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.

```
int gcd(int a, int b) {
    while (a != b){
        if (a > b) a = a - b;
        else b = b - a;}
    return a;
}
```

Figure 1: Source code for gcd function.

```
int palindrome_r(const char *s, int b, int e)
{
   if ( (e - 1) <= b ) return 1;
   if ( s[b] != s[e-1] ) return 0;
   return palindrome_r(s, b+1, e-1);
}
```

Figure 3: Source code for palindrome function.

In the following sections, the paper covers the intuition of the asymptotic path complexity metric, reviews previous work on computing the APC for intraprocedural functions, and the theoretical changes and optimizations that produce a correct and efficient APC-IP algorithm. Finally the paper will cover the experimental results comparing our APC-IP with previous APC metrics as well as Klee.

2 PATH COMPLEXITY BACKGROUND

Here, we introduce and define path complexity and asymptotic path complexity. Path complexity is a function expressing the number of execution paths through a program within a certain execution depth n. Computing path complexity relies on: (1) the control flow graph (CFG), a standard representation of a program's structure, (2) formal language theory including context free grammars, and (3) the theory of generating functions.

2.1 Asymptotic Path Complexity Intuition

Asymptotic path complexity (APC) is an asymptotic upper bound of the path complexity metric. APC is calculated in terms of n, which is the program's execution depth, or the length of the path through the Control Flow Graph. We define the path length to be the number of edges in the path. The asymptotic path complexity of a program P is expressed as a function $f(n)$ such that $f(n) = O(path(n))$, where $path(n)$ is the number of different executions of a program with maximum execution length n. Earlier studies have computed APC for non-recursive as well as recursive programs [3, 19]. Our work extends this metric for interprocedural functions.

2.2 Path Complexity Examples

To explain the intuition of path complexity, we demonstrate computing path complexity by hand for small depths. We count the paths until a certain depth for three examples: one non-recursive function, one recursive function, and one interprocedural function.

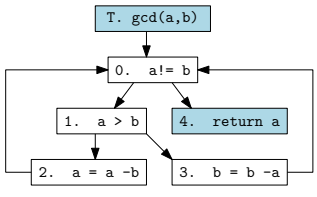

Figure 2: Control Flow Graph for gcd function.

Example: gcd (Non-Recursive). In this control flow graph (CFG) of a gcd function (Figure 2), the nodes are labeled by the line number of the code. Any counted path must go from start (node T) to exit (node 4). Since gcd contains loops, there are infinitely many paths through this graph. Hence, a function in terms of maximum path length n is used to express path complexity. Below is a table showing both the number of paths *with* length n ($\text{len}(p) = n$) and the number of paths *within* depth n ($\text{len}(p) \leq n$).

Depth, n	0	1	2	3	4	5	6	7	8	9	10	11	12
$\lvert\{p : \text{len}(p) = n\}\rvert$	0	0	1	0	0	2	0	0	4	0	0	8	0
$\lvert\{p : \text{len}(p) \leq n\}\rvert$	0	0	1	1	1	3	3	3	7	7	7	15	15

For example, the shortest path, $T \to 0 \to 4$, is length 2. The next shortest paths are 2 paths of length 5, $T \to 0 \to 1 \to 2 \to 0 \to 4$, and $T \to 0 \to 1 \to 3 \to 0 \to 4$ respectively. We continue to complete the table above. One can manually verify that the function $2^{\lfloor (n+1)/3 \rfloor} - 1$ correlates with the numerical series in the table. With Metrinome, the APC of the gcd function is upper-bounded by $2^{\lfloor (n+1)/3 \rfloor} - 1$, approximately $O(1.26^n)$.

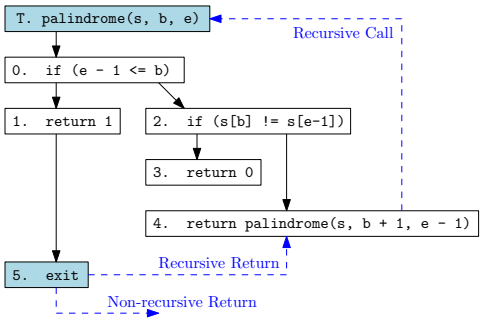

Figure 4: Control Flow Graph for palindrome function.

Example: palindrome (Recursive). A recursive palindrome function is given in Figure 4. When the recursive call is made at node 4, the control flow passes back to the entry point of the function at node T. Similar to the previous path counting methods, the path number is bounded by the length of execution given by n.

Depth, n	0	1	2	3	4	5	6	7	8	9	10	11	12
$\lvert\{p : \text{len}(p) = n\}\rvert$	0	0	0	1	1	0	0	1	1	0	0	1	1
$\lvert\{p : \text{len}(p) \leq n\}\rvert$	0	0	0	1	2	2	2	3	4	4	4	5	6

Here, the shortest path is $T \to 0 \to 1 \to 5$, with length 3. The next shortest path is $T \to 0 \to 2 \to 3 \to 5$, with length 4. Then, there is the first path with a recursive call, $T \to 0 \to 2 \to 4 + (T \to 0 \to 1 \to 5) \to 5$. This path has length 7. By convention, the recursive edge doesn't count in the length, since no code is executed in that step. Furthermore, the final 5 must occur once we return from the recursive call to finish the original call. For recursive functions, call and return locations must be appropriately matched within a path. By using the table, we can calculate the path complexity to be $\lfloor n/2 \rfloor$ (for $n > 2$) which is $O(n/2)$.

```
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);
}
```

Figure 5: Source code for `is_even` and `is_odd` functions.

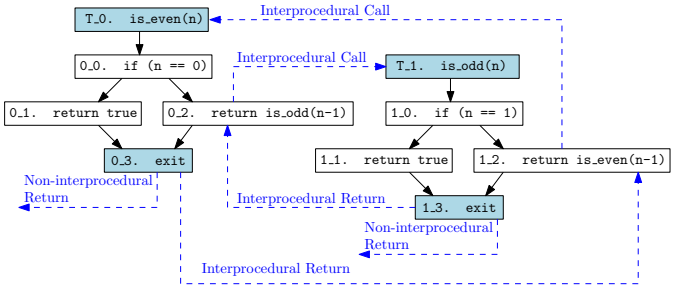

Figure 6: Control Flow Graph for `is_even` and `is_odd` functions (dashed lines are interprocedural calls).

Example: `is_even` and `is_odd` (Interprocedural, mutually recursive). The functions `is_even` and `is_odd` (CFGs in Figure 6) have mutually recursive interprocedural calls where the node 0_2 calls `is_odd` and the node 1_2 calls `is_even`.

Depth, n	0	1	2	3	4	5	6	7	8	9	10	11	12
$\lvert\{p : \text{len}(p) = n\}\rvert$	0	0	0	1	0	0	1	0	0	1	0	0	1
$\lvert\{p : \text{len}(p) \leq n\}\rvert$	0	0	0	1	1	1	2	2	2	3	3	3	4

Here, we see that our shortest path, $T_0 \to 0\_0 \to 0\_1 \to 0\_3$, is length 3. Our next shortest path is $T_0 \to 0\_0 \to 0\_2 + (T_1 \to 1\_0 \to 1\_1 \to 1\_3) \to 0\_3$ is length 6. This path contains nested interprocedural calls. Then we have $T_0 \to 0\_0 \to 0\_2 + (T_1 \to 1\_0 \to 1\_2 + (T_0 \to 0\_0 \to 0\_1 \to 0\_3) \to 1\_3) \to 0\_3$, of length 9. Again, calls and returns should be matched. Similar to the previous examples, interprocedural path complexity can also be manually calculated with the the table. The number of paths within a given depth can be expressed as $\lfloor n/3 \rfloor$ for this function (where $n > 2$).

Observations. The gcd function's CFG can be thought of as a finite automaton where paths in the graph form a regular language [22]. The recursive palindrome example instead requires matching calls and returns, hinting at the need for a context-free language to represent paths [22]. The analytic combinatorics behind counting members of regular and context-free languages parameterized by length is well understood [8, 10, 12], and it is these methods that are used straightforwardly in existing APC analyses [3, 19]. However, interprocedural paths (as in the `is_even` and `is_odd` example) can be thought of as generated by an *interdependent set* of recursive context-free grammars. Now, one could treat such a system of coupled context-free grammars as a single large grammar and apply the same approaches in order to compute interprocedural APC, but we observed that this is too inefficient to be an effective method for computing APC. Similar to solving coupled systems of difference equations [15], carefully simplifying subsystems of constraints that arise from these interdependent systems of graphs and grammars is the key to making our approach efficient for computing interprocedural APC (i.e. Algorithms 5 and 6 of Section 3.3).

2.3 Recursive Path Complexity Analysis

An existing approach, APC-R for non-interprocedural recursive code [19], is reproduced here in Algorithm 1. The first two lines which compute the control flow graph and the system of equations are elided in this paper, but control flow graph construction is well-known, and our following examples elucidate how the system of equations is constructed from the CFG.

Algorithm 1 APC-R: Asymptotic-Analysis of Recursive Functions (Program P)

```
G ← Control-Flow-Graph(P)
S ← System(G)
V ← Variables(S)                          ▷ get the variables in S
Γ ← Eliminate(S, V/{T})                   ▷ Γ in terms of T
D ← Discriminant(Γ)
R ← Roots(D)
if (R = ∅) then                           ▷ Case 1
    g(z) = Solve(Γ, T)/(1 − z)
    apc = Symb-Calc(g, R) ← upper bound of g
else                                      ▷ Case 2
    r* ← minimum positive real root in R
    apc ← (1/r*)^n
return apc                                ▷ Return Asymptotic Path Complexity
```

To compute APC for a program P, the algorithm produces a control flow graph G, and then converts it to a context free grammar R and then to a system of equations S. The grammar R is constructed such that its language $\mathcal{L}(R)$ corresponds with paths through G. A context free grammar is used because all strings that can be generated by the context-free language correspond to paths through the control-flow graph, which is what we wish to count. Each string-replacement grammar rule encodes information about a node in the control graph: which nodes it can lead to, if there are any recursive calls, and whether or not it's a return node. The Chomsky-Schützenberger Enumeration Theorem is then applied to compute a system of equations that describes constraints on a combinatorial generating function that counts the number of strings of a given length, as explained in the following examples.

Theorem (Chomsky–Schützenberger, 1963) *If L is a context-free language with unambiguous context-free grammar, and a_k is the number of words of length k in L, then $g(x) = \sum_{k=0}^{\infty} a_k x^k$ is a power series over $\mathbb{N}$ that is algebraic over $\mathbb{Q}(x)$.* □

Continued Example: `palindrome`. We start with the control flow graph on the left of Figure 7, then create the context-free grammar (below left) and use Chomsky–Schützenberger to transform it into a system of equations (below right).

$$\begin{array}{l} T \to 0V_0 \\ V_0 \to 1V_1 | 2V_2 \\ V_1 \to 5V_5 \\ V_2 \to 3V_3 | 4V_4 \\ V_3 \to 5V_5 \\ V_4 \to T5V_5 \\ V_5 \to \varepsilon \end{array} \xrightarrow[\text{Transformation}]{\text{Chomsky-Schützenberger}} \begin{array}{l} T = V_0 x \\ V_0 = V_1 x + V_2 x \\ V_1 = V_5 x \\ V_2 = V_3 x + V_4 x \\ V_3 = x \\ V_4 = TV_5 x \\ V_5 = 1 \end{array}$$

Looking at the context-free grammar, we note that string replacement rules correspond to traversing the control flow graph. For example, the first rule goes from T to $0V_0$, representing a path that just started at node 0, and is currently deciding what to do. The second rule represents a branch in the control flow graph: V_0 can be replaced with either $1V_1$ or $2V_2$, representing the split in the graph. Finally, suppose we replace V_1 with $5V_5$ and then replace V_5 with ε

– this is a terminal character, and the final generated string is 015, representing following the leftmost path in the control flow graph and returning. On the other hand, if we traverse the rightmost path of the graph, we have the option to replace V_2 with $4V_4$, and then V_4 with $T5V_5$: since T is the start symbol, this represents a recursive call. Note that the T is before the 5: the recursive call must return before the calling node can return, so the recursive call's string replacement is before the path ends at node 5.

Substituting to eliminate variables V_i gives us $T = \frac{x^3(1+x)}{1-x^4}$. The APC-R algorithm branches into two cases here. If the discriminant D of T has any real roots, we can directly calculate the APC as $(\frac{1}{r^*})^n$, where r^* is the minimum positive real root of D. In this example, the discriminant of T has no real roots, so we must take further steps to compute the APC, using a generating function $g(x)$. This generating function always has the formula $g(x) = \frac{T}{1-x}$. This $g(x)$ encodes, in its series expansion, the integer sequence of $path(n)$ for $n = 1, 2, 3, \ldots$, as the coefficients of $x^1, x^2, x^3, \ldots$. That is, the coefficient $[x^n]g(x)$ is the number of paths through the control flow graph starting at T with max depth n. For this example, the Taylor expansion is $g(x) = \frac{T}{1-x} = \frac{x^3(1+x)}{(1-x^4)(1-x)}$

$= x^3 + 2x^4 + 2x^5 + 2x^6 + 3x^7 + 4x^8 + 4x^9 + 4x^{10} + 5x^{11} + \ldots$

Observe that the coefficients of this series exactly correspond to the path counts of the earlier manually computed palindrome example. A complete treatment of generating functions for encoding sequences of integers can be found in several references [21, 24, 26].

After the generating function $g(x)$ is produced, symbolic calculus is used to compute the APC using the Taylor series of the generating function. Given the generating function, g, one computes

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

where r_i represents the distinct roots of the denominator of g, j as the multiplicity of r_i, and each $c_{i,j}$ is the coefficient of the corresponding term in $path(n)$, which is solved for as a system of equations using linear algebra. This equation gives the upper bound of path complexity, and the asymptotic value is APC. Using this approach, the APC-R of `palindrome` function is $O(n/2)$. A full discussion of this method is found in [2]. The overall algorithm for APC-R is summarized in Algorithm 1.

3 INTERPROCEDURAL PATH COMPLEXITY

3.1 Introduction

APC-R is capable of computing the path complexity of recursive functions by including recursive calls in the context-free grammar. We adapt this approach to develop APC-IP, which can compute the path complexity of interprocedural functions. We implement three major changes: the first is sufficient for correctness, while the second and third make the algorithm computationally tractable.

Our first change, covered in Section 3.2, is to relabel the vertices of the control flow graph and thus the variables in the system of equations to handle multiple graphs. We produce control flow graphs $G_0, \ldots, G_n$ for each function in program P, instead of one control flow graph for the singular function. These control flow graphs also contain metadata representing function calls. Subsequently, we produce a set of interrelated systems to solve, with each system encoding a single graph, instead of a single system encoding all the graphs. This relabeling is sufficient to produce a theoretically correct yet naive algorithm for interprocedural path complexity analysis, which we call NAPC-IP. However, empirically, NAPC-IP is computationally intractable for interprocedural functions of any significant size or complexity. Therefore, we optimize solving the system of equations in Section 3.3, implementing a Eliminate-Optimized in Line 4 that is designed to solve lengthy and numerous interrelated systems. In addition, we replace the asymptotic analysis of the generating function via symbolic calculus (line 9 of Alg 1, Symb-Calc) with a more effective analytic combinatorial approach (line 9 of Alg 2, GetRgf) in Section 3.4. Our final algorithm, APC-IP (Alg 2), implements both these optimizations. The three major changes from APC-R to APC-IP are detailed in section 3.2, 3.3, and 3.4 respectively.

Algorithm 2 APC-IP: Asymptotic-Analysis of Interprocedural Functions (Program P)

```
1:  G_0, ..., G_n ← Control-Flow-Graphs(P)
2:  S = S_0, ..., S_n ← Systems(G_0, ..., G_n)                ▷ Alg 3
3:  V = V_0, ..., V_n ← Variables(S_0, ..., S_n)
4:  Γ ← Eliminate-Optimized(S, V/{T_0})                       ▷ Alg 5
5:  D ← Discriminant(Γ)
6:  R ← Roots(D)
7:  if (R = ∅) then                                           ▷ Case 1
8:      g(z) = Solve(Γ, T_0)/(1 − z)
9:      apc = GETRGF(g, R) ← upper bound of g                 ▷ Alg 7
10: else                                                      ▷ Case 2
11:     r* ← minimum positive real root in R
12:     apc ← (1/r*)^n
13: return apc                          ▷ Return Asymptotic Path Complexity
```

3.2 From Code to Systems of Equations

As we saw in section 2.3, the first step of APC-R is to convert the source code of a function into a control flow graph (CFG). We first establish a start node T that points to the function's first node. All other nodes are named V_n, where n represents that node's number.

We adapt this labeling process for interprocedural functions. Each separate function needs unique labeling and a start symbol we can refer to. Thus, for the i-th function we assign a start node T_i and non-terminal nodes $V_{i,j}$, for the j-th node of the i-th function.

Continued Example: `palindrome`. Recall from section 2.3 the code of the palindrome function. Using the labeling convention established for APC-R, the CFG for the palindrome function would correspond to the left graph of Figure 7. With our convention, the CFG for this function corresponds to the graph in the right.

Continued Example: `is_even` and `is_odd`. We manually computed this APC in Section 2. We now use APC-IP.

Below is the APC-IP algorithm to generate the systems of equations from the control flow graphs. It operates identically to the System algorithm inAPC-R, except that it produces a system for each graph, and the variables are indexed by both graph and node. Systems (Algorithm 3) produces interrelated systems, where the $V_{i,j}$ vertex variables are contained within a graph's system S_i, but the T_i variables are interrelated by function calls.

Here are the systems this algorithm produces from the control flow graphs for `is_even`.

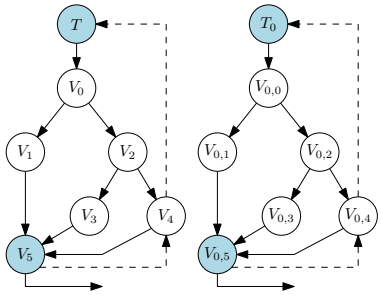

Figure 7: CFG for `palindrome` using APC-R labeling in the left and APC-IP labeling in the right (dashed lines are recursive calls / returns).

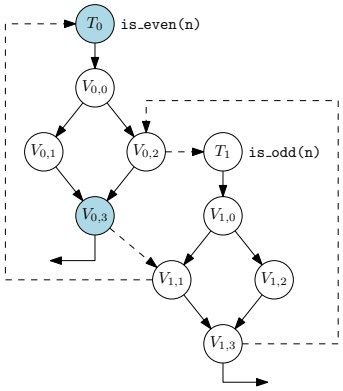

Figure 8: CFGs for `is_even` and `is_odd` using APC-IP labels (dashed lines are interprocedural calls / returns).

Algorithm 3 Systems (Graphs $G_0, \ldots, G_n$)

1: **for each** G_i **do** ▷ Make system for each graph
2: $S_i = \{T_i = xV_{i,0}\}$ ▷ Initial equation for each system of equations
3: **for each** $j \in$ Nodes(G_i) **do**
4: $expr = 0$
5: **for each** $k \in$ Out-Neighbors(j) **do**
6: $expr = expr + V_{i,k}$
7: **if** expr = 0 **then** ▷ j has no children
8: $expr = 1$ ▷ To preserve terminal nodes' calls
9: **for each** $T_w \in$ Calls(j) **do** ▷ A node can have >1 outgoing calls
10: $expr = T_w \cdot expr$
11: eqn $= \{V_{i,j} = expr \cdot x\}$
12: $S_i = S_i \cup$ eqn ▷ Add equation to system
13: **return** $S = \{S_0, S_1, \ldots, S_n\}$ ▷ Return systems

System 0 (`is_even`)	System 1 (`is_odd`)
$T_0 = V_{0,0}x$	$T_1 = V_{1,0}x$
$V_{0,0} = V_{0,1}x + V_{0,2}x$	$V_{1,0} = V_{1,1}x + V_{1,2}x$
$V_{0,1} = V_{0,3}x$	$V_{1,1} = V_{1,3}x$
$V_{0,2} = T_1V_{0,3}x$	$V_{1,2} = T_0V_{1,3}x$
$V_{0,3} = 1$	$V_{1,3} = 1$

To find our Γ function, instead of solving for T, we solve for T_0, which is the T of the function whose path complexity we are calculating. This modification to APC-R is sufficient to produce a theoretically correct path complexity algorithm for interprocedural functions. As such, we refer to APC-R with this modification alone as NAPC-IP, or naive asymptotic path complexity for interprocedural functions. However, empirically, NAPC-IP is inefficient and unable to process interprocedural functions with any significant complexity. For example, processing an interprocedural `merge sort` function took over 2000 seconds. As such, we implement two further changes in the following subsections, one to the step of solving the systems, and one to the step of analysis on the generating function $g(x)$, to produce our final APC-IP.

3.3 Computing the Generating Function

The first of two major performance issues with NAPC-IP was solving the system of equations, which occurs in line 4 of the Algorithms 2 and 3. When multiple graphs or long systems of equations were involved, the former Eliminate-Naive, which was left unchanged from APC-R, often stalls, crashs, or times out. Eliminate-Naive solves the system in its entirety, solving for T in APC-R, and T_0 for NAPC-IP. Our new method Eliminate-Optimized is optimized for function calls and larger systems of equations. These optimizations slow down the algorithm for small computations, but greatly enhance performance for difficult source code. Our original Eliminate-Naive, shown below, iterates through the system of equations to eliminate until the first variable, T in APC-R and T_0 in NAPC-IP. It also makes 2 interior loops through the system of equations, one to search for a solution if need be, and one to substitute that solution in. It searches for a solution in the case where an equation of the form $X = A$, where we are solving for X, contains X within A.

Algorithm 4 Eliminate-Naive (system S, vars V)

1: **if** len$(S) = 1$ **then**
2: **return** $S[0]$
3: sub ← expression for last variable $V[-1]$ from last equation $S[-1]$
4: **if** $V[-1] \in$ sub **then** ▷ If last equation has last variable on both sides
5: **for each** eq $\in S$ **do**
6: **if** $V[-1] \in$ eq **then**
7: sub $= SOLVE($eq$, V[-1])$
8: **if** len(sub) $= 1$ **then** ▷ Unique solution for variable
9: **break**
10: **for each** eq $\in S$ **do** ▷ Substitute solution throughout the system of equations
11: **if** $V[-1] \in eq$ **then**
12: eq ← substitute sub for $V[-1]$ in eq
13: **return** Eliminate-Naive$(S[:-1], V[:-1])$

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

1: $S = S_0, S_1, \ldots, S_n$
2: $T = \{\}$
3: **for each** $i \in len(S)$ **do** ▷ Solve each system for T_i
4: d ← substitution dictionary for eliminating
5: d $= \{\{V_k :$ all eqns containing $V_k\} \forall V_k \in S_i\}$
6: $T \leftarrow$ add Partial-Eliminate$(S_i, V_i,$ d$)$
7: d $= \{T_i : \{$all eqns $\in T$ containing $T_i\}\}$
8: $v = \{T_0, T_1, \ldots, T_n\}$ ▷ Variables for eliminating Ts
9: **return** Partial-Eliminate(T, v, d) ▷ Solve Ts for T_0

Our new algorithm, Eliminate-Optimized, makes two key changes: (1) solving each graph's system of equations separately before combining them, (2) using a dictionary to optimize iteration through each system. Solving each system of equations is done by a helper function, Partial-Eliminate, and entails eliminating all the $V_{i,j}$

Algorithm 6 PARTIAL-ELIMINATE (sys s, vars v, dict d)

```
1: if len(s) = 1 then
2:     return s[0]                                  ▷ Return T_i = A
3: var = v[−1] ← var to eliminate
4: eqn = s[−1] ← eqn of form var = A
5: sub ← right side of eqn, equal to var
6: if var ∈ sub then                                ▷ Must solve for var
7:     for each eq ∈ d[var] s.t. eq in bounds do
8:         sub = solve(eq, var)
9:         if len(sub) = 1 then                     ▷ Unique solution for var
10:            break
11: for each eq ∈ d[var] do
12:     eq ← substitute var with sub
13:     d ← update dict d after substitution
14: return PARTIAL-ELIMINATE(s[: −1], v[: −1], d)
```

vertex variables, which are specific to a single system of equations, since a graph can only call another graph's T variable. This results in a single equation for T_i in terms of other T_ks. Solving these lightly-coupled systems separately at first allows us to avoid working with complex polynomial equations for longer, thus speeding up our symbolic solving package. PARTIAL-ELIMINATE also uses a substitution dictionary `d` to avoid iterating once through the whole system of equations to find solutions and iterating through again to substitute them in. This replaces the loops on lines 5 and 10 in Algorithm 4 that iterate through the whole system for shorter loops that only pass through the necessary equations (usually 3 or less). After solving each system of equations for T_i, we combine these equations into a final system of equations containing only T variables, and solve these for T_0 with a final call to PARTIAL-ELIMINATE.

Continued Example: `is_even` and `is_odd`. We continue our example of `is_even` and `is_odd` to illustrate some of the differences in our methods. Recall the systems of equations:

System 0 (`is_even`)	System 1 (`is_odd`)
$T_0 = V_{0,0}x$	$T_1 = V_{1,0}x$
$V_{0,0} = V_{0,1}x + V_{0,2}x$	$V_{1,0} = V_{1,1}x + V_{1,2}x$
$V_{0,1} = V_{0,3}x$	$V_{1,1} = V_{1,3}x$
$V_{0,2} = T_1V_{0,3}x$	$V_{1,2} = T_0V_{1,3}x$
$V_{0,3} = 1$	$V_{1,3} = 1$

The old method would backsolve the system, treating these interrelated systems of equations as one singular, large system of equations, solving from $V_{1,3}$ to T_1 to $V_{0,3}$ to T_0. In cases where a variable being solved for was on both sides of the equation, it would iterate through the list of equations to isolate and solve it in another equation, and then plug that solution in. It results in the final equation $T_0 = x^3 + x^6 + T_0x^6$, which we refer to as Γ.

The new method would solve as follows. We show the separation of systems of equations for easier backsolving; recall that all the substitutions shown would be found using a dictionary. We begin with the two separated systems shown above, and start eliminating the first system until T_0 (step 1-3) to get:

$$T_0 = x^3 + T_1x^3$$

$$\begin{aligned} T_1 &= V_{1,0}x \\ V_{1,0} &= V_{1,1}x + V_{1,2}x \\ V_{1,1} &= x \\ V_{1,2} &= T_0V_{1,3}x \\ V_{1,3} &= 1 \end{aligned}$$

Steps 4-7: Eliminate the second system until T_0

$$T_0 = x^3 + T_1x^3 \qquad T_1 = V_{1,0}x = x^3 + T_0x^3$$

Step 8: Eliminate T_1 by substituting the equation on the right into the one on the left. We get $T_0 = x^3 + x^6 + T_0x^6$. Our ELIMINATE-OPTIMIZED also yields $T_0 = x^3 + x^6 + T_0x^6$ as our Γ.

After elimination, we calculate the discriminant of Γ, as described in [19]. If the discriminant has no real roots, we move into Case 1. Else, we move into Case 2. This example is Case 1, so we now solve for T_0 in Γ to find our generating function, exactly as we did in APC-R. After algebraic manipulation, we get $T_0 = \frac{x^3}{1-x^3}$. Therefore, we get the generating function $g(x) = \frac{T_0}{1-x} = \frac{x^3}{(1-x)(1-x^3)}$.

Our ELIMINATE-OPTIMIZED thus solves multiple systems of equations more efficiently, and arrives at the same gamma function, which can be turned into a generating function exactly as in APC-R. Now that we have our generating function $g(x)$, we must find an upper bound of g as the last step in computing APC.

3.4 From Generating Function to APC

Recall that a generating function encodes a sequence using the coefficient of the series expansion. Our generating function $g(x)$ represents the sequence $path(n)$. Then, the coefficient of x^n in the Taylor expansion of $g(x)$, $[x^n]g(x)$, is $path(n)$, or the number of paths through the CFG within depth n. In this step of the algorithm, we compute the asymptotic behavior of the closed form of $[x^n]g(x)$, resulting in the APC, which bounds $path(n)$ above.

In NAPC-IP, the second step that was computationally costly was the symbolic calculus inside Case 1 (SYMB-CALC). When the APC from the generating function is calculated, the number of required computations rapidly increase with more complex functions, growing faster than the number of nodes in the control flow graphs. For more complicated functions, part of this method requires the Taylor series of the generating function up to 100 terms. In practice, it can take more than 1000 seconds to compute all the required Taylor series terms. In APC-IP, we replace the SYMB-CALC step with our new method GETRGF, where the computations are bounded by the roots of the generating function $g(x)$'s denominator. In GETRGF, $g(x)$ and the General Expansion Theorem for Rational Generating Functions [11] are enough to compute APC.

General Expansion Theorem for Rational Generating Functions (GETRGF). *If $g(x) = P(x)/Q(x)$, where $Q(x) = q_0(1-\rho_1x)^{d_1}(1-\rho_2x)^{d_2}\dots(1-\rho_tx)^{d_t}$ and the numbers $(\rho_1, \rho_2, \dots, \rho_t)$ are distinct, and if $P(x)$ is a polynomial of degree less than $d_1+d_2+\cdots+d_t$, then $[x^n]g(x) = f_1(n)\rho_1^n + f_2(n)\rho_2^n + \cdots + f_t(n)\rho_t^n$, where each $f_k(n)$ is a polynomial of degree $d_k - 1$ with leading coefficient*

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

Below outlines a step by step procedure of our novel approach for getting the APC from generating function.

(1) From the generating function ($g(x)$), first get its denominator ($Q(x)$), numerator ($P(x)$), and the dictionary *rootsDict*. The dictionary *rootsDict* stores all the distinct roots r_i of the denominator along with its multiplicity d_i:

$$rootsDict = \{(r_i : d_i) \mid r_i \text{ is a root of } Q(x)\}$$

(2) If the degree of $P(x)$ is greater than the degree of $Q(x)$, the generating function does not satisfy the requirements of the theorem. Thus, we use polynomial long division to find the remainder $R(x)$ of $P(x)/Q(x)$, which has a lesser degree than $Q(x)$. Since $g(x) = P(x)/Q(x) = S(x) + R(x)/Q(x)$ for some finite polynomial $S(x)$, $S(x)$ can only affect the Taylor series expansion of $g(x)$ for small powers of n, and thus would not change the asymptotic path complexity of $[x^n]g(x)$. Therefore, we can simply replace $P(x)$ with $R(x)$ when finding the asymptotic upper bound of $g(x)$'s coefficients, in order to satisfy the theorem conditions.

(3) Then, the reciprocal ρ_i of each of the roots r_i in *rootsDict* is computed. The multiplicity d_i is also recorded:

$$rhoDict = \{(\rho_i = \frac{1}{r_i} : d_i) \mid \forall\, r_i \in rootsDict\}$$

(4) To find the dominant term in $[x^n]g(x)$, the first step is to find the maximum magnitude of all the ρ_i, name it ρ_{max}:

$$\rho_{max} = max\{|\rho_i| \mid \rho_i \in rhoDict\}.$$

Let m be the highest multiplicity of all the ρ_i such that $|\rho_i| = \rho_{max}$:

$$m = max\{d_i \mid \forall\, |\rho_i| = \rho_{max}\}$$

Among all the ρ_i such that $|\rho_i| = \rho_{max}$, only consider the ones with multiplicity m. Let

$$S = \{\rho_k \mid (d_k = m) \wedge (|\rho_k| = \rho_{max})\}.$$

(5) Then the a_k can be computed for each $\rho_k \in S$ using the above formula. That is,

$$A = \{a_k \mid \forall\, \rho_k \in S, \quad a_k = \frac{P(1/\rho_k)}{(d_k-1)!q_0 \prod_{j\neq k}(1-\rho_j/\rho_k)^{d_j}} \quad \text{where } \rho_j \in rhoDict\}. \tag{2}$$

Note that q_0 is the constant term in $Q(x)$. Further note that the product includes all $\rho_j \in rhoDict$, not merely the $\rho_j \in S$.

(6) After computing all the a_k, sum them to compute the coefficient

$$c = \sum_{a_k \in A} a_k$$

for the leading term of $[x^n]G(x)$. The leading term itself is the APC, and it can be described by $APC = c \cdot n^{m-1} \cdot \rho_{max}^n$.

Instead of computing path complexity and then obtaining its asymptotic behavior to find the APC, this new method allows us to go straight from the generating function directly to APC.

Example: `partition`. To further explain, we present as an example an interprocedural `partition` function that calls a `swap` function. We compute the APC starting from the generating function using our new method.

The generating function for `partition` is computed by the methods we have already explained:

$$g(x) = \frac{P(x)}{Q(x)} = \frac{x^6}{(x-1)(x^8+x^4-1)}.$$
$$= x^6 + x^7 + x^8 + x^9 + 2x^{10} + 2x^{11} + 2x^{12} + 2x^{13} + 4x^{14} + 4x^{15} \ldots$$

Then, the APC for `partition` can be computed using GETRGF.

```
int partition(int arr[], int low, int high){
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++){
        if (arr[j] < pivot){
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

Figure 9: Source code for `partition` function.

Step 1-2: The numerator is $P(x) = x^6$ and the denominator is $Q(x) = (x-1)(x^8+x^4-1)$. The degree of the numerator is less than the denominator, so the generating function satisfy GETRGF restriction. The roots of $Q(x)$ are $r_1 = -0.887$, $r_2 = 0.887$, $r_3 = 1$, $r_4 = -0.798 - 0.798i$, $r_5 = -0.798 + 0.798i$, $r_6 = -0.887i$, $r_7 = 0.8872i$, $r_8 = 0.798 - 0.798i$, $r_9 = 0.798 + 0.798i$. They all have multiplicity 1.
Step 3: Compute the ρ's by taking the reciprocal of the roots: $\rho_1 = -1.128$, $\rho_2 = 1.128$, $\rho_3 = 1$, $\rho_4 = 0.627 + 0.627i$, $\rho_5 = -0.627 - 0.627i$, $\rho_6 = 1.128i$, $\rho_7 = -1.128i$, $\rho_8 = 0.627 + 0.627i$, $\rho_9 = 0.627 - 0.627i$.
Step 4: The maximum magnitude of ρ is $\rho_{max} = 1.128$. Note that $\rho_1, \rho_2, \rho_6, \rho_7$ all have the same maximum magnitude. Among them, the maximum multiplicity is 1, so $m = 1$. Thus, the S contains

$$\begin{aligned} S &= \{\rho_k \in rhoDict \mid (d_k = m = 1) \ \wedge \ (|\rho_k| = \rho_{max} = 1.128)\} \\ &= \{\rho_1, \rho_2, \rho_6, \rho_7\} \\ &= \{-1.128, 1.128, 1.128i, -1.128i\} \end{aligned}$$

Step 5: Compute the respective $A = \{a_1, a_2, a_6, a_7\}$. To do this, the constant term q_0 in $Q(x)$ is needed, which is 1 for this example. We use equation 2 with all $d_k = 1$, and $q_0 = 1$. Below, we show a detailed calculation of a_1 as an example.

$$\begin{aligned} a_1 &= \frac{P(1/\rho_1)}{(d_1-1)!q_0 \prod_{j\neq 1}(1-\rho_j/\rho_1)^{d_1}} = \frac{(1/\rho_1)^6}{(1-1)! \cdot 1 \prod_{j\neq 1}(1-\rho_j/\rho_1)} \\ &= \frac{(1/\rho_1)^6}{(1-\frac{\rho_2}{\rho_1})(1-\frac{\rho_3}{\rho_1})(1-\frac{\rho_4}{\rho_1})\cdots(1-\frac{\rho_9}{\rho_1})} \\ &= 0.047 - 3.968 \cdot 10^{-18} i \end{aligned}$$

With similar computation, we get $a_2 = 0.776$, $a_6 = -0.049 + 0.044i$, and $a_7 = -0.049 - 0.044i$.
Step 6: Now we have

$$c = \sum_{a_k \in A} a_k = 0.724 - 1.39 \cdot 10^{-17} i \approx 0.724$$

Finally we calculate

$$apc = c \cdot n^{m-1} \cdot \rho_{max}^n = 0.724 \cdot n^{1-1} \cdot 1.128^n = 0.724 \cdot 1.128^n$$

Thus, the APC for `partition` is $0.724 \cdot 1.128^n$. Alg 7 summarizes the algorithm GETRGF to compute asymptotic path complexity from the generating function.

4 EXPERIMENTS

We conducted a series of experiments to validate our new approach APC-IP and compare to existing approaches.

Algorithm 7 GETRGF ($g(x)$)

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

4.1 Experimental Setup

4.1.1 APC Implementation Versions. To demonstrate that our approach subsumes and improves upon prior work, we investigate multiple APC implementations:

APC-IP. This is our implementation of fully interprocedural and optimized asymptotic path complexity (Algorithm 2). We implemented this directly on top of the existing Metrinome software. As part of our ablation studies (see next section), we investigate the two modifications from NAPC-IP in APC-IP to evaluate the effects of each one.

APC-R. This is the implementation of APC analysis from Metrinome as given in the 2023 paper "Formalizing Symbolic Execution Path Explosion for Recursive Functions via Asymptotic Path Complexity" from the 2023 Formal Methods in Software Engineering (FormaliSE) proceedings [19]. This constitutes the most recent advance in APC analysis available for comparison. APC-R is able to perform *intra*prodecural APC analysis on single functions that do not call any other functions or APC analysis on functions that only make self-recursive function calls. That is, APC-R as implemented did not handle *inter*procedural APC analysis.

NAPC-IP. This is a "naive," non-optimized implementation of interprocedural asymptotic path complexity. This implementation takes APC-R and adds the absolute minimum necessary modification to apply APC-R to interprocedural code. APC-R solves a system of equations derived from the control flow graph of the analyzed function to produce its result. Our minimal modification amounts to relabelling variables in the several systems of equations created for mutually dependent functions under analysis such that the APC-R analysis can treat them as one large system of equations.

4.1.2 Summary of Experiments Conducted. We frame our experimental analysis around the three aforementioned APC implementations. This allows us to isolate and measure the effects of each optimization in our implementation, compare to prior implementations, and compare our approach to a naive but straightforward baseline. To that end we conducted the following experiments:

Ablation Study 1. We compare the optimized variable elimination strategy of APC-IP (Algorithm 5 Eliminate-Optimized) to the naive variable elimination strategy of NAPC-IP (Algorithm 4 Eliminate-Naive).

Ablation Study 2. We compare APC-IP's novel use and implementation of the generalized expansion theorem for regular generating functions (GETRGF, Algorithm 7) to NAPC-IP's symbolic calculus approach to bounding generating functions, (Symb-Calc).

Head-to-Head Comparisons. We run APC-IP, APC-R, and NAPC-IP on the same code to compare runtimes and the resulting APCs.

Klee Path Explosion and APC. We ran Klee on a subset of our benchmark to measure path explosion as symbolic execution exploration depth increases and compare to APC-IP.

4.1.3 Benchmark Programs. We choose an externally sourced benchmark of C algorithms, found at https://github.com/TheAlgorithms/C. This repository includes common C algorithms, such as sorting, searching, and dynamic programming algorithms. We also chose this repository for readability, documentation, and credibility. It is reviewed with 17.4K stars on GitHub and has 4.2K forks. This benchmark also contains programs which exercise various aspects of the different implementations. First, the benchmark contains non-recursive, recursive, and interprocedural functions. Second, the benchmark has functions that well-represent a variety of interacting control structures likely to yield variation in resulting APC values: straight-line code, nested conditionals, nested loops, loops and conditionals nested within one another, function calls in loops, and so on. Our benchmark contains the 76 functions from this repository, and we supplement it with three functions from our running examples: an even or odd function, a partition function, and a palindrome function, to create a benchmark of 79 functions.

4.1.4 Experiment Hardware Specifications. We performed the experiments on a 12th Gen Intel 505 MHz i7-12700 computer with Python v3.10.6, Ubuntu 22.04.2, and 16GB RAM.

4.2 Experimental Results

4.2.1 Ablation Experiments. To evaluate each of our two improvements, we conduct ablation studies, isolating our changes in APC-IP (Algorithm 2). Our first change is optimizing the elimination of variables in systems of equations (Algorithm 4) and the second is to completely replace our algorithm bounding the generation function using GETRGF (Algorithm 7). We refer to these processes as Eliminate and GETRGF.

Ablation Study 1. For the Eliminate tests, we compare not only the elimination, but the processing of the graphs into the system of equations. This is because these two processes are tightly coupled in Algorithm 5, which needs to do additional processing on the graphs to create dictionaries of graph edges that we use to optimize the variable elimination process. We compute the ratios of the runtimes of the non-optimized elimination strategy to the optimized elimination strategy for all benchmark functions. When this ratio is below 1, the additional inital overhead of the optimized version actually causes the runtime to be worse, and so the naive version is better. When the ratio is greater than 1, then the additional overhead of the optimization does indeed provide improvement.

Results of the first ablation study are in Figure 10. This graph shows the runtime performance ratios for benchmark functions, sorted by run-time of the non-optimized (naive) elimination method,

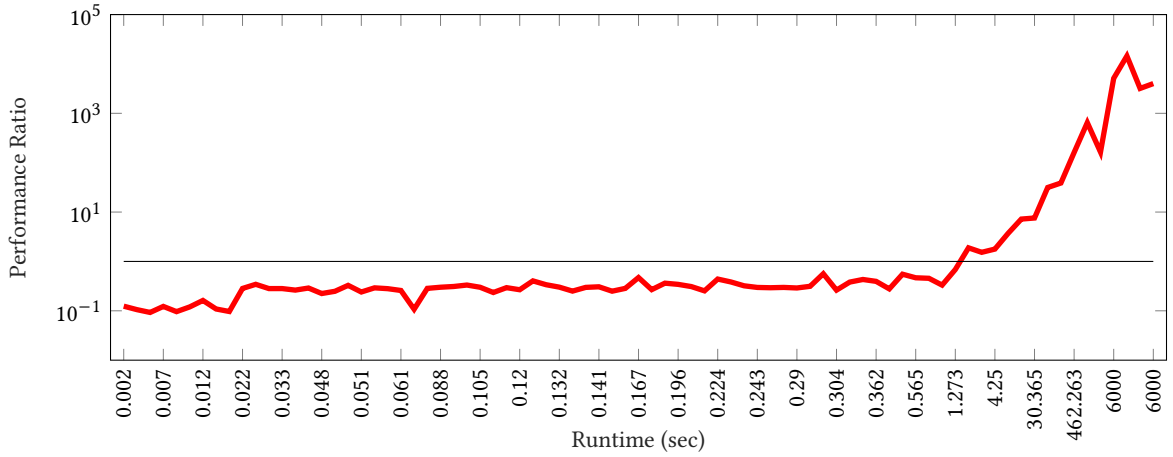

Figure 10: Eliminate Ablation Study Results. Performance ratio of Eliminate-Optimized to Eliminate-Naive (y-axis, log scale) for different run times in seconds (x-axis, linear scale).

with runtimes on the x-axis. Note that we ran the experiments with a timeout of 6000s, and so for several programs, the optimized version was able to complete quickly in cases where the non-optimized version could not even finish; this is the rightmost column labeled as 6000s. We see that APC-IP does add small overhead in processing time, which is visible for simpler programs (those for which elimination runtime is less than 1.273 seconds and where the ratio is less than 1), but is substantially faster for more complex programs. The optimized version completed well within the timeout for all programs. Seen in the figure, our optimization of the elimination method can result in significant performance improvements "when it matters." Thus, we conclude that our new eliminate expands Metrinome's scope to more complex programs, with only a minor cost in efficiency for programs that are already reasonably fast to analyze using either method anyway.

Ablation Study 2. For our GETRGF tests, we only compare the subset of the benchmark functions that engage Case 1 of Algorithm 2 (resp. Algorithm 1). Since in Case 2, both methods use the same, quick method of bounding the generating function, we do not compare the benchmark programs that fall into this case. This leaves us with 67 benchmark programs in our GETRGF study. For this study, we measure the runtime starting from directly after the generating function $g(z)$ has been computed to the time APC is determined.

Results of this study are summarized in Table 1. In this table, we show the time performance ratio of the naive approach which does not use GETRGF to our new approach which does use GETRGF. The table shows the number of functions from the benchmark that resulted in buckets of ranges of performance ratios. That is, 10 functions resulted in a performance ratio between 1 and 10, 34 functions resulted in a performance ratio between 10 and 100, 19 functions resulted in a performance ratio of 100 to 1000, and 4 functions resulted in a ratio greater than 1000. Also included in this table are the ranges of run times for the two different methods.

First we observe that the performance ratio is always greater than 1, and so GETRGF is always faster. We also see that without GETRGF, the approach does not scale well to some functions, requiring almost a minute of analysis time, whereas using GETRGF was always in the range of 8 to 191 milliseconds.

Table 1: GETRGF Ablation Study Results

Performance Ratio (Naive/GETRGF)	Count of functions	Naive time range (s)	GETRGF time range (s)
1-10	10	0.014 - 0.208	0.008 - 0.043
10-100	34	0.129 - 9.442	0.009 - 0.127
100-1000	19	10.218- 58.034	0.065 - 0.191
>1000	4	50.089 -50.923	0.026 - 0.040

4.2.2 APC-R, NAPC-IP vs APC-IP. We ran APC-IP, APC-R, and NAPC-IP on 79 benchmark programs. We summarize results in Table 2, highlight a representative subset of results in Table 3, and compare performance ratios of APC-IP to APC-R and NAPC-IP in Figures 11 and 12. All data is available and inspectable.

In the summarized results in Table 2 we separate functions that are intraprocedural or self-recursive only (first line) and functions that are interprocedural or may contain mutually recursive calls (second line). For intraprocedural and self-recursive functions, among 42 of them, APC-R runs the fastest in 7 of them, NAPC-IP runs fastest in 5 of them, and APC-IP runs fastest for 30 of them. For interprocedural functions or interprocedurally called recursive functions (37 total) APC-R cannot compute APC as it is not implemented to do so (NA for that column in the second line), NAPC-IP was fastest for 3 functions and APC-IP was fastest for 34 functions.

Specific outcomes from a subset of the summarized experimental results of Table 2 are shown in Table 3. The results are chosen to holistically represent our benchmark with the goal of showing a variety of functions and performing result comparisons. In all 79 functions, all three methods produces the same APC values. For example, we can look at lines 6 (Fibonacci Search) and line 10 (Multikey Quick Sort). We see that APCs are consistant with the value of 2.33×1.22^n, and that APC-IP was fastest as 0.88s compared over two seconds for the other two approaches. Furthermore, APC-IP

Table 2: Summary Results for APC-IP vs. APR-R vs. NAPC-IP. Table entries indicate how many times the given method was fastest for a specific benchmark function.

Function type	APC-R	NAPC-IP	APC-IP	Count
Intraproc. / Recursive	7	5	30	42
Interprocedural	NA	3	34	37
Total	7	8	64	79

elimination algorithm was slightly slower than a naive approach, and that using the GETRGF method for APC-IP was much faster. For line 10, APC-R does not apply, since that function makes interprocedural calls. NAPC-IP and APC-IP has the same result, but APC-IP is almost 9 times faster, the APC-IP elimination time saves us a lot of time.

In Figure 11, we show runtime performance ratios for APC-R compared to APC-IP. The x-axis is performance ratio ranges and the y-axis is the number of functions resulting in that performance ratio. We also indicate in blue the cases in which APC-R's runtime was under 1 second and in red the times when it was greater than 1 second. Note that the first bar with ratios 0.4 to 1 are cases in which APC-R does *better* than APC-IP, since the performance ratio is less than 1, and there are 10 such cases. What we see is that when the simple APC-R method is already quite fast, APC-IP does not help as much, but when APC-R is slower, the benefit of our innovations in APC-IP is more pronounced. Similar analysis and interpretation applies to Figure 12, which compares the performance of APC-IP and NAPC-IP on all 79 benchmark functions. Overall we conclude that APC-IP outperforms the other two methods in situations where analysis is more costly and so the benefits pay off for source code that is more difficult to analyze using path complexity analysis.

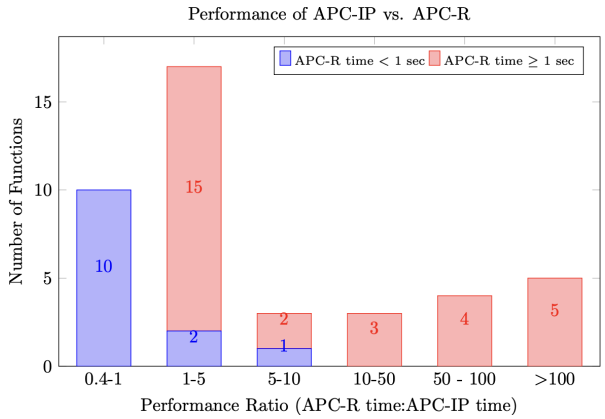

Figure 11: Performance of APC-IP vs APC-R

4.2.3 APC-IP vs Klee. Klee is a popular symbolic execution tool for C programs [6]. Previous work [3, 19] showed that intraprocedural asymptotic path complexity bounds the complexity class of the growth rate of number of paths found by Klee for increasing symbolic exploration depth. That is, if the asymptotic path complexity of a function is, say, quadratic, then the number of paths

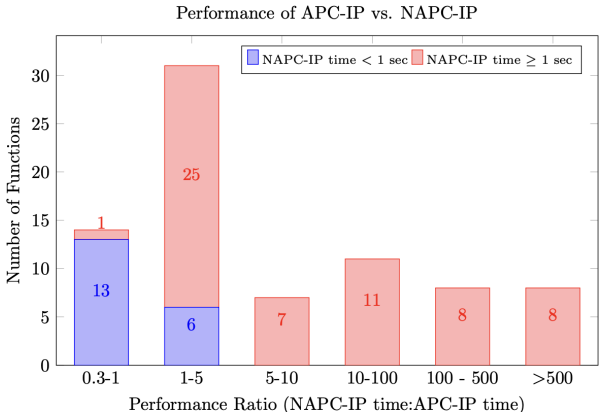

Figure 12: Performance of APC-IP vs NAPC-IP

explored by Klee is no worse than quadratic for that function as the exploration depth of Klee increases. And, if APC is exponential, then Klee's path explosion is no worse than exponential, and in most cases actually does become exponential. Further, APC can be computed faster than it takes to run Klee, so APC can be used to predict the path explosion behavior of Klee *before* running it.

Though the relation between APC and symbolic execution has been established, we still seek to confirm that our new APC-IP predicts Klee path explosion growth rates for interprocedural functions. It is important to note that symbolic execution discards infeasible paths through the code, while our APC-IP does not consider the actual path conditions. As such, APC-IP in some cases returns a non-strict upper bound of the complexity class of Klee's path explosion. Further, APC-IP uses path depth as a basis for its metric, while Klee uses branch count, causing mismatches in the exact coefficients. However, overall, APC-IP achieves its goal of predicting a coarse complexity class bound on Klee's path explosion.

We were able to run Klee on 57 out of our 79 benchmark functions. We were not able to meaningfully run Klee under its standard configurations on our entire benchmark due to limitations in the constraint solver backend, which prevents us from getting results from Klee, e.g. when complex floating point constraints are encountered during symbolic exploration. We tracked the number of paths explored by Klee for increasing exploration depths and found the best-fit curve through that data, as in prior works [19], to compute the growth rate of Klee's path explosion. We compared these path explosion rates to the APC values of the functions computed using APC-IP. While we do not provide all of the results directly in this paper, a hand-selected sample of representative results is given in Table 4, and the complete data is provided in our artifact.

In all but one case of Table 4, APC-IP bounds or matches the best fit for Klee. APC lies in the same complexity class as the Klee best fit expression in 4 of the 7 cases, and upper bounds it in 2 more. The exponent base is not expected to match Klee exactly because Klee bounds exploration by branch count, not CFG edge count. In our full data, in 46 out of the 57 programs, APC-IP is in the same complexity class as the Klee best fit line, and in 53 out of 57, we bound the Klee best fit line. In 3 cases, we do not have enough data for Klee to determine a best fit line, and in the one case shown in

Table 3: APC data on C files showing asymptotic path complexity (APC-R: recursive, NAPC-IP: naive interprocedural, APC-IP: interprocedural, APC runtime, ablation runtime, the best APC).

	Program Name	Fastest Metric[1]	APC-R	NAPC-IP	APC-IP	APC-R time(s)	NAPC-IP time(s)	APC-IP time	APC-IP eliminate time(s)	NAPC-IP eliminate time(s)	GETRGF time(s)	non-GETRGF time(s)
1	Even Odd §	APC-IP	NA	$n/3$	$n/3$	NA	0.22	**0.14**	0.08	0.01	0.01	0.16
2	Fibonacci (DP)	APC-IP	$n/3$	$n/3$	$n/3$	0.31	0.32	**0.26**	0.15	0.05	0.02	0.21
3	Palindrome (R) †	APC-R	$n/2$	$n/2$	$n/2$	**0.11**	**0.11**	0.12	0.05	0.01	0.01	0.06
4	Partition §	APC-IP	NA	$0.72 * 1.13^n$	$0.72 * 1.13^n$	NA	1.10	**0.40**	0.21	0.05	0.06	0.69
5	Shaker Sort §	APC-IP	NA	$0.12 * 1.26^n$	$0.12 * 1.26^n$	NA	32.04	**1.51**	0.92	0.27	0.09	30.69
6	Fibonacci Search	APC-IP	$2.33 * 1.22^n$	$2.33 * 1.22^n$	$2.33 * 1.22^n$	2.14	2.18	**0.88**	0.63	0.16	0.05	1.54
7	Insertion Sort (R) †	APC-IP	$0.11 * 1.35^n$	$0.11 * 1.35^n$	$0.11 * 1.35^n$	2.06	2.08	**0.52**	0.33	0.10	0.03	1.52
8	Cycle Sort §	APC-IP	NA	$0.04 * 1.36^n$	$0.04 * 1.36^n$	NA	63.76	**5.07**	4.01	30.36	0.14	31.46
9	Bucket Sort §	APC-IP	NA	NA	$0.004 * 1.39^n$	NA	>6000	**42.39**	36.53	NA	0.20	NA
10	Multikey Quick Sort §	APC-IP	NA	1.46^n	1.46^n	NA	5242.50	**102.10**	5.39	3489.53	case 2	32.24

[1] The "Fastest Metric" column shows the fastest APC metric applicable to the function.
† This version of the function is implemented recursively.
§ This version of the function involves interprocedual calls.
* In each row, the best time is highlighted in bold.

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.

Row 3 of Table 4, the APC-IP is quadratic while the Klee best fit line is exponential, though we suspect that this is due to overfitting in our bext-curve-fitting function. Overall, APC-IP predicts Klee path explosion behavior.

4.3 Experimental Takeaways

We demonstrated that APC-IP efficiently computes asymptotic path complexity, and provides a sound upper bound on the degree of Klee's path explosion when testing simple, recursive, or interprocedural programs. It successfully computes path complexity for interprocedural functions, which the previous APC-R could not. For intraprocedural functions, APC-IP matches APC-R's results, with faster runtime on complex functions. For interprocedural functions, by optimizing the elimination step and implementing GETRGF to bound the generating function, APC-IP efficiently computes correct APC for complex interprocedural functions, usually in under 10 seconds. APC-IP thus subsumes earlier APC work, and with drastic improvements on performance cost.

5 RELATED WORK

APC-IP builds on the 2023 recursive path complexity research[19], which itself extended earlier studies on non-recursive functions [2, 3]. Our approach covers both interprocedural and intraprocedural code, including recursive and mutually recursive functions. In the broader field, code complexity research measures the complexity of programs [9, 18, 20] including cognitive complexity focused on human code comprehension [7, 23, 25]. Other complexity metrics include Halstead complexity, based on code size and code element uniqueness [13] and Dependency degree for measuring code coupling complexity [4]. Control flow graph-based metrics like McCabe's cyclomatic [16] complexity and NPATH complexity [17] quantify different aspects of code complexity.

6 CONCLUSION

We described our work on computing asymptotic path complexity for interprocedural functions (APC-IP). Prior methods could not handle the theoretical complications or complexity and scale of interprocedural code. Our work provides algorithmic and mathematical formalization that not only allows for the computation of path complexity in the presence of interprocedural calls but also subsumes all previous methods in terms of runtime and accuracy to make asymptotic path complexity tractable for a wider range of programs.

APC-IP serves as a useful tool to compute code complexity of medium-sized programs in the context of automated software testing. Future work involves continuing to scale APC to meet the scale of today's industry code-bases. We also hope to extend APC to process programs in more common programming languages, such as Python and Java. Finally, we hope to conduct more robust experiments directly comparing the performance and refinement of the APC-IP algorithm with other complexities, such as NPATH and cyclomatic complexity.

7 DATA AVAILABILITY

The source code of APC-IP is avaialable along with our experimental results, benchmark programs, and scripts for reproducing our data. An explanation on how to run APC-IP and how to replicate these experiments is included in the provided artifact as a README file. The most up-to-date information about Metrinome can be found at our public repository.[1]

ACKNOWLEDGMENTS

We thank Eli Pregerson, Sora Cullen-Baratloo, Josh Cordova, and Gabe Bessler for their support and consultation. This project supported by the Harvey Mudd College Anguilo Summer Research fund and NSF Awards 2008640 and 1950885.

[1] https://github.com/hmc-alpaqa/metrinome

REFERENCES

[1] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-Driven Compositional Symbolic Execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 367–381.

[2] Lucas Bang, Abdulbaki Aydin, and Tevfik Bultan. 2015. Automatically computing path complexity of programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 61–72. https://doi.org/10.1145/2786805.2786863

[3] Gabriel Bessler, Josh Cordova, Shaheen Cullen-Baratloo, Sofiane Dissem, Emily Lu, Sofia Devin, Ibrahim Abugharar, and Lucas Bang. 2021. Metrinome: Path Complexity Predicts Symbolic Execution Path Explosion. In *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*. IEEE.

[4] Dirk Beyer and Ashgan Fararooy. 2010. A Simple and Effective Measure for Complex Low-Level Dependencies. *2010 IEEE 18th International Conference on Program Comprehension* (2010), 80–83.

[5] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. 2008. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 351–366.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[7] G. Ann Campbell. 2018. Cognitive Complexity: An Overview and Evaluation. In *Proceedings of the 2018 International Conference on Technical Debt* (Gothenburg, Sweden) *(TechDebt '18)*. Association for Computing Machinery, New York, NY, USA, 57–58. https://doi.org/10.1145/3194164.3194186

[8] N. Chomsky and M.P. Schützenberger. 1959. The Algebraic Theory of Context-Free Languages. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 26. Elsevier, 118–161.

[9] Nasir Eisty, George Thiruvathukal, and Jeffrey Carver. 2018. A Survey of Software Metric Use in Research Software Development. 212–222. https://doi.org/10.1109/eScience.2018.00036

[10] P. Flajolet and R. Sedgewick. 2009. *Analytic Combinatorics*. Cambridge University Press.

[11] Robert L. Graham, Donald E. Knuth, and Oren Patashnik. 1994. *Concrete Mathematics: A Foundation for Computer Science* (2nd ed.). Addison-Wesley Publishing Company, USA.

[12] Hermann Gruber, Jonathan Lee, and Jeffrey O. Shallit. 2012. Enumerating regular expressions and their languages. *CoRR* abs/1204.4982 (2012).

[13] Maurice H. Halstead. 1977. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., USA.

[14] Saparya Krishnamoorthy, Michael S. Hsiao, and Loganathan Lingappan. 2010. Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs. In *2010 19th IEEE Asian Test Symposium*. 59–64. https://doi.org/10.1109/ATS.2010.19

[15] H. Levy and F. Lessman. 1992. *Finite Difference Equations*. Dover Publications.

[16] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Trans. Software Eng.* 2, 4 (1976), 308–320.

[17] Brian A. Nejmeh. 1988. NPATH: A Measure of Execution Path Complexity and Its Applications. *Commun. ACM* 31, 2 (Feb. 1988), 188–200.

[18] Alberto S. Nuñez-Varela, Héctor G. Pérez-Gonzalez, Francisco E. Martínez-Perez, and Carlos Soubervielle-Montalvo. 2017. Source code metrics: A systematic mapping study. *Journal of Systems and Software* 128 (2017), 164–197. https://doi.org/10.1016/j.jss.2017.03.044

[19] Eli Pregerson, Shaheen Cullen-Baratloo, David Chen, Duy Lam, Max Szostak, and Lucas Bang. 2023. Formalizing Path Explosion for Recursive Functions via Asymptotic Path Complexity. In *2023 IEEE/ACM 11th International Conference on Formal Methods in Software Engineering (FormaliSE), Melbourne, Australia, May 14-15, 2023*. IEEE.

[20] R.S. Pressman and B.R. Maxim. 2019. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education.

[21] R. Sedgewick and P. Flajolet. 2013. *An Introduction to the Analysis of Algorithms: Introdu Analysi Algori_p2*. Pearson Education.

[22] Michael Sipser. 2013. *Introduction to the Theory of Computation* (third ed.). Course Technology, Boston, MA.

[23] Harry M. Sneed. 1995. Understanding Software through Numbers: A Metric Based Approach to Program Comprehension. *Journal of Software Maintenance* 7, 6 (nov 1995), 405–419. https://doi.org/10.1002/smr.4360070604

[24] Richard P. Stanley. 2011. *Enumerative Combinatorics: Volume 1* (2nd ed.). Cambridge University Press, USA.

[25] Dinuka R. Wijendra and K. P. Hewagamage. 2022. Cognitive Complexity Reduction through Control Flow Graph Generation. In *2022 IEEE 7th International conference for Convergence in Technology (I2CT)*. 1–7. https://doi.org/10.1109/I2CT54291.2022.9824923

[26] Herbert S. Wilf. 2006. *Generatingfunctionology*. A. K. Peters, Ltd., USA.

Received 16-DEC-2023; accepted 2024-03-02