Using Dynamically Inferred Invariants to Analyze Program Runtime Complexity

ThanhVu Nguyen
The University of Nebraska-Lincoln

Didier Ishimwe
The University of Nebraska-Lincoln

Alexey Malyshev
The University of Nebraska-Lincoln

Timos Antonopoulos
Yale University

Quoc-Sang Phan
Synopsys

ABSTRACT

Being able to detect program runtime complexity can help identify security vulnerabilities such as DoS attacks and side-channel information leakage. In prior work, we use dynamic invariant generation to infer nonlinear numerical relations to represent runtime complexity of imperative programs. In this work, we propose a new dynamic analysis approach for learning *recurrence relations* to capture complexity bounds for recursive programs. This approach allows us to efficiently infer simple linear recurrence relations that represent nontrivial, potentially nonlinear, complexity bounds. Preliminary results on several popular recursive programs show that we can learn precise recurrence relations capturing worst-case complexity bounds such as $O(n \log n)$ and $O(c^n)$.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → **Software security engineering**.

KEYWORDS

dynamic invariant generation, complexity analysis, recurrence relations, numerical relations

ACM Reference Format:
ThanhVu Nguyen, Didier Ishimwe, Alexey Malyshev, Timos Antonopoulos, and Quoc-Sang Phan. 2020. Using Dynamically Inferred Invariants to Analyze Program Runtime Complexity. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Security from Design to Deployment (SEAD '20), November 9, 2020, Virtual, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3416507.3423189

1 INTRODUCTION

The automated discovery of *program invariants*—relations among variables that are guaranteed to hold at certain locations of a program—is an important research area in program analysis, verification, and synthesis. Generated invariants can be used to understand undocumented programs, prove correctness assertions, establish security properties, provide formal documentation, and more [5, 13–15, 25, 29].

In [32, 34], we developed DIG, a dynamic invariant generation tool that learns numerical invariants involving relations among numerical program variables. In particular, DIG supports *nonlinear polynomial relations*, e.g., $x \leq y^2$, $x = qy + r$. These relations arise in many scientific, engineering, and safety- and security-critical software, e.g., to verify the absence of errors in Airbus avionic systems [11]. A rather surprising use of DIG's nonlinear invariants is that they can help characterize program runtime complexity, by instrumenting a counter for the number of blocks executed and inferring a relationship involving that counter and the program's input variables, at the end of the program's execution [32]. For example, it can be shown this way that a program runs in $O(n^2 + 2m)$ for certain inputs and $O(m)$ for other inputs.

In this paper, we propose a new dynamic analysis for learning *recurrence relations* to capture complexity bounds for recursive programs. At high level, a recurrence relation defines the complexity to solve a problem in terms of the complexities to solve its subproblems. The dynamic technique allows us to efficiently infer simple linear recurrence relations that represent nontrivial, potentially nonlinear, complexity bounds. When applied to several classical divide-and-conquer algorithms, we were able to learn precise recurrence relations capturing worst-case complexity bounds such as $O(n \log n)$ or $O(c^n)$ from execution traces obtained by running the programs using few randomly generated inputs.

What distinguishes our work from other complexity analyses (e.g., [21, 22, 27, 35] and those reviewed in Section 4) is the use of dynamic, instead of static, analysis to learn program complexity bounds. In general, a static analysis can reason about all program paths soundly, but doing so is often expensive and is only possible for relatively simple forms of invariant relations or restricted classes of programs. Dynamic analyses limit their attention to only some of a program's paths, and thus provide no guarantee that those invariants are correct, but can often be more efficient and produce more expressive results [16, 34]. We can also improve correctness by using symbolic execution techniques to check for spurious results [33] and generate worst-case complexity inputs leading to high-complexity program paths [6, 30, 35].

As shown in many works [21, 22, 27, 35], complexity analysis, in particular through worst-case execution time (WCET) analysis [41] and high-security input dependent resource analysis, can help detect several important security vulnerabilities [9, 12, 19] (e.g., by allowing an attacker to exhaust the system's resources (time or memory) and perform Denial-of-Service attacks on servers or by

SEAD '20, November 9, 2020, Virtual, USA

ACM ISBN 978-1-4503-8126-0/20/11…$15.00
https://doi.org/10.1145/3416507.3423189

```
def tripple(M, N, P):
  t = 0  #ctr variable
  assert (0 <= M and 0 <= N and 0 <= P);
  i = 0; j = 0; k = 0
  while i < N:
    j = 0; t++
    while j < M:
      j++; k = i; t++
      while k < P:
        k++; t++
      i = k
    i++
  [L]
```

Figure 1: A program with several complexity bounds.

exhibiting side-channel information leakage). By knowing the execution times of different high-security dependent branches (e.g., one branch takes linear time while the other takes quadratic time), the developer can mitigate an attack by "padding" the computation so that all executions take the same time (e.g., instrumenting the program to add dummy loops, instructions, or delays).

2 LEARNING POLYNOMIAL RELATIONS

In [32, 34], we developed DIG, a dynamic analysis tool that learns (potentially nonlinear) numerical invariants, which describe relations over numerical variables at arbitrary program locations. DIG's invariants can help understand programs and characterize their runtime complexities, which is useful for identifying possible security problems [4, 31].

Example. Figure 1 shows the program `tripple`, adapted from Figure 2 of [22], with nontrivial runtime complexity. At first, `tripple` appears to take $O(NMP)$ due to the three nested loops. A closer analysis [22] shows a more precise bound $O(N + NM + P)$ because the inner most loop, which is updated each time the middle loop executes, changes the behavior of the outer most loop.

When given this program, DIG discovers an interesting and complex postcondition at location L about the variable `t`, which is a ghost variable introduced to count loop iterations:

$$P^2Mt + PM^2t - PMNt - M^2Nt - PMt^2 + MNt^2 + PMt-$$
$$PNt - 2MNt + Pt^2 + Mt^2 + Nt^2 - t^3 - Nt + t^2 = 0.$$

This nonlinear equality is valid, but incomprehensible and quite different than the expected bound $N + NM + P$ or even NMP. However, when solving this equation (finding the roots of `t`), we obtain three solutions that describe the exact bounds of this program:

$$t = \begin{cases} 0 & \text{when} \quad N = 0 \\ P + M + 1 & \text{when} \quad N \leq P \\ N - M(P - N) & \text{when} \quad N > P \end{cases}$$

These results are more precise than the bound $N + MN + P$ given in [22] and can help developers analyze inputs causing the program to run in different time complexities. This example also shows complexity analysis can help detect side-channel information leakage, e.g., if some of the inputs from M, N and P are high-security and some are public, an attacker can infer valuable information about the high-security inputs by observing the running time of the program.

3 LEARNING RECURRENCE RELATIONS

Polynomial relations can help capture general program complexity. However, for recursive, e.g., divide-and-conquer, programs we can generate *recurrence relations* [10] to compute complexity more precisely. Using dynamic analysis to infer recurrent relations is relatively straightforward and a linear recurrence relation can capture complex program bounds such as those involving log or nonlinear degrees.

A recurrence relation (or simply recurrence) defines the complexity to solve a problem in terms of the complexities to solve its subproblems. For example, we can compute the recurrence for the standard `mergesort` algorithm as $T(n) = 2T(\frac{n}{2}) + O(n)$, i.e., the algorithm splits the problem into two subproblems of half the sizes of the original problem and merges the results of the subproblems in linear time). Next, solving this recurrence, e.g., using the well-known Master Theorem [10], gives the asymptotic complexity $O(n \log n)$. Thus, we can obtain difficult program complexity bounds by inferring and solving relatively simple recurrence relations, e.g., we obtain `mergesort`'s complexity involving log from a recurrence that does not directly involve log.

Example. For the `mergesort` program in Figure 2, we instrument the program with the new variables `id` and `t` to keep track of recursive calls. We also record execution traces at the program entrance to capture the length of the input and the unique id of each recursive call.

The tree in Figure 2 shows the program execution traces when applying `mergesort` to a list of 7 elements. The root node $(7, [1])$ is the first `mergesort` call with id [1] on the list of 7 elements. The children nodes $(3, [1, 1])$ and $(4, [1, 2])$ respectively represent the first and second recursive calls on the first 3 and the remaining 4 elements of the original list.

We now analyze the recursive parts of the program. From the execution traces, we form tuples of the form (t_0, t_1), where t_0 represents the input length of the original call and t_1 the input length of the first recursive call. Then we use a learning technique such as linear regression to find a relation of the form $t_1 \approx ct_0$, which represents the relation between the sizes of the original problem and the subproblems. From the data $(7, 3), (3, 1), (2, 1), (4, 2)$ in the execution tree in Figure 2, we obtain the relation $t_1 \approx \frac{1}{2}t_0$. Similarly, we obtain $t_2 \approx \frac{1}{2}t_0$ as the relation between `mergesort` and its second recursive call. The combination of these two, $T_0 = \frac{T_0}{2} + \frac{T_0}{2}$, gives the recurrence $T(n) = T(\frac{n}{2}) + T(\frac{n}{2})$, indicating that `mergesort` makes two recursive calls over inputs that are approximately half of the original input.

For the non-recursive `merge` function, we can find a general polynomial relation to capture its complexity. First, we instrument the program using the counter variable t and increment it in each loop to count the number of executed blocks. Next, from traces recorded at the progarm exit (using the `trace` function), we can compute the relation $t \approx len(A) + len(B)$, indicating `merge` runs in linear time.

The combination of the recursive and non-recursive results gives the recurrence $T(n) = 2T(\frac{n}{2}) + O(n)$ (`merge` takes linear time). We

```
def mergesort(L, id):
  #id is a list, e.g., [1]
  trace(len(L), id)

  t = 0 #ctr variable
  n = len(L)
  if n == 0 or n == 1:
    return copy(L)

  mid = n // 2

  A = mergesort(L[0:mid],
      id+[++t]) #id = [1,1]

  B = mergesort(L[mid:n],
      id+[++t]) #id = [1,2]
  C = merge(A, B)
  return C
```

```
def merge(A, B):
  t = 0 #ctr variable
  a = 0; b = 0; C = []
  while (a < len(A)
       and b < len(B)):
    t++
    if A[a] <= B[b]:
      C += [A[a]]; a++
    else:
      C += [B[b]]; b++

  while a < len(A):
    t++; C += [A[b]]
  while b < len(B):
    t++; C += [B[b]];

  trace(len(A), len(B), t)
  return C
```

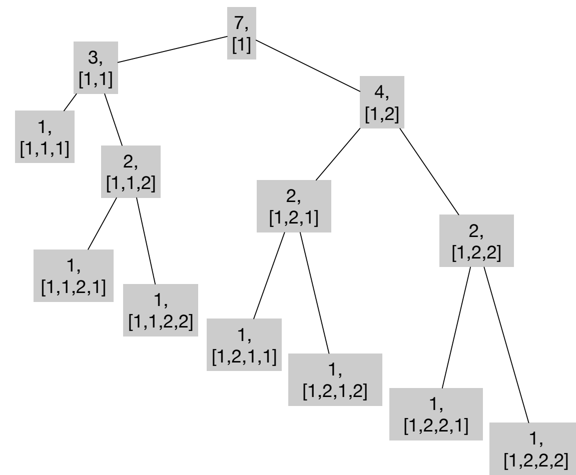

Figure 2: The Mergesort Algorithm.

can now apply the Master Theorem [10] to solve this recurrence to obtain the complexity $O(n \log n)$.

Evaluation. We applied our approach to compute recurrence relations for several recursive programs in OCaml. We manually instrumented each program, ran it on randomly generated inputs to obtain execution traces (e.g., for `mergesort` we randomly generated input lists of various sizes), and finally computed the recurrences.

Table 1 shows the results. We were able to obtain the correct recurrences for all considered programs. We used the Master Theorem, which supports recurrences of the form $T(n) = aT(\frac{n}{b}) + f(n)$, to obtain the complexities of `binary search` and `merge sort`. For the other programs, we manually convert their recurrences to program complexities.

Table 1: Results

Program	Recurrence	Complexity
Binary Search	$T(n) = T(\frac{n}{2}) + 1$	$O(\log n)$
Merge Sort	$T(n) = 2T(\frac{n}{2}) + n$	$O(n \log n)$
Insertion Sort	$T(n) = T(n-1) + n$	$O(n^2)$
Selection Sort	$T(n) = T(n-1) + n$	$O(n^2)$
List Rotation	$T(n) = T(n-1) + 1$	$O(n)$
Depth First Search	$T(n) = T(n-1) + 1$	$O(n)$
Fibonacci	$T(n) = T(n-1) + T(n-2) + 1$	$O(2^n)$
Tower of Hanoi	$T(n) = 2T(n-1) + 1$	$O(2^n)$

4 RELATED WORKS

There are many static analyses for program complexity, e.g., the SPEED project [21–23] and others [26–28]. Chatterjee et al. [7, 8] use ranking functions and linear programming to compute termination property and non-polynomial worst-case upper bounds. Hansel et al. [24] use symbolic execution to obtain an integer transition system to derive upper runtime bounds. Several techniques focus on recurrence relations for worst-case complexity analysis [1–3, 17, 20]. For example, the work in [2] solves recurrence relations using evaluation trees and can derive the complexity bound for `mergesort`. These works use static or symbolic analyses while we dynamically learn complexity invariants.

There are also works on verifying given complexity bounds [37]. In particular, the TiML functional language [39] allows a user to specify time complexity as types and then uses type checking to verify the specified complexity. We can use these works to check our candidate invariants.

Several worst-case execution time (WCET) analyses use symbolic execution or fuzzing to find inputs or program paths leading to worst-case program behaviors [6, 30, 35, 36, 40]. The recent work in [38] uses automatic amortized resource analysis, a type-based technique to compute symbolic bounds and generate worst-case input for OCaml functions. We can leverage these inputs to obtain useful execution traces for dynamic analysis.

5 CONCLUSION AND FUTURE WORK

We propose new dynamic analysis techniques to learn numerical and recurrence relations to capture program complexity bounds. In addition to developing tools implementing these ideas, we are extending DIG with other learning techniques such as linear regression [18] and neural networks [42] to compute more general relations to represent inexact, e.g., lower and upper, complexity bounds. We are also exploring existing WCET techniques to generate worst-case complexity inputs and check candidate invariants (e.g., using the guess-and-check approach [33] to remove spurious results and generate counterexamples to help dynamic inference).

ACKNOWLEDGMENTS

We thank the anonymous reviewers for helpful comments and NSF CCF-1948536 and ARO W911NF-19-1-0054 for funding support.

REFERENCES

[1] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, German Puebla, Diana Ramírez, G Román, and Damiano Zanardini. 2009. Termination and cost analysis with COSTA and its user interfaces. *Electronic Notes in Theoretical Computer Science* 258, 1 (2009), 109–121.

[2] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2008. Automatic inference of upper bounds for recurrence relations in cost analysis. In *International Static Analysis Symposium*. Springer, 221–237.

[3] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. 2007. Cost Analysis of Java Bytecode. In *European Symposium on Programming*. Springer, 157–172.

[4] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition instead of self-composition for proving the absence of timing channels. In *Programming Language Design and Implementation*. 362–375.

[5] Thomas Ball and Sriram K. Rajamani. 2001. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN Symposium on Model Checking of Software*. Springer, 103–122.

[6] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. 2009. WISE: Automated test generation for worst-case complexity. In *International Conference on Software Engineering*. IEEE Computer Society, 463–473.

[7] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016. Termination analysis of probabilistic programs through Positivstellensatz's. In *Computer Aided Verification*. Springer, 3–22.

[8] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2019. Non-polynomial worst-case analysis of recursive programs. *ACM Transactions on Programming Languages and Systems* 41, 4 (2019), 1–52.

[9] Php Classes. [n.d.]. https://www.phpclasses.org/blog/post/171-PHP-Vulnerability-May-Halt-Millions-of-Servers.html.

[10] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[11] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The Astrée analyzer. In *European Symposium on Programming*. Springer, 21–30.

[12] Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security Symposium* (Washington, DC) *(SSYM'03)*. USENIX Association, USA, 3.

[13] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: path-sensitive program verification in polynomial time. *SIGPLAN Notices* 37, 5 (2002), 57–68.

[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[15] Michael D. Ernst. 2000. *Dynamically detecting likely program invariants*. Ph.D. Dissertation. University of Washington.

[16] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* (2007), 35–45.

[17] Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. 1991. Automatic average-case analysis of algorithms. *Theoretical Computer Science* 79, 1 (1991), 37–109.

[18] Aurélien Géron. 2017. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.".

[19] John Graham-Cumming. [n.d.]. Details of the Cloudflare outage on July 2, 2019. https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/.

[20] Bernd Grobauer. 2001. Cost recurrences for DML programs. *ACM SIGPLAN Notices* 36, 10 (2001), 253–264.

[21] Sumit Gulwani. 2009. SPEED: Symbolic Complexity Bound Analysis. In *Computer Aided Verification*. Springer-Verlag, 51–62.

[22] Sumit Gulwani, Sagar Jain, and Eric Koskinen. 2009. Control-flow Refinement and Progress Invariants for Bound Analysis. In *Programming Language Design and Implementation*. 375–385.

[23] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: precise and efficient static estimation of program computational complexity. In *Principles of Programming Languages*. ACM, 127–139.

[24] Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder. 2018. Termination and complexity analysis for programs with bitvector arithmetic by symbolic execution. *Journal of Logical and Algebraic Methods in Programming* 97 (2018), 105–130.

[25] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. 2002. Lazy Abstraction. In *Principles of Programming Languages*. ACM, 58–70.

[26] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate amortized resource analysis. In *Principles of Programming Languages*. 357–370.

[27] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *Computer Aided Verification*. 781–786.

[28] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static determination of quantitative resource usage for higher-order programs. In *Principles of Programming Languages*. 223–236.

[29] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Principles of Programming Languages*. ACM, 42–54.

[30] Kasper Luckow, Rody Kersten, and Corina Pasareanu. 2020. Complexity vulnerability analysis using symbolic execution. *Software Testing, Verification and Reliability* (2020), e1716.

[31] V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *Symposium on Security and Privacy (SP)*. IEEE, 710–728.

[32] ThanhVu Nguyen, Timos Antopoulos, Andrew Ruef, and Michael Hicks. 2017. A Counterexample-guided Approach to Finding Numerical Invariants. In *Foundations of Software Engineering*. ACM, 605–615.

[33] ThanhVu Nguyen, Matthew Dwyer, and William Visser. 2017. SymInfer: Inferring Program Invariants using Symbolic States. In *Automated Software Engineering*. IEEE, 804–814.

[34] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using Dynamic Analysis to Discover Polynomial and Array Invariants. In *International Conference on Software Engineering*. IEEE, 683–693.

[35] Yannic Noller, Rody Kersten, and Corina S Păsăreanu. 2018. Badger: complexity analysis with fuzzing and symbolic execution. In *International Symposium on Software Testing and Analysis*. 322–332.

[36] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Conference on Computer and Communications Security*. ACM, 2155–2168.

[37] Akhilesh Srikanth, Burak Sahin, and William R Harris. 2017. Complexity verification using guided theorem enumeration. *ACM SIGPLAN Notices* 52, 1 (2017), 639–652.

[38] Di Wang and Jan Hoffmann. 2019. Type-guided worst-case input generation. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

[39] Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: a functional language for practical complexity analysis with invariants. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 79.

[40] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. 2018. Singularity: Pattern Fuzzing for Worst Case Complexity. In *Foundations of Software Engineering*. ACM, to appear.

[41] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. 2008. The worst-case execution-time problemâĂŤoverview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 1–53.

[42] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning nonlinear loop invariants with gated continuous logic networks. In *Programming Language Design and Implementation*. ACM, 106–120.