DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants

THANHVU NGUYEN and DEEPAK KAPUR, University of New Mexico
WESTLEY WEIMER, University of Virginia
STEPHANIE FORREST, University of New Mexico

This article describes and evaluates DIG, a dynamic invariant generator that infers invariants from observed program traces, focusing on numerical and array variables. For numerical invariants, DIG supports both nonlinear equalities and inequalities of arbitrary degree defined over numerical program variables. For array invariants, DIG generates nested relations among multidimensional array variables. These properties are nontrivial and challenging for current static and dynamic invariant analysis methods. The key difference between DIG and existing dynamic methods is its generative technique, which infers invariants directly from traces, instead of using traces to filter out predefined templates. To generate accurate invariants, DIG employs ideas and tools from the mathematical and formal methods domains, including equation solving, polyhedra construction, and theorem proving; for example, DIG represents and reasons about polynomial invariants using geometric shapes. Experimental results on 27 mathematical algorithms and an implementation of AES encryption provide evidence that DIG is effective at generating invariants for these programs.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; F.3.1 [**Logics and Meanings of Programs**]: Specifying, Verifying and Reasoning about Programs—*Invariants*

General Terms: Algorithms, Experimentation, Verification, Theory

Additional Key Words and Phrases: Program analysis, dynamic analysis, invariant generation, nonlinear invariants, array invariants, geometric invariant inference, theorem proving

ACM Reference Format:
Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. DIG: A dynamic invariant generator for polynomial and array invariants. ACM Trans. Softw. Eng. Methodol. 23, 4, Article 30 (August 2014), 30 pages.
DOI: http://dx.doi.org/10.1145/2556782

1. INTRODUCTION

Program invariants are asserted properties, such as relations among variables, at certain locations in a program. By encoding program semantics as logical formulas, invariants allow for program verification using knowledge from mathematical logic, for example, to prove that a program meets required specifications or is free of certain types of errors. Invariants are also useful in other phases of programming, including documentation, design, coding, testing, debugging, optimization, and maintenance [Ernst 2000]. Thus, the study of invariants is a cornerstone of program analysis [Karr 1976; Jones et al. 1993; Ernst et al. 2007] and has been a major research area since the

This work was partially supported by the AFOSR (FA9550-07-1-0532, FA9550-10-1-0277), DARPA (P-1070-113237), DOE (DE-AC02-05CH11231), NSF (SHF-0905236, CCF-0709097, CNS-0905222) and the Santa Fe Institute.
Authors' addresses: T. Nguyen (corresponding author), D. Kapur, and S. Forrest, Computer Science Department, University of New Mexico; W. Weimer, Computer Science Department, University of Virginia; corresponding author's email: nguyenthanhvuh@gmail.com.

1970s [Wegbreit 1974; German and Wegbreit 1975; Katz and Manna 1976; Karr 1976; Suzuki and Ishihata 1977; Dershowitz and Manna 1978].

Invariants can be identified from programs using static or dynamic analysis. Static analysis discovers invariants by inspecting program code directly, and thus often has the advantage of providing sound results that are valid for any program input. The requirement that invariants be sound leads to expensive computations arising from the difficulty of analyzing complex program structures. In contrast, dynamic analysis infers invariants from traces gathered from program executions over a sample of test cases. The accuracy of the inferred invariant thus depends on the quality and completeness of the test cases. However, dynamic analysis is generally efficient and scales well to complex programs because it focuses on traces, rather than program structures. For these reasons, dynamic methods have received considerable attention in practice. Recently, for example, dynamically inferred invariants have been used to prevent security attacks in Mozilla Firefox [Perkins et al. 2009].

A dynamic invariant detector is typically initialized with a predefined collection of invariant templates postulated to be useful and likely to occur in programs. The detector filters out invalid templates based on observed program traces and returns the remainders as candidate invariants. Such an approach of checking and filtering is efficient, but it cannot find invariants that are inexpressible under the predefined templates. This article is concerned with two such examples: relations over general polynomials of numerical and array variables. Polynomials are fundamental to many scientific and engineering applications. Nonlinear polynomials, for example, are especially useful for the analysis of hybrid systems [Roozbehani et al. 2005; Sankaranarayanan et al. 2005]. ASTRÉE [Cousot et al. 2005a; Blanchet et al. 2003], a successful static analyzer used to verify the absence of runtime errors in Airbus avionic systems, implements a static analysis involving the ellipsoid abstract domain to represent and reason about a class of quadratic inequality invariants.[1] Arrays are a widely used data structure that is essential to the success of many programs. Fixed-size arrays are also present in many systems programs, and proper reasoning is often critical for security (e.g., buffer overruns) and performance (e.g., for bounds check elimination [Bodík et al. 2000]). The major impediment for finding these invariants dynamically is the prohibitively large number of possible templates, for example, each relation involving polynomials of different degrees or different numbers of variables would require a separate template. For example, Daikon [Ernst 2000; Ernst et al. 2007], the pioneer of dynamic invariant analysis, detects only linear relations over at most three variables and has limited support for array relations.

This article describes and evaluates DIG, a dynamic invariant generator that finds polynomial and array relations over program execution traces. DIG uses parameterized templates and computes the unknown coefficients in the templates directly from traces. Consequently, the resulting invariants represented by the instantiated templates are precise over the input traces. DIG also creates terms to represent information about variables such as nonlinear polynomials over numerical variables. These techniques allow us to discover invariants that are more expressive than those considered by current dynamic methods.

DIG takes as input a set of traces consisting of values from numerical (reals and integers) or array variables captured at any program points, including entrance/exit points of functions or heads of loops. Depending on the invariants of interest, different techniques are used to generate invariants over the input traces. We view polynomial

[1]The ellipsoid domain for this case is expressed by the quadratic form $x^2 + axy + y^2 \geq k$, where $0 < b < 1$ and $a^2 < 4b$ [Feret 2004].

equations and inequalities over variables as geometric shapes in multidimensional space: trace data as points, equations as hyperplanes, and conjunctions of inequalities as convex polyhedra. Thus, we can exploit well-known concepts and efficient algorithms from algebra and geometry to compute these properties accurately.

DIG also generates flat (non-nested) and nested array relations among multidimensional array variables (and functions that can be viewed as arrays). To find linear equations among flat arrays, we first find equalities among array elements and then identify the relations among array indices from the obtained equalities. We also find nested array relations by performing reachability analysis. Real-world programs often use large arrays, thus we employ automatic theorem proving technologies to reason about large arrays more efficiently.

The main contributions of this article are as follows.

—DIG, a dynamic invariant generator that automatically discovers polynomial and array invariants from program execution traces. We integrate concepts and tools from mathematical fields such as linear algebra, geometry, and formal methods to improve dynamic analysis. This is the first work to demonstrate the use of dynamic analysis to generate nontrivial program invariants such as nonlinear polynomial and nested array relations.
—A mapping from the task of inferring polynomial invariants to that of generating geometric shapes over points created from input traces. DIG represents equality and inequality constraints among multiple variables as hyperplanes and polyhedra. Polyhedra in high dimensions are expensive to compute, thus we also consider simpler geometric shapes, such as octagons, which are more tractable because they encode less expressive constraints. When additional inputs from the user are available, we can also deduce new inequalities from previously obtained equality relations.
—The use of equation solving and automatic theorem proving to dynamically infer relations among array variables of multiple dimensions and functions of multiple arguments. In particular, we encode the problem of finding nested array relations as a satisfiability formula that can be efficiently solved by a satisfiability modulo theories (SMT) solver.
—An empirical evaluation of DIG on a set of 27 programs that have documented invariants[2] involving nonlinear polynomials. We also evaluated DIG on an implementation of AES encryption that contains documented invariants involving array relations. The tool successfully discovers all documented invariants of the considered polynomial and array forms.

A preliminary version of some of these points was published in Nguyen et al. [2012]. This article extends those results to include the following.

—*Geometric Invariant Inference*. By interpreting polynomial relations as geometric shapes, we take advantage of proven concepts and existing algorithms in linear algebra and geometry to reason about these invariants efficiently. Using geometric reasoning, we prove an underapproximation property of polynomial invariants that is guaranteed in DIG, but not in other template-based analyses.
—*New Forms of Invariants*. We extend our earlier work to include octagonal inequalities, a special form of constraint that is useful in array bound checks. The technique for flat array relations has been extended to identify relations over certain subsets of array elements, that is, a form of conditional invariants. A modified version of

[2]*Documented* invariants refer to information found in the program documentations or source-code comments describing the program behaviors.

```
def egcd(A,B):
  x = A; y = B; i = 1
  j = 0; k = 0; m = 1
  while [L] x >= y:
    if x > y
       x = x - y
       i = i - k
       j = j - m
     else
       y = y - x
       k = k - i
       m = m - j
  return x, i, j
```

a	b	x	y	i	j	k	m
9	5	9	5	1	0	0	1
9	5	4	5	1	-1	0	1
9	5	4	1	1	-1	-1	2
			⋮				
182	255	182	255	1	0	0	1
182	255	182	73	1	0	-1	1
182	255	109	73	2	-1	-1	1
			⋮				

Fig. 1. An extended GCD algorithm and its traces at location L on inputs ($a = 9, b = 5$) and ($a = 182, b = 255$). From such traces, DIG generates three nonlinear invariants $x = ai + bj$, $y = ak + bm$, $1 = im - jk$.

reachability analysis for generating nested array relations with polynomial runtime complexity is also provided.

—*Formal Analysis*. We describe the time complexity of all presented algorithms. In particular, we show that the problem of generating nested array relations belongs to the polynomial class of complexity by presenting a polynomial-time algorithm for it.

The rest of this article is organized as follow: Section 2 provides a motivating example and an overview of DIG. Sections 3 and 4 describe our algorithms for generating polynomial and array invariants in detail. Section 5 analyzes the complexity of the given algorithms and shows interesting properties of the generated invariants. Section 6 reports experimental results. Section 7 surveys related work. Section 8 concludes.

2. MOTIVATING EXAMPLE

We use an example program to highlight the important insights underlying DIG and to motivate key design decisions. Figure 1 shows an implementation of `egcd`, an extended GCD algorithm in number theory that takes as input a pair of integers (a, b) and returns $x = \gcd(a, b)$ and two integers i, j satisfying the Bézout identity $x = ai + bj$. The main computation of `egcd` consists of a while loop on lines 4–12, whose semantics is captured by its loop invariant at location L. The table in Figure 1 consists of several sets of trace values from the eight variables $\{a, b, x, y, i, j, k, m\}$ in scope at L for the two inputs ($a = 9, b = 5$) and ($a = 182, b = 255$).

From such traces, DIG identifies three nonlinear relations $x = ai + bj$, $y = ak + bm$, $1 = im - jk$ at location L. The first two are documented invariants for `egcd`, which assert the computation and preservation of the Bézout identity in the loop. The third relation is a valid but undocumented invariant, revealing a potentially useful detail: the product im is exactly 1 more than the product jk whenever the program reaches location L.

At a high level, DIG treats numerical trace data as points in Euclidean space and computes geometric shapes enclosing these points. For example, the trace values of the two variables v_1, v_2 are points in the (v_1, v_2)-plane. DIG then determines if these points lie on a line, represented by a linear equation of the form $c_0 + c_1v_1 + c_2v_2 = 0$. If such a line does not exist, DIG builds a bounded convex polygon from these points. The edges of the polygon are represented by linear inequalities of the form $c_0 + c_1v_1 + c_2v_2 \geq 0$. This technique generalizes to equations and inequalities among multiple variables by constructing hyperplanes and polyhedra in a high-dimensional space. To generate nonlinear constraints, DIG uses terms to represent nonlinear polynomials over program variables, for example, $t_1 = v_1, t_2 = v_1v_2$. This allows DIG to generate

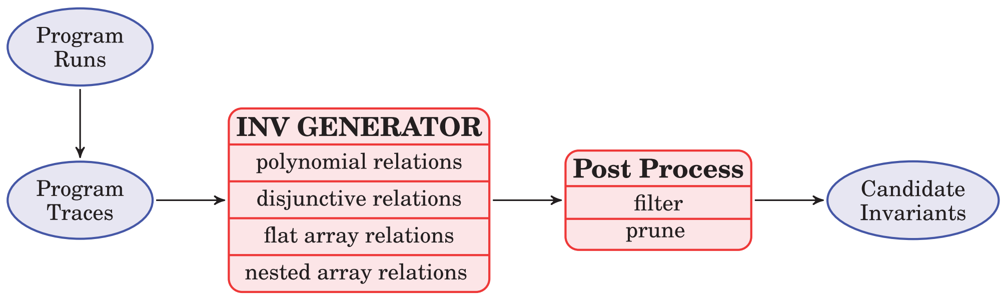

Fig. 2. An overview of DIG. The generator finds different types of invariants from input traces. The post-processing step removes redundant and spurious invariants.

equations such as $t_1 + t_2 = 1$, which represents a line over t_1, t_2 and a hyperbola over v_1, v_2.

DIG builds geometric shapes in high dimensions that are represented by nonlinear constraints over multiple variables or terms. However, constructing such complex shapes from many points is expensive. Thus, DIG also supports constraints representing simpler geometric shapes such as octagons. Octagonal inequalities are less expressive than general inequalities, but they are useful for detecting bugs such as array bounds errors and memory leaks. The modular design of DIG allows for easy extensions to other geometric shapes for other specific forms of invariants.

Returning to the `egcd` example, terms are generated to represent monomials up to a certain degree over the variables $\{a, b, x, y, i, j, k, m\}$. An equation template of the form $c_1t_1 + \cdots + c_nt_n = 0$ is created from the terms t_i. We use the traces in Figure 1 to instantiate the template, obtaining a set of equations, which we then solve for the unknowns c_i using a standard equation solver. This allows DIG to identify the three equations $x = ia + jb, y = ka + mb, 1 = im - jk$ at location L from the execution traces of `egcd`. These nonlinear invariants cannot be discovered by current dynamic analysis tools and are also challenging for methods based on static analysis.

We explore these ideas and describe concrete implementation details in the next sections.

2.1. Overview of DIG

Figure 2 gives an overview of the DIG framework that generates invariants from input traces consisting of values from numerical or array variables.[3] First, terms are created to represent variables whose values are captured in the traces. Depending on the type of the variables from input traces, DIG next generates polynomial relations or array relations over terms. Finally, the post-processing step removes redundant and spurious invariants[4] and reports the remaining invariants as its output. Optionally, the user can modify the parameters of DIG for better performance or specify additional information to aid the invariant generation process, for example, loop conditions as described in Section 3.2.1.

2.2. Terms

We use *terms* to represent nonlinear properties over program variables and other information of interest. From a set V of variables and a degree d, a set T of terms

[3]Currently we do not support variables from dynamic data structures that may have values in some executions and may not exist in other executions.

[4]*Spurious* invariants refer to candidate relations that hold over the observed traces at a program location, but they might not hold over all possible traces at that location.

is created to represent monomials up to degree d from V. For instance, the set T of ten terms $\{1, x, y, i, xy, xi, yi, x^2, y^2, i^2\}$ contains all monomials up to degree 2 over the variables $\{x, y, i\}$. Nonlinear relations over program variables can now be specified as linear relations over terms, which allows us to generate nonlinear invariants from existing techniques for linear constraint solving.

In addition to monomials, the user can manually define terms to capture other desirable properties, for example, $t_1 = \frac{x}{y}, t_2 = x^i, t_3 = \text{mod}(x, 256)$. This idea is related to the concept of *derived variables* used in Daikon to express additional information [Perkins and Ernst 2004]. Users can also query DIG for relations among a specific set of terms, for example, only inequalities among $\{x, y, i^2\}$. These customizations allow DIG to identify specific relations among potentially interesting terms and reduce the overall complexity of the process.

2.3. Post Processing

DIG uses two techniques, pruning and filtering, to help remove redundant and spurious invariants. We note that existing strategies from other approaches could also be integrated with DIG-generated invariants. For example, if DIG's algorithms were incorporated into Daikon, then most of its optimization techniques [Perkins and Ernst 2004] could be applied directly to the resulting invariants. Recently, the work reported in Sharma et al. [2013] has integrated ideas from DIG to infer sound equality invariants.

Pruning. To reduce the number of candidate invariants, DIG removes any invariants that are logical implications from other invariants. For instance, we suppress the invariant $x^2 = y^2$ if another invariant $x = y$ is also found, because the latter implies the former. These redundant invariants arise because we treat each term as an independent variable for the purpose of finding nonlinear polynomials. For example, if $t_1 = x, t_2 = y, t_3 = x^2, t_4 = y^2$, then $x = y$ implies $x^2 = y^2$; however, their corresponding term relations, $t_1 = t_2$ and $t_3 = t_4$, have no direct relation. To verify an implication, we use an SMT solver to show that the negation of that implication is unsatisfiable.

Filtering. DIG uses a subset of the input traces for invariant generation and the remaining traces to check the resulting invariants. Because a program invariant holds for any set of traces, it is likely that we can find that same invariant using a smaller subset of the available traces. The candidate invariant, which is obtained using a subset of traces and might not be true for all observed traces, is then verified against the remaining traces and removed if it fails for any of these traces. This strategy improves the runtime of DIG since it is more expensive to generate invariants, especially complex ones like $1.2xy - 2.3yz + 3.4zw = 0$, than to check that they hold over given traces.

Currently DIG randomly chooses traces for invariant generation and training. For example, DIG selects a set of random traces whose size is $1.5 \times$ the number of terms for invariant generation and 1,000 random traces for filtering. In future work, we intend to use machine-learning heuristics for more effective partitioning of traces for training and learning purposes. We elaborate further the application of filtering for reducing spurious inequality relations in Section 5.2.

3. POLYNOMIAL INVARIANTS

DIG takes as input the set V of variables that are in scope at location L, the associated traces X, and a maximum degree d, and returns a set of possible polynomial relations among the variables in V whose degree is at most d. The post-processing techniques in Section 2.3 are applied to the obtained relations to suppress redundant relations and to filter out spurious invariants.

3.1. Equalities

DIG treats polynomial equalities as unbounded geometric shapes, for example, lines and planes, to obtain equality invariants of the form

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

where c_i are real-valued and t_i are terms.

Algorithm 1 outlines the four steps to generate equality invariants. First, `genTerms` creates the set T of terms to represent monomials over the input variables V up to degree d, as covered in Section 2.2. These terms are used by `genTemplate` to form the equation template F in Eq. (1). Next, we instantiate each trace containing values of the program variables by F to form an equation. Repeating this process of instantiating equations from the traces X with `genEqts` gives a system of linear equations $E = \{e_1, \ldots, e_{|X|}\}$. We then solve E for c_i using a standard equation solver in linear algebra. The nontrivial solutions of E, if any, suggest relations among the terms in T.

ALGORITHM 1: Algorithm for Finding Polynomial Equations

```
input  : set $V$ of variables, set $X$ of traces, max degree $d$
output: set $S$ of polynomial relations of the form given in Eq. (1)
$S \leftarrow \emptyset$
$T \leftarrow$ genTerms($V$,$d$)
$F \leftarrow$ genTemplate($T$)
$E \leftarrow$ genEqts($F$,$X$)
$S \leftarrow$ solve($E$)
return $S$
```

The nontrivial solutions of E for the unknowns c_i are of the form $c_i = v_i$. The values v_i are free variables that range over the reals. The terms in the template F that have zero-valued coefficients are not related, because the only way to satisfy equations in E is by setting the coefficients of these terms to zero. In contrast, terms that have coefficients sharing some free variable v are related. To find relation among the terms sharing the variable v, we fix v to a concrete value, for example, $v = 1$ and other v' to 0, and instantiate F with $v = 1$ and $v' = 0$. This step is repeated for each shared variable v to get relations among terms sharing v.

Example. We demonstrate these steps by deriving the nonlinear equalities $x = ai + bj$, $y = ak + bm$, $1 = im - jk$ for `egcd`. For illustration, we focus on the case where $d = 2$, in which the algorithm generates quadratic equations.

For the eight variables $\{a, b, x, y, i, j, k, m\}$, together with degree $d = 2$, the set $T = \{1, a, \ldots, m^2\}$ of monomials of degree ≤ 2 contains 45 terms. We use T to form the template $F : c_1 + c_2a + \cdots + c_{45}m^2 = 0$ with 45 unknown coefficients c_i to be solved for. F is instantiated with the elements in X to form the set E of equations. For example, instantiating F with the values $(a = 9, \ldots, m = 1)$ from the first trace in Figure 1 gives the equation $c_1 + 9c_2 + \cdots + c_{45} = 0$. Solving E for the unknowns c_i results in the nontrivial solution

$$\begin{array}{lll} c_5 = v_1, & c_{16} = -v_1, & c_{24} = -v_1, \\ c_4 = v_2, & c_{14} = -v_2, & c_{22} = -v_2, \\ c_1 = v_3, & c_{39} = -v_3, & c_{41} = v_3, \\ c_2 = v_4, & c_{30} = -v_4, & c_{33} = v_4, \\ c_3 = -v_5, & c_{29} = -v_5, & c_{32} = v_5, \end{array}$$

and all other $c_i = 0$.

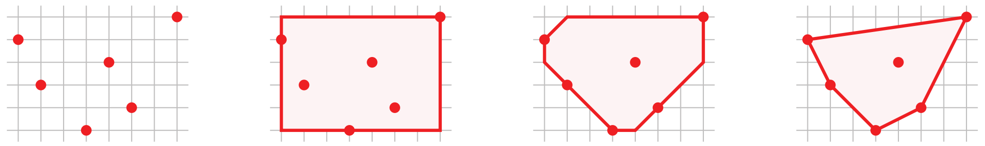

Fig. 3. (a) A set of points in 2D and its approximation using the (b) interval, (c) octagonal, and (d) polyhedral regions.

To find the relation among the terms t_5, t_{16}, t_{24} whose coefficients c_5, c_{16}, c_{24} share the value v_1, we set $v_1 = 1$ and $v_2 = v_3 = v_4 = v_5 = 0$ (since the terms t_5, t_{16}, t_{24} are not related by the values $v_{2,3,4,5}$). The template F, when being instantiated with $(v_1 = 1, v_2 = v_3 = v_4 = v_5 = 0)$, gives the relation $t_5 - t_{16} - t_{24} = 0$. The terms t_5, t_{16}, t_{24} represent the monomials y, bm, ak, thus we obtain the relation $y - bm - ak = 0$.

After repeating this process for all shared variables, the following equations are achieved:

$$\begin{array}{llllll}
t_5 = y, & t_{16} = bm, & t_{24} = ak & \rightarrow & y = bm + ak, \\
t_4 = x, & t_{14} = ai, & t_{22} = bj & \rightarrow & x = ai + bj, \\
t_1 = 1, & t_{39} = im, & t_{41} = jk & \rightarrow & 1 = im - jk, \\
t_2 = a, & t_{30} = mx, & t_{33} = jy & \rightarrow & a = mx - jy, \\
t_3 = b, & t_{29} = kx, & t_{32} = iy & \rightarrow & b = -kx + iy.
\end{array}$$

The first three equations are program invariants. The last two relations are redundant, that is, they can be obtained from the first three relations through variable substitutions. The post-processing step in Section 2.3 suppresses these redundant invariants using theorem proving. The resulting set of equations for `egcd` after post processing is $\{y = bm + ak, x = ai + bj, 1 = im - jk\}$.

3.2. Inequalities

We interpret sets of inequalities among terms as geometric shapes over points created from program traces. Figure 3 depicts several geometrical shapes corresponding to the types of inequalities currently supported in DIG. For illustration purposes, two-dimensional shapes are used to represent linear relations between two terms. Figure 3(a) shows a set of trace points created from input traces. Figures 3(b), 3(c), 3(d) approximate the area enclosing these points using the interval, octagonal, and polyhedral shapes that are represented by systems of constraints of the forms $c_1 \leq v \leq c_2$, $c_1 \leq \pm v_1 \pm v_2 \leq c_2$, and $c_1 v_1 + \cdots + c_n v_n \leq 0$, respectively. These constraints are sorted by expressive power: polyhedral constraints can express octagonal constraints, which can express interval constraints. This order is reversed in complexity: interval shapes are cheaper to compute than octagons, which are cheaper to compute than polyhedra.

3.2.1. General (Polyhedral) Inequalities. DIG finds inequality invariants of the form

$$c_1 t_1 + \cdots + c_n t_n \geq 0, \tag{2}$$

where c_i are real-valued and t_i are terms. These general inequalities also represent octagonal inequalities (two terms with specific integral coefficients) and interval inequalities (single terms with unit coefficients).

Algorithm 2 outlines two techniques for finding general inequalities. The polyhedral technique consists of three steps: using terms to represent program variables (`genTerms`), instantiating points from terms using input traces (`genPoints`), creating a convex polyhedron enclosing the points (`createPolyhedron`), and extracting its facets

```
def cohen(x,y):
  q = 0; r = x
  while r >= y:
    a = 1; b = y
    while r >= 2 * b:
      [L]
      a = 2 * a
      b = 2 * b
    r = r - b
    q = q + a
  return q
```

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. 4. Cohen's integer division algorithm and its traces on inputs ($x = 15, y = 2$) and ($x = 4, y = 1$).

ALGORITHM 2: Algorithm for Finding Polynomial Inequalities

```
input : set of variables $V$, set of traces $X$, max degree $d$
input : (optional) set of inequalities ieqs from additional information such as loop
        conditions
output: set $S$ of polynomial inequalities
$S \leftarrow \emptyset$
if ieqs $= \emptyset$ then
    $T \leftarrow$ genTerms($V$,$d$)
    $P \leftarrow$ genPoints($T$,$X$)
    $H \leftarrow$ createPolyhedron($P$)
    $S \leftarrow$ extractFacets($H$)
end
else
    // if additional information is given
    eqts $\leftarrow$ genInvs$_{eqts}$($V$,$X$,$d$)
    $S \leftarrow$ deduce$_{ieqs}$(eqts,ieqs)
end
return $S$
```

to represent inequalities among terms (extractFacets). When additional information is available, the alternative technique combines the discovered equations (genInvs$_{eqts}$) with the given information to deduce new inequalities (deduce$_{ieqs}$). Both techniques give sound relations with respect to input traces; however the deduction method, with the help of additional information, runs much faster.

We demonstrate these methods using the cohen program in Figure 4, which has a nonlinear inequality invariant and other information that is useful for deduction. cohen implements the integer division algorithm by Cohen [1990], which takes as input a pair of integers (x, y) and returns the integer q as the quotient of x and y. We consider invariants at location L, the head of the inner while loop. There are six variables $\{a, b, q, r, x, y\}$ in scope at L. The table in Figure 4 consists of several sets of values representing traces obtained from the variables at L for inputs ($x = 15, y = 2$) and ($x = 4, y = 1$).

The documented invariants $b = ya, x = qy + r, r \geq 2ya$ describe precisely the semantics of the inner while loop in Cohen's algorithm.[5] The first two equations are obtained using the technique described in Section 3.1. This section focuses on the third invariant

[5]The invariant $x = qy + r$ asserts that the dividend x equals the divisor y times the quotient q plus the remainder r.

that is an inequality of the form given in Eq. (2). For illustration, we again focus on the case where $d = 2$, in which the algorithm generates quadratic inequalities.

Using Polyhedra. After the set T of terms is created, the traces X are used to generate points in $|T|$-dimensional Euclidean space, and the convex hull of these points is computed to represent a polyhedron H. The bounded convex polyhedron H can also be described by a system of linear inequalities of the form given in Eq. (2). This is called the *half-space* representation of a polyhedron. The facets of H, corresponding to the solutions of the system of linear equalities, represent the inequalities among the terms in T. Figure 3(d) depicts a 2D polyhedron (polygon) that has five facets.

The complexity of building H in $|T|$ dimensions is exponential in $|T|$, as discussed in Section 5.1. The set T generated from the six variables in `cohen` with degree 2 has 28 terms. Building a convex polyhedron in 28 dimensions is not computationally feasible, so DIG uses several heuristics to identify possible inequality relations.

We first observe that a program invariant often involves just a small subset of all possible program variables. For example, the invariant $b - ay = 0$ involves only $\{a, b, y\}$ even though all six variables in scope were considered. We experimented with several heuristics based on this observation, such as iteratively searching for invariants involving all possible combinations of a small, fixed number of variables. The ability to determine which variables are important improves performance greatly and is further discussed in Section 6.3.

Example. DIG first generates possible inequality relations in which at most three of the six program variables $\{a, b, q, r, x, y\}$ appear. There are $\binom{6}{3} = 20$ combinations that contain three variables, one of which is $\{r, y, a\}$. To find nonlinear inequalities, terms of degree d are built on the variables under consideration. With $d = 2$, DIG generates the set $T = \{1, r, y, a, ry, ra, ya, r^2, y^2, a^2\}$ of terms.

The elements of T are instantiated with the traces X to form a set P of points. For instance, the first trace in Figure 4 gives the point [1, 15, 2, 1, 30, 15, 2, 225, 4, 1] in ten-dimensional Euclidean space corresponding to the terms in T. The convex polyhedron H is then constructed to enclose the points in P. One of the facets of H corresponds to the documented invariant $r - 2ya \geq 0$. The inequalities represented by other facets are also valid with respect to the input traces, although they might be spurious invariants. Section 5.2 provides additional discussion on these spurious invariants.

Deduction From Loop Conditions. As previously shown, the polyhedral method for general inequalities does not scale to large numbers of terms. Consequently, we developed an alternative technique using *deduction* to find inequalities of the form given in Eq. (2) if some additional information is available. More specifically, if some inequalities are asserted at location L, then DIG can use them together with the discovered equalities from Section 3.1 to deduce new nontrivial inequalities. For instance, if the location L is the head of a loop, then L can be reached if and only if the loop conditions are met. Such loop conditions are an example of additional information, which can be given as input from the user (or automatically mined from the source code as in the `cohen` program) to facilitate the process of generating additional invariants. Deduction is related to the strategy of adding known facts or proved result as lemmas in interactive theorem provers such as PVS [Owre et al. 1992].

Example. We demonstrate how deduction is applied to `cohen`. First, the set of equations $\{b - ay = 0, qy + r - x = 0\}$ representing possible invariants at location L is obtained, as described in Section 3.1. The head of the inner loop at location L is reached only when the condition of that loop $r \geq 2b$ is met, thus $r - 2b \geq 0$ is also an invariant at L. New and nontrivial inequalities can be deduced from this additional information using deduction, term rewriting, and substitution. In the current implementation, we pair inequalities from the loop conditions with the obtained equations to deduce new inequalities. For the running example, $r - 2ay \geq 0$ is deduced from the pair $(r - 2b \geq 0, b - ay = 0)$, and $x - qy - 2b \geq 0$ is deduced from $(r - 2b \geq 0, qy + r - x = 0)$. Hence, deduction finds the inequalities $r - 2ya \geq 0, x - qy - 2b \geq 0$ among the variables $\{a, b, q, r, x, y\}$, both of which are program invariants at location L in `cohen`.

Deduction could theoretically produce many results by combining discovered equalities and loop conditions. However, the technique is efficient in our experiments because the number of loop conditions and generated equality invariants is few (one or two guards at most loops and less than four equalities at a particular program location). Moreover, although we could just use the obtained equalities and loop conditions, these relations might be opaque to the human user, but the deduced ones are easier to understand (e.g., match the program descriptions). Our experiments in Section 6.2 shows that deduction allows for effective inequality invariant discovery that otherwise would require the more expensive polyhedral method or would not be possible in the case of incomplete traces.

3.2.2. Octagonal Inequalities. DIG builds an octagon—a polygon with eight edges—depicted in Figure 3(c), to obtain constraints of the form

$$c_1t_1 + c_2t_2 \geq k, \tag{3}$$

where t_1, t_2 are terms, $c_1, c_2 \in \{-1, 0, 1\}$ are coefficients, and k is real-valued.

To obtain the half-space representation of an octagon enclosing the points $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, we compute

$$\begin{aligned} u_1 &= \max(x_i),\ l_1 = \min(x_i), \\ u_2 &= \max(y_i),\ l_2 = \min(y_i), \\ u_3 &= \max(x_i - y_i),\ l_3 = \min(x_i - y_i), \\ u_4 &= \max(x_i + y_i),\ l_4 = \min(x_i + y_i) \end{aligned}$$

and form the system of linear constraints $\{u_1 \geq x \geq l_1, u_2 \geq y \geq l_2, u_3 \geq x - y \geq l_3, u_4 \geq x + y \geq l_4\}$. The algorithm to find octagonal invariants from inputs X, V, d is similar to the one listed in Figure 2, where the *createPolyhedron* function computes candidate octagonal invariants for each pair of terms in T. The post-processing techniques from Section 2.3 also apply to the obtained invariants.

Like polyhedral constraints, octagonal constraints can also represent interval constraints, for example, $u_1 \geq x \geq l_1, u_2 \geq y \geq l_2$, as illustrated in Figure 3(a). However, octagonal constraints are less expressive than general constraints due to the restriction to two terms with specific integral coefficients. For instance, octagonal constraints cannot represent the inequality $t_1 \leq 2t_2$, where $t_1 = r, t_2 = ya$, in the `cohen` program due to the coefficient 2. However, if $2ya$ is represented by a term, then the inequality $r - 2ay \geq 0$ can be generated with octagonal constraints. Octagons can be computed efficiently, and their constraints are useful for detecting bugs in flight-control software, performing array bounds and memory leaks checks [Cousot et al. 2005b; Miné 2004].

Example. Consider the following code fragment `flatten` often seen in C programs that puts the contents of a two-dimensional array $A[M][N]$ into a one-dimensional array $B[MN]$.

```
for (i = 0; i < M; ++i){
  for (j = 0; j < N; ++j){
    k  = i * n + j;
    [L]
    B[k] = A[i][j];
  }
}
```

The nonlinear relation $0 \le k \le MN-1$ at location L is essential for the safety of `flatten` and is identified by DIG using octagonal constraints with terms representing quadratic polynomials over variables. The array relation $A[i][j] = B[iN+j]$, which asserts the correctness of `flatten`, is also generated by DIG using the technique described in the next section.

4. ARRAY INVARIANTS

DIG takes as input the set V of (possibly multidimensional) array variables that are in scope at location L and the associated traces X, and returns a set of possible relations among the elements of arrays in V. Currently, we do not consider nonlinear array relations (e.g., $A[i]^2 = B[i]^2 + C[i]^2$) and therefore do not use terms to represent array variables. The filtering technique given in Section 2.3 also applies to the obtained relations to help deal with spurious invariants.[6]

4.1. Flat Array Relations

DIG finds flat (non-nested) relations among array elements of the form

$$A = b_1B^1 + \cdots + b_nB^n + c, \tag{4}$$

where A, B^i are distinct (possibly multidimensional) arrays whose elements are real-valued. The array A, called the *pivot array*, is privileged in our approach because the indices of arrays B^i and the coefficients b_i, c are hypothesized as linear expressions ranging over the indices of A. The invariant $A[i][j] = B[iN+j]$ (N is a constant) introduced at the end of Section 3.2.2 is an example of flat array relation. DIG also supports more complex relations of this form (e.g., $A[i][j] = \frac{1}{2}jB[2i+j] - (j+1)C[7i][3]+5$).

ALGORITHM 3: Algorithm for Finding Flat Array Relations

```
input  : set $V$ of array variables, set $X$ of traces
output: set $S$ of array relations of the form given in Eq. (4)
$S \leftarrow \emptyset$
// obtain linear relations among array elements
$V' \leftarrow$ genNewVars ($V$)
eqts $\leftarrow$ genInvs_eqts ($V'$,$X$,$d = 1$)
$Rs \leftarrow$ group(eqts)
if $Rs \neq \emptyset$ then
    foreach $R \in Rs$ do
        pivot $\leftarrow$ genPivot ($R$)
        exps $\leftarrow$ genLinExps (pivot)
        $s \leftarrow$ solve(exps,$R$)
        $S \leftarrow S + \{s\}$
    end
end
return $S$
```

Algorithm 3 for finding flat array relations consists of two steps: (i) identifying groups of relations among individual array elements such as $\{A[1] = B[0]+2, A[4] = 3B[7]-4\}$ and $\{C[0] = D[1], C[1] = D[2], C[2] = D[3], \ldots\}$ and (ii) analyzing these information for potential flat array relations like $C[i] = D[i+1]$ in the second group. To identify relations among array elements, we create new variables (`genNewVars`) to represent array elements, find equality relations among these array elements (`genInvs`$_{\texttt{Eqts}}$), and

[6]The *pruning* technique in Section 2.3 is unnecessary because polynomial terms are not used to find array relations.

group the obtained relations (`group`). To analyze the obtained groups for flat array relations, we represent the relations among the indices of a selected pivot array (`genPivot`) and other arrays as a parameterized linear expression (`genLinExps`), instantiate this expression with information from the obtained group of equalities, and solve these equations (`solve`).

For simplicity, the following explains the algorithm for two one-dimensional arrays, that is, $V = \{A, B\}$, although the method generalizes to multidimensional arrays.

Relations among Array Elements. We first generate a set V' of new variables representing elements of the arrays in V. Next, the technique from Section 3.1 is used to identify linear equalities of the form given in Eq. (1) over the variables in V' from the input traces X. The obtained equations represent relations among array elements (e.g., $A_4 = 3B_7 - 4$), where the variables A_4, B_7, represent the array elements $A[4], B[7]$, respectively. Currently, we do not find relations among similar arrays (e.g., $A[i] = A[2i]$), and thus keep only equations that express relations among array elements of different arrays. These relations are then grouped so that each group contains relations among elements from a same set of arrays. For example, $\{A_1 = B_0 + 2, A_4 = 3B_7 - 4\}$ and $\{C_0 = D_1, C_1 = D_2, C_2 = D_3\}$ are two different groups.

Relations among Array Indices. From each obtained group, we consider only the set R of relations of the form

$$\begin{aligned} A_{i_0} &= b_0 B_{j_0} + c_0, \\ A_{i_1} &= b_1 B_{j_1} + c_1, \\ &\vdots \end{aligned}$$

where b_x, c_x are real-valued and A_{i_x}, B_{j_x} are the variables in V' representing $A[i_x], B[j_x]$, respectively.

In such a set R, we select A as the pivot array and hypothesize that the coefficients b_x, c_x and the indices j_x of array B are linear expressions ranging over the indices i_x of A. For instance, we represent the relation between j_x and i_x through the parameterized linear expression $j_x = p_1 i_x + q_1$, where p_1 and q_1 are unknowns to be solved for. This expression is then instantiated with the information from R to obtain a system of equations $\{j_0 = p_1 i_0 + q_1, j_1 = p_1 i_1 + q_1, \dots\}$. Any solution for p and q of these equations implies a relation of the form $A[i_x] = (p_0 i_x + q_0)B[p_1 i_x + q_1] + (p_2 i_x + q_2)$, where i_x are the indices of A obtained from R.

The resulting relation $i_x \in \{\dots\} \Rightarrow r$ has a conditional form where the relation r holds only for specific indices i_x of A. Such invariants are useful and appear in many programs, for example, in the following codefragment.

```
for (i=0; i < M; ++i){
  if (i < 6){
     A[i] = [B[4*i], B[4*i+1],
              B[4*i+2], B[4*i+3]];
  }
}
[L]
```

For this code fragment, DIG generates the invariant $A[i][j] = B[4i + j]$ for $i = \{0, \dots, 5\}$ and $j = \{0, \dots, 3\}$, indicating a relation among certain elements of the arrays A and B at location L.

Example. We illustrate the algorithm by finding the relation $A[i] = 7B[2i] + 3i$ between two arrays A, B, using traces X that exhibit the relation. An example trace in X contains the values $A = [-546, -641, 34]$ and $B = [-78, 3, -92, -34, 4]$.

Eight variables are created to represent the elements of A and B. Based on the given trace, the set $R = \{A_0 = 7B_0, A_1 = 7B_2 + 3, A_2 = 7B_4 + 6\}$ of linear equations is obtained using the technique in Section 3.1. From R, we choose A as the pivot and extract the information $i_x = \{0, 1, 2\}$. The relation between j_x and i_x is expressed as $j_x = p_1 i_x + q_1$. We instantiate $j_x = p_1 i_x + q_1$ with the information from R and obtain the set of equations $\{0 = 0p_1 + q_1, 2 = 1p_1 + q_1, 4 = 2p_1 + q_1\}$. The unique solution $\{q_1 = 0, p_1 = 2\}$ of these equations yields $j_x = 2i_x$, that is, $A[i_x] = b_x B[2i_x] + c_x$. Similarly, we instantiate the analogous equations for b_x and c_x. After solving these, the array relation $i_x = \{0, 1, 2\} \Rightarrow A[i_x] = 7B[2i_x] + 3i_x$ is obtained.

Notice that all relations in R have 7 as the coefficient of B_i, thus we can divide these equations by 7 to obtain $R' = \{B_0 = \frac{1}{7}A_0, B_2 = \frac{1}{7}A_1 - \frac{3}{7}, B_4 = \frac{1}{7}A_2 - \frac{6}{7}\}$. From R', we select B as the pivot array and extract the information $i_x = \{0, 2, 4\}$. Applying the preceding process of creating and solving linear equations gives the relation $i_x = \{0, 2, 4\} \Rightarrow B[i_x] = \frac{1}{7}A[\frac{1}{2}i_x] - \frac{3}{14}i_x$. DIG can recognize such a scenario and thus is able to generate both array relations.

4.2. Nested Array Relations

DIG finds nested array relations (relations among nested array structures) of the form

$$A = B, \tag{5}$$

where the left-hand side is the pivot array A and the right-hand side is a nested array expression consisting of an array B whose indices are nested array expressions or linear expressions ranging over the indices of A. An example of this form is the nested relation $A[i][j] = B[i + 2][C[D[3j]]]$.

Algorithm 4 outlines the three steps to generate nested array relations. The first step (`genNestings`) enumerating nestings representing hypothesized nested array structures such as $A = B[C[\ldots]], B = A[C[\ldots]], \ldots$. The next step (`reachAnalysis`) applies reachability analysis to identify relations among individual array elements according to the hypothesized nesting such as $A[0] = B[C[1]], A[1] = B[C[2]], A[2] = B[C[3]]$. The last step analyzes these information for potential nested array relations like $A[i] = B[C[i + 1]]$ by encoding the problem as a satisfiability formula that can be solved using an SMT solver (`genFormula` and `SMT`). The last two steps for finding relations among array elements and nested array structures are conceptually similar to

ALGORITHM 4: Algorithm for Finding Nested Array Relations

```
input  : set $V$ of array variables, set $X$ of traces
output: set $S$ of array relations of the form given in Eq. (5)

$S \leftarrow \emptyset$
nestings $\leftarrow$ genNestings ($V$)
foreach nesting $\in$ nestings do
    $R \leftarrow$ reachAnalysis(nesting,$X$)
    if $R \neq \emptyset$ then
        $f \leftarrow$ genFormula ($R$)
        $s \leftarrow$ SMT ($f$)
        if $s \neq \emptyset$ then
            $S \leftarrow S$ + {s}
        end
    end
end
return $S$
```

those of Section 4.1 for flat array relations, but relies on reachability analysis and SMT solving instead of equation solving.

For simplicity, we illustrate these steps next using three one-dimensional arrays, that is, $V = \{A, B, C\}$, although the algorithm generalizes to multidimensional arrays.

Nestings. We first enumerate nested structures among the arrays in V. A nested array structure, or *nesting*, from a set V of arrays is a tuple (P, S), where P is an array in V designated as the pivot and S is a nonempty and nonrepeating[7] sequence of arrays in V that does not contain the array P. For the input $V = \{A, B, C\}$, we generate the nestings $(A, [B]), (A, [C]), \ldots, (C, [B, A])$.

Reachability Analysis. A nesting $(A, [B, C])$ implies the relation $A[i] = B[C[k]]$, where elements of the pivot array A are related to elements of B using elements of C as indices into B. For such a relation to hold, the elements of A must be in B. Moreover, the indices of B, where the elements of A appear in, must also be in C. Reachability analysis is our method to determine how the elements of A are related to the elements of B using C as indices into B.

The analysis could start by checking if all elements of A are in B. However, this naïve approach has an exponential complexity when the elements of A occur multiple times in B. Instead, reachability analysis can be done in polynomial time (Section 5.1) as follows.

We arbitrarily choose two distinct elements $A[x] \neq A[y]$ from the pivot array A (the reason for using two elements will be justified). For $A[x]$, we find the indices j_x in B, where $B[j_x] = A[x]$. For each of the obtained indices j_x in B, we again find the indices k_x in C, where $C[k_x] = j_x$. We then form a set of relations of the form $A[x] = B[C[k_x]]$ from these results, which indicate that the element $A[x]$ is related to elements of B using elements $C[k_x]$ as indices into B. Repeating this process for $A[y]$, we obtain a set of relations of the form $A[y] = B[C[k_y]]$. Each set R from the cross product of the two sets of relations consists of two equations of the form $\{A[x] = B[C[k_x]], A[y] = B[C[k_y]]\}$.

Note that the relation $A[i] = B[C[k]]$ is determined invalid if any of the preceding checks fails (e.g., $A[x]$ is not in B or the obtained indices j_x of B are in C). We can further optimize this algorithm by starting with two distinct elements of A that occur least often in B. However, such a greedy approach does not guarantee the smallest number of relation sets generated at the end because the indices j_x of B can occur many times in C.

Relations among Array Indices. From a set $R = \{A[x] = B[C[k_x], A[y] = B[C[k_y]]\}$ of relations obtained from reachability analysis, we determine the relation between the indices of A and C. This step is conceptually similar to that of Section 4.1 in which the relation between the indices i, k of arrays A, C is represented by the parameterized linear expression $k = ip + q$.

Instantiating $k = pi + q$ with the information from R, we get a system of two equations $\{k_x = xq + q, k_y = yp + q\}$. The solution for p, q of these equations gives a relation of the form $A[i] = B[C[pi + q]]$, for $i = \{x, y\}$. We now verify that this relation also holds for other indices i of A (instead of just x, y). If it is verified, we return it as the candidate invariant. Otherwise, we repeat this step on another set R of relations to find a different nested array relation.

A relation of the form $A[i] = B[C[k]]$ that holds for all indices i of A must also hold for the two indices (x, y). Thus, we can find such a relation, if it exists, by trying all possible sets R of relations generated by reachability analysis on the two elements

[7]The nonrepeating constraint is used to enforce finite depth in the sequence.

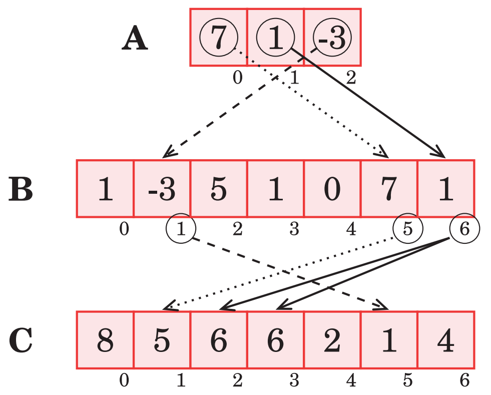

Fig. 5. Reachability analysis showing $A[0] = B[C[1]]$ (dotted), $A[1] = B[C[2]] \vee B[C[3]]$ (solid), and $A[2] = B[C[5]]$ (dashed).

$A[x]$, $A[y]$. Moreover, it is sufficient to apply the analysis on two distinct[8], instead of all, elements of the one-dimensional A is because only two independent equations of the form $k = ip + q$ are needed to solve for the two unknowns p, q. In generally, we apply reachability analysis on a tuple of $d+1$ elements of a d-dimensional array A because the relation among the indices of A and C is represented by a linear expression consisting of $d + 1$ unknowns $p_1, \ldots, p_d, q$.

Example. We demonstrate the algorithm by finding the relation $A[i] = B[C[2i + 1]]$ from the trace $A = [7, 1, -3]$, $B = [1, -3, 5, 1, 0, 7, 1]$, and $C = [8, 5, 6, 6, 2, 1, 4]$. Figure 5 illustrates reachability analysis on the three elements of array A over the nesting $(A, [B, C])$.

Among the nestings generated from the input $V = \{A, B, C\}$, those representing relations such as $B[i] = C[\ldots]$ are ruled out immediately because the element -3 of B is not in C. Note that the use of traces is essential here, as it allows us to quickly filter out invalid nestings. For the nesting $(A, [B, C])$, we apply reachability analysis on two arbitrarily chosen elements $A[1]$ and $A[2]$ of A. For $A[1]$, the analysis generates $\{A[1] = B[C[2]], A[1] = B[C[3]]\}$ because $A[1] = B[0], B[3], B[6]$ and $6 = C[2], C[3]$ (the index values $0, 3$ of B do not occur in C). For $A[2]$, we obtain the set $\{A[2] = B[C[5]]\}$ because $A[2] = B[1]$ and $1 = C[5]$. The cross product of these two sets yields the sets $R_1 = \{A[1] = B[C[2]], A[2] = B[C[5]]\}$ and $R_2 = \{A[1] = B[C[3]], A[2] = B[C[5]]\}$ of relations.

The information from either set R_1 or R_2 suggests the possibility of a nested relation $A[i] = B[C[k]]$ for $i = \{1, 2\}$ and k is the parameterized linear expression $k = pi + q$. Instantiating $k = pi + q$ with the information from R_1 gives two equations $\{2 = p + q, 5 = 2p + q\}$. The unique solution $\{p = 2, q = -1\}$ for these equations yields the relation $A[i] = B[C[3i - 1]]$, where $i = \{1, 2\}$. This relation does not hold for all indices of A (e.g., $(A[0] \neq B[C[-1]])$, and is thus disregarded. Next, we instantiate $k = pi + q$ with the information from R_2 and obtain the equations $\{3 = p + q, 5 = 2p + q\}$. The unique solution $\{p = 2, q = 1\}$ for these yields the relation $A[i] = B[C[2i + 1]]$, for $i = \{1, 2\}$. This relation holds for all indices of A and therefore is returned as the candidate invariant.

[8]If all elements of A are the same (or A has only one element), we apply reachability analysis on $A[0]$ and obtain a set of equations of the form $A[0] = B[C[k]]$. Any of these equations is a candidate relation because any relation that holds for $A[0]$ also holds for other elements of A.

Satisfiability Problem Formulation. In practice, arrays often have large sizes with multiple duplicate elements, causing reachability analysis to generate many sets R of relations to be solved for. Hence, we encode the results of reachability analysis as a satisfiability formula in the theory of linear integer arithmetic, which can be solved efficiently with modern SMT technologies [Dutertre and De Moura 2006].

Returning to the running example, we create a clause consisting of two atoms $(2 = p + q \;\vee\; 3 = p + q)$ to represent the result $\{A[1] = B[C[2], A[1] = B[C[3]]\}$ from reachability analysis. Similarly, the atom $5 = 2p + q$ is created for $\{A[2] = B[C[5]]\}$. Since the relation should hold for the two chosen elements of A, that is, $A[i] = B[C[pi + q]]$ for $i = \{1, 2\}$, we combine these formulas into the final CNF formula $f = (2 = p + q \vee 3 = p + q) \wedge (5 = 2p + q)$. Next, we query the SMT solver to return, if possible, an assignment of integers (since array indices are integers) to the variables p and q that satisfies f. In this example, the solver might assign $p = 3, q = -1$ for f, which implies the relation $A[i] = B[C[3i - 1]]$ for $i = \{1, 2\}$. This relation cannot be verified because it does not hold for all indices of A (e.g., $A[0] \neq B[C[-1]]$). We then add the constraint $\neg(p = 3 \wedge q = -1)$ to f and query the SMT solver for a new assignment for p, q. The solver now assigns $p = 2, q = 1$, implying the relation $A[i] = B[C[2i + 1]]$. This relation is verified to hold for all indices of A and thus is returned as the candidate invariant.

We can avoid having to verify each relation by applying the analysis on all elements of A. Doing so for the running example results in the CNF $f = (1 = q) \wedge (2 = p + q \vee 3 = p + q) \;\wedge\; (5 = 2p + q)$ (the atom $1 = q$ represents the relation $A[0] = B[C[1]]$, as illustrated in Figure 5). The solution $\{p = 2, q = 1\}$, returned by the SMT solver on the formula f, implies the similar relation $A[i] = B[C[2i + 1]]$ as before. Moreover, this relation is valid for all elements of A because the analysis is applied on all of those elements. Thus, we only need to invoke the solver once, but over a more complex formula f (the number of clauses in f is the size of A).

The problem of finding nested array relations has a polynomial-time complexity (by the algorithm and the analysis given in Section 5.1). However, our implementation in DIG for nested array relations involves SMT technologies and hence does not guarantee a polynomial runtime.[9] The experimental results on finding nested arrays in Section 6 were obtained when applying reachability analysis on all elements of A.

4.3. Functions

Array invariants involving user-defined functions, for example, $A[i] = f(C[i], g(D[i]))$, require special treatment. We view a function f with n arguments as an n-dimensional array F, where the element $F[i_1] \ldots [i_n]$ contains the output of $f(i_1, \ldots, i_n)$. Thus, if f is the `mult` function, then $F[4][7] = F[7][4] = 28$. For efficiency, F is represented as a partial array that stores only observed values. For example, if $A = [4, 7]$ and $B = [5]$ are considered, then F contains just the elements $F[4][4], F[4][5], F[4][7], \ldots, F[7][7]$. Our approach extends to invariants involving function composition, such as $g(f(A[\ldots], B[\ldots]))$. For instance, if g is $\texttt{mod}_2$ which maps even and odd inputs to 0 and 1 respectively, then the corresponding array G has as its indices the elements of A, B, F (e.g., $G[4] = G[28] = 0, G[5] = G[7] = 1$). Just like with array nesting, we enforce finite depth in nested expressions by disallowing a function to appear in the scope of one of its arguments, for example, $g(f(g(\ldots), f(\ldots)))$ is not allowed.

DIG predefines a set of basic functions such as `mult`,`add`,`xor`,`mod`,... and automatically generates the corresponding partial arrays based on given traces, as previously. Once

[9]This depends on the technique implemented in SMT solvers for satisfiability checking over CNFs of the discussed form.

Table I. Time Complexity of Invariant Generation Algorithms in DIG

Invariant Type	Form	Complexity
Equality	(1)	$O(\|T\|^3)$
Polynomial (general) inequality	(2)	$O(\|X\|^{\frac{\|T\|}{2}})$
Octagonal inequality	(3)	$O(\|X\|\|T\|^2)$
Flat array relation	(4)	$O(\|E\|^3)$
Nested array relation	(5)	$O(\|X\|\|V\|!\|E\|^{d\|V\|})$

Note: For polynomial invariants, T represents the set of terms and X the set of traces. For array invariants, V represents the set of array variables, E the set of array elements, d the highest array dimension among the arrays in V.

functions are included as arrays, DIG can then generate invariants involving functions, such as the nested array relation $R[i] = T(\texttt{mod255}(\texttt{add}(L(A[i]), L(B[i]))))$ in the `multWord` function in AES.

5. ANALYSIS

We first give the computational complexity of DIG's algorithms for generating different forms of invariants. Next we show that the polyhedral method generates precise inequality invariants but can also give many spurious results if the program invariants do not appear in the traces.

5.1. Complexity

Table I summarizes the time complexity of DIG's algorithms for generating invariants of different forms. The filtering technique in Section 2.3 takes $O(|X||T|)$ to instantiate and check a candidate invariant with $|T|$ terms over $|X|$ traces.

5.1.1. Polynomial Invariants. We analyze the complexity of the algorithms for generating polynomial invariants in the number of traces $|X|$ and terms $|T|$. Recall that terms are used to represent polynomials over variables (Section 2.2). Given a set V of variables and a degree d, the set T of terms representing monomials over V up to degree d has size $\binom{|V|+d}{d}$. The number of terms thus increases exponentially in the number of variables and degrees.

For equalities of the form given in Eq. (1), a standard equation solver is used to find equality invariants in Section 3.1. We use the traces in X to instantiate $|T|$ independent equations. The complexity of using Gaussian elimination to solve $|T|$ linear equations for $|T|$ unknowns is $O(|T|^3)$ [Farebrother 1988]. Hence, generating invariants representing equations among $|T|$ terms takes $O(|T|^3)$, cubic in the number of terms.

In practice, the number of traces often exceeds the number of terms, that is, $|X| \gg |T|$, which is desirable because the complexity depends on the smaller parameter $|T|$. The current implementation of DIG uses a small random subset of the input traces to generate equations among terms.

For general inequalities of the form given in Eq. (2), we build polyhedra in Section 3.2.1 to obtain polyhedral (general) inequalities. Constructing a convex polyhedron over $|X|$ points in $|T|$ dimensions has a theoretical exponential upper bound $\Theta(|X|^{\lfloor\frac{|T|}{2}\rfloor})$ [de Berg et al. 1997]. Thus, the cost of generating polyhedral inequalities is $O(|X|^{\frac{|T|}{2}})$, exponential in the number of terms (because a term is essentially a new variable representing a new dimension).

If one is interested only in inequalities among a fixed number c of terms over program variables, then the heuristic described in Section 3.2.1 builds polyhedra for all term combinations of size $|c|$. The complexity of such a heuristic is $O(\binom{|T|}{c}|X|^{|c|})$, which is polynomial in $|X|$ and $|T|$ because c is fixed.

For octagonal inequalities of the form given in Eq. (3) representing relations between two terms, we instantiate each pair of terms with the traces in X to obtain the set of $|X|$ points in two dimensions and apply the min, max operations on these points as shown in Section 3.2.2. These two operations run in linear time in $|X|$, thus identifying the octagonal constraints for each pair of terms takes $O(|X|)$. There are $O(|T|^2)$ such pairs from the set of terms T, hence generating octagonal constraints for all pairs of terms takes $O(|X||T|^2)$.

5.1.2. Array Invariants. The complexity of the algorithms for generating array invariants is analyzed in terms of the number of traces $|X|$, array variables $|V|$, array elements $|E|$ consisting of elements from all arrays in A, and the highest dimension d among the arrays.

The complexity of the algorithm to find flat array relations of the form given in Eq. (4) is dominated by solving equations. As described in Section 4.1, we create $|E|$ new variables to represent array elements and use the equation solving technique in Section 3.1 to find equalities among them. As previously analyzed, generating equalities among these variables (terms) takes $O(|E|^3)$, the time of solving $|E|$ equations for $|E|$ unknowns.

For nested array relations of the form given in Eq. (5), reachability analysis is applied on the $|V|!$ nestings enumerated among the arrays in V. For a nesting representing the relation $A[i] = B_1[\dots[B_l[ip+q]]\dots]$, we apply the analysis on two arbitrarily chosen elements $A[i_x], A[i_y]$ of the one-dimensional array A. In the worst case, $A[i_x]$ could occur $O(|B_1|)$ times at B_1. Each index value of these $O(|B_1|)$ locations could again occur $O(|B_2|)$ times at B_2. Thus, the analysis generates $O(|B_1|\cdots|B_l|)$ relations of the form $A[i_x] = B_1[\dots[B_l[i_x p+q]]\dots]$ at B_l. This is $O(|E|^{|V|})$, because $|B_i| < |E|$ and $l < |V|$. Similarly, we obtain $O(|E|^{|V|})$ relations for $A[i_y]$. The cross product of these two sets results in $O(|E|^{2|V|})$ sets of relations, each set has two equations and two unknowns. More generally, we apply the analysis on $d+1$ elements of a d-dimensional A and thus obtain $O(|E|^{d|V|})$ sets of relations. Observe that we obtain $O(|A|^{|E||V|})$ sets of relations if the analysis is applied on all elements of $|A|$ and thus the algorithm becomes exponential in the number of array elements ($|A| = O(|E|)$).

From each set of relations, a system of $d+1$ equations is instantiated and solved for the $d+1$ unknowns to find relations among array indices. Doing this for $O(|E|^{d|v|})$ sets of relations takes $O(d^3|E|^{d|v|})$, which becomes $O(|E|^{d|v|+3})$ since $|E| \geq d$. Finally, assuming array indexing is $O(1)$, the verification that a nested relation $A[i] = B_1[\dots[B_l[k]]\dots]$ holds for all indices of A takes $O(l|A|)$. To be comprehensive, we check the candidate relation over the traces in X and hence verification takes $O(l|A||X|)$.

The algorithm given in Section 4.2 is thus $O(l|A||X||V|!|E|^{d|V|+3})$, which is $O(|X||V|!|E|^{d|V|+3})$, because $|E| > l|A|$. Moreover, we can fix $|V|$ and d because, in practice, the number of array elements is typically much larger than the number of arrays or the array dimensions. Hence, the complexity of finding nested array relations is polynomial in the number of array elements $|E|$.

5.2. Polyhedra and Inequalities

The polyhedral method described in Section 3.2.1 merits additional discussion because it generates precise inequalities that guaranteed to underapproximate the desired invariants[10] expressible under the considered inequality forms. However, if the desired

[10] *Desired* invariants refer to program invariants, that is, relations that are guaranteed to hold at a program location for all possible traces observed at that location.

invariants do not fall under the considered forms, this method could generate a complex polyhedron whose facets represent many spurious invariants.

Underapproximation. A dynamically inferred invariant could either be equivalent to underapproximate (i.e., be a spurious invariant that is too strong and does not always hold) or overapproximate (i.e., be too weak and possibly not useful) the desired invariant. For instance, when the template $x \leq y$ is used to infer the desired invariant $x \leq y - 10$, then this template, an overapproximation of the desired invariant, is returned as the candidate invariant. This section shows that this overapproximation situation cannot happen in DIG. More precisely, assuming the desired invariant belongs to an inequality form supported by DIG, then a candidate inequality generated from DIG using convex hulls can only be equivalent to or underapproximates the desired invariant. This property is useful because its falsification, that is, the inferred invariant (strictly) overapproximates the desired one, indicates that the desired invariant fails for some observed traces and thus the program has a bug. For example, consider the `flatten` code in Section 3.2.2 with an off-by-one error.

```
for (i = 0; i < M; ++i){
  //bug, should be j < N
  for (j = 0; j <= N; ++j){
    k  = i * n + j;
    [L]
    B[k] = A[i][j];
  }
}
```

Depending on the given traces, DIG may generate at L the octagonal relation $0 \leq k \leq MN + 5$, which is an overapproximation of the desired invariant $0 \leq k \leq MN - 1$. This indicates an error because DIG would never generate such a relation unless the value $k = MN + 5$ is in the traces, that is, a counterexample that violates the desired invariant.

The proof of the underapproximation property is relatively straightforward, using the facts that a convex hull of a set of points is the smallest convex set containing those points and that the observed traces are a subset of all possible traces. Formally, let F be the desired invariant of a shape considered in this article (i.e., a conjunction of inequalities representing a bounded convex object in multidimensional Euclidean space), then our candidate invariant F' of that shape is equivalent to or underapproximates F, that is, $F' \Rightarrow F$. To see this, observe that the object represented by F encloses all trace points and the object of the same shape represented by F' encloses a subset of all trace points. Moreover, because F' is computed as the convex hull of that subset of trace points, the object represented by F' is enclosed in the object represented by F. Thus, $F' \Rightarrow F$.[11]

Observe that equivalence is achieved when the input traces consist of the extreme points describing the desired shape. For instance, we can find the exact inequalities representing an octagon from any set of traces consisting of the eight extreme points of that octagon.

We note that the underapproximation property also holds for equalities generated from DIG, as proved in Sharma et al. [2013].

[11]The underapproximation property $F' \Rightarrow F$ also holds if the shape of F' is more precise than the shape of F. This property is not guaranteed if the shape of F' is less precise than the shape of F. In Figure 3, a desired invariant representing an octagon (Figure 3(c)) is an overapproximation of a candidate invariant representing a polygon (Figure 3(d)) and is an underapproximation of a candidate invariant representing an interval (Figure 3(b)).

Spurious Invariants. The polyhedral method has a high theoretical complexity because it could produce a complex polyhedron with multiple facets in high dimensions depending on the given trace points. Importantly, if the traces do not precisely capture the desired invariant, then the polyhedron consists of many facets representing spurious inequalities. For instance, if x, y can take any value over the reals, then an n-facet polygon computed over any set of traces for x, y produces n spurious invariants because no bounded polygons can capture the unbounded ranges of x, y.

Although filtering (Section 2.3) reduces spurious invariants by removing facets of the polyhedron (i.e., widening it), the modified polyhedron can still have many remaining facets representing faux relations because it is rare to have inequalities among all involved terms. Thus, DIG does not automatically invoke the polyhedral method for general inequalities. The method is effective when the user has certain expectations about the desired invariants. The user can ask DIG for octagonal relations if only inequalities among pairs of terms over program variables are of interest. The user can also hypothesize a spherical shape $c_1x^2 + c_2y^2 +_c 3z^2$ and query DIG to search for that exact sphere (i.e., compute the coefficients c_i) from the convex hull built over trace points for these terms. The availability of source code allows for hybrid approaches with static analysis such as *slicing* [Reps et al. 1995] to automatically find variables that are likely related to one another, reducing the number of term combinations.

In contrast to the convex hull construction, methods using equation solving give few spurious equalities, because equalities are stricter constraints than inequalities. For example, we can always compute a convex polygon representing many inequalities over any set of finite points in 2D, but can only have at most a line representing an equality over these points. Moreover, assuming traces are obtained from random program inputs, it is unlikely that a large set of traces would exhibit random false equalities. The next section shows that DIG does not generate spurious equality relations for both numerical and array variables in our experiments.

6. EXPERIMENTAL RESULTS

Our prototype, DIG, is implemented in Python using the Sage mathematical environment [Stein 2012]. The prototype uses built-in Sage functions to solve equations and construct polyhedra. It also uses Z3 [De Moura and Bjørner 2008] to check the satisfiability of SMT formulas. The experiments reported here were performed on dual-core 2.3GHz Intel Unix-based system with 8GB of RAM.

6.1. Programs

We evaluated DIG on programs taken from a test suite which we call NLA (nonlinear arithmetic) and an implementation of the Advanced Encryption Standard (AES) [Rijmen and Daemen 2001]. The details of NLA and AES are given in Tables II and III, respectively.

The NLA test suite consists of 27 programs from various sources collected by Rodríguez-Carbonell and Kapur [Carbonell and Kapur 2007a, 2007b; Carbonell 2006]. These programs implement classic arithmetic algorithms that are widely used in programming, such as `mult, div, mod, sqrt, gcd`. The programs are relatively small, about 20 lines of `C` code each. However, they implement nontrivial mathematical algorithms and are often used to benchmark static analysis methods. Importantly, the complexity of our method depends on the size of the traces and the invariant forms of interest—not the size of the program per se. Among the 27 programs from NLA, there are 41 documented nonlinear relations: 39 equations and 2 inequalities.

The second benchmark, AES, is an annotated AES implementation from Yin et al. [2009]. It exemplifies a real-world security-critical application and contains nontrivial array invariants. To show that the implementation conform to the formal AES specification, the authors of AES inspected and documented the invariants of each function in AES and then fully verified the result using SPARK Ada [Barnes 2003] and PVS [Owre et al. 1992]. The annotated invariants represent the manual effort required to fully verify the functionality of an AES implementation using axiomatic semantics. AES contains 868 lines of Ada code organized into 25 functions containing 30 invariants: 8 flat array relations, 7 nested array relations, 2 linear equations, and 13 other relations.

Program Locations and Execution Traces. Our test programs come with documented invariants at various locations such as loop heads and function exits. For evaluation purpose, we find invariants at those locations automatically and compare them to the human-documented invariants. We manually instrumented the program source code to trace values of all variables in the scope at each program location containing a known invariant. Specifically, for NLA, invariants are obtained mainly at loop entrances. For AES, invariants are obtained mainly at function exits. For NLA, it turns out that most of the loop invariants specify the behaviors of the programs (e.g., the `egcd` program in Figure 1 and `cohen` in Figure 4). In addition, we apply DIG over traces captured at systematically chosen program locations (e.g., all function entries) and verify its results at these locations manually.

The instrumented programs were run against a set of randomly selected inputs. The number of obtained traces is different across programs and program locations. For example, locations inside loops may be visited many times while function exits may be visited rarely. DIG automatically selects a set of random traces whose size is 1.5 times the number of created terms for invariant generation and another set of 1,000 random traces for filtering.

6.2. Quality of Results

DIG selects the appropriate algorithms for finding invariants depending on the variables that appear in the trace file. For numerical variables, DIG first generates equalities among terms, and next proceeds to inequalities using the deduction method when additional information such as loop guards is available. By default, we do not generate inequalities using the convex hull methods unless specified by the user. For array variables, DIG first generates flat relations and then nested relations. For invariants involving user-defined functions, this information must be specified by the user because it is not available from traces. The post-processing step refines the candidate invariants when each invariant generation algorithm finishes.

For the experiments reported here, we define a single parameter, $\alpha = 200$ which bounds DIG's running times. For polynomial invariants, DIG automatically adjusts the maximum degree so that the number of generated terms does not exceed α. For flat array relations, DIG automatically adjusts the sizes of the considered arrays in such a way that the total number of array elements does not exceed α. There is no parameter for nested array relations because reachability analysis enumerates all possible non-repeating nestings to consider array relations up to any nesting depth.

6.2.1. NLA. Table II reports experimental results on 27 programs from NLA, with running times averaged over 20 runs. The *Invs* column reports the number of nonlinear invariants generated by DIG and their types (equality or inequality). The *V, D* column reports the number of distinct variables (a subset of the considered variables) and the highest polynomial degree in the generated invariants. The T_α column reports the average time in seconds to discover the invariants using parameter α, including the time to refine the results. The T_{deg} column reports the average time in seconds to discover these invariants if we had restricted the search space to the polynomial degree given in the *V, D* column. The *Vs Doc* column reports the number of documented invariants

Table II. Experimental Results on 27 Programs from NLA

Program	Desc	Invs	V, D	T_α	T_{deg}	Vs Doc
divbin	div	1 eq	5, 2	27.0	0.3	1/1
cohendiv	div	2 eq, 2 ieq	6, 2	8.5	0.8	3/3
mannadiv	int div	1 eq	5, 2	23.1	1.0	1/1
hard	int div	2 eq	6, 2	9.2	0.9	2/2
sqrt1	square root	2 eq, 2 ieq	4, 2	39.6	0.9	3/3
dijkstra	square root	4 eq	5, 3	31.4	4.0	1/1
freire1	square root	1 eq	3, 2	33.9	0.2	1/1
freire2	cubic root	4 eq	4, 2	45.5	2.9	2/2
cohencube	cubic sum	3 eq	5, 3	39.5	8.4	3/3
egcd	gcd	3 eq	8, 2	53.9	2.2	2/2
egcd2	gcd	3 eq	10, 2	4.2	4.5	3/3
egcd3	gcd	4 eq	12, 2	8.2	8.8	4/4
lcm1	gcd, lcm	1 eq	6, 2	8.7	0.5	1/1
lcm2	gcd, lcm	1 eq	6, 2	11.5	0.7	1/1
prodbin	product	1 eq	5, 2	32.5	0.4	1/1
prod4br	product	1 eq	6, 3	8.2	8.5	1/1
fermat1	divisor	1 eq	5, 2	31.6	0.4	1/1
fermat2	divisor	1 eq	5, 2	31.6	0.4	1/1
knuth	divisor	4 eq	8, 3	53.8	53.8	1/1
geo1	geo series	1 eq	4, 2	14.8	0.1	1/1
geo2	geo series	1 eq	4, 2	24.4	0.1	1/1
geo3	geo series	1 eq	5, 3	23.4	2.4	1/1
ps2	pow sum	1 eq	3, 2	29.2	0.2	1/1
ps3	pow sum	1 eq	3, 3	27.9	0.3	1/1
ps4	pow sum	1 eq	3, 4	29.7	0.8	1/1
ps5	pow sum	1 eq	3, 5	30.2	2.8	1/1
ps6	pow sum	1 eq	3, 6	28.2	8.5	1/1
27 programs		52 invs		709.2s	114.7s	41/41

matched by our results. For example, DIG generated four quadratic relations (two equalities and two inequalities) over six variables for `cohendiv` in 8.5s (or 0.8s if the maximum degree 2 was specified). The obtained invariants also match with the three documented invariants.

To evaluate DIG, we compared its results to the documented invariants or verified them manually against the program source code. Comparing to the documented invariants, DIG found all 41 documented relations from the 27 programs in NLA. In most cases, the results matched the documented invariants exactly as written. Occasionally we achieved results that are mathematically equivalent to the documented invariants. For example, `sqrt1` has two documented equalities $2a + 1 = t, (a + 1)^2 = s$; our results gave $2a + 1 = t, t^2 + 2t + 1 = 4s$, which is equivalent to $(a + 1)^2 = s$ by substituting t with $2a+1$. In many cases, DIG discovered undocumented invariants, for example, the relation $1 = im - jk$ in the example program `egcd` in Section 2. We also found invariants that are stronger than the documented ones, for example, the two quadratic relations discovered in `freire2` imply a documented relation of degree 3 (DIG also found this cubic invariant but automatically pruned it in post processing because it is weaker than the other two). We obtained no spurious results for NLA using the method based on equation solving. We believe and intend to formally prove that our method for generating equality invariants has this desirable property. That is, assuming random and adequate traces (e.g., $\geq n$ unique traces for n unknowns), the method will generate only true *equality* invariants. Note that this property will not hold for *inequalities*, because

Table III. Experimental Results on 25 Functions from AES

Function	Desc	Invs	V, D	T_α	Vs Doc
multWord	mult	1 N_4	7, 2	11.0	1/1
xor2Word	xor	3 N_1	4, 2	0.8	1/1
xor3Word	xor	4 N_1	5, 3	2.0	1/1
subWord	subs	2 N_1	3, 1	1.3	1/1
rotWord	shift	1 F	2, 1	0.5	1/1
block2State	convert	1 F	2, 2	4.1	1/1
state2Block	convert	1 F	2, 2	4.2	1/1
subBytes	subs	2 N_1	3, 2	3.2	1/1
invSubByte	subs	2 N_1	3, 2	3.3	1/1
shiftRows	shift	1 F	2, 2	3.7	1/1
invShiftRow	shift	1 F	2, 2	3.6	1/1
addKey	add	2 N_1	4, 2	3.5	1/1
mixCol	mult	0	-	1.0	0/1 O_3
invMixCol	mult	0	-	1.0	0/1 O_3
keySetEnc4	driver	1 F	2, 2	76.4	1/2 O_2
keySetEnc6	driver	1 F	2, 2	78.8	1/2 O_2
keySetEnc8	driver	1 F	2, 2	79.3	1/2 O_2
keySetEnc	driver	1 F	2, 1	76.3	0/1 O_3
keySetDec	driver	0	-	73.0	0/1 O_3
keySched1	driver	0	-	77.9	0/1 O_1
keySched2	driver	1 F	2, 2	79.5	0/1 O_1
aesKeyEnc	driver	1 F, 1 eq	2, 1	76.2	1/2 O_3
aesKeyDec	driver	1 eq	2, 1	73.6	1/2 O_3
aesEncrypt	driver	1 F	2, 2	70.5	0/1 O_3
aesDecrypt	driver	1 F	2, 2	73.8	0/1 O_3
25 functions		30		878.5s	17/30

the convex hull method would likely generate many spurious polyhedral inequalities as discussed in Section 5.2.

As shown in column T_{deg}, the runtime of DIG can be improved significantly by limiting the search to invariants of a given maximum degree. For instance, DIG took 2.2s to find the three quadratic relations in `egcd` using maximum degree 2, but it took 53.9s to find the same relations using the parameter $\alpha = 200$, which queries DIG for all invariants up to degree 3 this program. The large difference in runtime is because the number of terms has an exponential dependency on the degree, for example, the number of terms created over eight variables is 45 for degree 2 and 165 for degree 3. The approach using a maximum degree follows that of earlier work [Carbonell and Kapur 2007b; Sharma et al. 2013; Nguyen et al. 2012], which assumes a default upper bound on degree (e.g., $d = 2$) to improve performance. The uniform parameter α, used in this experiment, takes more time, but it allows DIG to generate invariants for all programs in NLA without a priori knowledge of specific polynomial degrees.

DIG supports nonlinear invariants that are not in other dynamic analysis tools. Conversely, Daikon has discovered useful properties not supported by DIG. For instance, the built-in PowerOfTwo template allows Daikon to find an interesting property in `egcd3`, which states that the values of x are always a power of 2. In several occasions Daikon also identified the relation $x \% y = 0$, that is, y divides x, due to its built-in templates involving the modulus operator.

6.2.2. AES. Table III reports experimental results on 25 functions from AES and has similar format as that of Table II. The *V, D* column reports the number of distinct array variables and the highest dimension of the arrays in the candidate invariants. The types of the generated invariants, reported in the *Invs* columns, include Flat, Nested, and linear equality invariants (N_l indicates that the depth of the generated nested array relation is l). The *Vs Doc* column also indicates the types of documented invariant (called *O*thers) that DIG could not identify. The *driver* functions are composed from other functions in this table.

DIG discovered 30 candidate invariants for AES, all of which are valid relations, using the parameter $\alpha = 200$ as in the NLA experiment. Comparing to the documented invariants, we found all 17 documented relations that are expressible in the considered forms. In many cases, DIG also discovered undocumented invariants. In the three relations $r[i] = xor(a[i], b[i]), a[i] = xor(r[i], b[i]), b[i] = xor(r[i], a[i])$ obtained from `xor2Word`, the first one is a documented invariant but the other two are indeed properties of the *xor* operator. In addition to the documented invariant $r[i][j] = S[t[i][j]]$ in `subWord`, we found the relation $t[i][j] = Si[r[i][j]]$, which is valid because the array Si contains the reversed values of the array S. The results generated for the `keySetupEnc` functions are conditional invariants, for example, in `keySetupEnc8`, we got $r[i][j] = k[4i + j]$ for $i = 0, \ldots, 7, j = 0, \ldots, 3$ and $r[i][j] = 0$ for other indices i, j. In several cases, the algorithms for both flat and nested array relations discover similar invariants such as $S[i][j] = R[4i + j]$ in `state2Block` because these flat array relations are also nested array relations with nesting depth 0. We did not obtain spurious invariants for the functions in AES.

Similar to NLA, DIG's runtime for AES can be significantly improved with some additional information about the desired invariants. For instance, DIG found the relation $t[i][j] = c[4i + j]$ in `aesDecrypt` under 10s using reachability analysis but over 60s using the algorithm for flat array relations (for solving 200 equations). The last eleven functions in Table III has similar run times (77s on average) because the considered arrays in each function were automatically resized to contain $\alpha = 200$ elements. We note that a smaller parameter value is also sufficient to obtain similar results for AES with much shorter runtime, for example, DIG took on average 14.3s for the last eleven functions when run with $\alpha = 50$.

The 13 documented invariants that were not discovered fall into categories that are not supported by DIG and are left for future work. These can be grouped into three categories: Others$_{1-3}$. Others$_1$ includes nested array relations such as $A[i] = 4B[6C[\ldots]]$. We do not currently handle nested invariants if the elements of A are not exactly nested in B. Others$_2$ includes nested array invariants such as $A[i] = B[C[\ldots]]$ and $A[j] = B[C[\ldots]]$, where $i \neq j$, that is, a conditional form of nested array relations. We require that generated relations such as $A[i] = B[C[\ldots]]$ hold for all i. Others$_3$ includes array invariants involving functions whose inputs are arrays, such as $f([1, 2])$. We only consider functions with scalar inputs such as $g(7, 8)$. We note that existing dynamic analysis methods cannot find these array relations either.

The manual annotation of AES with sufficient invariants to admit machine-checked full formal verification was a significant undertaking involving hours of tool-assisted manual effort [Yin et al. 2009, 2008]. Annotating pre- and post-conditions and loop invariants has not been solved in general and is known to be a key bottleneck in approaches based on axiomatic semantics [Flanagan and Leino 2001]. It is not surprising that our approach was unable to discover all relevant invariants; indeed, we view reducing the manual verification annotation burden by one-half as a strong result.

6.2.3. Other Program Locations. DIG can be applied over traces captured at arbitrary locations, including those with invariants that are not currently considered or those that do not likely have useful invariants. Table III shows that DIG generated ten valid invariants and no spurious results when applied to locations in AES with invariants under the unsupported forms Others$_{1-3}$. When being applied to systematically chosen locations like all program entry points in NLA, DIG generates no invariants because these programs take in random input values, for example, $-\infty \le x \le \infty$ (an unsupported form of invariant that cannot be captured by finite traces). Note that if we had used the convex hull methods to generate inequalities, then we will get spurious relations such as $c_1 \le x \le c_2$, where c_1, c_2 correspond to the lower and upper bounds of x from the input traces. In general, we expect DIG's candidate invariants at arbitrary locations from any program to have similar quality as those reported here (these invariants were obtained without a priori knowledge about their forms).

To summarize, DIG found all of the invariants under consideration: 100% of the documented nonlinear invariants in NLA and 17 out of 30 documented invariants in AES. The other 13 invariants were beyond the scope of this article and are left for future work. To the best of our knowledge, no other dynamic invariant analysis approaches have analyzed the forms of invariants discussed in this article.

6.3. Limitations

Two of the operations, namely, reachability analysis and polyhedra construction, are expensive, because the analysis can generate many relation sets over large arrays and many inequalities representing complex polyhedra in high dimensions. We address these issues using SMT solvers by deducing new inequalities through loop conditions, or by generating simpler invariants by building octagons. Nonetheless, these techniques have trade-offs: having high theoretical complexities, requiring additional information, or loosing precisions by representing invariants with simpler shapes.

Floating point values are subject to round-off errors that may confound some exact checks. Given an abundance of available traces, we filter out certain traces containing rounded float values. DIG also allows comparison within ε instead of exact comparisons (e.g., $0.33333 \approx \frac{1}{3}$ and $0.99998 \approx 1.0$). The use of established techniques from numerical analysis to handle other corner cases in floating point arithmetic is left for future work.

As discussed in Section 5.2, the effectiveness of our method depends on program traces produced by test inputs. DIG cannot derive properties that are not exhibited by the traces. For example, if the relation $x + y > 10$, but not $x + y = 10$, is implicated by the traces, then we cannot find the inequality $x + y \ge 10$. We note the existence of many active research projects investigating high-coverage test inputs and efficient test suites. In particular, we can take advantage of an entire body of work on generating test suites specifically for dynamic invariant detection [Gupta and Heidepriem 2003; Harder et al. 2003; Xie and Notkin 2003].

Finally, our work focuses on specialized types of invariants. It is unlikely that DIG will find invariants of other, unrelated forms, as seen in the AES benchmark. We believe that our approach strikes a balance between rich expressive power, allowing it to find many invariants with real-world uses (e.g., documentation in NLA, verification in AES) and efficiency, allowing it to complete in seconds per program.

7. RELATED WORK

Daikon [Ernst 2000; Ernst et al. 2007; Perkins and Ernst 2004], the paragon of dynamic invariant analysis, infers candidate invariants from traces and templates. By default, the system reports invariants at the entry and exit points of a function, although it is possible to extract invariants at other locations, such as inside loops, by manual instrumentation. The system comes with a large list of assorted invariant templates that are considered as useful to programmers and allows user-supplied invariants. For polynomial relations, the system can find linear relations over at most three variables, for example, $x + 2y - 3z + 4 = 0$, and has fixed nonlinear templates such as $x = y^2$. Relations among arrays have limited support in Daikon, for example, the relations $A[i] = B[C[i]], A[i] = 2B[i] + C[5] + 7$ are not considered.

There is related work on dynamic methods for finding invariants for debugging (e.g., the detected properties are used to find certain type of errors). The Diduce [Hangal and Lam 2002] tool analyzes what happens when an error occurs by looking at the differences between the previous and current values of variables. Statistical debugging [Liblit et al. 2005], a fault localization technique, looks for relations (e.g., $\{<, =, >\}$) between two variables or a variable and a constant. The Spin model checker [Vaziri and Holzmann 1998] can also find relations over two variables. In general, these approaches find invariants that are relatively simple compared to those given by Daikon and DIG.

Abstract Interpretation [Cousot and Cousot 1976, 1977; Cousot and Halbwachs 1978] is a popular framework in static invariant analysis that approximates program properties under a given domain, for example, the polyhedra domain expresses general linear relations over variables. The method starts from an initial approximation and gradually improves the approximation based on the structure of the program until no more improvements can be made (a fixed-point). A *widening* heuristic operator is often used to ensure termination.

Carbonell and Kapur [2007; Carbonell 2006] provide an abstract interpretation framework for polynomial equalities of the form given in Eq. (1). They first observe that a set of polynomial invariants form the algebraic structure of an ideal, then compute the polynomial invariants using Gröbner basis and operations over ideals based on the structure of the program until a fixed point is reached. Scalability is an issue because the approach must consider all possible program paths to guarantee sound results. The method can analyze precisely only programs with assignments and loop guards expressible as polynomial equalities. To ensure termination when analyzing programs with nested loop, the method uses a widening operator that depends on a priori bound on the degrees of the polynomials. Carbonell and Kanpur [2007a] does not require upperbounds on polynomial degrees but is restricted to non-nested loops. This work does not support the forms of inequalities and array relations in DIG.

Recently, Sharma et al. [2013] proposed a complete and sound approach that hybridizes dynamic and static analysis to generate equality invariants. They use the algorithm of Section 3.1 to compute equality invariants from traces and use SMT solving to verify that the candidate invariants are correct with respect to the program source code. Counterexamples to candidate invariants that fail to prove are subsequently used to produce more traces to generate better candidate invariants. They formally prove that the candidate equalities are sound (i.e., always underapproximate the program invariant), and the approach terminates after a finite number of steps to generate and verify candidate invariants. It would be interesting if this approach could be extended to inequality invariants generated by DIG, which are also underapproximations and can be checked for satisfiability using SMT solvers.

8. CONCLUSION

We present DIG, the first dynamic invariant generator that can discover nonlinear polynomial and linear array invariants. Our method applies mathematical techniques not previously employed to aid dynamic invariant detection. By generating invariants directly based on input traces, our results are very accurate with respect to given traces. For nonlinear equality relations, we generate terms representing nonlinear polynomials among variables and use an equation solver to find linear relations among the terms; this yields nonlinear relations among the original variables. We represent inequality constraints using geometric shapes and reduce the task for inferring general inequalities to generating convex polyhedra. For nonlinear inequality relations, we generate terms and then build convex polyhedra, obtaining the desired relations from their facets. Building convex polyhedra in high dimensions is expensive, thus, we consider less expressive forms of inequalities that represent simpler geometric shapes, such as octagons. When additional information, such as a loop entry condition, is available, we can efficiently take advantage of it to deduce new inequalities by combining discovered equality relations with the provided loop condition. For flat array relations, we look for relations among individual array elements and extract from those results the possible relations among the array indices. These flat array relations also express conditional information, capturing array relations that hold for specific indices. For nested array relations, we build an SMT query using information obtained from a reachability analysis; the satisfying assignment provided by the SMT solver yields the desired invariant.

Our evaluation demonstrates the feasibility and potential of DIG by successfully identifying 100% of the nonlinear invariants in 27 nontrivial algorithms as well as 60% of the documented array relations necessary for full formal verification of an AES implementation.

ACKNOWLEDGMENTS

This work is a continuation of a project initiated by Jack Pullikottil with Deepak Kapur in 2003. We thank Matthias Horbach and Hengjun Zhao for insightful discussions as well as Xiang Yin for providing the AES implementation.

REFERENCES

J. Barnes. 2003. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jerome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical Software. In *Proceedings of the Conference on Programming Languages Design and Implementation*. 196–207.

Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. 2000. ABCD: Eliminating array bounds checks on demand. In *Proceedings of the Conference on Programming Language Design and Implementation*. 321–333.

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

Enric Rodríguez Carbonell and Deepak Kapur. 2007a. Generating all polynomial invariants in simple loops. *Symbol. Computa.* 42, 4 (2007), 443–476. DOI: http://dx.doi.org/10.1016/j.jsc.2007.01.002

Enric Rodríguez Carbonell and D. Kapur. 2007b. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Sci. Comput. Program.* 64. (Jan. 2007), 54–75. DOI: http://dx.doi.org/10.1016/j.scico.2006.03.003

Edward Cohen. 1990. *Programming in the 1990s: An Introduction to the Calculation of Programs*. Springer-Verlag.

P. Cousot and R. Cousot. 1976. Static determination of dynamic properties of programs. In *Proceedings of the International Symposium on Programming*. 106–130.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the International Synposium on Principles of Programming Languages*. 238–252.

Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005a. The AstrÉe analyzer. In *Proceedings of the European Symposium on Programming*. 21–30.

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. 2005b. The Astrée analyzer. In *Proceedings of the Conference on European Symposium on Programming (ESOP'05)*. Lecture Notes in Computer Science, vol. 3444, Springer, 21–30.

Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the International Symposium on Principles of Programming Languages*. 84–96.

Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. 1997. *Computational Geometry: Algorithms and Applications*. Springer-Verlag.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 337–340. http://research.microsoft.com/en-us/um/redmond/projects/z3/.

Nachum Dershowitz and Zohar Manna. 1978. Inference rules for program annotation. In *Proceedings of the International Conference on Software Engineering*. 158–167.

B. Dutertre and L. De Moura. 2006. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification*. Springer, 81–94.

M. D. Ernst. 2000. Dynamically detecting likely program invariants. Ph.D. Dissertation, University of Washington.

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 1–3. (2007), 35–45.

R. W. Farebrother. 1988. *Linear Least Squares Computations*. Marcel Dekker, Inc.

Jérôme Feret. 2004. Static analysis of digital filters. In *Programming Languages and Systems*, Springer, 33–48.

Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an annotation assistant for ESC/Java. In *Proceedings of the Intenational Symposium on Formal Methods for Increasing Software Productivity*. 500–517.

Steven M. German and Ben Wegbreit. 1975. A synthesizer of inductive assertions. *IEEE Trans. Softw. Eng.* 1, 1 (1975), 68–75.

N. Gupta and Z. V. Heidepriem. 2003. A new structural coverage criterion for dynamic detection of program invariants. In *Proceedings of the International Conference on Automated Software Engineering*. 49–58.

Sudheendra Hangal and Monica S. Lam. 2002. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*. 291–301.

M. Harder, J. Mellen, and M. D. Ernst. 2003. Improving test suites via operational abstraction. In *Proceedings of the International Conference on Software Engineering*. 60–71.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall.

Michael Karr. 1976. Affine relationships among variables of a program. *Acta Informatica* 6 (1976), 133–151.

Shmuel Katz and Zohar Manna. 1976. Logical analysis of programs. *Commun. ACM* 19, 4 (1976), 188–206.

Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable statistical bug isolation. In *Proceedings of the Conference on Programming Language Design and Implementation*. 15–26.

Antoine Miné. 2004. Weakly relational numerical abstract domains. Ph.D. Dissertation, École Polytechnique, Palaiseau, France.

ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using dynamic analysis to discover polynomial and array invariants. In *Proceedings of the International Conference on Software Engineering*. IEEE, 683–693.

S. Owre, J. Rushby, and N. Shankar. 1992. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (CADE)*. Lecture Notes in Computer Science, vol. 607, Springer-Verlag, 748–752.

Jeff H. Perkins and Michael D. Ernst. 2004. Efficient incremental algorithms for dynamic detection of likely invariants. *ACM SIGSOFT Softw. Eng. Notes*, 29, 6 (2004), 23–32.

Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. 2009. Automatically patching errors in deployed software. In *Proceedings of the Symposium on Operating Systems Principles*. 87–102.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the symposium on Principles of Programming Languages*. 49–61.

V. Rijmen and J. Daemen. 2001. Advanced encryption standard. Federal Information Processing Standards Publications, National Institute of Standards and Technology (2001), 19–22.

Mardavij Roozbehani, Eric Feron, and Alexandre Megrestki. 2005. Modeling, optimization and computation for software verification. In *Proceedings of the 8th International Workshop on Hybrid Systems: Computation and Control*. 606–622.

Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2005. Scalable analysis of linear systems using mathematical programming. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation*. 25–41.

Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013. A data driven approach for algebraic loop invariants. In *Proceedings of the European Conference on Programming Languages and Systems*. Springer, 574–592.

W. A. Stein. 2012. *Mathematics Software.* Sage. http://www.sagemath.org.

Norihisa Suzuki and Kiyoshi Ishihata. 1977. Implementation of an array bound checker. In *Proceedings of the International Symposium on Principles of Programming Languages*. 132–143.

M. Vaziri and G. Holzmann. 1998. Automatic detection of invariants in Spin. In *Proceedings of the SPIN Model Checking and Software Verification Workshop*.

Ben Wegbreit. 1974. The synthesis of loop predicates. *Commun. ACM* 17, 2 (1974), 102–113.

T. Xie and D. Notkin. 2003. Tool-assisted unit test selection based on operational violations. In *Proceedings of the International Conference on Automated Software Engineering*. 40–48.

Xiang Yin, John C. Knight, Elisabeth A. Nguyen, and Westley Weimer. 2008. Formal verification by reverse synthesis. In *Proceedings of the Conference on Computer Safety, Reliability, and Security*. 305–319.

Xiang Yin, John C. Knight, and Westley Weimer. 2009. Exploiting refactoring in formal verification. In *Proceedings of the International Conference on Dependable Systems and Networks*. 53–62.

Received December 2012; revised June, September 2013; accepted November 2013