Counterexample-Guided Approach to Finding Numerical Invariants

ThanhVu Nguyen
University of Nebraska-Lincoln, USA

Timos Antonopoulos
Yale University, USA

Andrew Ruef
University of Maryland, USA

Michael Hicks
University of Maryland, USA

ABSTRACT

Numerical invariants, e.g., relationships among numerical variables in a program, represent a useful class of properties to analyze programs. General polynomial invariants represent more complex numerical relations, but they are often required in many scientific and engineering applications. We present NumInv, a tool that implements a counterexample-guided invariant generation (CEGIR) technique to automatically discover numerical invariants, which are polynomial equality and inequality relations among numerical variables. This CEGIR technique infers candidate invariants from program traces and then checks them against the program source code using the KLEE test-input generation tool. If the invariants are incorrect KLEE returns counterexample traces, which help the dynamic inference obtain better results. Existing CEGIR approaches often require sound invariants, however NumInv sacrifices soundness and produces results that KLEE cannot refute within certain time bounds. This design and the use of KLEE as a verifier allow NumInv to discover useful and important numerical invariants for many challenging programs.

Preliminary results show that NumInv generates required invariants for understanding and verifying correctness of programs involving complex arithmetic. We also show that NumInv discovers polynomial invariants that capture precise complexity bounds of programs used to benchmark existing static complexity analysis techniques. Finally, we show that NumInv performs competitively comparing to state of the art numerical invariant analysis tools.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; **Software verification**; **Automated static analysis**; **Dynamic analysis**;

KEYWORDS

Dynamic and Static Invariant Analyses, Counterexample-guided Algorithms, Numerical Domains, Program and Correctness Analyses, Test-input Generation

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106281

ACM Reference Format:
ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017. Counterexample-Guided Approach to Finding Numerical Invariants . In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 11 pages.
https://doi.org/10.1145/3106237.3106281

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 and verification. Generated invariants can be used to prove correctness assertions, reason about resource usage, establish security properties, provide formal documentation, and more [2, 15, 16, 21, 29, 31].

A particularly useful class of invariants are *numerical invariants*, which involve relations among numerical program variables. Within this class of invariants, *nonlinear polynomial* relations, e.g., $x \leq y^2, x = qy + r$, arise in many scientific, engineering, and safety- and security-critical applications.[1] For example, the commercial analyzer Astrée, which has been applied to verify the absence of errors in the Airbus A340/A380 avionic systems [5, 13], implements the ellipsoid abstract domain [22] to represent and analyze a class of quadratic inequality invariants. Complexity analysis, which aims to determine a program's performance characteristics [25, 26, 30], perhaps to identify possible security problems [1, 33], also makes use of polynomial invariants, e.g., $O(n^2 + 2m)$ where n, m are some program inputs. In addition, such polynomial invariants have been found useful in the analysis of hybrid systems [40, 41], and in fact are required for implementations of common mathematical functions such as mult, div, square, sqrt and mod.

Numerical invariants can be discovered via static and dynamic program analyses. 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 invariants [34]. Dynamic analyses limit their attention to only some of a program's paths, and as a result can often be more efficient and produce more expressive invariants, but provide no guarantee that those invariants are correct [21, 36]. Recently, several systems (such as PIE [37], ICE [23] and *Guess-and-Check* [42]) have been developed that take a hybrid approach: use a dynamic analysis to infer *candidate invariants* but then confirm these invariants are correct for all inputs using a *static verifier*. When invariants are incorrect the verifier returns

[1] We refer to nonlinear polynomial relations such as $x = qy + r$, $x \leq y^2$ simply as *polynomial relations*.

```
int cohendiv(int x, int y){
  assert(x>0 && y>0);
  int q=0; int r=x;
  while(r ≥ y){
    int a=1; int b=y;
    while[L1](r ≥ 2*b){
      a = 2*a; b = 2*b;
    }
    r=r-b; q=q+a;
  }
  [L2]
  return q;
}
```

Traces:

x	y	a	b	q	r
15	2	1	2	0	15
15	2	2	4	0	15
15	2	1	2	4	7
			⋮		
4	1	1	1	0	4
4	1	2	2	0	4
			⋮		

Figure 1: An integer division program and example trace values at location $L1$ on inputs $(x = 15, y = 2)$ and $(x = 4, y = 1)$. Among other invariants, two key loop invariants discovered at L1 are $b = ya$ and $x = qy + r$, with the latter also found as the postcondition at L2.

counterexample traces which the dynamic inference engine can use to infer more accurate invariants. This iterative process is called *CounterExample Guided Invariant geneRation* (CEGIR).

While the CEGIR approach is promising, existing tools have some practical limitations. One limitation is that they find invariants strong enough to prove a particular (programmer-provided) postcondition where the quality of the generated invariants depends on the strength of the postcondition. As such, they are not well suited for automated analyses on code that lacks such formal specifications. Another limitation is that these tools employ a *sound* static verifier, which aims to definitively prove that an invariant holds. While this is a good goal, it turns out to be a significant restriction on the quality of the invariants that can ultimately be inferred—it can be quite challenging to do when invariants are nonlinear polynomials and involve many program variables. For example, consider the program in Figure 1, which implements Cohen's algorithm for integer division [10]. Two important loop invariants (at L1) are $b = ya$ and $x = qy + r$, as they both point directly to the correctness of the algorithm.[2] Neither PIE nor ICE can infer these invariants (both tools time out).

In this paper we present a new CEGIR algorithm called NumInv that overcomes these limitations. It has two main components. First, it uses algorithms from DIG [35, 36] to dynamically infer expressive polynomial equality invariants and linear inequality relations from traces at specified program locations. Second, it uses KLEE [6], a symbolic executor, to check candidate invariants and produce counterexamples when they fail to hold. To check that an invariant p holds at location L, NumInv transforms the input program so that L is guarded by the conditional $\neg p$. If KLEE is able to reach L then p must not be an invariant, and so it outputs a counterexample consisting of the relevant input values at that location. On the other hand, if KLEE never reaches that location prior to timing out, then NumInv accepts the invariant as correct. Although this technique is unsound, KLEE, by its nature as a symbolic executor, turns out to be very effective in discovering counterexamples to refute invalid candidates.

For the example in Figure 1, NumInv is able to find the critical equalities mentioned above, along with many other useful inequalities. These invariants help understand the precise semantics of the program and verify its correctness properties. Moreover, by instrumenting the program with a counter variable, NumInv can even infer program running times as a function of the inputs. For example, NumInv is able to infer the precise running time of the program in Figure 5 (page 7) which has a tricky, triple-nested loop.

We evaluated NumInv by using it to infer invariants on more than 90 benchmark programs taken from the NLA [35] and HOLA [17] suites for program verification and from examples in the literature on complexity bound analysis [25–27]. Our results show that NumInv generates sufficiently strong invariants to verify correctness and to understand the semantics of 23/27 NLA programs containing nontrivial arithmetic and polynomial relations. We also find that NumInv discovers highly precise invariants describing nontrivial complexity bounds for 18/19 programs used to benchmark static complexity analysis techniques (in fact, for 4 programs, NumInv obtains more informative bounds than what were given in the literature). We note that both ICE and PIE cannot find any of these invariants produced by NumInv, even when we explicitly tell these tools that they should attempt to verify these invariants. Finally, on the 46 HOLA programs, we compare NumInv directly with PIE. We find it performs competitively: in 36/46 cases its inferred invariants match PIE's, are stronger, or are more descriptive.

Thus, although NumInv can potentially return unsound invariants, our experience shows that it is practical and effective in removing invalid candidates and in handling difficult programs with complex invariants. We believe that NumInv strikes a practical balance between correctness and expressive power, allowing it to discover complex, yet interesting and useful invariants out of the reach of the current state of the art.

2 OVERVIEW

NumInv generates invariants using the technique of *counterexample-guided invariant generation* (CEGIR). At a high level, CEGIR consists of two components: a *dynamic analysis* that infers candidate invariants from execution traces, and a *static verifier* to check candidates against the program code. If a candidate invariant is spurious, the verifier also provides counterexamples (*cexs*). Traces from these cexs are recycled to repeat the process, hopefully producing accurate results. These steps of inferring and checking repeat until no new cexs or (true) invariants are found. The CEGIR approach is basically exploiting the observation that inferring a sound solution directly is often harder than checking a (cheaply generated) candidate solution.

Other promising CEGIR algorithms, e.g., the ICE, PIE and *Guess-and-Check* tools, have been developed in recent years that take the same approach [23, 37, 42], though they refer to it differently. In particular they refer to CEGIR as a *data driven* or *black-box* approach, where the dynamic analysis is called the *student* or *learner*, and the static verifier is called the *teacher* or *oracle*. These approaches have been able to prove correctness of specifications by inferring

[2] $x = qy + r$ describes the intended behavior of integer division: the dividend x equals the divisor y times the quotient q plus the remainder r.

inductive loop invariants, or sufficient and necessary preconditions. Some of these works (ICE and PIE) are verification oriented, i.e. they infer invariants to specifically prove a given assertion. In this approach, the computation of these "helper" invariants strictly depends on the given assertions, e.g., if the intended assertion is True then the inferred invariant can be just True. We review these works in more detail in Section 6.

NumInv has different goals and takes a different approach. Our goals are both discovery and verification, and our approach is to find the strongest possible invariant at any arbitrarily given location. When given an undocumented program, NumInv can discover interesting properties and provide formal specifications. For example, NumInv can reveal a stronger postcondition than the user might think to write down, and the user doesn't have to write down any postconditions at all. Moreover, when given a specific assertion, the resulting invariant from NumInv can help prove it (e.g., if the invariant matches or is stronger than the assertion). Empirically, NumInv can frequently infer invariants that are at least as strong as the postcondition, and frequently, stronger.

2.1 NumInv

NumInv infers candidate invariants using the algorithms from DIG [35, 36], which produce equality and inequality relations from traces. To check invariants, NumInv invokes KLEE [6], a symbolic executor that is able to synthesize test cases for failing tests.

KLEE as a "verifier". NumInv generates candidate invariants at program location L of interest (e.g., at the start of loops or at the end of functions). To check whether a property p holds at a location L, NumInv asks KLEE to determine the reachability of the location L when guarded by $\neg p$. For example, to check whether the relation $x = qy + r$ is an invariant at some location L, NumInv modifies the program as follows

```
...
if (!(x==qy+r)){
  [L]
  save(x,y,q,r); //cex traces
  abort();
}
...
```

KLEE then runs this program, systematically exploring the space of possible inputs. If, during this process, location L is reached, then the relation does not hold, so a cex consisting of the values of the relevant input variables is saved for subsequent inference. On the other hand, KLEE may be able to explore all program paths and thus verify that indeed that invariant p holds. Or, if this is infeasible, NumInv terminates KLEE after some timeout.

The use of KLEE as the verifier is a key feature of NumInv. Because programs often contain a very large number of possible paths, KLEE rarely explores all of them. However, in our experience (Section 5), if it does not quickly find a counterexample for p then p very likely holds. This is true even when p is a nonlinear polynomial relation. As such, KLEE serves as a practical improvement over existing theorem provers and constraint solvers, for which reasoning over general polynomial arithmetic is a significant challenge.

Inferring polynomial equalities and linear inequalities. NumInv uses two CEGIR algorithms to find candidate numerical relations p at program locations of interest. The first algorithm finds *polynomial equalities*. To do this, for each program location L, NumInv produces a *template* equation $c_1t_1 + c_2t_2 \cdots c_nt_n = 0$. This equation contains n unknown coefficients c_i and n *terms* t_i, with one term for each possible combination of relevant program variables, up to some degree d. NumInv calls KLEE on the program to systematically obtain many possible valuations of relevant variables at L. Each distinct observed valuation, which we call a *trace*, is substituted into the template to form an instantiated equation. After obtaining at least n traces, NumInv solves the c_i using the resulting set of equations. Substituting the solutions back into the template, we can extract candidate invariants. At this point, NumInv enters a CEGIR loop that tests the candidate invariants by using KLEE as described above. Any spurious invariants are dropped, and the corresponding cex traces are used to infer new candidates, as described above, until no additional true invariants are found.

NumInv's second algorithm tries to infer *linear inequalities* in the form of *octagons*, which are inequalities over two variables, containing eight edges. It refines the bounds on the candidate invariants using a divide-and-conquer algorithm. Once again, NumInv estimates and obtains an initial set of traces. It enumerates all possible octagonal inequality forms involving one and two variables and uses KLEE to check inequalities under these forms are within certain ranges $[minV, maxV]$. It then narrows this range, iteratively seeking tighter lower and upper bounds.

Finally, from the obtained equality and inequality invariants, NumInv removes any invariants that are logical implications of other invariants. For instance, we suppress the invariant $x^2 = y^2$ if another invariant $x = y$ is also found because the latter implies the former. We check possible implications using an SMT solver (checking whether the negation of the implication is unsatisfiable).

2.2 Example

Recall the program cohendiv in Figure 1, which takes as input two integers x, y and returns the integer q as the quotient of x and y. Given this program and locations of interest L1 and L2, NumInv automatically discovers the following (loop) invariants at L1:

$$\begin{array}{lll} x = qy + r & b = ya & \\ y \leq b & b \leq r & \\ r \leq x & a \leq b & 2 \leq a + y \end{array}$$

and the following (postcondition) invariants at L2:

$$\begin{array}{lll} x = qy + r & 1 \leq q + r & \\ r \leq x & r \leq y - 1 & 0 \leq r \end{array}$$

These equality and inequality relations are sufficiently strong to understand the function's semantics and verify the correctness of cohendiv. More specifically, the nonlinear equation $x = qy + r$ describes the precise behavior of integer division: the dividend x equals the divisor y times the quotient q plus the remainder r. The other inequalities also provide useful information for debugging. For example, these invariants reveal several required properties of the remainder r such as r is non-negative ($r \geq 0$), is at most the dividend ($r \leq x$), but is strictly less than the divisor ($r \leq y - 1$). In addition, these invariants can help prove assertions if they exist in the program. For example, if we want to assert and prove the postcondition stating that the returned quotient is non-negative ($q \geq 0$), then we can easily do so because the discovered invariants at L2 imply $q \geq 0$.[3]

As mentioned above, ICE and PIE generate invariants to prove specific assertions. Thus, given a program with no specific assertion, they will not provide anything useful. Even when asked to verify a specific assertion, e.g., $x = qy + r$ or other simpler invariants above found by NumInv, these tools fail to prove them (PIE does not converge and ICE fails to generate invariants to prove the given assertions). We do not have the implementation of the *Guess-and-Check* algorithm in [42] to run on this example, however this work does not support inequalities and thus would not generate the inequality invariants shown.

3 INFERRING POLYNOMIAL EQUALITIES

We now discuss NumInv's CEGIR algorithm for generating polynomial equalities among program variables. This algorithm integrates the equation solving technique in DIG with KLEE to find invariants.

3.1 Terms, Templates, and Equation Solving

NumInv infers polynomial equalities by searching for solutions to instantiations of a *template* equation having the form $c_1t_1 + c_2t_2 \cdots + c_nt_n = 0$, where c_i are real-valued and t_i are *terms*. Terms consist of monomials over program variables. More specifically, given a set V of variables and a degree d, NumInv creates a set of n terms consisting of monomials up to degree d from V. For instance, the $n = 10$ terms $\{1, r, y, a, ry, ra, ya, r^2, y^2, a^2\}$ consist of all monomials up to degree 2 over the variables $\{r, y, a\}$.

NumInv seeks to solve the c_i in the template equation by *instantiating* the t_i with values observed from traces. For our example, instantiating the template with the trace $r = 3, y = 2, a = 6$ would yield the equation $c_1 + 3c_2 \cdots + 36c_n = 0$. If there are n terms, we need at least n distinct valuations of the variables in V. Given the (at least) n equations that result after instantiation, we solve for the c_i, substituting their solutions into the template to produce equations over the (combinations of) variables in V.

3.2 Algorithm

Figure 2 shows the CEGIR algorithm for finding polynomial equalities. Given a program P, location L, and a degree d, NumInv automatically computes all equalities with degree up to d over the numerical variables at L. (In Section 5, we discuss our use of a single parameter that automatically adjusts the degree d depending on the program).

The first steps are to identify the variables at the program location of interest, and then to construct the terms and template as described above. Then, in the first loop, we use KLEE to obtain traces to instantiate the template and thereby produce equations over the coefficients associated with the generated terms. To obtain traces, we simply ask KLEE to find cexs producing traces reaching L (more specifically, the location guarded by ¬False at L). To avoid getting old inputs, we explicitly ask KLEE to return only new inputs (by adding assertions that the input variables are not any of the observed ones). After producing enough equations, we solve them using an off-the-shelf linear equation solver and extract results representing candidate equality relations among terms.

```
input  : a program $P$, a location L, a degree $d$
output : polynomial equalities over the variables at L up to degree $d$
vars ← extractVars($P$, L)
terms ← createTerms(vars, $d$)
template ← createTemplate(terms)
inps, traces, eqts, invs ← $\emptyset$, $\emptyset$, $\emptyset$, $\emptyset$
while |eqts| < |terms| do
    cexInps ← verify ($P$, L, False, inps)
    if cexInps ≡ $\emptyset$ then
        if inps ≡ $\emptyset$ then return {False} //unreachable
        else return NotEnoughTraces
    inps ← inps ∪ cexInps
    traces ← exec($P$, L, cexInps)
    eqts ← eqts ∪ instantiate(template, traces)
sols ← solve(eqts)
candidates ← extractEqts(sols, terms)
while candidates ≠ $\emptyset$ do
    cexInps ← verify ($P$, L, candidates, inps)
    foreach candidate ∈ candidates do
        if candidate.stat ≠ False then invs.add(candidate)
    if cexInps ≡ $\emptyset$ then break
    inps ← inps ∪ cexInps
    traces ← exec ($P$, L, cexInps)
    eqts ← eqts ∪ instantiate(template, traces)
    sols ← solve(eqts)
    candidates ← extractEqts(sols, terms)
    candidates ← candidates − invs
return invs
```

Figure 2: CEGIR algorithm for finding equalities.

Next, the algorithm enters a second loop that iteratively verifies candidate invariants and obtains cex traces, allowing the inference algorithm to discard spurious results and generate new invariants. NumInv accepts a candidate invariant as long as KLEE cannot find a cex for it within the timeout period. We repeat the steps of verifying candidate invariants, obtaining cexs, and inferring new results until we can no longer find cexs or new results.

Note that unlike the popular CEGAR (counterexample-guided abstract refinement) technique in static analysis [9] that usually starts with a weak invariant and gradually strengthens it, NumInv's CEGIR algorithm starts with a strong invariant (i.e., False) and iteratively weakens it. This is because the algorithm dynamically infers invariants using observed traces. We start with few traces and thus likely generate too strong or spurious invariants. We then accumulate more traces to refute spurious results and create more general invariants that satisfy all obtained traces.

We also note that an interesting property of nonlinear polynomial equalities is that they can represent a form of *disjunctive* invariants. For example, $x^2 = 4$ indicates that $x = 2 \vee x = -2$. In Section 5.2 we exploit this useful property to find multiple complexity bounds of a program.

[3] NumInv also found this assertion and other postconditions at L2, but discarded them because they are implied by other discovered invariants and are thus redundant.

3.3 Example

We demonstrate this technique by finding the equalities $b = ya$ and $x = qy + r$ at location L1 in the cohendiv program in Figure 1, when using degree $d = 2$.

For the six variables $\{a, b, q, r, x, y\}$ at L1, together with $d = 2$, we create 28 terms $\{1, a, \ldots, y^2\}$. NumInv uses these terms to form the template $c_1 + c_2 a + \ldots c_{28} y^2 = 0$ with 28 unknown coefficients c_i. Next, in the first loop, NumInv uses KLEE to obtain traces such as those given in Figure 1 to form (at least) 28 equations. From this set of initial equations, NumInv solves and extracts seven equalities.

Now NumInv enters the second loop. In iteration #1, KLEE cannot find cexs for two of these candidates $x = qy + r, b = ya$ (which are actually true invariants) and save these as invariants. KLEE finds cexs for the other five,[4] and NumInv forms new equations from the cexs. Next, NumInv combines the old and new equations to obtain another seven candidates, two of which are the already saved ones (because we also use the old equations). In iteration #2, KLEE obtains cexs for the other five candidates. With the help of the new cex equations, NumInv now infers three candidates, two of which are the saved ones. In iteration #3, NumInv uses KLEE to find cexs disproving the remaining candidate and again uses the new cexs to infer new candidates. This time NumInv only finds the two saved invariants $x = qy + r, b = ya$ and thus stops.

4 INFERRING OCTAGONAL INEQUALITIES

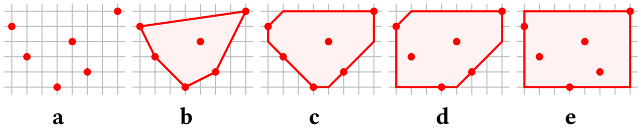

Figure 3: (a) A set of points in 2D and its approximation using the (b) polyhedral, (c) octagonal, (d) zone, and (e) interval regions. These shapes are represented by the conjunctions of inequalities of the forms $c_1v_1 + c_2v_2 \geq c$, $\pm v_1 \pm v_2 \geq c$, $v_1 - v_2 \geq c$, and $\pm v \geq c$, respectively.

NumInv's second algorithm aims to infer linear inequalities among program variables, essentially by attempting to find a convex polyhedron in a multi-dimensional space that contains all observed trace points. Figure 3 illustrates several examples of polygons in two-dimensional space. Figure 3a shows a set of points created from input traces. Figures 3b, 3c, 3d, and 3e approximate the area enclosing these points using the polygonal, octagonal, zone, and interval shapes that are represented by conjunctions of *inequalities* of different forms as shown in Figure 3. These forms of relations are sorted in decreasing order of expressive power and computational cost. For example, interval inequalities are less expressive than zone inequalities, and computing an interval, i.e., the upper and lower bound of a variable, costs much less than computing the convex hull of a zone.

NumInv infers octagonal inequalities. These can be computed efficiently (linear time complexity) and are also relatively expressive (e.g. represent zone and interval inequalities as illustrated in Figure 3). Thus, the computation of octagonal inequalities also produces zone and interval inequalities for free. By balancing computational cost with expressive power, octagonal relations are especially useful in practice for detecting bugs in flight-control software, and performing array bound and memory leak checks [13, 32].

```
Function findUpperBound(term, minV, maxV, $P$, $L$)
    if minV ≡ maxV then return maxV
    else if maxV − minV ≡ 1 then
        cexInps ← verify ($P$, L, {term ≤ minV}, {})
        if $cexInps$ ≡ ∅ then return minV
        else return maxV
    else
        midV ← $\lceil \frac{\text{maxV}+\text{minV}}{2} \rceil$
        cexInps ← verify ($P$, L, {term ≤ midV}, {})
        if $cexInps$ ≡ ∅ then
            maxV = midV
        else
            //disproved
            traces ← exec ($P$, L, cexInps)
            minV = max (instantiate (term, traces))
        return findUpperBound(term, minV, maxV, $P$, $L$)
```

Figure 4: CEGIR algorithm for finding inequalities.

4.1 Terms

The edges of an inferred octagon are represented by a conjunction of eight inequalities of the form $a_1v_1 + a_2v_2 \geq k$, where v_1, v_2 are variables, $a_1, a_2 \in \{-1, 0, 1\}$ are coefficients, and k is a real-valued constant. For example, from the traces in Figure 1, we could infer octagonal inequalities such as $4 \leq r \leq 15$ and $3 \leq r - y \leq 13$ at location L1.

NumInv infers octagonal inequalities by trying to prove invariants $t \leq k$ for some constant k. Here, t is a term involving two variables so that $t \leq k$ is an octagonal constraint, e.g., t could be $x - y$ or $x + y$. More precisely, we consider all possible terms for n variables: we create n^2 variable pairs from a set of n variables and obtain 8 octagonal terms $\{\pm v_1, \pm v_2, \pm v_1 \pm v_2\}$ for each pair v_1, v_2. For each such term we attempt to prove its upper bound k_1 and lower bound k_2, if they exist, using the algorithm described next.

4.2 Algorithm

One idea for inferring inequalities would be to iteratively refine conjectured bounds using cexs, but this can take a long time. For example, to find the invariant $x \leq 100$, we can first infer $x \leq 1$ from traces such as $x \in \{0, 1\}$. We can then disprove this candidate with cexs such as $x \in \{2, 3\}$ and weaken the relation to $x \leq 3$, which can also be disproved and weakened. This keeps going until we get the cex $x = 100$, which would allow us to obtain and prove $x \leq 100$. Even worse than taking a long time to reach the bound k, this brute-force approach does not terminate when x has no constant bound. As such, we use a divide-and-conquer-style search instead.

Finding Upper and Lower Bounds. We use the CEGIR algorithm shown in Figure 4 to compute a precise integral upper bound k of a

[4] These spurious results often have many terms and large coefficients, e.g., the simplest of these seven is $ry^2 - xy^2 - 72ry + 72xy + 8190q + 1397r - 1397x = 0$.

term t. Similar to a binary search, this algorithm computes k from a given interval by repeatedly dividing an interval into halves that could contain k. We start with the interval $[minV, maxV]$ where $maxV = -minV$; our experience is that inequalities are most useful with small constants, so by default we set $maxV = 10$. Next we check $t \leq midV$ where $midV = \lceil \frac{maxV+minV}{2} \rceil$. If this inequality is true, then k is at most $midV$ and thus we reduce the search to the interval $[minV, midV]$. Otherwise, we obtain counterexample traces showing that $t > midV$ and reduce the search to $[minV', maxV]$, where $minV'$ is the largest trace value observed for t. Thus this approach gradually strengthens the guess of k by repeatedly reducing the interval containing it.

We also use the same approach to find the lower bound of a term t by computing the upper bound of $-t$. This is possible because the semantics and results of all computations are reversed when we consider $-t$. For example, the max over the traces $t \in \{2, 3\}$ with respect to $-t$ is -2 and $-t \leq midV$ indicates the lower bound of t is at least $-midV$.

The algorithm terminates and gives a precise upper bound value when t ranges over the integers. The algorithm stops when $minV$ and $maxV$ are the same (because we no longer can reduce the intervals) or when their difference is one (because we cannot compute the exact $midV$). Currently NumInv does not support real-valued bounds. However, we believe that this algorithm can be extended to handle the case when t ranges over the reals. More specifically, we can approximate the results by using only whole numbers or values up to certain decimal places. This sacrifices precision but preserves soundness and termination, e.g., the invariant is $x \leq 4.123$ but we obtain $x \leq 4.2$, which is also an invariant, but less precise.

4.3 Example

Recall the program cohendiv from Figure 1. Suppose NumInv wishes to find inequality invariants at L1 (within $[-10, 10]$). It first uses KLEE to check candidate relations $r \leq 10, y \leq 10, r + y \leq 10, r - y \leq 10, \ldots$ and removes those that KLEE refutes. The remaining relations have upper bounds less than or equal to 10.

For each remaining inequality candidate, NumInv iterates to find tighter upper bounds. For example, suppose we wish to find k such that $r - y \leq k$. Since $r - y \leq 10$, the algorithm sets $midV = (10 + -10) \div 2$ which is 0 and thus tries to check $r - y \leq 0$. This succeeds. However, this turns out to be weaker than necessary. In the next iteration #2, NumInv tightens the bound to $(0-10) \div 2 = -5$ and checks $r - y \leq -5$. This time KLEE returns a cex showing that $r - y = -3$. In iteration #3, NumInv relaxes the bound to $(0 - 3) \div 2 = -1$ and KLEE cannot refute $r - y \leq -1$. In iteration #4, NumInv guesses and checks $(-1 - 3) \div 2 = -2$, in which case KLEE can find cexs stating that $r - y = -1$. At this point NumInv accepts the tightest bound $r - y \leq -1$ found in iteration #3. The process for finding the lower bounds is similar as described above.

5 EXPERIMENTAL RESULTS

NumInv is implemented in Python and uses the linear equation solver in the SAGE mathematical environment [43]. NumInv takes as inputs a C program, a list of locations, and interested numerical variables at these locations, and it returns relations among these variables at the considered locations. As mentioned, NumInv uses DIG's algorithms to infer invariants and calls the symbolic execution tool KLEE to check results and obtain counterexamples for refinement. The final step that removes redundant invariants uses the Z3 solver [16] to check SMT formulas.

We generate numerical invariants of two forms: nonlinear polynomial equations and octagonal inequalities. For octagonal invariants, NumInv by default considers the bounds within the range $[-10, 10]$. For equalities, NumInv by default sets a single parameter $\alpha = 200$ so that it can generate invariants without a priori knowledge of specific degrees. NumInv automatically adjusts the maximum degree so that the number of generated terms does not exceed α. For example, NumInv considers equalities up to degree 5 for a program with four variables and equalities up to degree 2 for a program with twelve variables. We acknowledge that inferring these parameter constants robustly and automatically is important future work. These constants can be chosen by the NumInv user; we chose values based on our experience. Note that the divide and conquer approach to inferring inequalities in Figure 4 is quite useful if the user decides to increase the bounds; for range $[-10, 10]$ the number of iterations is $\log(20) \approx 5$ rather than 20 (if we use a brute force algorithm) but for range $[-100, 100]$ it is $\log(200) \approx 8$, not 200 (using brute force).

Experiments. We evaluate and compare NumInv to other invariant analysis systems by considering three experiments. The first experiment in Section 5.1 determines if NumInv can discover invariants representing precise semantics and correctness properties of programs having complex arithmetic. The second experiment in Section 5.2 explores the use of NumInv's invariants to represent precise program complexity bounds. The last experiment in Section 5.3 compares NumInv's performance with the state of the art CEGIR tool PIE. The experiments reported below were performed on a Linux system with a 10-core Intel i7 CPU and 32 GB of RAM.

5.1 Analyzing Program Correctness

Programs. In this experiment, we focus on generating invariants that capture semantics and correctness properties of programs with nonlinear polynomial invariants. For this task, we evaluate NumInv on the NLA [35] test suite consisting of programs involving complex arithmetic. The suite, shown in Table 1, consists of 27 programs from various sources collected previously by Rodríguez-Carbonell and Kapur [7, 8, 39]. These programs are relatively small, on average two loops of 20 lines of code each. However, they implement nontrivial mathematical algorithms involving general polynomial properties and are often used to benchmark numerical invariant analysis methods [7]. To the best of our knowledge, NLA contains the largest number of numerical algorithms with nonlinear polynomial invariants.

Each program in NLA comes with documented or annotated correctness assertions requiring polynomial invariants, mostly loop invariants having nonlinear polynomial equalities. For evaluation purposes, we consider invariants at the annotated locations and compare them to the documented invariants.

Results. Table 1 summarizes the results and reports the medians across 11 runs. Column **Locs** gives the number of locations in the programs where we consider invariants. Column **Invs** reports the

Table 1: Results for 27 NLA programs. ✓: NumInv generates sufficiently strong results to prove known invariants.

Prog	Desc	Locs	V, T, D	Invs	Time (s)	Correct
cohendiv	div	2	6,3,2	11	24.57	✓
divbin	div	2	5,3,2	12	116.83	✓
manna	int div	1	5,4,2	5	30.86	✓
hard	int div	2	6,3,2	13	71.47	✓
sqrt1	sqr root	1	4,4,2	5	19.35	✓
dijkstra	sqr root	2	5,7,3	14	89.32	✓
freire1	sqr root	1	-	-	-	-
freire2	cubic root	1	-	-	-	-
cohencu	cubic sum	1	5,5,3	5	22.56	✓
egcd1	gcd	1	8,3,2	9	284.52	✓
egcd2	gcd	2	-	-	-	-
egcd3	gcd	3	-	-	-	-
prodbin	gcd, lcm	1	5,3,2	7	45.13	✓
prod4br	gcd, lcm	1	6,3,3	11	87.37	✓
knuth	product	1	8,6,3	9	84.69	✓
fermat1	product	3	5,6,2	26	185.36	✓
fermat2	divisor	1	5,6,2	8	101.83	✓
lcm1	divisor	3	6,3,2	22	175.29	✓
lcm2	divisor	1	6,3,2	7	163.86	✓
geo1	geo series	1	4,4,2	7	24.41	✓
geo2	geo series	1	4,4,2	9	24.33	✓
geo3	geo series	1	5,4,3	7	32.38	✓
ps2	pow sum	1	3,3,2	3	17.08	✓
ps3	pow sum	1	3,4,3	4	17.86	✓
ps4	pow sum	1	3,4,4	4	18.55	✓
ps5	pow sum	1	3,5,5	4	19.36	✓
ps6	pow sum	1	3,5,6	3	21.09	✓

number of equality and inequality invariants discovered by NumInv. Column **V, T, D** shows the number of distinct variables, terms, and the highest polynomial degree in those invariants. Column **Time** reports the time in seconds to generate these results, including the time to remove redundant results. Column **Correct** indicates whether these invariants matched or were strong enough to prove (imply) the documented invariants.

NumInv found invariants that matched or were sufficiently strong to prove the documented invariants of 23/27 programs in NLA. For these programs, we discovered results matched the documented invariants exactly as written in most cases. NumInv also achieved invariants that are logically equivalent to the documented ones. For example, `sqrt1` has two documented equalities $2a+1 = t, (a+1)^2 = s$; our results gave $2a + 1 = t, t^2 + 2t + 1 = 4s$, which is equivalent to $(a + 1)^2 = s$ by substituting t with $2a + 1$. In many cases, NumInv also found undocumented invariants, e.g., most of the discovered octagonal inequalities in the cohendiv program in Figure 1 are undocumented. For dijkstra, NumInv found the documented invariant describing the semantics of a loop computation, but also discovered an undocumented loop invariant $h^3 = 12hnq - 16npq + hq^2 + 4pq^2 - 12hqr + 16pqr$. Manual analysis shows that this strange relation is correct and captures detailed dependencies among variables in the loop. Thus, NumInv's strong invariants can help with understanding both *what* the program does and also *how* the program works. In Section 5.2, we further exploit such complex invariants to analyze program complexity.

```
void triple(int n, int m, int N){
  assert (0 <= n && 0 <= m && 0 <= N);
  int i = 0, j = 0, k = 0; int t = 0;
  while(i < n){//loop 1
    j = 0; t++;
    while(j<m){//loop 2
      j++; k=i; t++;
      while (k<N){k++; t++;}// loop 3
      i=k;
    }
    i++;
  }
  [L]
}
```

Figure 5: An example program that has muliple polynomial complexity bounds.

For these programs, the run time for finding equality invariants is dominated by solving equations because we are solving hundreds of equations with hundreds of unknowns each time. The run time significantly improves if we restrict the search to invariants up to a certain given degree. For example, NumInv took 2s to find the invariants in sqrt1 using degree 2, but it took 20s to find the same invariants using the parameter $\alpha = 200$, which queries NumInv for all invariants up to degree 5 in this program. For egcd1, the running time is also cut by more than half if we only focus on quadratic invariants. For inequality invariants, the running time is dominated by checking because we rapidly guess the bound values and check them with KLEE. Moreover, NumInv has to perform this "guess and check" computation for octagonal constraints over all possible pairs of variables.

We were not able to find invariants for 4/27 programs. NumInv was able to infer results matching the documented invariants for freire1 and freire2, but KLEE cannot run on these programs because they contain floating point operations. For egcd2 and egcd3, the underlying SAGE equation solver stopped responding for more than half of the 11 runs (though we observed all correct results for the runs during which the solver worked). These problems might occur because the solver has to consider hundreds of equations with very large coefficients for hundreds of unknowns. We are investigating and reporting these problems to the SAGE developers.

5.2 Analyzing Computational Complexity

We use NumInv to discover invariants capturing a program's computational complexity, e.g., $O(n^3)$ where n is some input. Figure 5 shows the program triple with three nested loops, adapted from the program in Figure 2 of Gulwani *et al.* [26]. The complexity of this program, i.e., the total number of iterations of all three loops at location L, appears to be $O(nmN)$ at first glance. Additional analysis yields a more precise bound of $O(n + mn + N)$ because the number of iterations of the innermost loop is bounded by N instead of nmN and it furthermore directly affects the running time of the outermost loop [26].

Table 2: Results for computing programs' complexities. ✓: NumInv generates the expected bounds. ✓✓: NumInv obtains more informative bounds than reported results. ✓*: program was slightly modified to assist the analysis.

Prog	V, T, D	Invs	Time (s)	Bound
cav09_fig1a	2,5,2	1	14.35	✓
cav09_fig1d	2,5,2	1	14.24	✓
cav09_fig2d	3,2,2	3	36.09	✓
cav09_fig3a	2,2,2	3	14.24	✓
cav09_fig5b	3,5,2	5	46.88	✓*
pldi09_ex6	3,8,3	7	54.18	✓
pldi09_fig2 (triple)	3,15,4	6	93.55	✓✓
pldi09_fig4_1	2,3,1	3	44.26	✓
pldi09_fig4_2	4,4,2	5	43.72	✓
pldi09_fig4_3	3,3,2	3	37.54	✓
pldi09_fig4_4	5,4,2	4	56.60	-
pldi09_fig4_5	3,4,2	3	31.60	✓*
popl09_fig2_1	5,12,3	2	211.73	✓✓
popl09_fig2_2	4,9,3	2	65.17	✓✓
popl09_fig3_4	3,4,3	4	54.70	✓
popl09_fig4_1	3,3,2	2	42.76	✓*
popl09_fig4_2	5,12,3	2	158.3	✓✓
popl09_fig4_3	3,3,2	5	39.28	✓
popl09_fig4_4	3,3,2	3	34.28	✓

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

$$N^2mt + Nm^2t - Nmnt - m^2nt - Nmt^2 + mnt^2 + Nmt$$
$$-Nnt - 2mnt + Nt^2 + mt^2 + nt^2 - t^3 - nt + t^2 = 0.$$

At first glance, this quartic (degree 4) equality with 15 terms looks incomprehensible and quite different than the expected bound $O(n+mn+N)$ or even $O(mnN)$. However, solving this equation for t, i.e., finding the roots, yields three solutions $t = 0$, $t = N + m + 1$, and $t = n - m(N - n)$. Careful analysis reveals that these results actually describe three distinct and exact bounds of this program:

$$\begin{array}{lll} t = 0 & \text{when} & n = 0, \\ t = N + m + 1 & \text{when} & n \le N, \\ t = n - m(N - n) & \text{when} & n > N. \end{array}$$

Thus, NumInv can find numerical invariants that represent precise program complexity. More importantly, the obtained relations can describe expressive and nontrivial *disjunctive* invariants, which capture different possible complexity bounds of a program.

Programs. We apply NumInv to find complexity invariants on programs adapted from [25–27].[5] These programs, shown in Table 2, are small, but they have nontrivial structures such as nested loops and represent examples drawn from Microsoft's production code [26]. For these programs, we introduce the counter variable t and obtain relations among t and other variables, such as inputs, at the program exit locations.

Results. Table 2 shows the median results across 11 runs and has similar format as that of Table 1. For column **Bound**, a checkmark denotes that NumInv generates invariants representing a similar bound to the one reported in the respective paper. A double checkmark (✓✓) denotes that NumInv obtains more informative bounds than reported results. A checkmark with an asterisk (✓*) denotes that the program was modified slightly to assist the analysis.

As can be seen, NumInv produced very promising results that capture the precise complexity bounds for these programs. For 18/19 programs, NumInv discovered expected or even more informative bounds than reported results in the respective papers. For many programs, NumInv generated equality invariants representing tight bounds, which can be combined with the discovered octagonal inequalities to produce expected bounds. For example, for popl09_fig3_4, NumInv obtained that the number of iterations t is either n or m. In addition, NumInv finds inequalities expressing that t is larger than both n and m, suggesting that t is equal to $\max(n, m)$, which is the bound also obtained in [27]. Thus, inequalities, though appearing much weaker compared to the obtained equalities, play an important role to achieve precise program analysis.

Interestingly, in some cases, NumInv produced results that are more informative than the ones given in the respective papers. This is particularly the case for the program triple analyzed earlier because the three distinct bounds produced by NumInv are strictly less than the bound $n + mn + N$ given in [26]. We note that in most other cases where NumInv obtained a better bound, the differences were not as apparent as they were for triple.

We performed some adaptations in certain programs to assist the bound analysis. For cav09_fig5b, we considered the invariant obtained as one close to the expected bound. For popl09_fig4_1, we inserted an assert statement that $m \ge 0$ at the beginning of the program. Finally, for pldi09_fig4_5, for the number of iterations t we obtained the three solutions $t = n - m$, $t = m$, or $t = 0$, which imply the correct upper bound $\max(0, n - m, m)$.

Finally, NumInv obtained invariants that are not strong enough to show the expected bound for pldi09_fig4_4. However, we would have obtained this bound if we had introduced a variable (or a term) representing the quotient from the division of two other variables in the program. In our experiments, when inserting such a variable, we obtained bounds that were tighter than the ones presented in [26]. Such cases suggest a possible extension to NumInv for predicting useful terms.

5.3 Comparing to PIE

NumInv automatically generates invariants for a program location without any given assertions or postconditions. Other state-of-the-art CEGIR tools such as PIE generate invariants in a goal-directed manner, driven by supplied postconditions. In this experiment, we compare NumInv with PIE's guided inference with postconditions. This experiment used the HOLA benchmark programs [17] (adapted by the developers of PIE). These programs, shown in Table 3, are short (10-40 LoC each) C programs already annotated with postconditions.

We first ran PIE on each program and recorded PIE's running time in seconds. Then, we removed the postcondition and ran NumInv, asking it to generate invariants at the location in the program

[5] We disable nondeterministic functions in these programs because currently NumInv assumes deterministic programs.

Table 3: Results for the HOLA benchmarks [17]. ✓: Made invariants from PIE. ✓✓: Made stronger invariants than PIE. An asterisk * indicates that verifying the invariant required additional investigation. ○: Failed to make any invariants, no running time reported in that case.

Benchmark	PIE time (s)	NumInv time (s)	Correct
01	21.88	8.75	✓✓
02	36.12	10.35	✓
03	56.28	108.20	✓✓
04	19.11	NA	○
05	25.19	13.20	✓✓
06	61.98	14.67	✓*
07	NA	16.83	✓
08	19.02	31.49	✓✓*
09	NA	30.19	✓
10	24.6	NA	○
11	27.95	NA	○
12	44.52	NA	○
13	NA	19.33	✓
14	25.98	11.65	✓✓
15	48.30	7.7	✓
16	33.19	29.07	✓
17	53.36	10.33	✓✓
18	21.70	7.7	✓✓
19	NA	25.79	✓
20	331.93	104.40	✓✓
21	25.65	11.60	✓*
22	25.40	10.90	✓
23	23.40	9.07	✓✓
24	51.22	NA	○
25	NA	16.76	✓
26	87.64	13.50	✓
27	55.41	376.80	✓
28	22.16	NA	○
29	58.82	NA	○
30	33.92	NA	○
31	88.10	20.39	✓
32	226.73	NA	○
33	NA	48.04	✓*
34	121.87	12.20	✓✓
35	20.07	13.23	✓
36	NA	14.98	✓
37	NA	14.23	✓✓*
38	37.37	10.83	✓
39	24.68	2.39	✓
40	60.71	17.07	✓*
41	34.10	15.47	✓ ✓*
42	54.93	13.13	✓ ✓*
43	21.16	11.3	✓
44	31.92	12.3	✓
45	84.00	15.3	✓
46	27.56	NA	○

where the postcondition was. If NumInv was able to generate invariants, we compared those invariants to the postcondition. If the invariants that NumInv generated were at least precise enough to establish the given postcondition, then NumInv earned a checkmark (✓). If the invariants were more precise, then NumInv earned a double checkmark (✓✓). For the programs that NumInv could not generate invariants, then the analysis is assigned the symbol ○. The results are in Table 3.

For 36/46 programs, NumInv found invariants that were at least as strong as the postconditions in the PIE programs. For the remaining 10/46 programs, NumInv failed to produce the necessary invariants. For 13 of the 36 programs where NumInv produced invariants, NumInv was able to generate stronger invariants. For example, for program 17, the target postcondition was $k \geq n$ given a precondition $n \geq 0$, and NumInv produced, among other invariants, that $k = (n^3 - n + 6)/6$, which implies that for all $n \geq 0$, $k \geq n$.

For programs having the ✓* or ✓✓*, NumInv found stronger invariants that imply the given postcondition, but require additional human effort to reason about. For program 42, the given postcondition is $a \ \% \ 2 = 1$, i.e., a is odd. NumInv found the invariants $xy = x + y - 1, u_1 - a \leq -2, a = x + y - 1$, and $2u_1 = x + y - 2$. This set of constraints implies that $x + y = 2(u_1 + 1)$ and $a = x + y - 1$, which indicates that a is indeed odd. But the first invariant in this set produced by NumInv, also points to another relation among those variables, namely that at least one of x and y is equal to 1, and thus we marked this example with a double checkmark and additionally annotated it with an asterisk.

Another interesting case is with program 8 that contains a postcondition $x < 4 \lor y > 2$, which has a disjunctive form of strict inequalities that NumInv does not support. Instead of generating this, NumInv returns a stronger relation $x \leq y$, which implies this postcondition and therefore proves it.

Summary. These experiments show that NumInv is effective in producing expressive and useful invariants. The NLA experiment in Section 5.1 shows that NumInv discovers necessary invariants to understand the semantics and check correctness properties of 23/27 NLA programs containing nontrivial arithmetic. The Complexity experiment in Section 5.2 indicates that NumInv discovers useful invariants that capture challenging complexity bounds for 18/19 programs used to benchmark static complexity analyses. We also note that the recent CEGIR tools ICE and PIE cannot find any of these nonlinear polynomial invariants produced by NumInv in these experiments, even when we explicitly tell these tools that they should attempt to verify these invariants. Finally, the HOLA experiment in Section 5.3 shows that NumInv competes well with PIE and in 36/46 programs discovers invariants that match or are more informative than PIE's.

5.4 Threats to Validity

As mentioned earlier, NumInv can return *unsound* results because KLEE cannot fully verify programs with complex polynomial properties. We can recover soundness by using a true verifier instead, e.g., we are considering the verification tools CPAChecker [4] and Ultimate Automizer [28], which performed well in the recent SV-COMP 2017 [3]. However, our experience shows that KLEE is effective in finding counterexamples disproving invalid results and thus results that KLEE cannot disprove have high likelihood of being correct. KLEE is also practical because it can consider challenging invariants that are not understandable to many sound verifiers.

KLEE does not fully support floating point arithmetic and thus NumInv is limited to finding invariants over integral variables. KLEE is also language dependent, thus NumInv considers only C programs. We are extending NumInv with additional verification backends that support richer semantics (e.g., arithmetic over the reals) and other languages (e.g., JPF [24] for Java programs).

DIG's algorithms focus on specialized classes of numerical invariants, thus NumInv is unlikely to find invariants of other, unrelated forms. However, our results show that NumInv can often generate invariants that are logically equivalent or sufficiently strong to prove other forms of complex invariants, e.g., disjunctive ones.

Although our benchmark programs have nontrivial structures (e.g., nested loops) with complex arithmetic and have been used to evaluate modern invariant generation systems, these programs are small and do not represent real-world applications containing hundreds of thousands of lines of code. Nonetheless, we believe that CEGIR is a promising approach to build invariant analysis tools that can scale and handle larger and more complex codebases. This is because dynamic analysis allows for inferring expressive invariants efficiently from traces and static checkers such as KLEE have become more powerful and practical in recent years.

6 RELATED WORK

We review related invariant generation techniques using pure static analysis, dynamic analysis, and CEGIR approaches.

Static invariant generation. Abstract interpretation [11, 12, 14] computes an invariant that over-approximates reachable program states. This method starts from a weak invariant representing an initial approximation and iteratively strengthens the invariant by analyzing the structure of the program until reaching a fixed point. Over-approximation can lead to imprecise information and produce false positive errors. Thus, major research directions in this area focus on finding *abstract domains* that are sufficiently expressive to retain important information from the programs. For example, the work in [13, 32] focus on the six-edged zone relations and the eight-edge octagon relations shown in Figure 3.

Rodríguez-Carbonell *et al.* [7, 8, 39] use abstract interpretation to generate nonlinear polynomial equalities. They first observe that a set of polynomial invariants forms the algebraic structure of an ideal, then compute the invariants using Gröbner basis and operations over the ideals, based on the structure of the program until reaching a fixed point. The work only analyzes programs with assignments and loop guards that are expressible as polynomial equalities. In addition, this technique does not find inequalities and does not support programs with nested loops.

Dynamic invariant generation. The popular tool Daikon [18–21, 38] infers candidate invariants from traces and templates. Daikon comes with a large list of invariant templates and tests them against program traces. Templates that are violated in any of the test runs are removed and the remainders are presented as the possible invariants. For numerical relations, Daikon can find linear relations over at most three variables and has a small number of fixed nonlinear polynomial templates such as $x = y^2$. In general, the tool has limited support for inequalities and disjunctive invariants.

CEGIR Approaches. Sharma *et al.* [42] present a *Guess-and-Check* technique for inferring equality invariants. This technique is the standard CEGIR approach, and the "guess" component infers equalities using the similar equation solving technique in DIG. Thus for equality, this technique has the same theoretical power as NumInv. The "check" component uses the Z3 SMT solver, and in this context, it is interesting to note the various differences in running time caused by the different choices made in the latter and our implementation, and specifically the use of KLEE instead of Z3. This *Guess-and-Check* approach is limited to equality relations and, as mentioned in Section 4.2, it is not trivial to extend to finding inequality invariants.

The PIE (Precondition Inference Engine) tool [37] can generate both preconditions and loop invariants to automatically verify given assertions. Given an assertion Q, the goal is to produce a predicate formula sufficiently strong to ensure the assertion. To do this, PIE iteratively learns and refines a set of features (predicates over inputs such as $x > 0$) that are sufficiently strong to separate"good" traces satisfying Q and "bad" traces violating Q. These predicates form the required precondition that proves the assertion. The novelty of PIE is that it does not rely on a fixed class of predicates and can construct necessary predicates during the inference process. Nonetheless, the tool cannot provide invariants for arbitrary locations in the program, especially if no additional assertions are given. More specifically, on the cohendiv example in Figure 1, PIE did not converge to an invariant.

The ICE (implication counter-example) learning model [23] is also a CEGIR approach that generates inductive invariants to prove given assertions. The "student" uses a decision learning algorithm to guess candidate invariants expressed over predicates, which separate the good and bad traces. The "teacher" uses the Boogie verifier to check and provide good, bad, and novel implication counterexamples to help the student infer more precise inductive invariants. For efficiency, they restrict attention to the octagon domain and search only for predicates that are arbitrary boolean combinations of octagonal inequalities. Similar to PIE, ICE infers only necessary invariants to prove assertions. Even when provided with assertions such as the postconditions of the program cohendiv in Figure 1, ICE fails to prove them. We note that part of the reason might be because ICE does not support arithmetic operations such as division and modulo.

7 CONCLUSION

We present NumInv, a CEGIR-based tool that discovers numerical invariants at arbitrary program locations. NumInv uses a dynamic analysis to infer invariants and the test-input generation tool KLEE to verify them. For invalid invariants, KLEE returns counterexample traces that are then used to help the inference algorithm discard invalid results and to find new invariants. The use of KLEE allows NumInv to work on programs with nontrivial arithmetic and discover useful and complex invariants. Preliminary experiments show that NumInv often outperforms state-of-the-art CEGIR systems in discovering invariants required to understand and analyze semantics, correctness, and complexity properties of programs.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their detailed feedback and helpful comments. This research was supported by DARPA under contracts FA8750-15-2-0104 and FA8750-16-C-0022.

REFERENCES

[1] 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 *PLDI*. 362–375.

[2] 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.

[3] Dirk Beyer. 2017. Software Verification with Validation of Results. In *TACAS*. Springer, 331–349.

[4] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *CAV*. Springer, 184–190.

[5] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jerome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A Static Analyzer for Large Safety-Critical Software. In *PLDI*. 196–207.

[6] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. USENIX Association, 209–224.

[7] Enric Rodríguez Carbonell. 2006. *Automatic generation of polynomial invariants for system verification*. Ph.D. Dissertation. Technical University of Catalonia, Barcelona, Spain.

[8] Enric Rodríguez Carbonell and Deepak Kapur. 2007. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation* 42, 4 (2007), 443–476.

[9] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2000. Counterexample-guided abstraction refinement. In *CAV*. Springer, 154–169.

[10] Edward Cohen. 1990. *Programming in the 1990s: An introduction to the calculation of programs*. Springer.

[11] P. Cousot and R. Cousot. 1976. Static Determination of Dynamic Properties of Programs. In *International Symposium on Programming*. 106–130.

[12] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*. ACM, 238–252.

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

[14] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *POPL*. ACM, 84–96.

[15] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: path-sensitive program verification in polynomial time. In *PLDI*. 57–68.

[16] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *TACAS*. Springer, 337–340.

[17] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive Invariant Generation via Abductive Inference. In *OOPSLA*. 443–456.

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

[19] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *TSE* 27, 2 (2001), 99–123.

[20] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. 2000. Quickly detecting relevant program invariants. In *ICSE*. ACM, 449–458.

[21] 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.

[22] Jérôme Feret. 2004. Static analysis of digital filters. In *ESOP*. Springer, 33–48.

[23] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning Invariants Using Decision Trees and Implication Counterexamples. In *POPL*. 499–512.

[24] The Java Pathfinder group. 2017. JPF: Java PathFinder. (2017). http://babelfish.arc.nasa.gov/trac/jpf/.

[25] Sumit Gulwani. 2009. SPEED: Symbolic Complexity Bound Analysis. In *CAV*. 51–62.

[26] Sumit Gulwani, Sagar Jain, and Eric Koskinen. 2009. Control-flow Refinement and Progress Invariants for Bound Analysis. In *PLDI*. 375–385.

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

[28] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2010. Nested interpolants. In *POPL*. ACM, 471–482.

[29] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. 2002. Lazy Abstraction. In *POPL*. ACM, 58–70.

[30] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *CAV*. 781–786.

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

[32] Antoine Miné. 2004. *Weakly relational numerical abstract domains*. Ph.D. Dissertation. École Polytechnique, France.

[33] Van Chan Ngo, Mario Dehesa-Azuara, Matthew Fredrikson, and Jan Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *2017 IEEE Symposium on Security and Privacy*. 710–728.

[34] ThanhVu Nguyen. 2014. *Automating Program Verification and Repair Using Invariant Analysis and Test-input Generation*. Ph.D. Dissertation. University of New Mexico.

[35] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using Dynamic Analysis to Discover Polynomial and Array Invariants. In *ICSE*. IEEE, 683–693.

[36] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. *TOSEM* 23, 4 (2014), 30:1–30:30.

[37] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven Precondition Inference with Learned Features. In *PLDI*. 42–56.

[38] Jeff H. Perkins and Michael D. Ernst. 2004. Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants. In *FSE*. 23–32.

[39] E. Rodríguez-Carbonell and D. Kapur. 2007. Automatic Generation of Polynomial Invariants of Bounded Degree Using Abstract Interpretation. *Science of Computer Programming* 64, 1 (Jan. 2007), 54–75.

[40] Mardavij Roozbehani, Eric Feron, and Alexandre Megrestki. 2005. Modeling, Optimization and Computation for Software Verification. In *Hybrid Systems: Computation and Control*. ACM, 606–622.

[41] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2005. Scalable analysis of linear systems using mathematical programming. In *VMCAI*. Springer, 25–41.

[42] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V Nori. 2013. A data driven approach for algebraic loop invariants. In *ESOP*. Springer, 574–592.

[43] W. A. Stein and others. 2017. Sage Mathematics Software. (2017). http://www.sagemath.org.