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,ssdissem,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:

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 exploration 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 repo1 as well as in a ready-to-run image on Dockerhub2.

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

1https://github.com/hmc-alpaqa/metrinome
2https://hub.docker.com/orgs/harveymudd/metrinome
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 $pc(n)$:

\[ g(z) = \frac{\det(I - zT : \lfloor N \rfloor, 1)}{(-1)^{|N|+1} \det(I - zT)}. \]

where $T$ is the augmented transfer-matrix (an adjacency matrix with $T_{N|N}$). $(M : i, j)$ 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 $I(n)$ as a sum of products of simple polynomial and exponential terms such that $pc(n) = \Theta(f(n))$. The form of $f(n)$ is determined by

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

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 $c_{i,j}$ are coefficients determined by $|N|$ terms of the Taylor expansion of $g(z)$. Since $\text{path}(n) = [z^n]g(z)$, 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.
```c
int parity(int num) {
    if (num % 2 == 0)
        return 0;
    else
        return 1;
}
```

```c
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;
}
```

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

**Table 1:** APC 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.).
work demonstrated that path complexity is scalable, analyzing the
Trautsch
The slight difference in exponential bases is explained by the fact
Metrinome
ting cutting-edge research-based software analysis tools due to the
bolic execution. We follow up on that line of work, and presented
libraries, approx. 177,000 methods total, for an average rate of 14
path complexities of the entire Java SDK and Apache Commons
fined complexity metric than cyclomatic and NPATH [1]. That
7 RELATED WORK
Klee
time faster than that of
Klee
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 comput-
ing 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 increas-
ing exploration depth bound. We ran polynomial and exponential
regressions to generate 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 affir-
matlive. 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 ex-
ponential 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 domi-
nant 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 Com-
mon 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 re-
fined 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 comput-
ating APC for Java in their study of reproducibility of 34 software
analysis tools [14]. They lament the excessive difficulty of run-
ning 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 gen-
erating 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, not-
tably 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

path complexity of programs. In Proceedings of the 2015 10th Joint Meeting on
Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy. 2015. ACM.
Graph Theory. Cambridge University Press.
Automatic Generation of High-Coverage Tests for Complex Systems Programs.
In Proceedings of the 8th USENIX Conference on Operating Systems Design and
Sequence Generation Based on All Synchronization Pair Coverage Criteria.
International Journal of Software Engineering and Knowledge Engineering 30 (1)
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.
Metrics in Software Maintenance. IEEE Transactions on Software Engineering
2006), 253–261. https://doi.org/10.1109/49.648679
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.
bridge University Press, USA.
Addressing Problems with Replicability and Validity of Repository Mining Studies