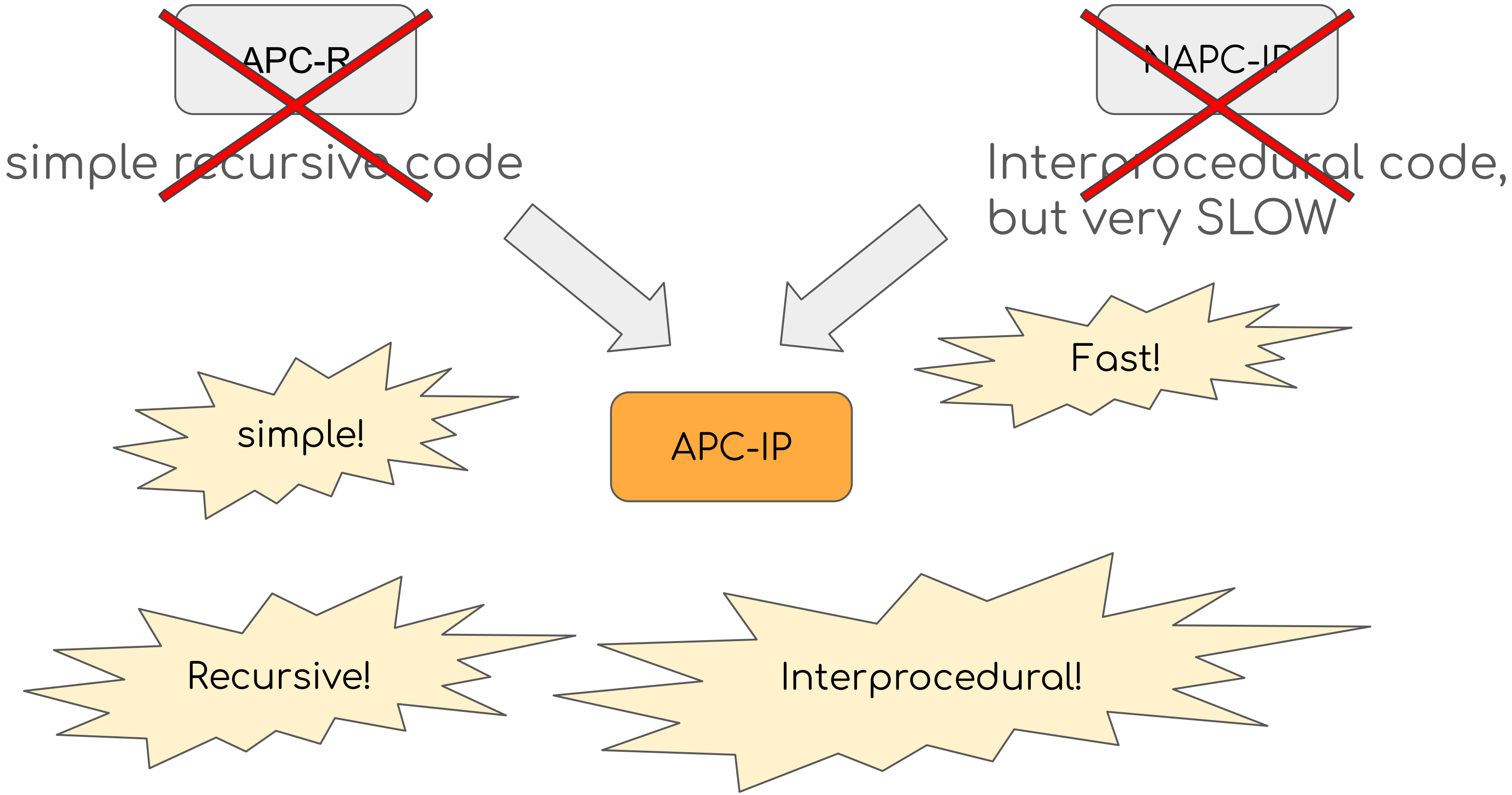

APC-R
simple recursive code
NAPC-IP
Interprocedural code,
but very SLOW
simple!
APC-IP
Recursive!

Conclusion

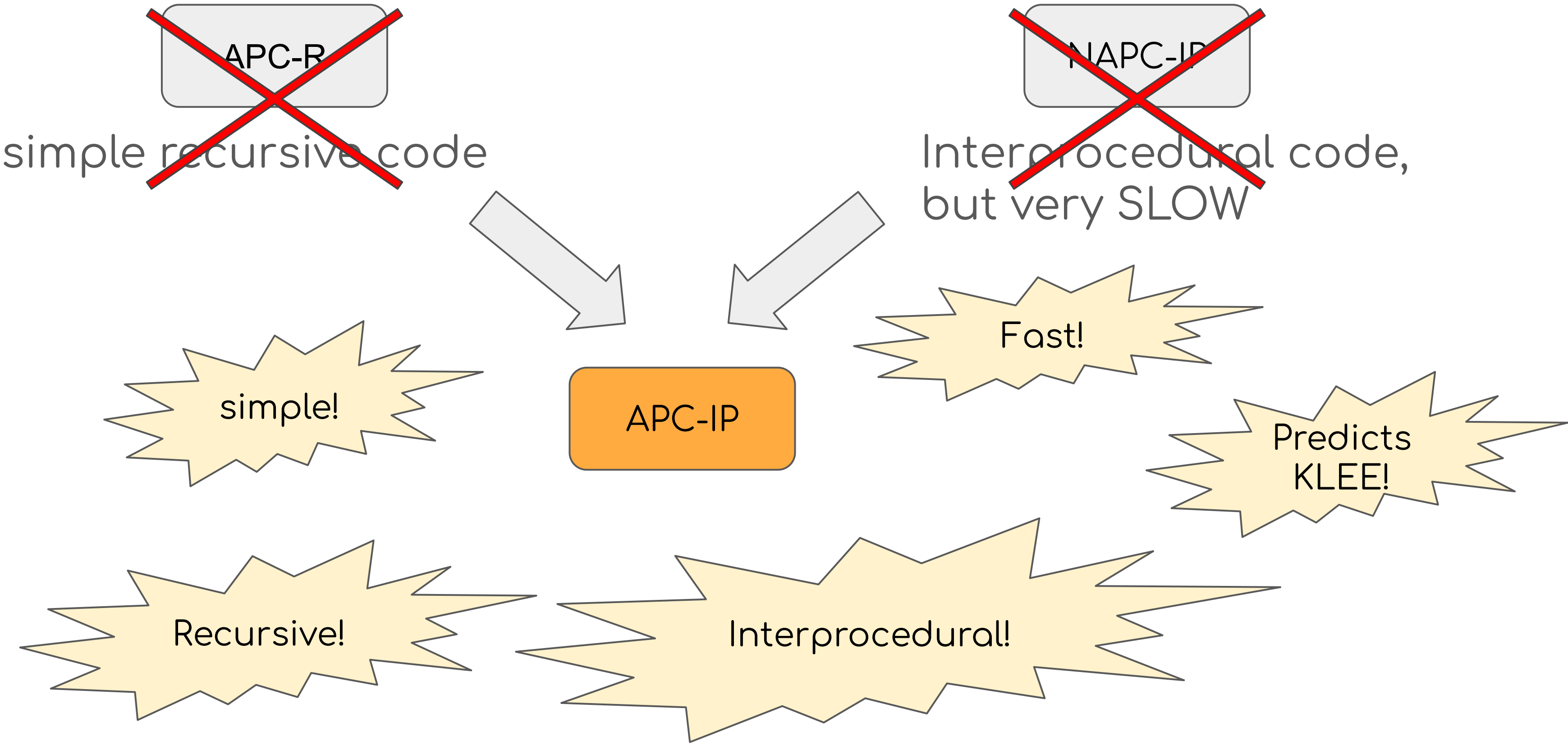


APC-R
simple recursive code
NAPC-IP
Interprocedural code,
but very SLOW
simple!
APC-IP
Recursive!
Interprocedural!

Conclusion


APC-R
simple recursive code
NAPC-IP
Interprocedural code,
but very SLOW
APC-IP
simple!
Fast!
Recursive!
Interprocedural!

Conclusion


APC-R
simple recursive code
NAPC-IP
Interprocedural code,
but very SLOW
APC-IP
simple!
Fast!
Predicts
KLEE!
Recursive!
Interprocedural!

Conclusion

- APC-IP provides a sound upper bound on the degree of KLEE's path explosion when testing simple, recursive, or interprocedural programs.
- For intraprocedural functions, APC-IP = APC-R and FASTER!
- For interprocedural functions, APC-IP can computes correct APC usually in under 5 seconds.
- APC-IP subsumes earlier APC, and with drastic improvements on performance cost.

Future Work

- [Done/Test stage] Expand APC to process programs in more common programming languages, such as Python and Java.
- [To do] Continuing to scale APC to meet the scale of today's industry code-bases.

Takeaways

Path Complexity

Asymptotic upper bound on the

number of paths in control flow graph from start to exit

up to a given execution depth.

APC-IP (Interprocedural Asymptotic Path Complexity)

APC-IP subsumes earlier APC work

- produces the same results on the simpler benchmarks

APC-IP extends earlier APC work

- handles fully interprocedural code, unlike previous work

APC-IP outperforms earlier APC work

- much faster when it matters

APC-IP predicts symbolic execution explosion rate

- upper bound on execution paths explored by KLEE

Thank you!

https://github.com/hmc-alpaqa/metrinome

Ablation Study

Optimization 1: Effects of GETRGF on the runtime

- Euclidean algorithm: ***54.5s → 0.06s***

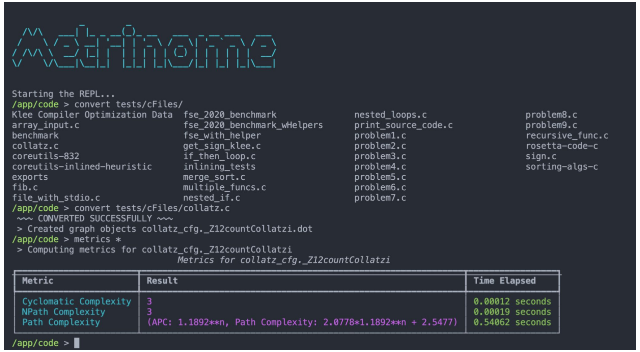
Optimization 2: Effects of “chunking” systems of equations

- Heap sort: ***>6000s → 1.88s***
- When Naive is fast (< 1 s), Optimized is not as fast but still < 1s
- When Naive explode (>100 s), Optimized can be up to 1000 times faster, still in 1-2s range

Takeaway: APC-IP performs much better for complex functions!

Conclusion and Future

- APC can be accurately calculated in Metrinome
- KLEE behavior can be predicted by Metrinome
- Next Steps
  - Further experimental validation
  - More robust numerical computing (e.g. fix APC computation for mergesort)
  - Implement full interprocedural analysis



Metrinome

https://github.com/hmc-alpaqa/metrinome