Metrinome: Path Complexity Predicts Symbolic Execution Path Explosion

Gabriel Bessler, Josh Cordova, Shaheen Cullen-Baratloo, Sofiane Dissem, Emily Lu*, Sofia Devin, Ibrahim Abughararh, Lucas Bang
{gbessler,jcordova,scullenbaratloo,sdissem,sdevin,iabughararh,lbang}@hmc.edu,elu3370@scrippscollege.edu
Harvey Mudd College, *Scripps College
Claremont, California, USA

ABSTRACT

This paper presents METRINOME, a tool for performing automatic path complexity analysis of C functions. The path complexity of a function is an expression that describes the number of paths through the function up to a given execution depth. METRINOME constructs the control flow graph (CFG) of a C function using LLVM utilities, analyzes that CFG using algebraic graph theory and analytic combinatorics, and produces a closed-form expression for the path complexity as well as the asymptotic path complexity of the function. Our experiments show that path complexity predicts the growth rate of the number of execution paths that KLEE, a popular symbolic execution tool, is able to cover within a given exploration depth. Metrinome is open-source, available as a Docker image for immediate use, and all of our experiments and data are available in our repository and included in our Docker image.

CCS CONCEPTS

• **Software and its engineering → Software organization and properties**; **Software notations and tools**.

KEYWORDS

Path complexity, automated testing, symbolic execution.

ACM Reference Format:
Gabriel Bessler, Josh Cordova, Shaheen Cullen-Baratloo, Sofiane Dissem, Emily Lu*, Sofia Devin, Ibrahim Abughararh, Lucas Bang. 2020. Metrinome: Path Complexity Predicts Symbolic Execution Path Explosion. In *Proceedings of FULL CONFERENCE NAME (ICSE 2021).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Confidence in modern automated software testing relies on the ability of tools to achieve path coverage. Symbolic execution is one of the most prominent automated verification techniques, but suffers from the path explosion problem [9]. Path complexity is a code metric that formalizes and quantifies the severity of path explosion for a given function [1]. Given an execution depth bound n for a function f, the *path complexity* (PC) of f is a function $pc(n)$ that provides an upper bound on the number of execution paths of f with length up to n. These expressions can be large and cumbersome. Thus, we compute the more succinct *asymptotic path complexity* (APC), the dominating term of the path complexity function. We show that APC correlates with the number of paths explored by symbolic execution within a given exploration depth bound.

This paper describes our implementation of APC in our METRINOME tool. Our experimental results indicate that APC is indeed able to predict the behavior of the symbolic execution tool KLEE [3] on several algorithms implemented in C. Our source code, benchmarks, experiment scripts, and experimental data are available in our public repo[1] as well as in a ready-to-run image on Dockerhub[2].

To summarize, the contributions of this work are:

(1) A practical tool, METRINOME for computing path complexity and asymptotic path complexity, as well as Cyclomatic complexity and NPath complexity. METRINOME can compute code metrics for C, C++, Java and Python, but we focus on C in this paper.

(2) Empirical demonstration that APC is a fast way to predict the behavior of KLEE before running it.

2 BACKGROUND

Various metrics for the complexity of a given piece of code have been proposed. The most well-known are McCabe's *cyclomatic complexity* (the number of linearly independent paths) [11] and Nejmeh's *NPATH complexity* (the number of paths that take no edge more than once) [12]. These metrics have been used to suggest code refactoring or to predict the difficulty of testing or maintaining a segment of code [7, 8]. Code complexity metrics typically look only at the structure of the code, and so their computation is based on a standard representation of the structure, the control flow graph.

Path complexity was proposed in 2015 by Bang *et al.*, and implemented as a tool called PAC for Java functions [1]. This previous work demonstrated that path complexity is a more refined metric than popular existing metrics, cyclomatic and NPATH complexity.

Our metrics, PC and APC, are both based on the CFG of a function. We define the path complexity of a function f to be a function $pc(n)$ such that for any depth $n > 0$, $pc(n)$ is the number of paths from the start node to the exit node in the CFG of f with length (number of edges) less than or equal to n. We then define the asymptotic path complexity $apc(n)$ as the dominating term of $pc(n)$.

Note that path complexity is *exactly* equal to the number of paths of a given path length in the CFG, but may be an over-approximation of the number of paths through the function f up to execution depth

[1]https://github.com/hmc-alpaqa/metrinome
[2]https://hub.docker.com/orgs/harveymudd/metrinome

ICSE 2021, CONFERENCE DATE, CONFERENCE LOCATION

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

n, since PC does not take into account the satisfiability of branch conditions and so counts paths that may not be feasible. On the other hand, PC is a sound upper bound on the number of paths of execution depth n through f and is fast to compute because it looks only at the graph structure and is computed by using efficient linear algebra packages with the adjacency matrix of the CFG. APC is obtained by taking the dominant term in the PC expression; that term dictates the overall path complexity of the underlying program. This makes comparing the complexity of two programs easier: many programs have path complexities that contain many polynomial and exponential terms and so reporting just the highest order term is a succinct way to summarize the path complexity.

Path Complexity Examples. We provide three examples of C code with their corresponding CFGs, PCs, and APCs in Figure 1.

3 COMPUTING PATH COMPLEXITY

We give a brief synopsis of the theory behind computing path complexity [1] in order to present a self-contained paper. We use techniques from algebraic graph theory and analytic combinatorics to count the number of execution paths of a CFG [2, 13]. Given a CFG G with nodes N and edges E, and a depth n, we can compute the generating function $g(z)$ such that the n^{th} Taylor series coefficient of $g(z)$, denoted $[z^n]g(z)$, is equal to pc(n):

$$g(z) = \frac{\det(I - zT : |N|, 1)}{(-1)^{|N|+1} \det(I - zT)}. \quad (1)$$

where T is the *augmented transfer-matrix* (an adjacency matrix with $T_{|N|,|N|} = 1$), $(M : i, j)$ denotes the matrix obtained by removing row i and column j from M, and I is the identity matrix. From $g(z) = p(z)/q(z)$ we can derive a closed-form function f(n) as a sum of products of simple polynomial and exponential terms such that $\text{pc}(n) = \Theta(f(n))$. The form of $f(n)$ is determined by

$$f(n) = \sum_{i=1}^{D} \sum_{j=0}^{m_i - 1} c_{i,j} n^j \left(\frac{1}{|r_i|}\right)^n, \quad (2)$$

where $q(z)$ had D distinct roots, r_i is the i^{th} root of $q(z)$, m_i is the multiplicity of r_i, and c_i are coefficients determined by $|N|$ terms of the Taylor expansion of $g(z)$. Since $path(n) = [z^n]g(z)$, we can define a system of $|N|$ equations and $|N|$ unknowns. This system can be solved for the coefficients $c_{i,j}$ via linear algebra. This gives a closed form function for $pc(n)$. We define $apc(n)$ as $O(pc(n))$ using standard asymptotic analysis. This allows us to determine if the PC asymptotically behaves as a constant, polynomial, or exponential.

4 MEASURING SYMBOLIC PATH EXPLOSION

Bang *et al.* suggested that can be used as a predictor of path explosion during symbolic execution, but did not empirically verify this [1]. In the current work, we seek to examine this claim. In order to do so, we needed to quantify the path explosion problem.

KLEE is a popular open-source tool that uses symbolic execution to discover bugs and automatically generate tests for a given C program. This can be a computationally intensive process due to the well-known path explosion problem of symbolic execution. For a given test function, we use METRINOME's built-in KLEE utilities to generate a symbolic execution driver that marks each function input parameter as symbolic. We then use KLEE's `max-depth` parameter to collect statistics on how the number of generated paths varies with exploration depth bounds. Finally, we find the best-fit constant, polynomial, or exponential function for the collected data. For example, in Figure 2 we can see the results of this procedure for two example functions: `Selection Sort` and `Monotone Array Check` (checks if an array is monotonically increasing or decreasing). The number of paths explored by KLEE on `Selection Sort` grew exponentially with the exploration depth, while `Monotone Array Check` exhibited a clearly quadratic trend.

We used METRINOME to compute APCs of $O(1.27^n)$ and $O(n^2)$ respectively. Our experimental results show that APC either matches or soundly upper bounds the asymptotic growth complexity class in the number of paths generated by KLEE.

5 IMPLEMENTATION

In the paper introducing path complexity, a tool was made for computing PC and APC of Java programs. Our tool includes this functionality and extends it substantially. METRINOME runs within a Docker image which can be built locally or downloaded from Dockerhub, ensuring all dependencies and examples are present within the environment. Overall, METRINOME is implemented as a REPL (read-eval-print-loop), which means that rather than executing individual commands in the shell, it provides its own 'path complexity shell' where a series of commands can be executed. In order to implement this, we use Python's built in Cmd module.

There are 4 main components to the architecture. The first of these is the Command module. This handles the parsing of user input and calling the necessary methods from other modules. The second component is the set of converters, which turn source code files into CFGs. Each converter follows the same `Converter` interface (abstract class in Python), which means it is simple to add converters for more languages in the future. The third component is the 'metrics component', responsible for computing a single metric from a CFG, and implementing the `Metric` interface. The fourth component is the KLEE handler, which converts standard C files into files which can be used by KLEE, and provides commands for running KLEE within the REPL.

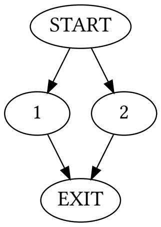
Given that METRINOME is meant to process a large number of files, performance is a strong priority. A key advantage of the REPL is that it caches all objects in memory. For example, CFGs are stored as Graph objects rather than files. This facilitates experiment execution and reduces runtime. In order to do symbolic computations in Python, we use sympy. This is the main bottleneck for computing APC as we need to obtain symbolic determinants. To work around this, we modified the APC metric component to use a graph search instead of one of the two determinants, significantly speeding up metric computation.

6 EXPERIMENTS

Our experiments address the following research questions:

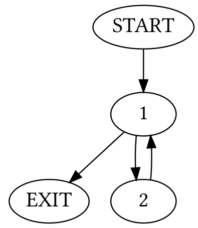
- **RQ1:** Is APC an effective way to predict the asymptotic rate at which the number of paths explored by KLEE grows with respect to the symbolic execution exploration depth bound?
- **RQ2:** If APC is an effective predictor, is it efficient compared to simply running KLEE and counting the paths generated?

```
int parity(int num) {
  if (num % 2 == 0)
    return 0;
  else
    return 1;
}
```



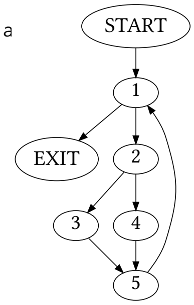
Path complexity: 2
Asymptotic Path complexity: $O(1)$

```
int palindrome(int num) {
  int rev_num = 0, rem, temp;
  temp = num;
  while (temp != 0) {
    rem = temp % 10;
    rev_num = rev_num * 10 + rem;
    temp /= 10;
  }
  return reverse_num == num;
}
```



Path complexity: $0.5n + 0.5$
Asymptotic Path complexity: $O(n)$

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



Path complexity: $2.21 \times 1.19^n + 6.05$
Asymptotic Path complexity: 1.19^n

Figure 1: Examples with resulting CFGs and APCs. Left: finds the parity of input integer, with APC of $O(1)$. Center: checks if input is a palindrome, with APC of $O(n)$. Right: finds the GCD of two inputs, with APC of $O(1.19^n)$

Table 1: APC and KLEE data on C files showing lines of code (LoC), cyclomatic (Cyclo) and NPATH complexity, asymptotic path complexity (APC), APC time, number of edges and nodes in the CFG ($|E|, |N|$), best fit curve for KLEE's path growth with respect to search depth, KLEE time, and indication of when APC matches the asymptotic complexity class of KLEE's fitted path growth function (constant, same polynomial, or exponential growth) (✓) or is a complexity class upper bound (U.B).

Function Under Test	LoC	Cyclo	NPATH	APC	APC Time (s)	|E|	|N|	Klee Best Fit	Klee Time (s)	Match?
Parity	7	2	2	$O(1)$	0.012	4	4	2	0.173	✓
Sign	9	3	3	$O(1)$	0.17	7	6	3	0.161	✓
Greatest of 3	9	5	7	$O(1)$	0.022	11	8	5	0.313	✓
Lexicographic Array Compare	11	4	4	$O(n)$	0.033	13	11	n	1.833	✓
Prime	8	3	3	$O(n)$	0.091	9	8	n	432.047	✓
Check Array Sorted	9	3	3	$O(n)$	0.092	9	8	n	4.705	✓
Check Arrays Equal	9	3	3	$O(n)$	0.091	9	8	n	2.243	✓
Find in Array	9	3	3	$O(n)$	0.090	9	8	n	2.269	✓
Check Heap Order	8	4	4	$O(n)$	0.313	11	9	n	2.98	✓
Check Sorted or Reverse	18	6	18	$O(n^2)$	0.809	19	15	$0.33n^2$	196.375	✓
Three Loops w/ variable bounds	14	4	8	$O(n^3)$	1.464	15	13	$0.17n^3$	301.162	✓
Three Loops with variable break	23	7	27	$O(n^3)$	1.952	24	19	$0.07n^3$	301.117	✓
Array Max	8	3	3	$O(1.17^n)$	0.752	8	7	1.41^n	301.112	✓
Euclid GCD	11	3	3	$O(1.19^n)$	0.304	8	7	1.41^n	307.592	✓
Binary Search	16	5	5	$O(1.22^n)$	0.832	16	13	1.27^n	119.797	✓
Bubble Sort	13	4	4	$O(1.27^n)$	0.884	13	11	1.55^n	301.166	✓
Selection Sort	13	4	4	$O(1.27^n)$	0.347	13	11	1.63^n	301.211	✓
Edit Distance	25	8	9	$O(1.29^n)$	2.797	29	23	1.42^n	17.423	✓
Insertion Sort	14	4	5	$O(1.35^n)$	1.153	12	10	1.58^n	301.218	✓
Quick Sort	34	6	13	$O(1.36^n)$	1.514	18	14	1.17^n	539.516	✓
Merge Sort	29	11	197	$O(1.42^n)$	13.000	41	32	1.78^n	47.263	✓
Heap Sort	62	20	4971	$O(1.41^n)$	47.318	72	54	1.72^n	301.512	✓
Palindrome	11	2	2	$O(n)$	0.108	4	4	11	2.474	U.B.
Variance	11	3	4	$O(n^2)$	0.607	10	9	n	4.585	U.B.
Position, Velocity, Acceleration	15	4	8	$O(n^3)$	1.697	15	13	n	97.929	U.B.
Newton's Method	20	4	5	$O(1.12^n)$	0.504	13	11	n	2.116	U.B.
Fibonacci	16	3	3	$O(1.15^n)$	0.390	9	8	n	1.99	U.B.
Sieve of Eratosthenes	10	5	8	$O(1.22^n)$	1.333	18	15	n	18.168	U.B.
Longest Common Inc. Subsequence	18	10	66	$O(1.36^n)$	7.233	35	27	$2n$	4.241	U.B.

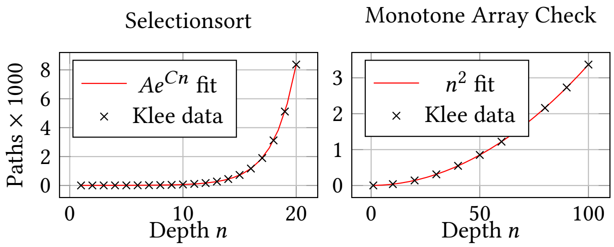


Figure 2: Paths explored by KLEE by execution depth.

Experimental Benchmark. We computed APC with METRINOME and KLEE path statistics for 29 C functions (Table 1). When computing path complexity for a function, METRINOME is agnostic to the complexities of any external calls. For each function under test we ran KLEE and collected the number of paths explored for increasing exploration depth bound. We ran polynomial and exponential regressions to generate the the best fitting curve of path count as a function of KLEE exploration depth and compared it to the APC results. The benchmark source code and KLEE drivers synthesized by METRINOME are available in our repo and Docker image (see repo README).

Experimental Results. We answered **RQ1** and **RQ2** in the affirmative. Overall, our results show that APC is an effective and fast predictor of path explosion by KLEE. APC always gave a complexity class upper bound for KLEE best fit, in the sense that constant < quadratic < cubic < . . . < exponential complexity of any base. APC had the same complexity class (up to differences in base for exponential classes) as the KLEE best fit expression in 22 cases. APC had a higher complexity class than that of the KLEE best fit expression in 7 cases. We found no examples of APC having a smaller dominant asymptotic class term than that of the KLEE best fit expression. The slight difference in exponential bases is explained by the fact that APC considers path lengths as the number of edges in the reduced control flow path, whereas KLEE considers path lengths as the number of branches. APC was significantly faster to compute than KLEE in 28 out of 29 cases, the exception being Longest Common Increasing Subsequence. The average runtime of APC was 49 times faster than that of KLEE. Overall, APC can be used to quickly predict the degree of path explosion when running KLEE.

7 RELATED WORK

Earlier work proposed APC and showed that it is a more refined complexity metric than cyclomatic and NPATH [1]. That work demonstrated that path complexity is scalable, analyzing the path complexities of the entire Java SDK and Apache Commons libraries, approx. 177,000 methods total, for an average rate of 14 methods per second. Future work was to demonstrate that APC can be used to predict the difficulty of path exploration during symbolic execution. We follow up on that line of work, and presented the METRINOME tool, which contains significant improvements. Trautsch *et al.* included our earlier replication package for computing APC for Java in their study of reproducibility of 34 software analysis tools [14]. They lament the excessive difficulty of running cutting-edge research-based software analysis tools due to the wide variation in system dependencies and configurations. Indeed, PAC relied on outdated versions of Java and MATHEMATICA (which requires a paid-license). We alleviate these issues in METRINOME by performing all symbolic algebra using sympy and providing a Docker image on Dockerhub. Fazli *et al.* propose a method for generating prime paths of a control flow graph [4] (paths that do not pass through a vertex more than once), which is closely related to NPATH complexity, and so in that context, NPATH is the correct metric to predict difficulty of prime path generation. We feel that APC is the correct analogous metric for symbolic execution path and test generation. In concurrent programming, metrics exist for measuring the difficulty of achieving *interleaving* coverage [5, 6, 10], analyzing process interleaving graphs rather than CFGs.

8 CONCLUSION

METRINOME enables computing complexity metrics for C, notably *asymptotic path complexity*. It provides a framework that can easily be extended to new languages, and incorporates a REPL environment to calculate the complexity metrics. The REPL also has features for running KLEE, a popular symbolic execution tool, and generating KLEE compatible files. Using the tool, we compared the number of paths generated by KLEE to APC. Our APC metric quickly and soundly predicts the growth rate of KLEE paths generated as a function of symbolic exploration depth.

REFERENCES

[1] Lucas Bang, Abdulbaki Aydin, and Tevfik Bultan. [n.d.]. Automatically computing path complexity of programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, 2015*. ACM.

[2] N. Biggs, N.L. Biggs, Cambridge University Press, and B. Norman. 1993. *Algebraic Graph Theory*. Cambridge University Press.

[3] 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 (OSDI'08)*. USENIX Association, USA, 209–224.

[4] E. Fazli and M. Afsharchi. 2019. A Time and Space-Efficient Compositional Method for Prime and Test Paths Generation. *IEEE Access* 7 (2019).

[5] JunXia Guo, Zheng Li, CunFeng Shi, and Ruilian Zhao. 2020. Thread Scheduling Sequence Generation Based on All Synchronization Pair Coverage Criteria. *International Journal of Software Engineering and Knowledge Engineering* 30 (01 2020), 97–118. https://doi.org/10.1142/S0218194020500059

[6] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. 2012. Testing Concurrent Programs to Achieve High Synchronization Coverage. In *Proc. of the 2012 International Symp. on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA.

[7] Dennis G. Kafura and Geereddy R. Reddy. 1987. The Use of Software Complexity Metrics in Software Maintenance. *IEEE Transactions on Software Engineering* SE-13 (1987), 335–343.

[8] T. M. Khoshgoftaar and J. C. Munson. 2006. Predicting Software Development Errors Using Software Complexity Metrics. *IEEE J.Sel. A. Commun.* 8, 2 (Sept. 2006), 253–261. https://doi.org/10.1109/49.46879

[9] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[10] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. 2007. A Study of Interleaving Coverage Criteria. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA.

[11] T. J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320.

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

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

[14] Fabian Trautsch, Steffen Herbold, Philip Makedonski, and Jens Grabowski. 2018. Addressing Problems with Replicability and Validity of Repository Mining Studies through a Smart Data Platform. *Empirical Softw. Engg.* 23, 2 (April 2018).