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:

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...
def triple(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++

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).

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 $T(t) = A t + B$, where $A$ 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 $t_2 \approx \frac{1}{4} t_0$ as the relation between mergesort and its second recursive call. The combination of these two, $T_0 = \frac{1}{2} + \frac{1}{4} t$, gives the recurrence $T(n) = T(\frac{n}{2}) + T(\frac{n}{4})$, 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 program exit (using the trace function), we can compute the relation $t \approx \text{len}(A) + \text{len}(B)$, 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
    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++
    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)
    trace(len(A), len(B), t)
    return C

def merge(A, B):
    t = 0  # ctr variable
    n = len(L)
    if n == 0 or n == 1:
        return copy(L)
    else:
        if n == 0:
            return copy(L)
        if n == 1:
            if A[0] <= B[0]:
                C += [A[0]]; a++
            else:
                C += [B[0]]; b++
    t = 0  # ctr variable
    A = mergesort(L[0:mid], id+[++t]) # id = [1,1]
    B = mergesort(L[mid:n], id+[++t]) # id = [1,2]
    C = merge(A, 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

<table>
<thead>
<tr>
<th>Program</th>
<th>Recurrence</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary Search</td>
<td>( T(n) = T(\frac{n}{2}) + 1 )</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>( T(n) = 2T(\frac{n}{2}) + n )</td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>( T(n) = (n - 1) + n )</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>( T(n) = (n - 1) + n )</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td>List Rotation</td>
<td>( T(n) = (n - 1) + 1 )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Depth First Search</td>
<td>( T(n) = T(n - 1) + T(n - 2) + 1 )</td>
<td>( O(2^n) )</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>( T(n) = T(n - 1) + T(n - 2) + 1 )</td>
<td>( O(2^n) )</td>
</tr>
<tr>
<td>Tower of Hanoi</td>
<td>( T(n) = T(n - 1) + 1 )</td>
<td>( O(2^n) )</td>
</tr>
</tbody>
</table>

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, 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 representations 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


