Using Symbolic States to Infer Numerical Invariants

ThanhVu Nguyen, KimHao Nguyen, and Matthew B. Dwyer

Abstract—Automatically inferring invariant specifications has proven valuable in enabling a wide range of software verification and validation approaches over the past two decades. Recent approaches have shifted from using observation of concrete program states to exploiting symbolic encodings of sets of concrete program states in order to improve the quality of inferred invariants. In this paper, we demonstrate that working directly with symbolic states generated by symbolic execution approaches can improve invariant inference further. Our technique uses a counterexample-based algorithm that iteratively creates concrete states from symbolic states, infers candidate invariants from both concrete and symbolic states, and then validates or refutes candidate invariants using symbolic states. The refutation process serves both to eliminate spurious invariants and to drive the inference process to produce more precise invariants. This framework can be employed to infer complex invariants that capture nonlinear polynomial relations among program variables. The open-source SymInfer tool implements these ideas to automatically generate invariants at arbitrary locations in Java or C programs. Our preliminary results show that across a collection of four benchmarks SymInfer improves on the state-of-the-art by efficiently inferring more informative invariants than prior work.

Index Terms—Program Invariants, Numerical Invariants, Dynamic Analysis, Symbolic Execution, CounterExample Guided Refinement, Program Testing and Verification

1 Introduction

The expressive power of programs lies in their ability to concisely represent repeated sub-computations that arise due to iteration or recursion. Developing software that correctly orchestrates those sub-computations is challenging as programmers learn when they study even basic sorting algorithms. Classic approaches for defining, and understanding, the correctness of such algorithms rely on specification of *program invariants* which define relationships that must hold between program variables at a given location in the program [1], [2], [3].

While invariants play a role in educating programmers about complex algorithms, they also offer the potential to improve programming practice. Research has demonstrated how invariant specifications can be leveraged for fault-detection and verification [3], [4], detecting security vulnerabilities [5], automating the repair of faults [6], and synthesizing low-level implementations [7]. A number of industrial-strength tools provide support for reasoning about invariant specifications [8], [9].

Despite the fact that programmers are exposed to the concept of invariants early in their education, writing specifications is viewed as a burden and developers don't generally add them to their code base [10]. One approach to addressing this is to define implicit specifications, e.g., that a null pointer should never be dereferenced or an array should never be indexed out of bounds. While valuable, this does not address the potential benefit from *program specific invariants*.

The seminal work by Ernst et al. on Daikon [11], [12] addressed this problem by observing that invariants can be thought of as latent properties of program behavior that can be inferred by observing sets of program runs. Techniques like Daikon can only determine *candidate invariants* – since there may be executions that are not observed which falsify the candidate. Nevertheless, these techniques have proven valuable in overcoming the specification burden and generating candidate invariants that can be subsequently verified or falsified by other techniques [13]. Moreover, the ability to reveal these latent properties serves as a rich source of information for understanding undocumented code [14], generating more formal documentation [11], localizing bugs [15], and even proving program termination and non-termination properties [16].

Daikon [12], [14] works by observing *concrete program states* that capture the values of variables at designated *locations of interest* in the program when a program is run on a given input. By sampling large numbers of inputs, Daikon can efficiently determine relationships that may hold among variables across those samples. Confirming that those relationships constitute a true invariant has been a focus of follow-on work to Daikon. Several invariant generation approaches (e.g., iDiscovery [17], PIE [18], ICE [19], NumInv [20], G-CLN [21]) use a hybrid approach that dynamically infers candidate invariants and then attempts to verify that they hold for all inputs. When verification fails, counterexamples are generated which help to refine the invariant inference process to obtain more accurate results – reporting only true invariants. This *CounterExample Guided Invariant Generation* (CEGIR) approach iterates the inference and verification processes until achieving stable results.

- *ThanhVu Nguyen is with the Department of Computer Science, George Mason University, USA.*
  Email: tvn@gmu.edu.
- *KimHao Nguyen is with the University of Nebraska-Lincoln, USA.*
  Email:kdnguyen@cse.unl.edu.
- *Matthew Dwyer is with the Department of Computer Science, University of Virginia, USA.*
  Email:matthewbdwyer@virginia.edu.

An important class of invariants capture numerical relations among program variables. Such *numerical invariants* can take on different mathematical forms. Daikon can infer conjunctive numerical invariant candidates, but its template matching engine makes it inefficient to infer disjunctive invariants. Disjunctive invariants are required to encode properties of programs, but fortunately rich forms of disjunction can be captured by more complex numerical relations. *Nonlinear polynomial* relations, e.g., $x^2 \leq y^2$, arise in many scientific, engineering, and safety- and security-critical software [22], and can encode disjunctive information, e.g., $x^2 \leq y^2$ implies $x \leq -y \vee x \leq y$. *Max/min-plus* relations encode properties that can be expressed in "tropical" algebra [23], [24] and are able to encode a complementary form of disjunctive information, e.g., the max inequality $\max(x, y) \geq 2$ is equivalent to $(x \geq y \wedge x \geq 2) \vee (x < y \wedge y \geq 2)$. As we demonstrate, when used together nonlinear and max/min invariants can express complex program properties, e.g., permutation and sortedness (S5.3), that cannot be expressed with purely conjunctive formulae.

In this work, we present SymInfer, a CEGIR technique that targets the inference of rich forms of numerical invariants using *symbolic program states* as a basis. Our key insight is that symbolic states generated by a symbolic execution engine are (1) compact encodings of large (potentially infinite) sets of concrete states, (2) naturally diverse since they arise along different execution paths, (3) explicit in encoding relationships between program variables, (4) amenable to direct manipulation and optimization, and (5) reusable across many different reasoning tasks within CEGIR algorithms.

We define algorithms for symbolic CEGIR that can be instantiated using different symbolic execution engines, and the SymInfer implementation uses symbolic states generated from Symbolic PathFinder [25] (SPF)—a symbolic executor for Java—and CIVL [26]—a symbolic executor for C. SymInfer uses symbolic states for both invariant inference and verification. For inference, SymInfer uses symbolic states to obtain concrete states to bootstrap a set of candidate invariants using DIG [27], [28], [29]—a dynamic analysis framework for inferring expressive numerical invariants. For verification, SymInfer formulates verification conditions from symbolic states to confirm or refute an invariant, solves those using an SMT solver, and produces counterexamples to refine the inference process.

We evaluated SymInfer over 4 distinct benchmarks, consisting of 108 programs, and compared its performance to state-of-the-art numerical invariant approaches. We find that the use of symbolic states allows SymInfer to overcome several limitations of existing CEGIR approaches. iDiscovery, which uses Daikon for inference, does not support nonlinear properties, and both ICE and PIE timeout frequently when nonlinear arithmetic is involved. NumInv also uses DIG to infer invariants, but it invokes KLEE [30] as a black box verifier for candidate invariants and which causes it to underperform relative to SymInfer for nonlinear and disjunctive invariant inference. G-CLN can infer nonlinear invariants for loops, but it requires manual problem-specific configuration to generate and prove invariants, and even then SymInfer infers more relevant invariants. Our evaluation demonstrates that SymInfer establishes the state-of-the-art for inference of complex nonlinear invariants. Across the benchmarks it is able to infer the ground truth specifications for 106 of 108 programs; the next best tool can infer only 89.

```
int cohendiv(int x, int y){
  assert(x >= 0 && y >= 1);
  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;
}
```

Concrete States

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
			⋮		

Fig. 1: The `cohendiv` integer division program and concrete states observed at location L1 on inputs $(x = 15, y = 2)$ and $(x = 4, y = 1)$. Among the invariatns discovered by SymInfer at L1, the key nonlinear equality $x = qy + r$ describes the precise semantics of the program.

Our prior work [31] made an initial step in exploiting symbolic states for invariant inference. This paper significantly extends those results to include: (1) a novel efficient algorithm to generate inequalities; (2) support for *max*- and *min-plus* formulae to represent disjunctive invariants; (3) proofs of correctness and termination for the presented algorithms; (4) support for Java and C programs; and (5) a broader experimental evaluation that demonstrates the cost-effectiveness of the approach and its superiority to existing invariant inference techniques. The implementation of SymInfer and all experimental data reported in this paper are available at https://github.com/unsat/dig/. These results strongly suggest that symbolic states form a powerful basis for computing program invariants. They permit an approach that blends the best features of dynamic inference techniques and purely symbolic techniques to enable a new state-of-the-art.

2 Overview

To illustrate SymInfer, we show how inferred numerical invariants can assist in understanding programs and analyzing program complexity. We then describe techniques using symbolic states to help remove spurious invariants to achieve expressive and accurate invariants.

2.1 Applications of Numerical Invariants

Program Understanding. SymInfer can help understand program behavior and discover unknown program properties. Consider the `cohendiv` integer division algorithm in Figure 1; L1 and L2 mark the locations of interest. Given this program and the considered locations, SymInfer automatically discovers at L1 the (loop) invariants:

$$x = qy + r \quad ay = b$$
$$b \leq x \quad y \leq r \quad 0 \leq q \quad 1 \leq b \quad 1 \leq y$$

```
void tripple(int M, int N, int P){
  assert (0 <= M && 0 <= N && 0 <= P);
  int i = 0, j = 0, k = 0;
  int t = 0; //counter variable
  while(i < N){
    j = 0; t++;
    while(j < M){
      j++; k = i; t++;
      while (k < P){
        k++; t++;
      }
      i = k;
    }
    i++;
  }
  [L]
}
```

Fig. 2: A program with three complexity bounds.

and at L2 the (postcondition) invariants:

$$x = qy + r \quad ay = b$$
$$r \leq y - 1 \quad 0 \leq r \quad r \leq x$$

These relations are sufficiently strong to understand the semantics and verify the correctness of cohendiv. The key invariant is the nonlinear equality $x = qy + r$, which captures 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 example, the invariants at the program exit reveal several required properties of the remainder r, e.g., non-negative ($0 \leq r$), at most the dividend ($r \leq x$), but strictly less than the divisor ($r \leq y - 1$).

Also, these invariants might help check assertions in the program. For example, we can assert and verify the postcondition stating that the returned quotient is non-negative because the discovered invariants at L2 implies $0 \leq q$.[1]

Complexity Analysis. Another rather surprising use of SymInfer's nonlinear numerical invariants is to characterize the computational complexity of a program, which is useful for identifying possible security problems [32], [33], [34]. Figure 2 shows the program tripple, adapted from Figure 2 of [35], with nontrivial runtime complexity; the program has been modified to include a variable t which serves to encode the number of loop iterations that are computed. At first, tripple appears to take $O(NMP)$ due to the three nested loops. A closer analysis [35] shows a more precise bound $O(N+NM+P)$ because the innermost loop, which is updated each time the middle loop executes, changes the behavior of the outermost loop.

When given this program, SymInfer discovers an interesting and complex postcondition at location L about the variable t:

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

1. SymInfer also found $0 \leq q$ and other invariants at L2, but discarded them because they are implied by other discovered properties and thus are redundant.

This nonlinear equality is valid, but looks 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 showing that this program has different time complexities:

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

Manual analysis shows these results represent the *exact* bound of tripple and are more precise than the bound $O(N + MN + P)$ given in [35] Note that $O(N + MN + P)$ is still a correct *upper bound* of tripple, e.g., when $N > P$ then $O(N + NM + P) = O(N + NM)$, which is equivalent to $O(N - M(P - N)) = O(N + MN)$.

2.2 SymInfer

To infer invariants, SymInfer integrates dynamic and symbolic analyses using the *counterexample-guided invariant generation* (CEGIR) approach. SymInfer's CEGIR consists of two phases: a *dynamic analysis* that infers candidate (equality and inequality) invariants from program execution traces or concrete states, and a *symbolic checker* that checks candidates against the program using symbolic states obtained from a symbolic execution tool. If a candidate invariant is spurious, the checker also provides counterexamples. Concrete states from these counterexamples are obtained and recycled to repeat the process, and produce more accurate results.

These steps of inferring and checking repeat until no new counterexamples or (true) invariants are found. The CEGIR technique exploits the observation that checking a (cheaply generated) candidate solution is often easier than directly inferring a sound solution [28].

2.2.1 Concrete States

Existing dynamic invariant analyses such as Daikon or DIG instrument the program at considered locations to record values of the local variables, and then, given a set of inputs, execute the program to record a set of *concrete states* of the program to generate candidate invariants. Figure 1 shows several concrete states obtained at location L1 when running cohendiv using inputs $(x = 15, y = 2)$ and $(x = 4, y = 1)$.

However, inferring invariants on just concrete states often produces undesirable results. On several hand-selected sets of inputs that seek to expose diverse concrete states, running Daikon on cohendiv results in very simple invariants, e.g., $4 \leq x$ and $0 \leq q$ at location L1. These are clearly much weaker than the key nonlinear invariant for this example. Moreover, the invariant on x is actually spurious since clearly values smaller than 4 can be passed as the first input which will reach L1. The more powerful DIG invariant generator permits the identification of the key equality invariant, but it too will yield the spurious $4 \leq x$ invariant. Spurious invariants are a consequence of the diversity and representativeness of the inputs used, and the observed concrete states. Leveraging symbolic states can help address this weakness.

Path Conditions (Π_{L1})	**Variable Mappings** (σ_{L1})
$0 < y \land y \leq x$	$q \mapsto 0; r \mapsto x; a \mapsto 1; b \mapsto y$
$0 < y \land 2y \leq x$	$q \mapsto 0; r \mapsto x; a \mapsto 2; b \mapsto 2y$
$0 < y \land 2y + y \leq x < 4y$	$q \mapsto 2; r \mapsto x - 2y; a \mapsto 1; b \mapsto y$
$\vdots$	$\vdots$

Fig. 3: Symbolic states at location L1 in the program `cohendiv` in Figure 1.

2.2.2 Symbolic States

SymInfer symbolically executes the program to compute the *symbolic states* at a considered program location L. A symbolic state compactly encodes a large (potentially infinite) set of concrete states. Symbolic states consist of path conditions describing execution paths to L and mappings from program variables at L to symbolic values. For example, Figure 3 shows the symbolic states at location L1 of `cohendiv`.

To check a candidate invariant p, SymInfer asks a constraint solver to determine the validity of p with respect to the symbolic states. If the solver finds a counterexample disproving p, SymInfer extracts concrete states from the counterexample and saves them for subsequent inference. Otherwise, SymInfer accepts and saves p as an invariant.

SymInfer can return *spurious* invariants because the solver might timeout or return unknown, or because symbolic execution might not be able to explore all program paths to compute precise symbolic states. Consequently, SymInfer is designed to explore program states incrementally and adaptively to minimize cost while finding accurate invariants (S3.2).

2.2.3 Inferring Numerical Invariants

SymInfer uses a CEGIR algorithm to find *polynomial equalities* at program locations of interest. For each considered program location, SymInfer creates an equation *template* $c_1t_1 + c_2t_2 + \cdots + c_nt_n = 0$. This template contains n unknown coefficients c_i and n *terms* t_i, with one term for each possible multiplicative combination of relevant program variables, up to some degree d.

SymInfer uses symbolic states to obtain many possible concrete states and substitute their concrete values into the template to form an instantiated linear equation. After obtaining at least n concrete states, SymInfer solves the resulting set of equations for the n unknown coefficients c_i. SymInfer then extracts candidate invariants by substituting the solutions back into the template. Now, SymInfer enters a CEGIR loop that checks the candidate invariants by using symbolic states. We discard any spurious invariants and use the corresponding counterexample concrete states to infer new candidates until no additional true invariants are found.

SymInfer also generates *linear inequalities* in the forms of (i) *octagons*, which are inequalities over two terms, and (ii) *max/min-plus constraints*, which are a form of disjunctive invariants. The early version of SymInfer [31] does not support max/min relations and can only infer octagonal inequalities using a CEGIR divide-and-conquer algorithm, e.g., repeatedly invokes symbolic states to guess and tighten lower and upper bounds of candidate inequalities.

The current version of SymInfer takes advantage of advances in constraint solvers and uses linear optimization to obtain directly from symbolic state the bounds for both octagonal and max/min inequalities. This optimization approach is much more efficient and allows SymInfer to discover more challenging and expressive invariants than the previous CEGIR approach (e.g., the user can configure SymInfer to generate max/min invariants as well as nonlinear inequalities over an arbitrary number of variables instead of the default octagonal linear inequalities).

Finally, in a post-processing phase, from the obtained invariants, SymInfer uses an SMT solver to check and remove any redundant invariants that are logical implications of other invariants. For instance, we suppress $x^2 = y^2$ if $x = y$ is also found because the latter implies the former.

3 Symbolic States

The behavior of a program at a location L can be precisely represented by the set of all possible values of the variables in scope at L. We refer to such values as *concrete L-states* of the program and define them as:

Definition 3.1 (Concrete State). *A concrete L-state is a mapping σ_L from program variables in scope at L to concrete values.*

Figure 1 shows several concrete L-states in the `cohendiv` program. Dynamic analyses such as Daikon and DIG analyze concrete states to infer invariants. These techniques instrument the program at location L to take "snapshots" of the state of the program at L and then execute the program on a set of inputs to record a set of concrete states, which are values of some/all variables at L.

In contrast, a *symbolic L-state* is formulated in terms of a set of *input variables* that capture the values of program inputs in order to represent a (potentially infinite) set of concrete states:

Definition 3.2 (Symbolic State). *A symbolic L-state is a tuple $\langle \sigma_L, \Pi_L \rangle$, where σ_L is a map from program variables in scope at L to symbolic expressions over input variables, and Π_L is the path condition, which is a logical formula over the input variables that the inputs must satisfy to reach L.*

The concrete states defined by a symbolic state are *feasible* – realizable at L on some program execution. Figure 3 shows several symbolic states at location L1 of the program `cohendiv`. SymInfer uses the technique described in S3.1 to collect symbolic states.

Finally, an *invariant* is a logical formula that always holds at a program location. For efficiency, invariant generation tools typically infer invariants under a certain template or form.

Definition 3.3 (Template-based Invariant). *An invariant under a template β_τ is a tuple $\langle L, \beta_\tau \rangle$ where L is a program point and β is a formula, which has the form τ and ranges over program variables in scope at L, that holds over all concrete or symbolic states of L.*

Section 2 shows invariants under different forms obtained by SymInfer for the example program, e.g., nonlinear equalities and octagonal linear inequalities.

```
input  : program $P$, location $L$, maxdepth $k$
output : symbolic states sstates
1 $\Pi, \sigma \leftarrow$ symexe($P, L, k$) //invoke symbolic execution
2 sstates $\leftarrow \Pi$
3 foreach variable $v \in \sigma$ do
4     sstates = sstates $\wedge (v \equiv \sigma(v))$
5 return sstates
```

Fig. 4: `getSymbolicstates`: calling a symbolic execution tool to obtain symbolic states

Concrete states, symbolic states, and invariants are different representations of properties at a program location. Concrete states describe program properties precisely, but there may be (infinitely) many to consider, and analyzing a smaller, finite subset of concrete states may lead to *spurious* invariants that dramatically underapproximate the set of program states. Program invariants overapproximate program states at L, but they generally have a compact form that can be leveraged to support understanding and reasoning about properties at L. Symbolic states serve as an intermediate representation between concrete states and invariants that might be inferred from the concrete states.

3.1 Obtaining Symbolic States

To obtain symbolic states, we modify the search process of a symbolic execution tool. We require that these tools produce underapproximating symbolic encodings of program states, which ensures that generated symbolic states only define feasible concrete states. First, we introduce a new method `vtrace` and insert a `vtrace` call at each location of interest in the program. Next, we intercept the search process of the symbolic execution tool whenever it enters a `vtrace` call. Most symbolic execution tools already maintain information such as location, path conditions Π, and variable mappings σ, and thus we just need to access and record this information. Thus, we obtain a symbolic L-state $\langle \sigma_L, \Pi_L \rangle$ whenever the program enters a `vtrace` call at some program location L and obtain a set of symbolic L-states if the program hits L multiple times (e.g., in a loop). There are potentially an infinite number of symbolic states at L, thus we adapt the symbolic execution tool to just return the symbolic L-states encountered during a search of a given depth k.

Figure 4 summarizes the process of invoking the symbolic execution tool to obtain symbolic states (line 1). SymInfer uses logical formulae to represent symbolic states and reuse these formulae for invariant checking and inference. For each symbolic L-state, we create the formula (lines 2 – 4)

$$\Pi_{\mathrm{L}} \wedge \bigwedge_{v \in \sigma} (v = \sigma_{\mathrm{L}}(v)).$$

For example, the formulae representing the symbolic state in the first row of Figure 3 is

$$(0 < y \wedge y \leq x) \wedge (q = 0 \wedge r = x \wedge a = 1 \wedge b = y)$$

This formula generalizes the first, $x = 15, y = 2, a = 1, b = 2, q = 0, r = 15$, and fourth, $x = 4, y = 1, a = 1, b = 1, q = 0, r = 4$, concrete states listed in Figure 1. Since each symbolic state represents a path leading to L, we can obtain a formula capturing multiple paths leading to L by taking the disjunction of formulae representing individual symbolic L-states.

The user of SymInfer can insert `vtrace` calls to multiple locations of interests and SymInfer will extract the appropriate set of symbolic states for those locations. Moreover, the user can specify subsets of variables in scope at L to `vtrace`, e.g., `vtrace(x,y,z)`, to obtain symbolic traces relevant to only those variables.

3.2 Using Symbolic States

Symbolic states can help invariant generation in many ways. We describe techniques using symbolic states to check and compute candidate invariants and to generate diverse concrete states.

As mentioned in S3.1, the number of symbolic states varies with the given symbolic execution depth. A low depth means few states. Few states will tend to encode a small set of concrete L-states, which limits verification and refutation power. Few states will also tend to solve verification condition faster. To address this cost-effectiveness tradeoff, rather than try to choose an optimal depth, our algorithm computes the lowest depth that yields symbolic states that change verification outcomes. In essence, the algorithm adaptively computes a good cost-effectiveness tradeoff for a given program, location of interest, and invariant.

3.2.1 Symbolic States as a "Verifier"

Figure 5 shows how we use symbolic states to `check`, or refute, a property. The algorithm is incremental and obtains symbolic states at a greater depth to increase the accuracy of verification.

The algorithm iterates with each iteration considering a different depth, k. The body of each iteration (lines 6 – 23) works as follows. For each iteration we use the function `getSymbolicStates`, which implements the technique described in S3.1, to generate the set of symbolic L-states reachable at depth less than or equal to k (line 7). Note that these states can be cached and reused for a given P and L.

We next create a verification condition (`vc`) by conjoining the symbolic states (i.e., the disjunction of individual symbolic states such as those in Figure 3) and the states to be blocked. We use these `block` states to avoid generating the same concrete states or counterexample inputs (when `check` returns a counterexample, we (e.g., the algorithm in Figure 8) use it to obtain concrete states, and then block the counterexample so that we do not generate it again). If the resulting formula implies a candidate property p then that candidate is consistent with the set of symbolic states. We use an SMT solver to check the negation of this implication.

The solver can return `sat` indicating that the property is not an invariant (lines 14 – 17). In this case, we query the solver for a model which represents a concrete state that is inconsistent with the proposed invariant. This counterexample state is saved so that the inference algorithm can search for invariants that are consistent with it. The solver can also return `unsat` indicating the property is a true invariant; at least as far as the algorithm can determine given the symbolic states at the current depth. Finally, the solver

```
input  : program $P$, location $L$, property $p$, clauses to block
output : proved status is_proved, counterexample cex
1  is_proved $\leftarrow$ unknown // is $p$ proved?
2  result, result$'$ $\leftarrow$ unknown, unknown
3  cex $\leftarrow \emptyset$
4  nochanges, nochanges$_{max}$ $\leftarrow 0$, NOCHANGES_MAX
5  $k, k_{max} \leftarrow d_{def}, d_{max}$ // default and max depth
6  while $k < k_{max}$ do
7      sstates $\leftarrow$ getSymbolicStates($P, L, k$)
8      vc $\leftarrow$ ($\bigvee$ sstates) $\wedge$ ($\neg$block)
9      result$'$ $\leftarrow$ checkSMT($\neg$(vc $\Rightarrow p$))
10     if result$'$ $\equiv$ result then
11         if nochanges $\equiv$ nochanges$_{max}$ then
12             break
13         nochanges $\leftarrow$ nochanges $+ 1$
14     if result$'$ $\equiv$ sat then
15         is_proved $\leftarrow$ False
16         cex $\leftarrow$ getModel()
17         break
18     else if result$'$ $\equiv$ unsat then
19         is_proved $\leftarrow$ True
20     else if result$'$ $\equiv$ unknown then
21         is_proved $\leftarrow$ unknown
22     result $\leftarrow$ result$'$
23     $k \leftarrow k + 1$
24 return is_proved, cex
```

Fig. 5: check: check a candidate property using symbolic states.

can also return unknown, indicating it cannot determine whether the given property is true or false.

For the latter two cases, we increment the depth and explore a larger set of symbolic states generated from a deeper symbolic execution. Lines 9 – 12 work to determine when increasing the depth does not influence the verification. In essence, they check if the same result is computed at several consecutive adjacent depths and if so, they return (line 12).

Correctness and Termination: This check function guarantees that refuted candidates are *not* invariant because running the program with the generated counterexample inputs would violate these candidates. However, results only hold over the symbolic states obtained up to the considered depth and might not hold in general. The function terminates because the loop executes for at most $d_{max} - d_{def}$ iterations.

3.2.2 Symbolic States as an "Optimizer"

Figure 6 shows how we use symbolic states to compute an upper bound of a term. The algorithm directly computes an inequality of the form $t \leq c$, where t is a term (e.g., $x - y$) and c is some integer value. The approach leverages the power of modern constraint solvers, which, in addition to finding satisfiability assignments, can find *optimal* assignments with respect to objective constraints using linear optimization techniques [36].

This algorithm, similarly to the one in Figure 5, incrementally obtains symbolic states at a greater depth to improve accuracy. The main difference is that instead of

```
input  : program $P$, location $L$, term $t$
output : upper bound value of $t$ within a predefined v_max range
1  $v$, result, result$'$ $\leftarrow \infty, \infty, \infty$
2  nochanges, nochanges$_{max}$ $\leftarrow 0$, NOCHANGES_MAX
3  $k, k_{max} \leftarrow d_{def}, d_{max}$ // default and max depth
4  while $k < k_{max}$ do
5      sstates $\leftarrow$ getSymbolicStates($P, L, k$)
6      result$'$ $\leftarrow$ MAX($\bigvee$ sstates, $t$)
7      if result$'$ $\equiv$ result then
8          if nochanges $\equiv$ nochanges$_{max}$ then
9              break
10         nochanges $\leftarrow$ nochanges $+ 1$
11     if result$'$ $\equiv$ unknown then
12         $v \leftarrow \infty$
13     else if result$'$ $\leq v_{max}$ then
14         $v \leftarrow$ result$'$
15     else if result$'$ $> v_{max}$ then
16         $v \leftarrow \infty$
17         break
18     result $\leftarrow$ result$'$
19     $k \leftarrow k + 1$
20 return $v$
```

Fig. 6: optimize: find the upper bound value of a term from symbolic states.

checking satisfiability of a formula, we use the "optimizer" component of the solver to find the maximum value of the given term from symbolic states (line 6).

The solver returns two possible values: a concrete integer value or unknown. If the returned value is less than or equal to a parameterized v_{max} bound, it is saved and the algorithm repeats the process using a higher depth. Otherwise, if the solver returns unknown or a value larger than the bound, then we save the result as ∞, which produces a trivial invariant $t \leq \infty$ that would be discarded. We also break out of the loop in the latter case because a result larger than some bound at some depth k will remain larger than that bound at any depth larger than k.

Correctness and Termination: Similarly to the check function in Figure 5, optimize guarantees that discarded results are those with bounds greater than the considered bound (or that cannot be determined). However, the results only hold over symbolic states at a given depth and might not hold in general. The function terminates because the loop executes for at most $d_{max} - d_{def}$ iterations.

3.2.3 Bootstrapping DIG with Concrete States

SymInfer generates candidate invariants using existing concrete state-based invariant inference techniques like DIG. In this application, we only need a small number of concrete states to bootstrap the algorithms to generate a diverse set of candidate invariants since symbolic states will be used to refute spurious invariants. In prior work [27], [29], fuzzing was used to generate inputs and that could be used here as well, but we instead exploit symbolic states which allows us to force diversity among generated concrete states, e.g., one per symbolic state.

```
input  : program $P$, location $L$, symbolic depth $k$,
         number of requested concrete states $n$
output : set of concrete states cstates, clauses to block
1  block ← false
2  cstates ← $\emptyset$
3  sstates ← getSymbolicStates($P, L, k$)
4  inputVars ← getInputs($P$)
5  foreach $s \in$ sstates do
6      if checkSMT($s.\Pi$) then
7          model ← getModel()
8          cstates ← cstates ∪ ($L$, eval($s.\sigma$, model))
9          block ← block ∨ ( $\bigwedge_{v \in \text{inputVars}}$ $v \equiv$ model[$v$])
10 while |cstates| < $n \wedge$ |sstates| > 0 do
11     $s$ ← choose(sstates)
12     if checkSMT($s.\Pi \wedge \neg$block) then
13         model ← getModel()
14         cstates ← cstates ∪ ($L$, eval($s.\sigma$, model))
15         block ← block ∨ ( $\bigwedge_{v \in \text{inputVars}}$ $v \equiv$ model[$v$])
16     else
17         sstates ← sstates − {$s$}
18 return cstates, block
```

Fig. 7: getConcreteStates: generate concrete states from symbolic states.

Figure 7 shows how we use symbolic states to generate a diverse set of concrete states—at least one for each symbolic state. The loop on line 5 considers each such state, checks the satisfiability of the states path condition Π and then extracts the model from the solver. Next, we bind concrete values to variables in the model to obtain a concrete state, which is then accumulated. Then we block the model to avoid generating the same concrete state in the future.

The loop on line 10 generates additional concrete states up to the requested number, n. We randomly pick a symbolic state and then call an SMT solver to generate a solution that has not already been computed. When a solution is found, we use the same processing as in lines 7–8 to create a new concrete state; otherwise, we block that symbolic state as in line 17 and continue. Note that it is possible that we cannot obtain exactly n concrete states as requested and therefore cannot perform intended tasks (e.g., for equation solving, as discussed in S4.1.1).

Correctness and Termination: SymInfer generates symbolic states representing all possible paths up to a given bound, so generated invariants will be correct with respect to the paths found in that bound. Function `getConcreteStates` terminates because (1) the depth bounded symbolic execution (line 3) returns a finite set of states which guarantees termination of the loop on line 5, and (2) the loop on line 10 terminates, since at each iteration either the set of concrete states computed increases monotonically (newly added states cannot carry over from prior iterations since prior states are explicitly blocked from the SMT call (lines 12 and 15)) or the set of symbolic states decreases monotonically (depth bounded symbolic execution produces a finite set of symbolic states).

3.3 Benefits of Symbolic States

Symbolic states are useful as a basis for efficient inference of invariants for several reasons:

Symbolic states are expressive: Dynamic analysis has to observe many concrete states to obtain useful results. Many of those states may be equivalent from a symbolic perspective because a symbolic state can encode a potentially infinite set of concrete states. SymInfer exploits this expressive power to infer and refute candidate invariants from a huge set of concrete states by processing a single symbolic state.

Symbolic states are relational: Symbolic states encode the values of program variables as expressions over free-variables capturing program inputs. This permits relationships between variables to be gleaned from the state.

Symbolic states can be reused: Invariant generation has to infer or refute candidate invariants relative to the set of observed concrete states. This can grow in cost as the product of the number of candidates and the number of concrete states. A disjunctive encoding of observed symbolic states can be constructed once and reused for each of the candidate invariants, which can lead to performance improvement.

Symbolic states can be used for optimization: We can use a constraint solver to check guessed invariants from symbolic states. However, for certain types of invariants, we can eschew guessing and directly use the solver to compute invariants. For example, instead of checking if $x + y \leq 10$ is an invariant, we just query the solver to find the least upper bound of the term $x + y$ from symbolic states. Thus, instead of performing multiple guesses and checks, we can obtain the desired invariant with a single call to the solver.

Symbolic states form a sufficiency test: The diversity of symbolic states found during depth-bounded symbolic execution combined with the expressive power of each of those states provides a rich basis for inferring strong invariants. We have observed that for many programs a sufficiently rich set of observed states for invariant inference will be found at relatively shallow depth. That is, the invariants generated and consistent with symbolic states at depth 10 are the same as those at depth 11. Consequently, we employ an adaptive and incremental approach that increases depth only when new states lead to changes in candidate invariants.

4 The SymInfer Approach

SymInfer uses symbolic states to generate equality and inequality forms of numerical invariants. First, we present a CEGIR technique that integrates a dynamic inference algorithm from DIG, which generates (nonlinear) *equality* invariants from concrete states, with symbolic states to check and refine invariants. Then, we present an optimization technique to compute *inequality* invariants directly from symbolic states.

4.1 Nonlinear Equalities

At a high level, we treat concrete state values as points in Euclidean space and compute geometric shapes enclosing these points. For example, the values of the two variables x, y are points in the (x, y)-plane. We then can determine if these points lie on a line, represented by a linear equation of the form $c_0 + c_1x + c_2y = 0$.

```
input  : program $P$, location $L$, degree $d$, symbolic
         depth $k$
output : nonlinear equalities up to deg $d$ at $L$
cstates ← $\emptyset$
invs ← $\emptyset$
vars ← getVars($P, L$)
terms ← createTerms(vars, $d$)
cstates, block ← getConcretStates($P, L, k$, |terms|)
candidates ← inferEqts(terms, cstates)
while candidates $\neq \emptyset$ do
    cexs ← $\emptyset$
    foreach $p \in$ candidates do
        is_proved, newCexs ← check($P, L, p$, block)
        cexs ← cexs ∪ newCexs
        if is_proved then invs ← invs ∪ $\{p\}$
    if cexs $\equiv \emptyset$ then break
    block ← block ∪ cexs
    cstates ← cstates ∪ cexs
    newCandidates ← inferEqts(terms, cstates)
    candidates ← newCandidates − invs
return invs
```

Fig. 8: Algorithm for finding (potentially nonlinear) polynomial equalities. The `check` function described in S3.2.1 is used to validate invariants.

Thus, we treat equalities as unbounded geometric shapes, e.g., lines and planes, to obtain a set or conjunction of linear equalities over program variables. To support nonlinear equalities, we create *terms* to represent nonlinear information from the given variables up to a certain degree. For example, the set of 10 terms $\{1, r, y, a, ry, ra, ya, r^2, y^2, a^2\}$ consists of all monomials up to degree 2 over the variables $\{r, y, a\}$. In total, we enumerate $\binom{|V|+d}{d}$ monomial terms over n variables up to the given degree d.

SymInfer uses the equation solving technique in DIG to find equality invariants over these terms by using concrete states. First, we form an equation *template*

$$c_1t_1 + c_2t_2 \cdots + c_nt_n = 0 \tag{1}$$

where t_i are the generated terms and c_i are real-valued unknowns to be solved for. Next, we instantiate the template with concrete state values to obtain concrete and linear equations. For example, instantiating the template with the concrete state $r = 3, y = 2, a = 6$ produces the equation $c_1 + 3c_2 \cdots + 36c_n = 0$. We then solve these equations for the unknown coefficients using a standard linear equation solving technique such as Gauss-Jordan elimination [37]. Finally, we combine solutions for the unknowns (if found) with the template to obtain equality relations.

4.1.1 *Inferring Equalities*

Figure 8 defines our CEGIR algorithm for computing nonlinear equality invariants. It consists of two phases: an initial invariant candidate generation phase and then an iterative invariant refutation and refinement phase.

Lines 4 – 6 define the initial generation phase. We first create terms to represent nonlinear polynomials. Because solving for n unknowns requires at least n unique equations, we use symbolic states as described in S3.2.3 to generate a sufficient set of concrete states.

The initial candidate set of invariants is iteratively refined on lines 7 – 17. The algorithm refutes or confirms them using symbolic states as described in S3.2.1. Any property that is proven to hold is recorded in `invs` and counterexample states, `cexs`, are accumulated across the set of properties. Generated counterexample states are also blocked so that they are not generated again.

If no property generated counterexample states, then the algorithm terminates returning the verified invariants. The counterexamples are added to the set of states that are used to infer new candidate invariants; this ensures that new invariants will be consistent with the counterexample states (line 15). These new results may include some already proven invariants, so we remove those from the set of candidates considered in the next round of refinement.

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

For the 6 variables $\{a, b, q, r, x, y\}$ at L1, together with $d = 2$, SymInfer generates 28 terms $\{1, a, \dots, y^2\}$. SymInfer uses these terms to form the template $c_1 + c_2a + \dots c_{28}y^2 = 0$ with 28 unknown coefficients c_i. SymInfer then uses the obtained symbolic states to generate concrete states such as those given in Figure 1 to form (at least) 28 linear equations. From this set of initial equations SymInfer extracts 6 equalities.

Now, SymInfer iteratively refines the inferred invariants. In iteration #1, SymInfer cannot refute 2 of these candidates $x = qy + r, b = ya$ (which are actually true invariants) and thus saves these as invariants. SymInfer finds counterexamples for the other 4 equalities[2], and creates new equations from the counterexamples. SymInfer next combines the old and new equations and solves them to obtain 4 candidates, 2 of which are the already saved ones. In iteration #2, SymInfer obtains counterexamples for the 2 new candidates. With the help of the new counterexamples, SymInfer generates 3 candidates, 2 of which are the saved ones. In iteration #3, SymInfer obtains counterexamples disproving the remaining candidate and again uses the new counterexamples to generate new candidates. This time SymInfer only finds the two saved invariants $x = qy + r, b = ya$ and thus stops.

Correctness and Termination: The algorithm returns (potentially nonlinear) equalities that are correct up to the considered symbolic depth (since it uses the `check` function in S3.2.1 to check invariants with respect to the given depth). The algorithm terminates because linear equation solving provides a solution space containing all possible coefficients for an equality invariant and each added counterexample input decreases the dimension of the solution space by one. Thus if we keep finding counterexamples, then the solution space will keep decreasing, and eventually, that will stop when the dimension of the solution space becomes zero (i.e., no equalities found). The full termination proof is provided in the supplementary appendix.

2. Spurious results often have many terms and large coefficients, e.g., $190qr + 10r^2 - 10x^2 - 551q - 2929r + 2929x = 0$

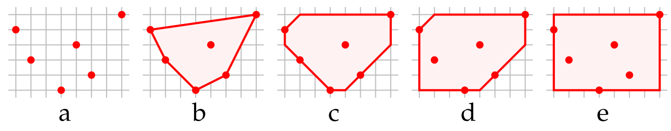


Fig. 9: (a) A set of 6 points in 2D and its approximation using the (b) polyhedral, (c) octagonal, (d) zone, and (e) interval shapes, represented by the conjunctions of inequalities of the forms $c_1v_1 + c_2v_2 \leq c$, $\pm v_1 \pm v_2 \leq c$, $v_1 - v_2 \leq c$, and $\pm v \leq c$, respectively.

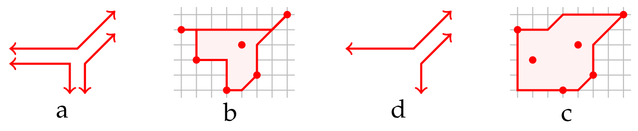


Fig. 10: (a) Three possible shapes of a max-plus line segment: $\max(c_1 + v_1, c_2) \leq v_2$ (top), $\max(c_1 + v_2, c_2) \leq v_1$ (right), $\max(c_1 + v_1, c_2 + v_2) \leq c_3$ (left); (b) approximation of the points in Figure 9a using a max-plus polygon; (c) two possible line segment of weak max-plus; and (d) approximation of points using a weak max-plus polygon.

Insufficient Traces: This algorithm might fail to produce equality invariants due to insufficient traces. More specifically, if the `getConcreteStates` call on line 5 fails to obtain |`terms`| concrete states, then we cannot form the necessary equations to solve for |`terms`| unknowns and thus the `inferEqts` call on line 6 produces no results.

```
if(x==1&&y==2){
  [L]
  ..
}
```

The example on the right illustrates the situation. The program has exactly 1 concrete L-state (i.e., $\{(x = 1 \land y = 2)\}$), thus we can form only a single equation and cannot solve for 2 or more unknowns. While it is possible to mitigate the problem in specific cases (e.g., equation solving is not even needed here), it is challenging for more general cases (e.g., if we do not have enough traces to solve n unknowns, do we consider only some subsets of those unknowns? and if so, which subsets to consider?).

Fortunately, this problem, in which every considered variable at a location has a fixed value, rarely happens. In the above program, if L has another variable z that is not fixed to a value, then we can generate a large number of concrete traces, e.g., $\{(x = 1, y = 2, z = 0), (x = 1, y = 2, z = 1), (x = 1, y = 2, z = 2), \dots\}$, and form equations to obtain the invariants $x = 1$ and $y = 2$. Thus, as long as some of the considered variables are "free", we can generate sufficient traces for equation solving.

Note that for this example, even if we cannot obtain the equalities $x = 1 \land y = 2$ due to insufficient traces, we can still infer inequalities that are exactly equivalent, i.e., $x \leq 1 \land 1 \leq x \land y \leq 2 \land 2 \leq y$, using the inference technique described next in S4.2.

4.2 Linear Inequalities

SymInfer supports several forms of inequality invariants. Below we describe inequalities using different geometric shapes and then present a linear programming approach that computes inequalities directly from symbolic states. SymInfer supports both linear (default) and nonlinear (configurable by the user) inequalities.

Figure 9 shows several types of convex polygons (polyhedra in 2D) that represent different forms of inequalities over two variables. Figure 9a shows a set of points created from concrete states. Figures 9b, 9c, 9d, and 9e approximate the region enclosing these points using the polyhedral, octagonal, zone, and interval shapes that are represented by *conjunctions of inequalities* of different forms as shown in Figure 9. Note that these forms of relations are sorted in decreasing order of expressive power and computational cost. For example, while interval relations are less expressive than zone relations, computing an interval, i.e., the upper and lower bounds of a variable, is much cheaper than computing the convex hull of a zone.

4.2.1 Octagonal Relations

While inequalities represented by a general polyhedron are expressive, computing a polyhedron is expensive (exponential time in the number of dimensions or variables) and often produces many complex and spurious invariants [29]. SymInfer thus focuses on *octagonal* invariants, whose shape and inequality form are shown in Figure 9c. Octagonal invariants can be computed efficiently from concrete states (linear time complexity [29]) and are also relatively expressive (e.g., can represent zone and interval inequalities as shown in Figure 9). Thus, the computation of octagonal relations also produces zone and interval relations for free. By balancing expressive power with computational cost, octagonal invariants are especially useful in practice for detecting bugs in flight-control software, and performing array bound and memory leak checks [22], [38].

The edges of an octagon are represented by a conjunction of eight inequalities of the form

$$c_1v_1 + c_2v_2 \leq k, \tag{2}$$

where v_1, v_2 are variables, $c_1, c_2 \in \{-1, 0, 1\}$ are coefficients, and k is a real-valued constant. For example, from the concrete states in Figure 1, we can build an octagon represented by inequalities such as $4 \leq x \leq 15$ and $3 \leq x - y \leq 13$.

4.2.2 Max- and Min-plus Relations

The convex polygons shown in Figure 9 represent conjunctions, but not disjunctions, of numerical relations. *Disjunctive invariants*, which represent the semantics of branching, are more difficult to analyze, but are crucial to many programs. For example, after `if (p) {a=1;} else {a=2;}`, neither $a = 1$ nor $a = 2$ is an invariant, but

$$(p \land a = 1) \lor (\neg p \land a = 2)$$

is a disjunctive invariant.

SymInfer supports a form of disjunctive invariants by using a special type of nonconvex polyhedra in the *max-plus* (max) algebra [23], [24], [39]. Briefly, max inequalities are analogous to polyhedra inequalities, but use $(\max, +)$ instead of the $(+, \times)$ of standard arithmetic. These operators allow max relations to form shapes that are nonconvex in the classical sense. For example, the max relation $\max(x, y) \leq 5$, which is equivalent to $(x \geq y) \Rightarrow x \leq 5 \land (x < y) \Rightarrow y \leq 5$, can be simplified to $x \leq 5 \lor y \leq 5$—a nonconvex area.

Figure 10a shows the three possible shapes of a max line segment in 2D. Figure 10b depicts a max polygon represented by a set of four lines connecting the points shown in Figure 9a.

Similarly to polyhedra, computing a general max convex hull is expensive and can result in many spurious inequalities. We thus focus on a "weaker" form of max invariants introduced in [28] that retains much of the general forms expressive power, but avoids the high computational cost.

Weak max invariants have the form:

$$\begin{aligned} \max(c_0, c_1+v_1,\dots,c_k+v_k) &\le v_j + d \\ v_j + d &\le \max(c_0, c_1+v_1,\dots,c_k+v_k), \end{aligned} \tag{3}$$

where v_i are program variables, $c_i \in \{0, -\infty\}$, d is a real numbers or $-\infty$, and k is a constant, e.g., $k = 2$ in 2D.

Weak max relations are thus a strict subset of general max relations. For example, the weak max form cannot represent general max relations like $\max(x+7, y) \le z$ or $\max(x, y) \le \max(z, w)$, but it does support zone relations like $x - y \le 10$ and $x = y$ and disjunctive relations like $\max(x, y) \le z$ and $\max(x, 0) \le y + 7$. Geometrically, weak max polyhedra are a restricted kind of max polyhedra in which a right angle cannot occur because their formula, $\max(x, y) \le 0$, is inexpressible using the weak max form. Figure 10 compares general and weak max polyhedra in 2D.

The advantage of these restrictions is that they admit a straightforward and efficient algorithm to compute the bounded weak max polyhedron over a set of finite points representing concrete states in k dimensions. The algorithm first enumerates all possible weak relations over k variables and then finds the unknown parameter d in each relation from the given points. The resulting set of relations represent the weak max polyhedron enclosing the points.

Dually, SymInfer also computes weak min invariants, whose form is similar to the one in Eq 3 but with `min` instead of `max`:

$$\begin{aligned} \min(c_0, c_1+v_1,\dots,c_k+v_k) &\le v_j + d, \\ v_j + d &\le \min(c_0, c_1+v_1,\dots,c_k+v_k). \end{aligned} \tag{4}$$

The algorithm for computing these invariants over concrete states is similar to the one for weak max invariants described above.

4.2.3 Inferring Inequalities

SymInfer uses a single algorithm, shown in Figure 11, to infer octagonal and (weak) max and min invariants. The key idea is to compute the invariant $t \le k$ where the term t represents different forms of octagonal and max/min inequalities and the constant k is the upper bound of t.

The algorithm consists of two phases. First, we invoke `createTerms` to enumerate terms over program variables (lines 3– 4). For octagonal inequalities, each term t involves two variables so that $t \le k$ is an octagonal constraint of the form in Eq 2. Given n variables, we create $\binom{n}{2}$ variable pairs, and for each pair x, y, obtain 8 terms of the octagonal form in Eq 2 in which x and y are associated with one of the 3 coefficients $\{-1, 0, 1\}$, e.g., $-1 \times x + 1 \times y$ gives the term $-x + y$ (and our goal is to find the upperbound k to form the octagonal inequality $-x + y \le k$). In total, we generate $\binom{n}{2} \times (2^3 - 1)$ octagonal terms (we exclude the term $0 \times x + 0 \times y$, which simplifies to 0).

```
input  : program $P$, location $L$, symbolic depth $k$
output : octagonal or max/min inequalities at $L$
1 invs ← $\emptyset$
2 vars ← getVars($P, L$)
3 terms$_{oct}$ ← createTerms$_{ieq}$(vars, $d = 1$, coefs =
      $(-1, 0, 1)$, subsetSize $= 2$)
4 terms$_{minmax}$ ← createTerms$_{minmax}$(vars, $d$)
5 foreach $t \in$ terms$_{oct}$ $\cup$ terms$_{minmax}$ do
6     $k$ ← optimize($P, L, t$)
7     if $k \neq \infty$ then
8         invs ← invs $\cup$ ($t \le k$)
9 return invs
```

Fig. 11: Algorithm for finding octagonal and max/min-plus inequalities. The `optimize` function described in S3.2.2 is used to find upper bound values of terms.

For max inequalities, t involves combinations of variables of the weak form in Eq 3. Given n variables, we create $\binom{n}{n-1}$ tuples of variables, and for each tuple of k variables, obtain 2^{k-1} terms of the form in Eq 3 in which each of the $k-1$ variables is associated with one of the 2 coefficients $\{0, -\infty\}$. For example, the invariant $\max(0, -\infty + x, 0 + y) \le z + d$ generates the term $\max(0, y) - z$ that we optimize to compute an upperbound k that forms the max inequality $\max(0, y) - z \le k$. In total, we generate $\binom{n}{n-1} \times 2^{k-1}$ max terms. We also do the same to obtain min terms of the form in Eq 4.

Then, we use the `optimize` function described in S3.2.2 to compute the smallest upper bound k of each term t from symbolic states (lines 5– 8). If k is found, we obtain the candidate invariant $t \le k$; otherwise, we discard the relation $t \le k$.

We also use the same approach to find the lower bound of a term t (i.e., $k \le t$, which is also an octagonal inequality) by computing the upper bound of the term $-t$. For example, if $t \in \{-2, 3, 7\}$ then we have $t \le 7$ and $-t \le 2$, which is equivalent to $-2 \le t$. The function `createTerms` generates both terms t and $-t$.

Correctness and Termination: This algorithm returns inequalities that are correct up to the considered symbolic depth (since it uses `optimize` to obtain the upper bounds with respect to the given depth). The algorithm terminates because the enumerated terms are finite and each `optimize` call terminates.

Note that the previous version of SymInfer [31] computes the upper bound for a term using a CEGIR approach, which repeatedly invokes symbolic states to guess and tighten the candidate upper bounds. This approach, which invokes the solver multiple times to check guessed results, is slower than the described optimization-based algorithm, which invokes the solver once to find the upper bound. However, the CEGIR algorithm, provided in the supplementary appendix, might be useful when computing the upper bound directly from symbolic states is not possible (e.g., when symbolic states or terms involve complex arithmetic not supported by the constraint solver's optimizer).

4.2.4 Nonlinear Inequalities

By default, SymInfer generates the described octagonal and max/min-plus relations. However, the user can easily configure SymInfer to generate more expressive inequalities by either specifying a command line argument and setting environment variables of SymInfer.

For example, by default function `createTerms`$_{ieqs}$ in Figure 11 generates octagonal terms such as $x, x - y$. These terms are linear (degree d = 1), have coefficients -1, 0, 1, and involve at most two variables (`subsetSize=2`). SymInfer allows the user to change these parameters to generate more expressive invariants involving more variables (e.g., using `subsetSize=3` would generate invariants such as $x + y - z \leq s$), having larger coefficients (changing the coefficients range to (-5,5) would produce invariants such as $2x - 5y \leq z$), and with higher degrees (setting $d = 2$ would generate nonlinear inequalities such as $x^2 + z + w \leq 7$). The user also can introduce new terms to represent desired information (e.g., we can obtain relations involving exponential properties such as $2^x \leq y$ by representing 2^x with a new term). The trade-off is that SymInfer would take longer to generate and analyze these richer invariants.

4.3 Post-Processing

Depending on the number of variables and form of invariants, SymInfer could generate many invariants (e.g., each octagonal and max/min term can produce an invariant candidate). Reporting too many invariants, even if they are all valid, would be a burden to the user and reduce the general effectiveness of the tool. Thus, SymInfer uses a post-processing step to reduce the number of reported invariants.

The post-processing consist of two parts; both aim to reduce the number of reported invariants. The first part simply checks generated invariants against all cached concrete states and removes violated ones. This part is efficient (we simply instantiate and check candidate relations with concrete values), but removes few results (because most generated invariants are already valid).

The second part attempts to remove redundant invariants, i.e., from a set of candidate invariants, we extract a subset of *independent* relations such that every member of the set is not implied by other relations in that set. This part is time-consuming because we essentially invoke the solver to check every candidate invariant, but is effective in reducing invariants because many invariants can be obtained by the combination of others. Our experiences show that this part can effectively reduce many inequalities to just a few strongest and relevant ones–making it much easier for the user to analyze and use the reported results.

5 Implementation and Evaluation

SymInfer is implemented in Python/SAGE [40]. The tool takes as input a program with marked target locations, i.e., using the `vtrace` method discussed in S3.1, and generates invariants at those locations.

Currently, SymInfer supports programs written in Java, (Java) bytecode, and C. SymInfer uses the Z3 SMT solver [41] to check and produce models representing counterexamples. We also use Z3 to identify and remove redundant invariants in post-processing. We use the same backend algorithms to infer and analyze invariants, but call different symbolic execution frontend tools to obtain symbolic states (Symbolic PathFinder (SPF) [25] for Java and bytecode programs and CIVL [26] for C programs). Our experiments focus on evaluating SymInfer on Java programs, and we discuss SymInfer's performance on C programs in S5.6.

SymInfer leverages the increasingly popular and affordable multicore architecture. The tool performs many independent tasks in parallel, e.g., running symbolic execution at different depths, generating invariants at different locations, computing upper bounds for terms, and checking candidate invariants. Parallel processing is crucial to the performance of SymInfer as it allows the tool to process and analyze thousands of candidate invariants at multiple program locations simultaneously.

To evaluate SymInfer we consider 6 research questions:

- **RQ1:** How well does SymInfer infer nonlinear invariants describing complex program semantics and correctness conditions?
- **RQ2:** How well does SymInfer generate expressive invariants to capture program runtime complexity?
- **RQ3:** How well does SymInfer infer min and max-plus invariants to prove disjunctive invariants?
- **RQ4:** How does SymInfer perform on programs involving non-trivial properties not directly supported by SymInfer?
- **RQ5:** How does the depth of symbolic states influence the ability of SymInfer to infer invariants?
- **RQ6:** How does SymInfer compare to other invariant generation tools?

To investigate these questions, we used 4 benchmark suites consist of 108 Java programs described in detail in the following subsections. These programs come with known or documented invariants. To compare the invariants inferred by SymInfer, we manually checked consistency with the documented invariants and we encoded documented invariants and used Z3 to determine that the inferred results imply them.

For our experiments, we use SymInfer's default settings to infer nonlinear equalities, octagonal and max/min relations. For equalities, SymInfer uses the default setting DIG [29] that limits the number of generated terms up to 200. This allows us, for example, to infer equalities up to degree 5 for a program with 4 variables and up to degree 2 for a program with 12 variables). For octagonal and max/min inequalities, we consider upper and lower bounds (the `v_max` value in Figure 6) within the range $[-20, 20]$; we rarely observe inequalities with large bounds. SymInfer can either choose random values in a range, $[-300, 300]$ by default, for bootstrapping, or use the algorithm in Figure 7. Our experience shows that we do not need very large input values to generate precise invariants. We start SPF with depth 7 and CIVL with depth 20; these seem to be good default search depths for almost all our test programs. All these parameters can be changed by SymInfer's user; we chose these default values based on our experience.

SymInfer has several sources of randomness, e.g., the generation of concrete states from the Z3 and the collection of symbolic states symbolic execution tools. In our experiments, we ran SymInfer 5 times on each program and report

TABLE 1: Experimental results for NLA programs. ✓: produce results that match or imply documented invariants. *: require minor modifications to work.

			Invs			Time(s)		
Prog	**L**	**V**	**T**	**E, I, M**	**NL(d)**	**T**	**Exp**	**Correct**
Bresenham	1	5	5	1,2,2	1(2)	78.5	64.5 S	✓
CohenCu	1	5	6	3,2,1	2(2)	16.0	12.5 E	✓
CohenDiv	2	6	20	4,15,1	4(2)	84.7	61.9 E	✓
Dijkstra	2	5	19	7,10,2	4(3)	98.2	76.4 E	✓
DivBin	2	5	15	3,8,4	1(2)	47.3	32.9 S	✓
Egcd	1	8	11	3,8,0	3(2)	104.7	67.6 S	✓
Egcd2	2	10	60	5,25,30	5(2)	165.2	71.7 R	✓
Egcd3	3	12	92	9,42,41	9(2)	406.0	220.4 R	✓
Fermat1	3	5	27	3,8,16	3(2)	241.8	110.5 M	✓
Fermat2	1	5	9	1,2,6	1(2)	310.3	190.2 M	✓
Freire1*	1	3	4	1,1,2	1(2)	3.8	1.8 E	✓
Freire2*	2	4	4	4,0,0	4(2)	6.4	3.0 E	✓
Geo1	1	4	7	1,6,0	1(2)	11.1	4.3 M	✓
Geo2	1	4	8	1,6,1	1(2)	15.4	4.3 M	✓
Geo3	1	5	10	1,7,2	1(3)	53.2	34.4 E	✓
Hard	2	6	21	5,11,5	3(2)	63.8	39.5 S	✓
Knuth*	1	8	13	4,5,4	4(3)	197.7	82.6 M	✓
Lcm1	3	6	32	4,21,7	4(2)	96.8	44.2 S	✓
Lcm2	1	6	9	1,6,2	1(2)	96.5	73.3 S	✓
MannaDiv	1	5	7	1,6,0	1(2)	202.9	195.5 E	✓
Prod4br	1	6	9	1,6,2	1(3)	69.6	31.4 E	✓
ProdBin	1	5	7	1,6,0	1(2)	75.3	35.4 S	✓
Ps2	1	3	4	1,3,0	1(2)	3.4	1.5 E	✓
Ps3	1	3	4	1,3,0	1(3)	3.4	1.5 E	✓
Ps4	1	3	4	1,3,0	1(4)	3.3	1.4 E	✓
Ps5	1	3	4	1,3,0	1(5)	3.7	1.7 E	✓
Ps6	1	3	4	1,3,0	1(6)	3.9	1.8 E	✓
Sqrt1	1	4	6	2,4,0	1(2)	4.6	1.8 E	✓

the median results (e.g., the median results of the runtimes and number of invariants collected over 5 runs).

The experiments reported were run on a 64-core AMD CPU 4 GHZ Linux system with 64 GB of RAM. SymInfer and all experimental benchmarks and results are available at https://github.com/unsat/dig/.

5.1 RQ1: Programs With Nonlinear Invariants

In this experiment, we use programs from the NLA testsuite [29] in the SV-COMP benchmark [42]. This testsuite consists of 28 programs implementing mathematical functions such as `intdiv`, `gcd`, `lcm`, and `power sum`. Although these programs are relatively small (under 50 LoCs), they contain nontrivial structures such as nested loops and nonlinear invariant properties. To the best of our knowledge, NLA is the largest benchmark of programs containing nonlinear arithmetic. Many of these programs have also been used to evaluate other numerical invariant systems [16], [31], [43], [44].

These programs come with known program invariants at various program locations (e.g., mostly nonlinear equalities for loop invariants). For this experiment, we evaluate SymInfer by finding invariants at these locations and comparing them with known invariants.

Results

Table 1 presents the results of SymInfer for the 28 NLA programs. Columns **L** and **V** show the number of locations where we obtain invariants and the number of variables at the location that has the largest number of variables, respectively. The **Invs** group shows the total number of discovered invariants (**T**) and from those the number of equalities (**E**), octagonal inequalities (**I**), min and max-plus inequalities (**M**), and nonlinear **NL** (equality) invariants and the highest degree (**D**) among those. The **Time** group shows the total time (**T**) in seconds. This time includes subtasks such as symbolic execution (S), equation solving (E), upper bound computation (M), and removing redundant results (R). The **Exp** column lists the time for the most expensive sub-task and indicates that task. Column **Correct** shows if the obtained results match or imply the known invariants. We also modified three programs (indicated with *) as they contain external calls and floating-point values that SymInfer currently does not support (details given below).

For all 28 programs, SymInfer generated correct invariants that match or imply the known results. In most cases, the discovered invariants matched the known ones exactly. Occasionally, we obtained results that are equivalent or imply the known results. For example, for some runs of `Sqrt1` we found the documented equalities $t = 2a + 1$ and $s = (a + 1)^2$, and for other runs we obtained $t = 2a + 1$ and $t^2 - 4s + 2t = -1$, which are equivalent to $s = (a + 1)^2$ by replacing t with $2a + 1$.

SymInfer also discovered undocumented invariants. For example, for `Egcd1`, which implements an extended GCD algorithm, DIG identifies three equalities for loop invariants: $x = ai + bj, y = ak + bm$, and $1 = im - jk$. The first two are documented invariants that assert the computation and preservation of the Bézout identity in the loop[3]. The third relation is a valid, but undocumented invariant, revealing a potentially useful implementation detail: the product im is exactly 1 more than the product jk whenever the program reaches location L. Also, as shown in S2.1, for `CohenDiv` SymInfer generated undocumented but useful inequalities such as $r \geq 0$, $r \leq x$, and $r \leq y - 1$ which state that the remain r is non-negative, is at most the dividend x, but is strictly less than the divisor y. Our experience shows that SymInfer is capable of generating many invariants that are unexpected yet correct and useful.

We had to modify three programs `Freire1`, `Freire2`, `Knuth` to work with SymInfer. We changed the floating point values used in `Freire1`, `Freire2` to integers because SymInfer currently does not support floating point arithmetic. We also remove the external library `sqrt` call from `Knuth` (by replacing `x == Math.sqrt(y)` with `x*x == y`) because SPF cannot obtain symbolic states from unknown functions.

From Table 1, we see that the number of invariants, especially inequalities, obtained at a location is largely dependent on the number of variables (i.e., the generated terms over these variables). Programs such as `Egcd2`, `Egcd3`, `LCM1` have more invariants because multiple locations are considered and also each location contains many variables. Nonetheless, we consider the final number of invariants reasonable, especially the stronger nonlinear equalities, and thus can be directly presented and analyzed by the user.

The runtime of SymInfer is largely dependent on the number of variables. While generally taking less than 2 minutes, for some programs the inference process can take

3. An extended GCD algorithm takes as input a pair of integers (a, b) and, in addition to computing the gcd of a, b, also produces two integers i, j satisfying the Bézout identity $x = ai + bj$

TABLE 2: SymInfer's results for computing programs' complexities. ✓: generates the expected bounds. ✓✓: obtains more precise bounds than reported results. *: require minor modifications.

Prog	V	T	E,I,M	NL(D)	Time(s)	Correct
cav09_fig1a	2	1	1,0,0	1(2)	5.7	✓
cav09_fig1d	2	1	1,0,0	1(2)	5.8	✓
cav09_fig2d	3	4	1,3,0	1(2)	23.8	✓
cav09_fig3a	2	3	1,1,1	1(2)	3.5	✓
cav09_fig5b	5	7	2,4,1	1(2)	10.0	✓
pldi09_ex6	4	9	5,1,3	4(3)	9.1	✓
pldi09_fig2	4	6	2,4,0	2(4)	43.3	✓✓
pldi09_fig4_1	3	7	1,2,4	0(1)	13.5	✓
pldi09_fig4_2	5	13	2,4,7	1(2)	14.4	✓
pldi09_fig4_3	3	3	1,2,0	1(2)	35.1	✓
pldi09_fig4_4*	4	6	1,3,2	1(2)	18.1	○
pldi09_fig4_5	3	3	1,2,0	1(2)	26.5	✓
popl09_fig2_1	5	2	1,1,0	1(3)	50.6	✓*
popl09_fig2_2	4	2	1,1,0	1(3)	70.7	✓✓
popl09_fig3_4	3	5	3,1,1	3(4)	44.0	✓
popl09_fig4_1	3	3	1,2,0	1(3)	134.1	✓
popl09_fig4_2	5	2	1,1,0	1(3)	51.1	✓*
popl09_fig4_3	5	19	3,1,15	1(2)	16.7	✓
popl09_fig4_4	3	4	1,3,0	1(2)	10.0	✓

a 5-7 minutes, e.g., `Fermat1` is the slowest because it infers invariants at 3 locations and `Egcd3` searches for invariant relations over 12 variables. Also, the most expensive subtasks vary across the programs: symbolic execution dominate for some programs (e.g., `Bresenham`, `Egcd`) while computing upper bounds is expensive for others (e.g., `Fermat1`, `Knuth`) because these programs involve many variables and thus generate many inequality candidate relations.

Note that our experiments use the typical *wall-clock* time to measure the time elapsed between the start and end of the program (by simply storing the start and end time values and obtaining their difference). If we instead use *user-cpu* time, which measures the CPU usage of a program, the time would be larger because SymInfer exploits parallelism. For example, for a run of `CohenDiv`, the user time is 370.51s but the wall-clock time is 68.48s. Multiprocessing can help program analysis scale, and SymInfer leverages this ability on increasingly available and affordable multicore computers.

RQ1: SymInfer was able to discover complex and precise nonlinear invariants to describe the semantics and correctness properties of 28/28 programs from the SV-COMP NLA benchmark. In many cases, SymInfer found undocumented but useful invariants revealing additional facts about program semantics.

5.2 RQ2: Analyzing Computational Complexity

As shown in S2.1, nonlinear invariants can represent precise program runtime complexity. More specifically, the roots of nonlinear relationships yield obtain disjunctive information that capture precise program complexity bounds.

To further evaluate SymInfer for discovering program complexity, we collect 19 programs, adapted[4] from existing static analysis techniques specifically designed to find runtime complexity [35], [45], [46]. These programs, shown in Table 2, are small, but contain nontrivial structures and represent examples from Microsoft's production code [35]. For this experiment, we instrument each program with a fresh variable t representing the number of loop iterations and generate postconditions over t and input variables (e.g., see Figure 2).

Results

Table 2, which has a similar format as Table 1, shows the results of SymInfer. Column **Correct** contains a ✓if SymInfer generates invariants matching the bounds reported in the respective work[5], and ✓✓if the discovered invariants represent more precise bounds than the reported ones.

For 18/19 programs, SymInfer discovered runtime complexity characterizations that match or improve on reported results. For `cav09_fig1a`, we found the invariant $mt - t^2 - 100m + 200t = 10000$, which indicates the correct bound $t = m + 100 \lor t = 100$. For these complexity analyses, we also see the important role of combining both inequality and equality relations to produce informative bounds. For `popl09_fig3_4`, SymInfer inferred a disjunctive equality showing that $t = n \lor t = m$ and inequalities asserting that $t \geq n \land t \geq m$, together indicating that $t = \max(n, m)$, which is the correct bound for this program. For `pldi09_fig4_5`, we obtained nonlinear results giving two bounds $t = n - m$ and $t = m$, which establish the reported upper bound $t = \max(n - m, m)$. In two programs, SymInfer obtained better bounds than reported results (marked with ✓✓). The `tripple` program shown in Figure 2 (`pldi_fig2` in Table 2) is a concrete example where the three inferred bounds are strictly less than the previously best known bound.

For `popl09_fig2_1` and `popl09_fig4_2` (marked with ✓*), we obtained similar complexity bounds as the reported results. However, the reported results also give the preconditions leading to the bounds (thus more informative than ours), but also have some *incorrect* bounds (our complexity results are correct). For example, in `popl09_fig2_1`, we got 3 bounds $t = m + n - a - b$, $t = n - a$, and $t = 0$ (when we do not enter the loop). The reported results give 4 bounds, three of which are similar to ours and also include preconditions (e.g., $n > a \land m > b \Rightarrow t = n + m - a - b$ indicates that $t = n + m - a - b$ occurs when $n > a \land m > b$). However, one of these 4 reported bounds, $(a \geq n \land b < m) \Rightarrow t = m - b$, is incorrect because under this condition the program does not enter the loop and thus has $t = 0$ instead of $t = m - b$ ($m - b$ is positive because of the condition $b < m$).

We were not able to obtain sufficiently strong invariants to show the reported bound for `pldi09_fig4_4`. However, if we create a new term representing the quotient of an integer division of two other variables in the program, and obtain invariants over that term, we obtain more precise bounds than those reported in [35].

4. We remove nondeterministic features in these programs because SymInfer assumes deterministic behavior.

5. In Table 2, results for programs prefixed with `pldi09`, `popl09`, and `cav09` are from [35], [45], [46], respectively.

TABLE 3: Disjunctive Invariant results. ✓: produce results that match or imply documented invariants. *: require minor modifications.

Prog	L	V	T	E,I,M	NL(D)	Time(s)	Correct
strncpy	1	3	4	0,2,2	0(1)	6.6	✓
oddeven2	1	4	5	2,1,2	1(2)	4.5	✓
oddeven3	1	6	7	3,2,2	2(3)	10.3	✓
oddeven4*	1	8	10	4,3,3	3(4)	92.4	✓
oddeven5	1	10	39	2,2,35	1(2)	207.0	✓
partd1	2	3	5	1,2,2	1(2)	10.3	✓
partd2	2	4	5	1,2,2	1(3)	58.1	✓
partd3	4	5	12	1,7,4	1(4)	151.9	✓
partd4	5	6	16	0,11,5	0(1)	158.5	✓
partd5	6	7	22	0,16,6	0(1)	192.9	✓
parti1	2	3	5	1,2,2	1(2)	10.2	✓
parti2	3	4	8	1,4,3	1(3)	66.0	✓
parti3	4	5	12	1,7,4	1(4)	182.0	✓
parti4	5	6	16	0,11,5	0(1)	185.0	✓
parti5	6	7	22	0,16,6	0(1)	218.0	✓

RQ2: We demonstrate a rather surprising application of SymInfer's invariants. SymInfer was able to discover unexpected and difficult invariants capturing the precise complexity bounds of 18/19 programs. In some cases, these results help reveal unknown or more informative complexity bounds.

5.3 RQ3: Disjunctive Invariant Results

In this experiment, we evaluate SymInfer's max/min invariants on benchmark programs used in existing disjunctive invariant analysis work [24], [28]. These programs, listed in Table 3, typically have many execution paths, e.g., the sorting method `oddeven5` contains 12 serial "if" blocks and thus 2^{12} paths.

The documented correctness assertions for these programs require reasoning about disjunctive invariants, but do not involve higher-order logic. For example, the sorting procedures are asserted to produce a sorted output, but are not asserted to produce a permutation of the input. Surprisingly, SymInfer discovers undocumented nonlinear relations that represent such permutation properties.

Results

Table 3, which has similar format as Table 1, shows the experimental results. For all 15 programs, the discovered invariants are sufficiently strong to prove the correctness of these programs (i.e., they match or imply the documented assertions).

For `strncpy`, which simulates the C `strncpy` function to copy the first n characters from a (null-terminated) source s to a destination d, SymInfer inferred two min-plus invariants

$$\min(|s|, n) - |d| \leq 0\ ,\ \ \min(|d|, n) - |s| \leq 0,$$

which represent the relation

$$(n \geq |s|\ \wedge\ |d| = |s|)\ \vee\ (n < |s|\ \wedge\ |d| \geq n)$$

This captures the desired semantics of `strncpy`: if $n \geq |s|$, then the copy stops at the null terminator of s, which is also copied to d, so d ends up with the same length as s. However, if $n < |s|$, then the terminator is not copied to d, so $|d| \geq n$.

As a second example, for $\texttt{oddeven}_N$, which sorts the input elements $x_0, \ldots, x_N$ and stores the results in $y_0, \ldots, y_N$, SymInfer's inferred max/min invariants prove the outputs y_0 and y_N hold the smallest and largest elements of the input, i.e., $y_0 = \texttt{min}(x_i)$ and $y_N = \texttt{max}(x_i)$. SymInfer's octagonal inequalities also show that the results are sorted, i.e., $y_0 \leq y_1 \leq \cdots \leq y_N$. These results are equivalent to the documented invariants and similar to those obtained using purely static analyses [24].

As shown in Table 3 SymInfer also found nonlinear relations, even though the documented invariants do not contain any such properties. Similarly to the complexity example mentioned in S2.1, these nonlinear properties are rather unexpected and complicated, but capture surprisingly useful and interesting program information. For example, for `oddeven2`, SymInfer found two equalities:

$$x_0 + x_1 - y_0 - y_1 = 0 \tag{5}$$

$$x_1^2 - x_1y_0 - x_1y_1 + y_0y_1 = 0 \tag{6}$$

The first linear equality shows that the sum of the inputs are the same as the outputs, which is true for sorting numbers. The second nonlinear inequality is undocumented, yet when combined with the first equation, yields useful information stating that the outputs $y's$ are permutations of the inputs $x's$. To see this, first notice that the second inequality contains the disjunctive information that x_1 is either y_1 or y_0:

$$(x_1^2 - x_1y_0 - x_1y_1 + y_0y_1 = 0)\ \Rightarrow\ (x_1 - y_0)(x_1 - y_1) = 0$$

Next, combining these two cases $x_1 = y_1 \vee y_0$ with Eq 5 shows that the outputs y_0, y_1 are permutations of the inputs x_0, x_1, i.e., $x_1 = y_0 \Rightarrow x_0 = y_1$ and $x_1 = y_1 \Rightarrow x_0 = y_0$:

$$(x_1 = y_0) \wedge (y_0 + y_1 - x_0 - x_1 = 0) \Rightarrow (x_0 = y_1)$$

and

$$(x_1 = y_1) \wedge (y_0 + y_1 - x_0 - x_1 = 0) \Rightarrow (x_0 = y_0)$$

For other programs, we also derive permutation properties from the obtained invariants through the same reasoning. For example, for `OddEven3`, SymInfer discovered three nonlinear equalities:

$$x_0 + x_1 + x_2 - y_0 - y_1 - y_2 = 0 \tag{7}$$

$$\begin{aligned} x_1^2 + x_1x_2 + x_2^2 - x_1y_0 - x_2y_0 - x_1y_1 - x_2y_1 +\\ y_0y_1 - x_1y_2 - x_2y_2 + y_0y_2 + y_1y_2 = 0 \end{aligned} \tag{8}$$

$$\begin{aligned} x_2^3 - x_2^2y_0 - x_2^2y_1 + x_2y_0y_1 - x_2^2y_2 + x_2y_0y_2 +\\ x_2y_1y_2 - y_0y_1y_2 = 0 \end{aligned} \tag{9}$$

As before, we first factor the highest-degree equality (Eq 9) to obtain $(x_2 - y_0)(x_2 - y_1)(x_2 - y_2) = 0$, i.e., $x_2 = y_0 \vee y_1 \vee y_2$). Then for each case we combine with the other equations to obtain all possible permutations (for 3 variables). We illustrate the case when $x_2 = y_0$, whose combination with Eq 8 shows that $x_1 = y_1 \vee x_1 = y_2$:

$$\begin{aligned} (x_1^2 + x_1x_2 + x_2^2 - x_1y_0 - x_2y_0 - x_1y_1 - x_2y_1 + y_0y_1\\ -x_1y_2 - x_2y_2 + y_0y_2 + y_1y_2 = 0) \wedge (x_2 = y_0)\\ \Rightarrow (x_1 - y_1)(x_1 - y_2) = 0 \end{aligned}$$

These results are then combined with Eq 7 (e.g., when $x_2 = y_0 \wedge x_1 = y_2$, we have $x_0 = y_1$) to derive permutation properties (e.g., $x_2 = y_0 \wedge x_1 = y_2 \wedge x_0 = y_1$ is a permutation of three numbers):

$$(x_0 + x_1 + x_2 - y_0 - y_1 - y_2 = 0) \;\wedge\; (x_2 = y_0 \wedge x_1 = y_2) \Rightarrow x_0 - y_1 = 0$$

Thus by doing this for all cases, these nonlinear results reveal that the resulting outputs form the six permutations of the three inputs for `OddEven3`.

While we obtained the documented invariants for all these benchmark programs with default settings in SymInfer, we had to increase the number of terms (parameterized in SymInfer) to find the undocumented nonlinear relations for `oddeven4` (indicated with *) because these relations have degree 4, which would require using more terms as the program involves 8 variables (by default SymInfer uses at most 200 terms).

SymInfer failed to obtain the undocumented nonlinear invariants for some challenging problems due to equation solving timeout. For example, the permutation of `oddeven5` would require a nonlinear equation of degree 5, which would require 3003 terms over the 10 variables in the program. Sage was not able to solve equations involving this many unknowns and thus SymInfer cannot infer the nonlinear equations to represent permutations over 5 numbers. Despite this SymInfer was able to infer other invariants about sortedness, largest, and smallest values, which match the documented invariants. A better, potentially external, equation solver might improve the scalability of SymInfer.

RQ3: SymInfer found sufficiently strong max/min and nonlinear invariants to establish the correctness of 15/15 programs requiring disjunctive invariants. We also discovered expressive undocumented nonlinear invariants that capture the higher-order permutation property of sorting algorithms.

5.4 RQ4: Checking Assertions

Several existing works generate invariants to verify given assertions or specifications. For example, to prove an assertion `assert(p)` in a program, PIE [18] computes an invariant `p'` that is sufficiently strong to prove `p`, i.e., $p' \Rightarrow p$. In contrast, SymInfer does not require given assertions to generate invariants (i.e., its goal is invariant discovery instead of finding invariants to prove specific goals). Nonetheless, we still can use SymInfer to discover invariants and compare them to the given assertions (e.g., using the Z3 solver).

In this experiment, we evaluate SymInfer on the 46 HOLA benchmark programs used in several static analyses (e.g., [47], [48], [49]). Similarly to the NLA programs, these programs are small (less than 50 LoC), but contain nontrivial structures including nested loops or multiple sequential loops and are part of the program synthesis competition SyGuS [50]. These programs are annotated with various assertions representing loop invariants and postconditions. These assertions do not involve nonlinear properties, but involve various non-trivial relations such as inequalities that are not expressible using octagonal relations and disjunctions that are not expressible using max/min invariants.

TABLE 4: SymInfer's runs on HOLA benchmarks. ✓: produce sufficiently strong invariants to prove assertions. ○: fail to make sufficiently strong invariants. *: require minor modifications.

Prog	L	V	T	E,I,M	NL(D)	Time(s)	Correct
H01	1	2	4	1,3,0	0(1)	2.8	✓
H02	1	2	4	1,3,0	0(1)	2.7	✓
H03	1	1	1	0,1,0	0(1)	48.2	✓
H04	1	1	1	0,1,0	0(1)	7.9	✓
H05	1	2	3	1,2,0	1(3)	3.7	✓
H06	1	2	4	1,3,0	0(1)	10.1	✓
H07	1	3	4	1,3,0	0(1)	6.9	✓
H08	1	2	2	0,2,0	0(1)	13.2	✓
H09	2	1	2	0,2,0	0(1)	127.5	✓
H10	1	2	3	0,3,0	0(1)	3.4	✓
H11	1	2	2	2,0,0	0(1)	15.6	✓
H12	1	1	2	0,2,0	0(1)	5.3	✓
H13	1	2	2	1,1,0	0(1)	2.7	✓
H14	1	2	4	1,3,0	1(2)	4.6	✓
H15	1	1	1	0,1,0	0(1)	1.9	✓
H16	1	3	4	0,1,3	0(1)	3.5	✓
H17	1	2	3	1,2,0	1(3)	2.9	✓
H18	1	2	4	2,1,1	2(2)	3.1	✓
H19	1	2	5	1,2,2	0(1)	8.8	✓
H20	1	5	3	2,1,0	1(2)	102.6	✓
H21	1	2	2	1,1,0	1(2)	5.1	✓
H22	1	3	6	2,4,0	0(1)	3.5	✓
H23	1	1	1	0,1,0	0(1)	1.7	✓
H24	1	2	2	0,2,0	0(1)	245.0	✓
H25	1	2	4	1,3,0	0(1)	10.0	✓
H26	1	2	4	1,3,0	0(1)	42.8	✓
H27	1	1	1	0,1,0	0(1)	43.5	✓
H28	1	2	3	0,3,0	0(1)	3.0	✓
H29	1	4	5	2,3,0	0(1)	12.3	✓
H30	1	2	2	2,0,0	0(1)	15.7	✓
H31	2	3	6	0,6,0	0(1)	63.1	✓
H32*	1	2	3	1,2,0	0(1)	2.9	✓
H33	1	2	3	1,2,0	0(1)	41.5	✓
H34	1	5	16	5,1,10	3(2)	20.4	✓
H35	1	2	5	1,2,2	0(1)	2.8	✓
H36	1	4	8	2,6,0	0(1)	64.2	✓
H37	1	2	4	1,1,2	0(1)	3.7	✓
H38	1	2	3	1,2,0	0(1)	2.8	✓
H39	1	2	2	0,2,0	0(1)	17.7	✓
H40	1	2	4	1,3,0	0(1)	8.1	✓
H41	1	4	11	2,2,7	1(2)	6.5	✓
H42	1	3	5	2,3,0	1(2)	8.9	○
H43	1	3	2	0,2,0	0(1)	4.2	✓
H44	1	3	6	0,3,3	0(1)	4.3	✓
H45	1	2	4	1,3,0	0(1)	53.4	✓
H46	1	1	2	0,2,0	0(1)	2.7	✓

Results

Table 4 shows the results. Column **Correct** shows whether SymInfer's generated invariants match or imply the annotated assertions.

For 45/46 programs, SymInfer discovered invariants are sufficiently strong to show the assertions. In most of these cases, we obtained correct and stronger invariants than the given assertions. For example, for H23, SymInfer inferred the invariants $i = n, n^2 - n - 2s = 0$, and $-i \leq n$, which imply the postcondition $s \geq 0$. For H29, we obtained the invariants $b + 1 = c, a + 1 = d, a + b \leq 2$, and $2 \leq a$, which imply the given postcondition $a + c = b + d$.

On one hand, this is expected because these assertions just involve linear properties and SymInfer has been shown to work with programs with much harder invariants. On the other hand, SymInfer was able to find undocumented nonlinear invariants, whose combinations with other invari-

TABLE 5: Symbolic states at different depths. **check**: checking candidate invariants (invalid (**S**AT), valid (**U**NSAT), unknown (**?**)) and **max**: optaining the upper bound values of terms (found (**S**AT), unknown (**?**)).

Prog	**T(s)**	**solver** check S/U(?)	max S(?)
Bresenham	64.5	242,192	170
CohenCu	1.1	228,230 (1)	182
CohenDiv	15.8	572,1771 (10)	1893
Dijkstra	1.0	194,140 (21)	117
DivBin	32.9	385,804	788
Egcd	67.6	1053,3102	2705
Egcd2	38.5	176,511 (1)	123
Egcd3	38.7	192,368 (5)	56
Fermat1	44.1	567,1153 (59)	629 (75)
Fermat2	48.3	200,425 (23)	213 (27)
Freire1	1.3	48,55	45
Freire2	0.9	0,968	0
Geo1	0.9	85,210	187 (2)
Geo2	0.9	85,201	184 (3)
Geo3	0.9	192,157 (1)	150
Hard	39.5	701,1253 (3)	1436
Knuth	36.3	1053,2922 (375)	1218 (135)
Lcm1	44.2	1050,2326 (2)	2410
Lcm2	73.3	330,788	852
MannaDiv	4.1	192,514 (20)	354
Prod4br	31.0	377,540 (5)	598
ProdBin	35.4	196,398 (5)	399 (1)
Ps2	1.0	35,131	110
Ps3	1.0	35,120	110
Ps4	1.0	35,120	110
Ps5	1.0	35,120	110
Ps6	1.0	35,120	110
Sqrt1	0.9	77,307	285

ants allow Z3 to establish assertions under forms that are *not* supported by SymInfer. For example, H08 contains a postcondition $x < 4 \lor y > 2$, which has a disjunctive form of strict inequalities. SymInfer did not produce this invariant, but instead produced a correct and stronger relation $x \leq y$, which implies this condition. Nonlinear invariants also allow us to check the assertions involving conditional information such as `if(c) assert (p);` where the property p only holds when the condition c holds. For example, for H18, we obtained the nonlinear relations $j^2 - 100j = 0$ and $fj = 100f$, which imply $j = 0 \lor j = 100$ and thus the annotated conditional assertion $f \neq 0 \Rightarrow j = 100$.

We were not able to generate sufficiently strong invariants to establish the assertion $a \equiv 1 \mod 2$ in H42 because this property cannot be expressed using SymInfer's supported invariants or combination with nonlinear invariants. Note that for H32, the path condition returned by SPF has a strange form (many nested parentheses) that crashes the Python AST parser, and thus we manually remove some parentheses from this condition.

RQ4: SymInfer was able to generate invariants that together establish the assertions in 45/46 HOLA programs. In many cases, SymInfer inferred correct and stronger invariants that prove asserted properties that are expressed in a form that is not directly supported by SymInfer (e.g., strict inequalities).

5.5 RQ5: Using Symbolic States

A main novelty of SymInfer is that it exploits the symbolic states computed by symbolic execution to improve invariant inference. Table 5 reports the uses of symbolic states in SymInfer from the NLA runs shown in Table 1. Column **T(s)** shows the total time of executing symbolic execution over multiple depths to obtain symbolic states. The next two columns show the numbers of calls to Z3 to check invariants (**check**) and compute the upperbounds of terms (**max**). Column **check** reports the number of times Z3 disproves (**S**) or proves (**U**) candidate invariants, or returns unknowns (**?**). Column **max** reports the number of times Z3 returns upper bound values (**S**) or unknown (**?**) (Z3's optimization technique returns ∞ for terms having no bounds).

Results

From these results, we see that SymInfer invokes Z3 many times. This is due to two factors: (i) SymInfer produces many octagonal and max/min-plus candidate invariants (e.g., every pair of terms produces eight candidate octagonal inequalities), and (ii) we analyze each candidate using symbolic states obtained at *multiple* depths as described in S3.2 (i.e., after proving a candidate using symbolic states at depth k, we check it again using symbolic states at depth $k + 1$ and only stop when the candidate is either disproved or remains unchanged for 3 consecutive depths).

We also see that Z3 returns more unknowns when computing upper bounds than checking candidate invariants (though for `knuth` the percentages of unknowns are approximately the same, 8% for checking and 10% for upper bound finding). This might be because Z3 is more optimized for finding satisfiability assignments than optimal assignments (especially for complex max/min-plus terms).

Note that while being used frequently, constraint solving tasks do not take up too much time as shown in Table 1. This is because Z3 is generally efficient for numerical reasoning, and SymInfer exploits parallelism and performs these tasks simultaneously.

Figure 12 shows the effect of varying symbolic depth in SymInfer from the above NLA runs. For the graph **check** on the left, the `y-axis` lists the number of the invariants remaining after being analyzed using the symbolic states at the depths given in the `x-axis`. The invariants shown at depth "zero" are newly-generated invariants that have not been analyzed at any depth. SymInfer analyzes an invariant starting at depth 7 (default) and increments the depth until it either disproves that invariant or makes no progress in 3 consecutive depths (S3.2.1).

These results show that for each program we generate a large number of invariants (i.e., depth 0) and disprove (and remove) many of them at the default depth 7. Additional depths help remove a modest number of additional invalid results. Moreover, most programs do not require depths beyond 7 (`egcd` is an exception that requires up to depth 24 to stabilize its results).

For the graph **max** on the right, the invariants at depth "zero" represent newly-generated terms that we need to find upper bounds for and at depth k are the number of remaining terms after obtaining the upper bound values using symbolic states at depth k. As described in S3.2.2

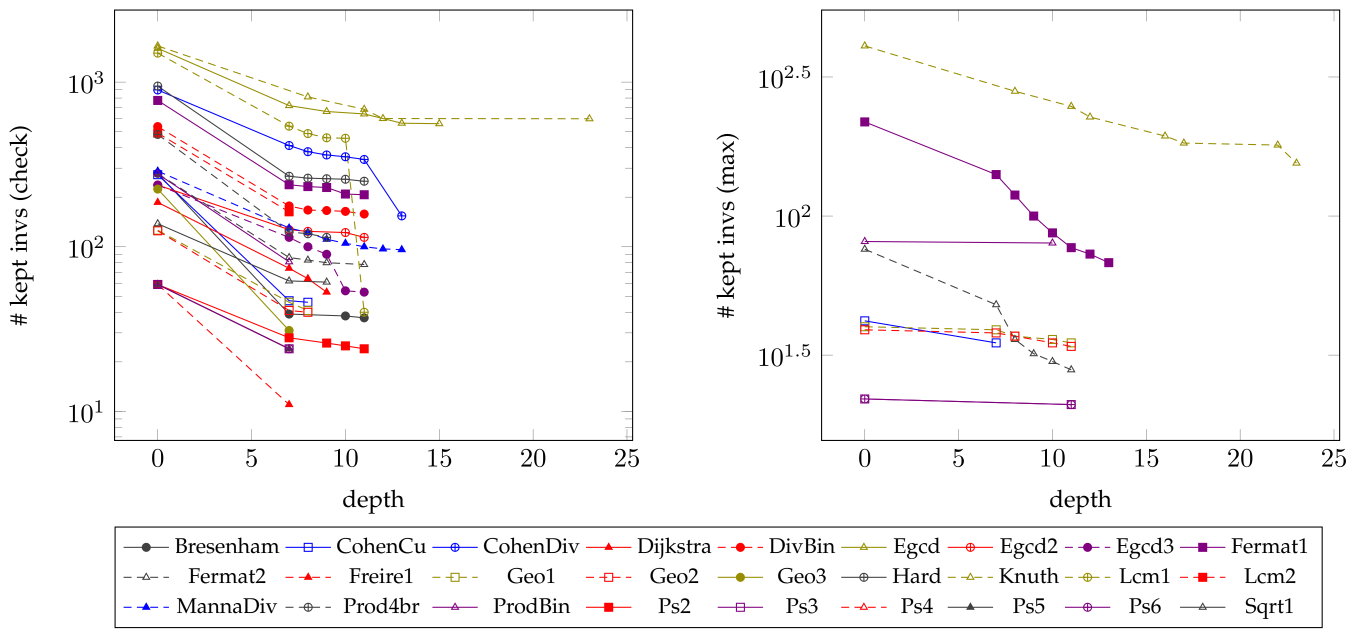


Fig. 12: Candidate invariants removed over incremental symbolic depths. Depth 0 indicates purely dynamic, i.e., results have not been checked with symbolic states at any depth. Depth 7 is the default and first depth that we check invariants.

SymInfer drops a candidate term if we found that it has no upper bound or is larger than a parameterized threshold value (by default set to 20). Similar to **check**, SymInfer increases the symbolic depths to find upper bounds for a term until the term is dropped or its value remains unchanged for 3 consecutive depths. Note that this graph does not show programs that have no changes (e.g., they start with n terms and never drop any of them).

Similarly to the **check** graph, we see that with additional depth inference converges quickly for most programs; the outlier `egcd` uses up to depth 24. However, different than the **check** graph, the upper bounds are relatively stable and do not change for most programs (those that are not shown).

RQ5: This experiment shows the effectiveness of using depth-adaptive symbolic execution. First, symbolic states are important and effective in detecting spurious invariants, even at the shallow depth of 7. Second, additional depths further help remove invalid results in more difficult programs. Third, SymInfer automatically increments depth based on the structure of the program and the state of the invariant inference algorithm, and thus can accommodate programs requiring a range of symbolic depths.

5.6 RQ6: Comparing Invariant Inference Approaches

SymInfer's Performance on Java and C programs: While the previous experiments report SymInfer's results on Java programs, SymInfer also supports C programs. SymInfer automatically invokes the CIVL symbolic execution frontend tool on C programs to obtain symbolic states and applies the same backend algorithms for invariant inference and checking.

SymInfer's results are similar for C or Java programs. By default, CIVL appears to run faster than SPF on more complex programs and slower than SPF on easier ones. For example, for `Lcm2` CIVL only took 27.3s while SPF took 73.3s, and for `Ps2--6` CIVL took 3 seconds while SPF only took a second. Thus, for complex programs whose runtimes are dominated by symbolic execution (e.g., as shown in Figure 12), SymInfer runs faster on the C versions. For the mentioned `Lcm2` program, the analysis of the took 48.5s for the C version and 96.8s for the Java version, but both analyses yield the exact 9 resulting invariants.

Thus, while different symbolic engine frontends produce different symbolic states and runtimes, the resulting invariant qualities are similar, showing the generality of using symbolic states. The modular design of SymInfer also makes it easy to add new front ends (e.g., to support another language or symbolic execution engine, we just override a few functions in SymInfer to invoke the new symbolic execution tool and parse its results).

SymInfer Compared to Other Invariant Approaches: We compare SymInfer to two other invariant generation tools NumInv and G-CLN, the verification tool PIE, and the CHC solvers Eldarica and GSPACER. Table 6 summarizes the results on the benchmarks used in our evaluation (- indicates that we were not able to run the tool on this benchmark as discussed below).

TABLE 6: Comparing SymInfer to other tools. Note that G-CLN comes with 27 NLA programs.

	NLA	COMPLE	DISJ	HOLA
SymInfer	**28/28**	**18/19**	15/15	**45/46**
NumInv	26/28	**18/19**	0/15	**45/46**
G-CLN (w/cust.)	26/27	-	-	-
G-CLN (w/default)	5/27	-	-	-
PIE	-	-	-	38/46
GSPACER	-	-	-	41/46
Eldarica	-	-	-	46/46

NumInv: Our previous invariant work NumInv [20] also relies on DIG's algorithms to infer numerical invariants, but calls the KLEE symbolic execution tool [30] as a black-box to check invariants. NumInv works with C programs and supports equalities of the form in Eq 1 and octagonal inequalities of the form in Eq 2. Thus, as shown in Table 6, NumInv achieved similar results as SymInfer for the NLA, COMPLE(XITY), and HOLA programs, whose correctness only rely on nonlinear equalities and linear inequalities. Note that SymInfer was able to show the correctness of 2 more NLA programs than NumInv (28 vs. 26) because SymInfer uses a better equation solver than the one used in NumInv, which timed out when solving large equations appeared in the two programs `Edgcd2` and `Egcd3`.

NumInv does not support min and max invariants. Thus, while it was able to generate similar equality and inequality invariants as SymInfer for the DISJ programs (e.g., nonlinear equations describing the permutation property of sorting algorithms shown in S5.3), it cannot generate any of the required max/min inequalities to capture the semantics of these programs (0/15 in Table 6). For example, NumInv cannot discover the min-plus invariants capturing the correctness property of `strncpy` and the max/min relations showing that the first and the last output elements represent the smallest and largest elements of the inputs for the $\texttt{oddeven}_N$ sorting programs.

It is difficult to directly compare the efficiency of SymInfer and NumInv. On one hand, using symbolic states allows SymInfer to reuse the results and directly compute inequalities as described in S5.5. On the other hand, the LLVM-based KLEE symbolic engine used in NumInv runs much faster than the Java SPF tool used in SymInfer. For example, for `Divbin`, SymInfer took 32.9s just to run SPF to obtain symbolic states but took 47.3s in total (thus only 14.4s for equality and inequality invariants (including max/min relations that NumInv does not consider) inference due to the use of symbolic states). For this program, NumInv took 50.51s in total, but it repeatedly invoked KLEE as a black box to check invariants. The algorithmic advantage of using symbolic states allows SymInfer to run faster, despite using a slower symbolic execution tool – SymInfer using KLEE to generate symbolic states would run substantially faster.

G-CLN: The recent tool G-CLN [21] uses a gated continuous neural network to learn candidate invariants from program traces and relies on user-supplied specifications (e.g., postconditions) to check the invariants. G-CLN focuses on nonlinear invariants (and also was evaluated on the NLA benchmark). The experimental data of G-CLN consists of pre-supplied concrete program traces for 27 NLA programs (it does not have `Bresenham`) and Z3 formulae representing the semantics and specifications (e.g., loop invariants or post-conditions) for each program.

We ran the provided runscript, which invokes G-CLN to learn invariants from given traces and checks candidate invariants with provided specifications. We confirmed that the generated invariants match or imply the correctness of 26/27 programs[6] as shown in Table 6. The runtime[7] of G-CLN ranges from 8.9s for `fermat2` to 78.7s for `lcm1` (with median around 26.5s).

We found a couple of limitations in the implementation of G-CLN. First, the tool requires user-supplied loop invariants and post-conditions to check its results. This guarantees sound results, but needs the user to provide this information (in most programs, the given specifications are either the exact documented invariants or something similarly informative)Second, G-CLN relies on the given traces and does not create new inputs or traces. G-CLN is very sensitive to both of these factors. We reran it eliminating the user-supplied specifications and using 90% of the traces and found that G-CLN only obtained sufficiently strong invariants for 20/27 programs. In addition to `knuth`, G-CLN failed 5 new programs: `egcd2`, `freire2`, `lcm1`, `lcm2`, and `prod4br`. For example, G-CLN was not able to infer the documented invariants $qx + sy = b, px + ry = a$ in `egcd2` and $4r^3 - 6r^2 + 3r + 4x - 4a = 1$ in `freire2`. Note that these issues can be mitigated by using a CEGIR approach like the one used in SymInfer, e.g., using symbolic states to check invariants and generate counterexample inputs and traces to improve inferred results.

Moreover, while learning invariants using gated neural networks can be effective, we found that G-CLN requires many specific settings from the users for *each program*. For example, the parameter `limit_poly_terms_to_unique_vars` is only used in `geo3`, and `drop_high_order_consts` is only used in `prod4br`. The `dropout` parameter is configured differently for different programs: for `fermat1`, `fermat2`, and `ps2–ps6` it is 0; for `mannadiv` is 0.1; for `freire2`, and `cohencu` it is 0.2; for `sqrt1` it is 0.5; and for the rest it is 0.3. The specifics of how inference is performed are explicitly controlled by configuration parameters. For some programs inequalities are disabled (e.g., `egcd2, egcd3, knuth, lcm1` have `ineq=-1`), while some use different inequality inference methods (e.g., `cohencu, cohendiv/2, divbin/2, hard/2, mannadiv, ps2–ps6, sqrt1` use `ieq=1` and others use `ieq=0`). Note here that `program/2` means the second loop of `cohendiv`) (thus these parameter settings are not for each program, but also for each program location in some cases). Moreover, for 12 programs, the runscript consists of degree information for *individual variables* to control the generation of terms. For example, `prod4br` has `max_deg=3, var_deg` $= \{q : 3, p : 3, a : 0, b : 0, x : 1, y : 1\}$ to specify that terms such as xq and pq are not considered because their degrees exceed 3.

In short, there are many parameters in G-CLN, and their uses and values depend on different programs. When we run without these customizations, G-CLN fails to discover the expected invariants and in many cases produced runtime errors. For example, with just the default settings (e.g., `max_deg=2`), G-CLN obtained sufficiently strong invariants for only 5/27 programs and also gave runtime errors for 6 programs. Surprisingly, when using `max_deg=4` (which technically would help generate more terms and thus invariants), G-CLN produced runtime errors for 5 programs and was not able to find sufficiently strong invariants for any programs (e.g., even missing *linear* equalities such as $z = 6n + 6$ in `cohencu`).

6. For `knuth`, G-CLN only infers linear equalities $a = d$ and $t = 0$.

7. The G-CLN paper runs its experiments on a GPU, which our machine does not have, and thus we only run G-CLN on CPU.

We were not able to run G-CLN on new programs because we do not know what parameters, settings, traces, and user-supplied specifications should be provided. Given that G-CLN requires extensive customization even on its own benchmark programs, we believe that it is difficult to make the tool work with new programs. Note that G-CLN does not support max/min invariants so it will likely fail to find those invariants, which are required in the DISJ benchmark programs.

PIE: PIE [18] uses a CEGIR and decision learning approach to infer invariants to prove given specifications, e.g., assertions or pre and postconditions. Thus, PIE aims to find sufficiently strong invariants to prove given specifications.

We were not able to directly run PIE because the original PIE tool that works with C is no longer available and the current version instead requires program models, which we find difficult to obtain from high-level languages such as C or Java. Nonetheless, in the NumInv work [20], we were able to run the original PIE and found that it failed to prove the annotated properties in 8 HOLA programs (e.g., it generates invariants that are too weak to establish them). For example, for H37, PIE failed to prove the postcondition `if (n > 0) assert(0 <= m && m < n)` which involves both conditional assertions and strict inequalities. For this program, SymInfer inferred 2 nonlinear equations and 3 inequalities[8], which are correct and together show the assertion. PIE also failed to find any of the high-degree nonlinear invariants found by SymInfer (e.g., in NLA), even when we ask it to find invariants to prove those nonlinear invariants.

CHC Solvers: Several verification works encode verification tasks (program semantics and desired property) as Horn clauses, and then use a CHC solver to find invariants to prove the given property. Thus, similar to PIE, these works generate invariants to prove specific goals.

We evaluated two popular CHC solvers Eldarica [51] and GSPACER [52] to prove the properties in the HOLA programs. These CHC solvers work on formulae encoded in the SMT-LIB format, and we directly use the SMT-LIB files provided for the 46 HOLA programs available at [53].

Eldarica solved 46/46 HOLA programs, most of them in under 3 seconds (except H34 took 11 seconds, and H32 took 18 minutes). GSPACER solved 41/46 programs, most of them in under 1 second (except H18 took 3.8 minutes). Thus, these tools are comparable to SymInfer and better than PIE.

However, just like PIE, these techniques focus on generating invariants to verify given goals (assertions or postconditions in the HOLA programs). Specifically, they cannot infer invariants in the absence of a given property whereas this is precisely what SymInfer is designed to do. For example, when we change the postcondition of H19 to `True`, Eldarica and GSPACER simply generate `True` as the invariants. These solvers also appear sensitive to the given conditions. For H30, Eldarica was able to prove the annotated postcondition $c \geq 0$, but fails to terminate when given something else, e.g., $c = 100$ or $c = 499500$. For H30, GSPACER got timeouts in all cases, even with the annotated postcondition. Eldarica and GSPACER also did not terminate after 30 minutes when we attempt to prove incorrect properties (e.g., the incorrect postcondition $y \neq 100$ in H19).

RQ6: SymInfer was able to infer more numerical invariants than NumInv, G-CLN, PIE, Eldarica, and GSPACER. The ability to exploit and reuse symbolic states allows SymInfer to strike a balance between expressive power and computational cost, while guaranteeing correctness, to establish state-of-the-art performance in numerical invariant inference.

5.7 Threats to Validity

The chief threat to external validity lies in the generalizability of the benchmarks used in our evaluation. Our evaluation uses 4 different benchmarks developed by other research groups and we use all of each of the benchmarks–we do not select subsets of benchmarks. The benchmarks are admittedly small programs and they clearly do not capture many aspects of complexity present in large software projects. However, they do include complex computational kernels that are characteristic of realistic programs, e.g., [35]. Moreover, invariant inference techniques can be applied modularly to individual functions, so the complexity of the enclosing software system is less relevant to assessing the cost-effectiveness of such techniques. Finally, we aim to promote comparative evaluation and reproducibility of our results which is achieved by using standard benchmarks and releasing our implementation[9].

SymInfer makes use of multiple underlying analysis tools, e.g., SPF, CIVL, SAGE, Z3, and DIG. These are widely used and robust systems which provides a degree of confidence that they are correct. That said, our primary means of addressing the internal validity of our findings was to perform manual and automated checking of all experimental results. For example, we ran independent checks, using an SMT solver to discharge validity claims for implication or equality formulae, to confirm that invariants computed by SymInfer were valid invariants. We then manually checked all of those results.

6 Related Work

Daikon-based Dynamic Analyses: Ernst et al.'s pioneering work on Daikon [12] demonstrated that specifications of program behavior can be inferred by observing concrete program states. Daikon used a template-based approach to define candidate invariants and, to mitigate cost, a rather modest set of templates is used that do not capture nonlinear or disjunctive properties. Many researchers have built on the foundation of Daikon by adopting its template-based approach. For example, iDiscovery [17] uses Daikon templates for inference and then attempts to verify or refute candidate invariants by running the symbolic execution tool SPF. However, neither Daikon nor iDiscovery is capable of inferring the expressive nonlinear or disjunctive invariants that SymInfer can infer for the programs in Figures 1 and 2.

8. $m^2 = nx - m - x, mn = x^2 - x, -m \leq x, x \leq m + 1, n \leq x$

9. SymInfer and all experimental data are available at https://github.com/unsat/dig/.

DIG: The DIG [27], [28], [29] dynamic invariant generation approach focuses on numerical invariants and supports more expressive families of templates, such as nonlinear equations and octagonal inequalities, and therefore can compute more expressive numerical relations than those supported by Daikon. However, DIG's results are only correct with respect to given program execution traces and might not generalize (i.e., they can be spurious).

In [28], DIG is extended with max/min invariants to infer disjunctive information and integrated with a theorem prover using k-induction to prove valid invariants and remove spurious ones. This work shows that many loop invariants, especially those in complex nonlinear programs, cannot be proved using standard induction (i.e., when $k = 1$) and requires k-induction where $k > 0$. However, the requirement that invariants being formally proved using k-induction makes this work very expensive, e.g., the sorting program `oddeven5` shown in S5 takes over half an hour to be proved. Due to this inefficiency, after disproving an invariant, this work does not use counterexample inputs to refine that invariant or to find new ones. NumInv [20] combines DIG's algorithms to infer nonlinear equations and octagonal invariants and the symbolic execution tool KLEE [30] to check and generate counterexamples to refine those invariants. We compared NumInv to SymInfer in S5.6.

Static Analyses and Goal-Oriented Invariant Generation: Static analyses based on the classical abstract interpretation framework [54], [55], [56] generate sound invariants under abstract domains (e.g., interval, octagonal, and polyhedra domains) to overapproximate program behaviors to prove the absence of errors [22]. Trade-offs occur between the efficiency and expressiveness of the considered domains. The work in [57] uses the domain of nonlinear polynomial equalities and Gröbner basis to generate equality invariants of the form in Eq 1. This approach is limited to programs with assignments and loop guards expressible as polynomial equalities and requires user-supplied bounds on the degrees of the polynomials to ensure termination. The work in [43] does not require upperbounds on polynomial degrees but is restricted to non-nested loops. SymInfer also generates invariants under various domains, but it integrates learning and checking candidate invariants using symbolic states, and does not have limitations on program constructs or require a priori degree knowledge.

Many static analyses generate invariants to prove specifications, e.g., assertions and pre and postconditions for a function or program, and thus can exploit the given specifications to guide the invariant inference process. PIE [18] and ICE [19] use CEGIR approach to learn invariants to prove given assertions. To prove a property, PIE iteratively infers and refines invariants by constructing necessary predicates to separate (good) states satisfying the property and (bad) states violating that property. ICE uses a decision learning algorithm to guess inductive invariants over predicates separating good and bad states and generates "implication" counterexamples to learn more precise invariants. For efficiency, they focus on octagonal predicates and only search for invariants that are boolean combinations of octagonal relations (thus do not infer nonlinear and disjunctive invariants such as those shown in Figures 1 and 2). The data-driven approach G-CLN [21] uses gated continuous neural networks to learn numerical loop invariants from program execution traces and uses traditional Hoare logic and Z3 to check inductive loop invariants. We compared PIE and G-CLN to SymInfer in S5.6, and found that without sufficiently strong goals (e.g., given postconditions), these approaches (PIE, ICE, G-CLN) cannot generate strong invariants like those discovered by SymInfer. G-CLN also relies on substantial problem-specific customizations to generate invariants.

CHC Solvers: Several verification works use constrained Horn clauses (CHC) to synthesize invariants to prove safety properties [58], [59], [60], [61], [62]. These works encode verification conditions, which consist of program semantics (e.g., initial states, infeasible post states, state transitions) and predicates with unknowns representing inductive and safe invariants (satisfying the initial states but avoiding the bad states) as Horn clauses, and then use a CHC solver to find satisfying assignments for the unknowns to generate the invariants. Thus, the problem of generating program invariants to prove programs is reduced to the problem of CHC satisfiability solving, which can be efficiently solved due to advancements in CHC solving technologies.

Examples of CHC solvers include Eldarica [51], which checks the satisfiability of Horn clauses over Presburger arithmetic by combining Predicate Abstraction [63] and CEGAR [64], and the solver works in [65], [66], which extend Eldarica to support formulae over the theories of integers, algebraic data-type, and bit vectors. SPACER [67], a popular SMT-based model checking Horn solver, is used in Z3 and CHC-based program analysis tools such as SeaHorn [58], and has been built upon by other CHC solving techniques such as GSPACER [52]. FreqHorn [59] learns candidate invariants by analyzing samplings representing frequency distributions of features found in the program (e.g, formulae involving variables, constants, arithmetic, and comparison operators in code). Other works, e.g., [62], [68], extend FreqHorn to use execution traces in addition to program features to learn invariants. In particular, [62] uses equation solving to infer candidate invariants and generates counterexamples to check if the invariants can be represented using purely polynomial equations or they would need conditional invariants.

Similar to the goal-oriented invariant generation techniques, CHC solvers synthesize invariants to solve verification conditions encoding specific program properties. Thus, the generated invariants largely depend on the verification goal. As a concrete example, for `ps2`, when given the documented postcondition $y^2 - 2x + y = 0$, FreqHorn quickly (within a second) found 3 invariants, $y^2 - 2x + y = 0$, $y^2 - 2x + y \geq 0$, and $y \geq 0$, to prove the given postcondition. When given a less precise post condition $y - x \leq 0$, FreqHorn finds 3 invariants: $y \geq 0$, $x - y \geq 0$, and $x - 2y \geq -1$ (and no equality invariants). Interestingly, when given something that is *not* an invariant, e.g., the wrong postcondition $y^2 - 2x = 0$, FreqHorn does not appear to terminate (we manually kill its process after 15 minutes). For a more complex example such as `ps6`, even when given the documented postconditions, FreqHorn cannot generate any invariants to prove the postconditions and also does not appear to terminate. These tools (FreqHorn and the CHC solvers Eldarica and GSPACER evaluated in S5.6) do not generate invariants if the goal is not specified.

7 Conclusion

We introduce the concept of symbolic states as an intermediate representation that can be leveraged to support the automated generation of useful and complex invariants for software systems. We propose a CEGIR approach that exploits symbolic states to generate candidate invariants and also to check or refute, and iteratively refine, those candidates. A key to the success of these methods is the ability to directly manipulate and reuse rich encodings of large sets of concrete program states.

We present SymInfer which implements CEGIR using symbolic states to efficiently discover rich invariants over numerical variables at arbitrary program locations. Evaluation on a set of 108 programs comprising 4 different benchmarks demonstrates that SymInfer is cost-effective in discovering useful invariants to describe precise program semantics, characterize the runtime complexity of programs, and check nontrivial correctness properties. This offers compelling evidence of the benefits of symbolic states in invariant inference.

Moreover, continuing advances in symbolic reasoning systems suggest that symbolic state representations are positioned to become increasingly attractive for invariant inference. For example, generating symbolic states can be sped up for invariant inference by combining directed symbolic execution [69] to target locations of interest, memoized symbolic execution [70] to store symbolic execution trees for future extension, and parallel symbolic execution [71] to accelerate the incremental generation of the tree. Moreover, we can apply techniques for manipulating symbolic states in symbolic execution [30], [72] to significantly reduce the complexity of the verification conditions sent to the solver. In future work, we plan to explore how to extend and adapt such optimizations from the general problem of symbolic execution to the problem of invariant inference.

Acknowledgments

We thank the anonymous reviewers for helpful comments. This material is based in part upon work supported by the UNL Faculty Award, by the National Science Foundation under grant numbers 1948536, 1617916, and 1901769, by the U.S. Army Research Office under grant number W911NF-19-1-0054, and by the DARPA ARCOS program under contract FA8750-20-C-0507.

References

[1] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT press, 2009.

[3] C. A. Hoare, "Proof of a program: FIND," *Communications of the ACM*, vol. 14, no. 1, pp. 39–45, 1971.

[4] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan *et al.*, "Automatically patching errors in deployed software," in *Symposium on Operating systems principles*, 2009, pp. 87–102.

[5] R. Bodik, R. Gupta, and V. Sarkar, "Abcd: eliminating array bounds checks on demand," in *Programming language design and implementation*, 2000, pp. 321–333.

[6] P. Cashin, C. Martinez, W. Weimer, and S. Forrest, "Understanding automatically-generated patches through symbolic invariant differences," in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 411–414.

[7] S. Srivastava, S. Gulwani, and J. S. Foster, "Template-based program verification and program synthesis," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5, pp. 497–518, 2013.

[8] "Coverity Scanner," https://scan.coverity.com, accessed on August 20, 2021.

[9] "The Infer Static Analyzer," http://fbinfer.com/, accessed on August 20, 2021.

[10] M. Das, S. Lerner, and M. Seigle, "Esp: Path-sensitive program verification in polynomial time," in *Programming Language Design and Implementation*, 2002, pp. 57–68.

[11] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[12] M. D. Ernst, "Dynamically detecting likely program invariants," Ph.D. dissertation, University of Washington, 2000.

[13] C. Csallner, N. Tillmann, and Y. Smaragdakis, "Dysy," in *International Conference on Software Engineering*. IEEE, 2008, pp. 281–290.

[14] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[15] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *International Symposium on Software Testing and Analysis*, 2016, pp. 177–188.

[16] T. Le, T. Antonopoulos, P. Fathololumi, E. Koskinen, and T. Nguyen, "Dynamite: dynamic termination and non-termination proofs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[17] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid, "Feedback-driven dynamic invariant discovery," in *International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 362–372.

[18] S. Padhi, R. Sharma, and T. Millstein, "Data-driven Precondition Inference with Learned Features," in *Programming Language Design and Implementation*. ACM, 2016, pp. 42–56.

[19] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," *ACM Sigplan Notices*, vol. 51, no. 1, pp. 499–512, 2016.

[20] T. Nguyen, T. Antonopoulos, A. Ruef, and M. Hicks, "Counterexample-guided approach to finding numerical invariants," in *Foundations of Software Engineering*, 2017, pp. 605–615.

[21] J. Yao, G. Ryan, J. Wong, S. Jana, and R. Gu, "Learning nonlinear loop invariants with gated continuous logic networks," in *Programming Language Design and Implementation*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 106–120.

[22] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The Astrée analyzer," in *European Symposium on Programming*. Springer, 2005, pp. 21–30.

[23] D. Maclagan and B. Sturmfels, *Introduction to tropical geometry*. American Mathematical Soc., 2015, vol. 161.

[24] X. Allamigeon, S. Gaubert, and É. Goubault, "Inferring min and max invariants using max-plus polyhedra," in *Static Analysis Symposium*. Springer, 2008, pp. 189–204.

[25] S. Anand, C. S. Păsăreanu, and W. Visser, "JPF–SE: A symbolic execution extension to Java Pathfinder," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007, pp. 134–138.

[26] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers, "Civl: the concurrency intermediate verification language," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

[27] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to discover polynomial and array invariants," in *International Conference on Software Engineering*. IEEE, 2012, pp. 683–693.

[28] ——, "Using dynamic analysis to generate disjunctive invariants," in *International Conference on Software Engineering*. IEEE, 2014, pp. 608–619.

[29] ——, "DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants," *Transactions on Software Engineering Methodology*, vol. 23, no. 4, pp. 30:1–30:30, 2014.

[30] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*. USENIX Association, 2008, pp. 209–224.

[31] T. Nguyen, M. Dwyer, and W. Visser, "Syminfer: Inferring program invariants using symbolic states," in *Automated Software Engineering*. IEEE, 2017, pp. 804–814.

[32] V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann, "Verifying and synthesizing constant-resource implementations with types," in *Symposium on Security and Privacy*. IEEE, 2017, pp. 710–728.

[33] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, "Decomposition instead of self-composition for proving the absence of timing channels," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 362–375, 2017.

[34] T. Nguyen, D. Ishimwe, A. Malyshev, T. Antonopoulos, and Q.-S. Phan, "Using dynamically inferred invariants to analyze program runtime complexity," in *International Workshop on Software Security from Design to Deployment*, 2020, pp. 11–14.

[35] S. Gulwani, S. Jain, and E. Koskinen, "Control-flow refinement and progress invariants for bound analysis," in *Programming Language Design and Implementation*, 2009, pp. 375–385.

[36] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "νz-an optimizing SMT solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 194–199.

[37] G. Strang, *Introduction to Linear Algebra*. Wellesley-Cambridge Press Wellesley, MA, 1993, vol. 3.

[38] A. Miné, "Weakly relational numerical abstract domains," Ph.D. dissertation, École Polytechnique, France, 2004.

[39] D. Kapur, Z. Zhang, M. Horbach, H. Zhao, Q. Lu, and T. Nguyen, "Geometric Quantifier Elimination Heuristics for Automatically Generating Octagonal and Max-plus Invariants," in *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*. Springer, 2013, vol. 7788, pp. 189–228.

[40] W. A. Stein *et al.*, "Sage Mathematics Software," https://www.sagemath.org, accessed on August 20, 2021.

[41] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[42] D. Beyer, "Software verification with validation of results," in *TACAS*. Springer, 2017, pp. 331–349.

[43] E. Rodríguez-Carbonell and D. Kapur, "Generating all polynomial invariants in simple loops," *Journal of Symbolic Computation*, vol. 42, no. 4, pp. 443–476, 2007.

[44] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori, "A data-driven approach for algebraic loop invariants," in *European Symposium on Programming*. Springer, 2013, pp. 574–592.

[45] S. Gulwani, K. K. Mehra, and T. M. Chilimbi, "SPEED: precise and efficient static estimation of program computational complexity," in *Principles of Programming Languages*. ACM, 2009, pp. 127–139.

[46] S. Gulwani, "SPEED: Symbolic complexity bound analysis," in *Computer Aided Verification*. Springer-Verlag, 2009, pp. 51–62.

[47] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The Software Model Checker BLAST," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 505–525, 2007.

[48] A. Gupta and A. Rybalchenko, "Invgen: An efficient invariant generator," in *International Conference on Computer Aided Verification*. Springer, 2009, pp. 634–640.

[49] B. Jeannet, "Interproc analyzer for recursive programs with numerical variables," 2014, https://pop-art.inrialpes.fr/interproc/interprocweb.cgi, accessed on August 20, 2021.

[50] "SyGuS: Syntax-Guided Synthesis Competition," https://www.sygus.org, accessed on August 20, 2021.

[51] P. Rümmer, H. Hojjat, and V. Kuncak, "Disjunctive interpolants for horn-clause verification," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 347–363.

[52] H. G. V. Krishnan, Y. Chen, S. Shoham, and A. Gurfinkel, "Global guidance for local generalization in model checking," in *International Conference on Computer Aided Verification*. Springer, 2020, pp. 101–125.

[53] "HOLA programs in SMT-LIB," https://github.com/chc-comp/eldarica-misc/tree/master/LIA/HOLA, accessed on August 20, 2021.

[54] P. Cousot and R. Cousot, "Abstract interpretation frameworks," *Journal of logic and computation*, vol. 2, no. 4, pp. 511–547, 1992.

[55] P. Cousot, "Abstract interpretation," *ACM Computing Surveys (CSUR)*, vol. 28, no. 2, pp. 324–328, 1996.

[56] A. Miné, "The octagon abstract domain," *Higher-order and symbolic computation*, vol. 19, no. 1, pp. 31–100, 2006.

[57] E. Rodríguez-Carbonell and D. Kapur, "Automatic generation of polynomial loop invariants: Algebraic foundations," in *International symposium on Symbolic and algebraic computation*, 2004, pp. 266–273.

[58] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 343–361.

[59] G. Fedyukovich, S. J. Kaufman, and R. Bodík, "Sampling invariants from frequency distributions," in *Formal Methods in Computer Aided Design*. IEEE, 2017, pp. 100–107.

[60] G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta, "Solving constrained horn clauses using syntax and data," in *Formal Methods in Computer Aided Design*. IEEE, 2018, pp. 170–178.

[61] G. Fedyukovich and R. Bodík, "Accelerating syntax-guided invariant synthesis," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2018, pp. 251–269.

[62] S. Prabhu, K. Madhukar, and R. Venkatesh, "Efficiently learning safety proofs from appearance as well as behaviours," in *International Static Analysis Symposium*. Springer, 2018, pp. 326–343.

[63] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in *International Conference on Computer Aided Verification*. Springer, 1997, pp. 72–83.

[64] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *Journal of the ACM (JACM)*, vol. 50, no. 5, pp. 752–794, 2003.

[65] H. Hojjat and P. Rümmer, "The ELDARICA horn solver," in *Formal Methods in Computer Aided Design*. IEEE, 2018, pp. 158–164.

[66] ——, "Deciding and interpolating algebraic data types by reduction," in *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2017, pp. 145–152.

[67] A. Komuravelli, A. Gurfinkel, and S. Chaki, "Smt-based model checking for recursive programs," *Formal Methods in System Design*, vol. 48, no. 3, pp. 175–205, 2016.

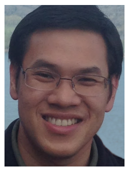
[68] G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta, "Quantified invariants via syntax-guided synthesis," in *International Conference on Computer Aided Verification*. Springer, 2019, pp. 259–277.

[69] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Static Analysis Symposium*. Springer-Verlag, 2011, pp. 95–111.

[70] G. Yang, C. S. Păsăreanu, and S. Khurshid, "Memoized symbolic execution," in *International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 144–154.

[71] M. Staats and C. Păsăreanu, "Parallel symbolic execution for structural test generation," in *International Symposium on Software Testing and Analysis*. ACM, 2010, pp. 183–194.

[72] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: Reducing, Reusing and Recycling Constraints in Program Analysis," in *Foundations of Software Engineering*. ACM, 2012, pp. 58:1–58:11.



ThanhVu Nguyen is an assistant professor in the Department of Computer Science at George Mason University, United States. His research interests include dynamic invariant inference, automatic program repair, and configurable systems analysis. His work in these areas has been recognized with several of impact and distinguished paper awards.

Kim Hao is a sophomore pursuing a bachelor's degree in Computer Science and Mathematics from the University of Nebraska-Lincoln, United States. He is a member of the UNSAT research group and advised by ThanhVu Nguyen. His research interests are in software analysis, testing, and verification. He has published in top software engineering conferences, including ICSE, OOPSLA, and ASE.

Matthew B. Dwyer is the Robert Thomson Distinguished Professor in the Department of Computer Science at the University of Virginia, United States. His research interests include software analysis, verification and testing and his work in these areas has been recognized over the years with several test-of-time and distinguished paper awards. He is a Fellow of the IEEE and of the ACM.

Appendix A
CEGIR Algorithm for Inferring Inequalities

The algorithm presented in Figure 6 exploits the capability of modern constraint solving to find the upper bound value for a given term directly from symbolic states. However, when the given term or symbolic states are too complex to directly compute the upper bounds, we can use a CEGIR, iterative guess and check style to compute the upper bounds.

```
1  Function FINDUPPERBOUND (term, minV, maxV, $P$, $L$)
2      if minV $\equiv$ maxV then return maxV
3      else if maxV $-$ minV $\equiv 1$ then
4          cexInps $\leftarrow$ check ($P$, L, {term $\leq$ minV}, {})
5          if cexInps $\equiv \emptyset$ then return minV
6          else return maxV
7      else
8          midV $\leftarrow \lceil \frac{\text{maxV}+\text{minV}}{2} \rceil$
9          cexInps $\leftarrow$ check ($P$, L, {term $\leq$ midV}, {})
10         if cexInps $\equiv \emptyset$ then
11             maxV = midV
12         else
13             //disproved
14             traces $\leftarrow$ exec ($P$, L, cexInps)
15             minV = max (instantiate (term, traces));
16         return FINDUPPERBOUND (term, minV, maxV, $P$, $L$)
```

Fig. 13: CEGIR algorithm for finding the upper bound value of a term.

Figure 13 presents a CEGIR approach using divide and conquer search technique to compute a integral upper bound k of a 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 $= 20$. Next we check $t \leq$ midV where $midV = \lceil \frac{\text{maxV}+\text{minV}}{2} \rceil$. 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 concrete state values showing that $t >$ midV and reduce the search to [minV$'$, maxV], where $minV'$ is largest the trace value observed for t. Thus this approach gradually strengthens the guess of k by repeatedly reducing the interval containing it.

The algorithm 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 SymInfer does not support real-valued bounds. However, the algorithm can be extended to handle the case when t ranges over the reals. More specifically, we can approximate real-valued results by using rationals or fixed-precision floating point values. This sacrifices precision, but preserves soundness, e.g., the invariant is $x \leq 4.123$ but we obtain $x \leq 4.2$, which is also an invariant, but less precise.

Example: We demonstrate the algorithm by finding the octagonal inequalities at location L1 of the `cohendiv` example. For demonstration purpose we restrict the bound to $[-10, 10]$. We first check candidate relations $r \leq 10, y \leq 10, r + y \leq 10, r - y \leq 10, \ldots$ and removes the invalid ones. The remaining relations have upper bounds less than or equal to 10.

For each remaining inequality candidate, we iterate 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 $midV = \lceil \frac{10+-10}{2} \rceil = 0$ 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, we tighten the bound to $\lceil \frac{0-10}{2} \rceil = -5$ and checks $r - y \leq -5$. This time the algorithm finds a cex showing that $r - y = -3$. In iteration #3, we relax the bound to $\lceil \frac{0-3}{2} \rceil = -1$ and cannot refute $r - y \leq -1$. In iteration #4, we strengthen the bound to $\lceil \frac{-1-3}{2} \rceil = -2$ and check $r - y \leq -2$, in which case it can find a cex stating that $r - y = -1$. At this point the algorithm accepts the tightest bound $r - y \leq -1$ found in iteration #3. The process for finding the lower bounds is similar as described above.

Termination: The CEGIR algorithm for finding the upper bound of a term terminates because each recursive call reduces the quantity maxV $-$ minV. This happens either at line 11, where the maxV is reduced, or at line 15, where minV is increased. Line 11 is guaranteed to reduce maxV, since the else case at line 7 guarantees that maxV $-$ minV > 1. Line 15 is guaranteed to increase minV, since the counterexample is a witness to the fact that the bound is greater than midV which in turn is greater than minV.

Appendix B
Equation inference and termination

We provide the termination proof for the equalities generation algorithm shown in Figure 8 of Section 4.1.1. The main idea is that a new counterexample decreases the dimension of the solution space of linear equation solving and the process terminates when the dimension reaches zero.

Suppose that we need to find the relations between n terms $x_1, x_2, \ldots, x_n$ using m traces $\boldsymbol{t_1}, \boldsymbol{t_2}, \ldots, \boldsymbol{t_m}$ such that

$$\boldsymbol{t_i} = \begin{bmatrix} 1 \\ x_{i1} \\ x_{i2} \\ \vdots \\ x_{in} \end{bmatrix}$$

where x_{ij} is the concrete value of x_j in the i-th trace. Let

$$M = \begin{bmatrix} \boldsymbol{t_1}^T \\ \boldsymbol{t_2}^T \\ \vdots \\ \boldsymbol{t_m}^T \end{bmatrix}$$

and $S = \text{Nul}\, M$ be the solution set of the homogeneous matrix equation $M \cdot \boldsymbol{c} = \boldsymbol{0}$. Then, each solution

$$\boldsymbol{c} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} \in S$$

represents one possible relationship between the terms x_is:

$$c_0 + c_1 x_1 + c_2 x_2 + \cdots + c_n x_n = 0.$$

Now let $\boldsymbol{t}_{\text{cex}}$ be a counterexample found by Z3, i.e., ${\boldsymbol{t}_{\text{cex}}}^T \cdot \boldsymbol{c} \neq \mathbf{0}$ for some $\boldsymbol{c} \in S$. $\boldsymbol{t}_{\text{cex}}$ cannot be a linear combination of $\boldsymbol{t_1}, \boldsymbol{t_2}, \ldots, \boldsymbol{t_m}$. Otherwise, we can write $\boldsymbol{t}_{\text{cex}} = a_1 \boldsymbol{t_{b_1}} + \cdots + a_k \boldsymbol{t_{b_k}}$ for some constants a_i, b_i. Then ${\boldsymbol{t}_{\text{cex}}}^T \cdot \boldsymbol{c} = (a_1 {\boldsymbol{t_{b_1}}}^T + \cdots + a_k {\boldsymbol{t_{b_k}}}^T) \cdot \boldsymbol{c} = a_1 {\boldsymbol{t_{b_1}}}^T \boldsymbol{c} + \cdots + a_k {\boldsymbol{t_{b_k}}}^T \boldsymbol{c} = \mathbf{0}$, which is a contradiction. Hence, the new matrix

$$M' = \begin{bmatrix} {\boldsymbol{t_1}}^T \\ {\boldsymbol{t_2}}^T \\ \vdots \\ {\boldsymbol{t_m}}^T \\ {\boldsymbol{t}_{\text{cex}}}^T \end{bmatrix}$$

has one more pivot position than M. The rank of a matrix equals the number of pivot positions, so rank $M' =$ rank $M + 1$. By the Rank theorem, rank $M + \dim \text{Nul } M =$ rank $M' + \dim \text{Nul } M' = n + 1$, hence $\dim \text{Nul } M' = \dim \text{Nul } M - 1$.

Because $0 \leq \dim \text{Nul } M' \leq n + 1$, we generate at most $n + 1$ counterexamples. Note that when $\dim \text{Nul } M' = 0$, $S = \text{Nul } M' = \{\mathbf{0}\}$, we found no meaningful relations and stop the inference algorithm.