Path Complexity Analysis for Interprocedural Code

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

ACM Reference Format:
Mira Kaniyur, Ana Cavalcante-Studart, Yihan Yang,, Sangeon Park, David Chen, Duy Lam, Lucas Bang. 2024. Path Complexity Analysis for Interprocedural Code. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3639478.3643527

1 INTRODUCTION

Symbolic execution's path explosion is a critical issue in software testing, quantified by Asymptotic Path Complexity (APC) [3]. APC, more precise than cyclomatic [6] or NPATH [7] complexities, measures the effort to cover paths in code analysis [1]. It's vital for testing, setting limits on path growth for tools like Klee [4], focusing previously on intraprocedural code [2, 8]. Our advancement, APC-IP, extends APC to interprocedural analysis, enhancing scalability and encompassing earlier models.

Contributions. We claim the following research contributions. (1) *APC-IP Formalization:* Extension of the theory and algorithms established for APC to account for interprocedural functions; (2) *Optimization Over All Previous APC Approaches.* Replacing theoretical steps in previous algorithms to improve performance for both interprocedural and intraprocedural code; (3) *APC-IP Implementation.* Implementing APC-IP atop Metrinome, an existing APC analysis tool; (4) *APC-IP Experimental Validation.* APC-IP computes accurate APC for both intraprocedural and interprocedural, and is the fastest option to process complex source code.

2 PATH COMPLEXITY BACKGROUND

Given a program, P, the APC of P is a function that bounds the number of execution paths of P as function of execution length. The execution length is the number of edges traversed in the control flow graph of P. For instance, if the APC of P is $O(n^2)$, then the number of different execution paths of P grows at most quadratically as execution length increases. Previous work described how to compute intraprocedural APC using the combinatorics of finite automata [1, 2], and more recent work implemented APC for self-recursive functions using the combinatorics of context free grammars, called APC-R [8].

ICSE-Companion '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0502-1/24/04.
https://doi.org/10.1145/3639478.3643527

3 NOVEL APC ALGORITHMS

Three major changes to APC-R produce a computationally viable APC algorithm for interprocedural functions. The first change is sufficient to create a *correct naive* algorithm (NAPC-IP). Two further optimizations are create a *tractable* APC-IP.

Naive Interprocedural APC. APC-R assigns a labelled variable to each node in the function's control flow graph. Using analytic combinatorics, a single system of equations is created, which is solved for a generating function g that captures APC behavior. g is a rational function whose Taylor expansion is an infinite polynomial encoding the exact execution path count in its coefficients. Asymptotic analysis of g yields the APC of self-recursive and non-recursive intraprocedural functions. Applying this approach to interprocedural code requires the introduction of mutually coupled systems of equations, one system for each control flow graph of each function. A naive interprocedural algorithm is then achieved by carefully relabelling variables for the nodes of the multiple control flow graphs so as to create one large system of combinatorial equations, and then using the same analysis as APC-R to compute APC. This relabeling modification is theoretically sufficient to extend APC-R's algorithm to operate on interprocedural programs. We thus describe APC-R with our labeling modification as NAPC-IP (naive APC interprocedural). However, empirically, NAPC-IP is inefficient and unable to process interprocedural functions with any significant complexity. For example, processing an interprocedural `MergeSort` function took over 2000 seconds (Table 1). 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.

Optimized Elimination of Equations. The first of two major performance issues with NAPC-IP was solving the system of equations. With multiple graphs or long systems of equations, the function Eliminate, which was left unchanged from APC-R, often stalled, crashed, or timed out. It solved the system in its entirety by iterating through the system of equations, eliminating one variable at a time using standard back-substitution techniques. To eliminate each variable, it also had to make 2 interior iterations through the system due to the nature of the systems of equations arising from the control flow graphs.

Our new method Eliminate-Optimized is designed and optimized to solve systems of equations from control flow graphs of interprocedural and complex functions. It separates each graph's system of equations and solves them independently. It uses multiple dictionaries for solving and substituting each variable to avoid interior iterations through the whole system of equations. After solving each system of equations, it combines these equations into a final system of equations and solves it. Our Eliminate-Optimized thus handles complex or interprocedural systems more efficiently, and

arrives at the same result, Γ, which becomes a generating function exactly as in APC-R.

GETRGF: Optimizing the Bounding of the Generating Function. The second bottleneck step was our old method of asymptotically bounding the generating function $g(x)$ using symbolic calculus. The algorithm has two cases after computing $g(x)$, depending on the roots of the discriminant, the more common of which needed symbolic calculus to bound $g(x)$. The number of required computations rapidly increased with function complexity, sometimes requiring the Taylor series of the generating function up to 100 terms, which can take more than 1000 seconds to compute. APC-IP replaces the symbolic asymptotic calculus of APC-R with the formula forthe General Expansion Theorem for Rational Generating Functions GETRGF, where the computations are bounded by the roots of the generating function g's denominator. Applying GETRGF[5] to $g(x)$ is enough to compute APC.

General Expansion Theorem for Rational Generating Functions (GETRGF). *If $g(x) = P(x)/Q(x)$, where $Q(x) = q_0(1-\rho_1 x)^{d_1}(1-\rho_2 x)^{d_2}\ldots(1-\rho_t x)^{d_t}$ and the numbers $(\rho_1, \rho_2, \ldots, \rho_t)$ are distinct, and if $P(x)$ is a polynomial of degree less than $d_1+d_2+\cdots+d_t$, 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*

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

We directly implement this algorithm, except when the degree of our $g(x)$'s $P(x)$ is greater than the degree of its $Q(x)$, violating the theorem's conditions. However, we can represent $g(x)$ as $S(x) + R(x)/Q(x)$ where the remainder $R(x)$ has a lower degree than $Q(x)$ and $S(x)$ is a finite polynomial. Since $S(x)$ only affects the Taylor series expansion of $g(x)$ for small powers of n, we can replace $P(x)$ with $R(x)$ when finding the asymptotic upper bound of $g(x)$'s coefficients to satisfy the theorem conditions. With this small modification, implementing this theorem can always bound $g(x)$ asymptotically. Note we calculate the APC directly, rather than calculating the path complexity and then simplifying to the APC, as teh symbolic asymptotic calculus of APC-R does.

APC-IP Algorithm Below is the full APC-IP algorithm. As noted, the first change to relabel the graph nodes is sufficient for NAPC-IP, but the additional two optimizations make the final APC-IP computationally viable.

Algorithm 1 APC-IP: Asymptotic-Analysis of Interprocedural Functions (Program P)

```
1: G_0, ..., G_n ← Control-Flow-Graphs(P)  ▷ Change 1 in APC-IP (& NAPC-IP)
2: S = S_0, ..., S_n ← Systems(G_0, ..., G_n)
3: Γ ← Eliminate-Optimized(S)                ▷ Change 2 in APC-IP
4: R ← roots of the discriminant of Γ
5: if (R = ∅) then                           ▷ Case 1
6:     g(z) = Solve(Γ, T_0)/(1 − z)
7:     apc = GETRGF(g, R) ← upper bound of g ▷ Change 3 in APC-IP
8: else                                      ▷ Case 2
9:     r* ← minimum positive real root in R
10:    apc ← (1/r*)^n
11: return apc
```

4 RESULTS

The two changes made with Metrinome produce significant outcomes. Table 1 shows a sample result of APC-IP compare with NAPC-IP and APC-R.

APC-IP subsumes APC-R. We see that APC-IP produces APCs consistent with APC-R on simple and recursive functions while running faster on more complex ones. The result for the functions APC-R cannot compute, the interprocedural ones, are shown as '-'.

APC-IP outperforms NAPC-IP on complicated functions. While Both APC-IP and NAPC-IP can handle interprocedural functions, with the optimization on Eliminate and the new algorithm using GETRGF, APC-IP outperforms NAPC-IP. In the case when APC-IP is slower than NAPC-IP, both is under 1 seconds. But when NAPC-IP times out in 2000 seconds, APC-IP can produce the APC in significantly less time. This is expected because Eliminate-Optimized is better at handling more complicated functions.

Table 1: APC-IP vs. NAPC-IP and APC-R

	Method			Runtime (sec)		
Function	APC-IP	NAPC-IP	APC-R	APC-IP	NAPC-IP	APC-R
Factorial-R	$2n/5$	$2n/5$	$2n/5$	0.20	50.32	50.30
Factorial -I	$n/2$	$n/3$	-	0.21	0.29	-
Palindrome -R	$n/2$	$n/2$	$n/2$	0.13	0.09	0.09
L.C. Subseq	$.68(1.31^n)$	$.68(1.31^n)$	$.68(1.31^n)$	3.92	65.92	64.80
Quicksort -I	1.24^n	1.24^n	-	0.50	0.27	-
Quicksort -R	1.35^n	1.35^n	1.35^n	0.75	0.39	7.29
Insertion sort	$.08(1.35^n)$	$.08(1.35^n)$	$.08(1.35^n)$	2.09	8.55	8.63
Merge Sort -I	1.39^n	timeout	-	19	> 2000	-
Merge Sort	$.006(1.42^n)$	$.006(1.42^n)$	$.006(1.42^n)$	3.27	1410	979
BST -I	$.0001(n^3 1.21^n)$	timeout	-	125	> 2000	-

5 CONCLUSION AND FUTURE WORK

Previous work on APC and Metrinome ignores interprocedural calls between functions. Renaming the vertices along with our two optimizations, allows Metrinome to accurately and efficiently compute the APC of interprocedural functions as well as more complex functions. Future work on APC might extend it to handle functions containing inter-file calls. This would likely require further computational optimizations of Metrinome, but would allow Metrinome to analyze much of industry code, making it a useful tool for software developers to analyze their code's path complexity.

REFERENCES

[1] 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

[2] Gabriel Bessler, Josh Cordova, Shaheen Cullen-Baratloo, Sofiane Dissem, Emily Lu, Sofia Devin, Ibrahim Abughararh, 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.

[3] 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.

[4] 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.

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

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

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

[8] 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.