Using Dynamic Analysis to Discover Polynomial and Array Invariants

ThanhVu Nguyen
Computer Science
University of New Mexico
tnguyen@cs.unm.edu

Deepak Kapur
Computer Science
University of New Mexico
kapur@cs.unm.com

Westley Weimer
Computer Science
University of Virginia
weimer@cs.virginia.edu

Stephanie Forrest
Computer Science
University of New Mexico
forrest@cs.unm.edu

***Abstract*—Dynamic invariant analysis identifies likely properties over variables from observed program traces. These properties can aid programmers in refactoring, documenting, and debugging tasks by making dynamic patterns visible statically. Two useful forms of invariants involve relations among polynomials over program variables and relations among array variables. Current dynamic analysis methods support such invariants in only very limited forms. We combine mathematical techniques that have not previously been applied to this problem, namely equation solving, polyhedra construction, and SMT solving, to bring new capabilities to dynamic invariant detection. Using these methods, we show how to find equalities and inequalities among nonlinear polynomials over program variables, and linear relations among array variables of multiple dimensions. Preliminary experiments on 24 mathematical algorithms and an implementation of AES encryption provide evidence that the approach is effective at finding these invariants.**

***Keywords*-program analysis; dynamic analysis; invariant generation; nonlinear invariants; array invariants**

I. Introduction

The study of program *invariants*—relations among variables that are guaranteed to hold at certain locations in a program—is a cornerstone of program analysis [1]–[3] and has been a major research area since the 1970s [2], [4]–[8]. Invariants can be identified using static or dynamic analysis. Static analysis is typically computationally expensive but more likely to provide provably sound results. Dynamic analysis is usually efficient, but its results are not guaranteed to be correct because the discovered properties may not generalize to all program traces. Nonetheless, dynamic invariant analysis is useful in practice because of its scalability and the help it can provide in program refactoring, documenting and debugging [9]–[11].

Nonlinear polynomial properties are essential to the success of many scientific, engineering and safety-critical applications. For example, Astrée [12], [13], a successful program analyzer used to verify the absence of run-time 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] Nonlinear invariants have also been found useful for the analysis of hybrid systems [14], [15].

Arrays are a widely used data structure that is fundamental to many programs. For example, in C.A.R. Hoare's seminal 1971 paper on algorithm verification, *Proof of a program: FIND*, the overall goal is to prove an array invariant that lies at the heart of the correctness of quicksort [16, p.40]. Fixed-size arrays are also present in many systems programs, and proper analysis is often critical for security (e.g., buffer overruns). Finally, the ubiquity of arrays in general software engineering makes reasoning about arrays crucial for performance (e.g., for bounds check elimination [17]).

Daikon [9] is a well-known dynamic analysis system that detects invariants from program traces. However, Daikon supports only a limited form of linear relations among program variables and arrays. For example, Daikon cannot discover (i) that the location of the chosen pivot in binary search is $l + u - 1 \leq 2p \leq l + u$ as these inequalities involve three variables, (ii) that the gcd of x, y is $nx + my$ because this is a nonlinear polynomial, (iii) the equation $v = 2x + 3y + 4z + 5$ because it involves four variables, or (iv) the relation $A[i] = B[C[i]]$ because it is a nested array relation. It is thus difficult to fully capture and reason about the semantics of programs that can only be expressed in such forms of invariants with Daikon.

To address the issues outlined above and improve dynamic invariant detection, we combine mathematical techniques that have not been previously applied to the problem: equation solving, polyhedra, and SMT (Satisfiability Modulo Theories) solving. More specifically, we focus on generating invariants expressed as nonlinear arithmetic relations among program variables and invariants on relations among complex data structures such as multi-dimensional arrays.

This paper makes the following contributions:

- Polynomial Invariants: We find equalities among nonlinear polynomials of program variables using equation solving, and we find nonlinear inequalities by constructing convex polyhedra. When additional inputs from the user are available, we can also deduce new inequalities from previously obtained equality relations.
- Array Invariants: We find linear equalities among arrays by first finding equalities among array elements and then identifying the relations among array indices from

[1] The ellipsoid domain used by Astrée [13] to examine the Airbus system is expressed in the quadratic form $x^2 + axy + by^2 \leq k$, where $0 < b < 1$ and $a^2 < 4b$.

the obtained equalities. We find nested array relations by performing reachability analysis. Our analysis has potentially high time complexity, thus we encode the problem as a satisfiability problem, which can be efficiently solved with an SMT solver.

- Evaluation: We implemented a prototype tool, depicted in Figure 1, based on this approach. The implementation was evaluated on a set of 24 programs that come with documented invariants involving nonlinear polynomials and on an implementation of AES encryption that contains annotated invariants involving array relations. The tool successfully discovered all documented invariants of the types described above.

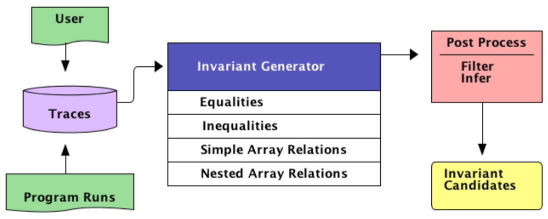

Figure 1. Automatic Generation of Dynamic Invariants. The generator finds different types of invariants from program traces. The post-processing step removes redundant and spurious invariants.

Dynamic invariant detection can be viewed as two separate subproblems: (i) fixing a priori candidate invariants over program variables and then (ii) ruling out invalidated candidates based on observed traces of program variables. We hypothesize that a key reason that previous approaches do not scale to nonlinear invariants or array invariants is that they enumerate candidates based on fixed templates (e.g., linear equations involving at most three variables). Such eager enumeration strategies do not scale to higher-degree polynomials or array invariants due to the large number of possible candidates. By contrast, our approach lazily explores the search space based on the structure of the trace data. It considers candidate invariants based on the traces available, rather than an eager enumeration. This insight, coupled with tools such as equation and SMT solvers, allows us to find human-relevant nonlinear and array invariants in nontrivial programs efficiently.

The paper is organized as follows: Section II provides a motivating example. Sections III and IV present our approach for generating polynomial and array invariants, respectively. Section V reports experimental results. Section VI surveys related work. Section VII concludes the paper.

II. Motivating Example

Invariants are typically placed at the entries and exits of functions corresponding to pre- and postconditions and/or the heads of loops corresponding to loop invariants. Given a location l, the program is instrumented to trace the values of the variables in scope at l. The instrumented program is then run against a set of inputs to obtain the traces.

```
1   int cohendiv(int x, int y){
2     int q = 0; // quotient
3     int r = x; // remainder
4     while (r >= y) {
5       int a = 1;
6       int b = y;
7       while (r >= 2 * b) {
8         // Invariant Location
9         // Invs: b=ya, x=qy+r, r >= 2ya
10        a = 2 * a;
11        b = 2 * b;
12      }
13      r = r - b;
14      q = q + a;
15    }
16    return q;
17  }
```

Figure 2. Cohen's integer division algorithm.

The program in Figure 2 implements the well-known integer division algorithm by Cohen [18], which takes as input two 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 on line 9. There are six variables $\{a, b, q, r, x, y\}$ in scope at l. Table I consists of five sets of values representing traces obtained from the variables at l for inputs $\{x = 15, y = 2\}$ and $\{x = 4, y = 1\}$.

Table I
Traces of the Cohen program on inputs $\{x = 15, y = 2\}$ and $\{x = 4, y = 1\}$.

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

We want to obtain the polynomial invariants over the variables $\{a, b, q, r, x, y\}$ based on such traces. The documented invariants $\{b = ya, x = qy + r, r \geq 2ya\}$, which cannot be identified with current dynamic invariant methods, describe precisely the semantics of the inner while loop in Cohen's algorithm.[2] The next two sections propose our technique for generating such invariants from such traces.

III. Finding Polynomial Invariants

We take as input the set V of variables at location l, the associated traces X, and a maximum degree d, and return possible polynomial relations among the variables in V whose degree is at most d. Two post-processing techniques

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

are applied to the obtained relations to suppress redundant relations and to filter out spurious invariants.

A. Polynomial Equations

Figure 3 outlines the procedure for finding equalities of the form

$$c_1 t_1 + \cdots + c_n t_n = 0, \tag{1}$$

where c_i are real-valued and t_i are terms in T, the set of polynomials over the variables in V of degree at most d.

```
input : set $V$ of variables, set $X$ of traces, maximal degree $d$
output: set $S$ of polynomial relations of the form (1)
$S \leftarrow \emptyset$
$T \leftarrow$ genTerms($V$,$d$)
$T_{\text{eq}} \leftarrow$ genTemplate($T$)
$S_{\text{eq}} \leftarrow$ genEqts($T_{eq}$,$X$)
$s \leftarrow$ solve($S_{eq}$)
if $s \neq \emptyset$ then
    $S \leftarrow s$
return $S$
```

Figure 3. Procedure for finding polynomial equations.

Terms: We first generate the set T of terms over V that can appear in polynomials with maximum degree d. For example, when $V = \{r, y, a\}$ and $d = 2$, the set $T = \{1, r, y, a, ry, ra, ya, r^2, y^2, a^2\}$ has ten terms[3] that can appear in any polynomial of at most degree 2 over variables $\{r, y, a\}$. In general, the number of elements in T is $\binom{|V|+d}{d}$.

The idea of using terms allows us to find nonlinear equalities using any existing and efficient technique that solves linear equations. The user can also specify additional terms to represent other information of interest. Thus, functions that are specific to a program or that come from standard libraries (e.g., sqrt, gcd, pow) can also appear in the discovered invariants.

Solving Equations: Using the terms in T, we create the equation template $T_{\text{eq}} : c_1 + c_2 r + c_3 y + c_4 a + c_5 ry + c_6 ra + c_7 ya + c_8 r^2 + c_9 y^2 + c_{10} a^2 = 0$. Each trace containing values of the program variables is then instantiated by T_{eq} to form an equation of the form (1). For instance, instantiating T_{eq} with the values $r = 15, y = 2, a = 1$ from the first trace in Table I forms the equation $c_1 + 15c_2 + 2c_3 + c_4 + 30c_5 + 15c_6 + 2c_7 + 225c_8 + 4c_9 + c_{10} = 0$. Repeating this process of instantiating equations from the traces X gives a system of linear equations $S_{\text{eq}} = \{e_1, \ldots, e_{|X|}\}$ over the parameters c_i. S_{eq} can be solved using standard equation solvers for linear algebra. The nontrivial solutions of S_{eq}, if any, suggest relations among the terms in T.

In general, to solve for the $|T|$ unknown coefficients c_i, S_{eq} must have at least $|T|$ independent equations. Moreover, solving a system of n linear equations for k unknown parameters has the complexity of $O(n^3)$, assuming $n \geq k$. Thus, the complexity of the procedure in Figure 3 is $O(n^3)$, where $n = |T|$.

[3]Terms of degree 0 are constants, e.g., $x^0 = 1$.

Example: We demonstrate these steps by finding the nonlinear equalities $b = ay, x = qy + r$ from the Cohen program. For illustration, we focus on the case where $d = 2$, in which the procedure generates quadratic equations.

For the six variables $\{a, b, q, r, x, y\}$, together with degree $d = 2$, the set T contains 28 terms. T is then used to form the template T_{eq} with 28 unknown parameters c_i to be solved for:

$$\begin{aligned} c_1 &+ c_2 y + c_3 q + c_4 x + c_5 b + c_6 a + c_7 r + c_8 y^2 \\ &+ c_9 qy + c_{10} xy + c_{11} by + c_{12} ay + c_{13} ry \\ &+ c_{14} q^2 + c_{15} qx + c_{16} bq + c_{17} aq + c_{18} qr \\ &+ c_{19} x^2 + c_{20} bx + c_{21} ax + c_{22} rx + c_{23} b^2 \\ &+ c_{24} ab + c_{25} br + c_{26} a^2 + c_{27} ar + c_{28} r^2 = 0. \end{aligned}$$

T_{eq} is instantiated with the elements in X to form S_{eq}. The nontrivial solutions of S_{eq} for the unknown parameters c_i are of the form $c_4 = -v_3, c_5 = v_1, c_7 = v_3, c_9 = v_3, c_{12} = -v_1, c_{16} = -v_2, c_{21} = v_2, c_{27} = -v_2$ and all other $c_i = 0$. The values v_i are free variables that range over the reals. The terms in T_{eq} that have zero-valued coefficients are not related; because the only way to satisfy equations in S_{eq} is by setting the coefficients of these terms to zero. In contrast, terms that have coefficients sharing some free variable v are related through v. To find the relation among the terms x, r, qy whose respective coefficients c_4, c_7, c_9 share the value v_3, we set $v_3 = 1$ and $v_1 = v_2 = 0$ (since the terms x, r, qy are not related by v_1, v_2). The template T_{eq}, when being instantiated with $\{v_1 = 0, v_2 = 0, v_3 = 1\}$, gives the relation $qy + r - x = 0$. Similarly, the assignment $\{v_1 = 1, v_2 = 0, v_3 = 0\}$ gives $b - ay = 0$ and $\{v_1 = 0, v_2 = 1, v_3 = 0\}$ gives $ax - ar - bq = 0$.

Observe that the equation $ax - ar - bq = 0$ is redundant because we can obtain it from the other two equations $b - ay = 0$ and $qy + r - x = 0$ by substitution. This happens because each term in T is treated as though it were independent of other terms, even though there is a relation among terms such as a, ax, a^2. Section III-C provides a refinement technique to suppress redundant invariants. The resulting set of equations for the running example after refinement is $\{b - ay = 0, qy + r - x = 0\}$.

B. Polynomial Inequalities

Figure 4 outlines the procedure for finding inequalities with two methods: one uses polyhedra and an alternative one uses deduction when additional information is available. Both methods give sound relations with respect to input traces; however the deduction method, with the help of additional information, runs much faster. We continue to use the Cohen program to show how the nonlinear inequality $r \geq 2ay$ can be obtained with both methods.

```
input : set $V$ of variables, set $X$ of traces, maximal degree $d$
input : (optional) set ieqs of inequalities from additional
        information such as loop conditions
output: set $S$ of polynomial relations of the form (2)
$S \leftarrow \emptyset$
if ieqs $= \emptyset$ then
    $T \leftarrow$ genTerms($V$,$d$)
    $S_p \leftarrow$ genPoints($T$,$X$)
    $P \leftarrow$ createPolyhedron($S_p$)
    $S \leftarrow$ extractFacets($P$)
else
    // if additional information is given
    eqts $\leftarrow$ genInvs$_{\texttt{Eqts}}$($V$,$X$,$d$)
    $S \leftarrow$ deduce$_{\texttt{ieqs}}$(eqts,ieqs)
return $S$
```

Figure 4. Procedure for finding polynomial inequalities.

1) Using Polyhedra: This method also consists of creating terms from variables as described previously. However, instead of solving equations, the method constructs points from traces and builds a bounded convex polyhedron that covers all the trace points. The facets or boundaries of the polyhedron represent possible inequalities among program variables of the form

$$c_1t_1 + \cdots + c_nt_n \geq 0, \qquad (2)$$

where c_i are real-valued and t_i are terms.

After the set T of terms is created, the traces X are used to generate points in $|T|$-dimensional Euclidean space, and a convex polyhedron P is computed to enclose all these points. A bounded convex polyhedron P can be described by a system of linear inequalities of the form (2). This is called the *half-space* representation of a polyhedron. The facets of P, corresponding to the solutions of the system of linear inequalities, are the inequalities among the terms in T. Figure 5 depicts a 2D polyhedron that has seven facets.

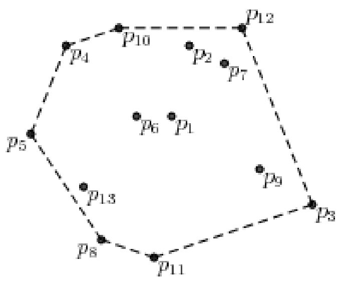

Figure 5. A bounded convex polyhedron with seven facets representing inequalities that constraint the points p_i.

The complexity of building P in n dimensions from k points has a theoretical exponential upper bound $\Theta(k^{\lfloor \frac{n}{2} \rfloor})$ [19]. In our case, n is the size of T. For the Cohen program, while solving equations for 28 unknown parameters is relatively efficient, building a convex polyhedron in 28 dimensions is not feasible. Consequently, several heuristics are employed to identify possible inequality relations.

We first observe that a program invariant often involves just a small subset of all possible program variables. The previously found invariant $b-ay=0$ involves only $\{y,a,b\}$ even though all six variables in scope were considered. To exploit this experience, we have experimented with heuristics, such as iteratively looking for invariants involving all possible combinations of a small, fixed number of variables. Being able to determine which variables are needed greatly speeds up the process and is further discussed in Section V-C.

Example: The Cohen program has six variables $\{a,b,q,r,x,y\}$. As a first step, we generate possible inequality relations in which at most three of these variables appear. There are $\binom{6}{3} = 20$ combinations containing 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 $d = 2$, we generate 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 S_p of points. For instance, the first trace in Table I gives the point $[1, 15, 2, 1, 30, 15, 2, 225, 4, 1]$ in 10-dimensional Euclidean space corresponding to the terms in T. The convex polyhedron P is then constructed to enclose the points in S_p. One of the facets enclosing P 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 not be program invariants that hold for every run. The *filtering* technique in Section III-D helps remove spurious invariants.

2) Deduction From Loop Conditions: Using convex polyhedra to find inequalities works well but does not scale to large numbers of terms. Consequently, we developed an alternative technique using *deduction* to find inequalities if additional information is available. More specifically, if some inequalities are asserted at location l, then we can use them together with the discovered equalities from Section III-A to deduce new nontrivial inequalities. For instance, if the location l is the head of a loop L, then l can be reached if and only if the loop conditions of L are met. Such loop conditions are an example of additional information, which can be given by the user to facilitate the process of generating additional invariants.

Example: We demonstrate how deduction is applied to the Cohen example. First, the set of equations $\{b - ay = 0, qy+r-x = 0\}$ representing possible invariants at location l is obtained as described in Section III-A. 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, the deduction technique finds the set of possible inequalities $\{r - 2ya \geq 0, x - qy - 2b \geq 0\}$ among variables $\{a, b, q, r, x, y\}$ at location l. We note that even though $x - qy - 2b \geq 0$ is not a documented invariant, it is indeed one of the loop invariants of the inner while loop in the Cohen program.

In contrast to finding inequalities by constructing convex polyhedra, using deduction requires additional information and is incomplete in the sense that it only deduces properties from the resulting equations and the supplied loop conditions. However, it has a low complexity of $O(n^3)$, where $n = |T|$ (the cost of running an equation solver). In our evaluation, this hybrid method efficiently discovers all the documented inequalities of the benchmark programs.

We now discuss two post-processing techniques that help remove redundant and spurious invariants. These techniques are useful because the obtained set of relations may contain redundant information (e.g., two relations may imply a third) or may contain spurious invariants (relations drawn from limited data sample that do not hold for additional inputs).

C. Implication

To reduce the size of the invariant set, we remove invariants that are logically implied by others. For instance, we suppress the invariant $x^2 = y^2$ if another invariant $x = y$ is also found because the latter implies the former. Redundant invariants can often occur in our approach, as seen in the Cohen example in Section III-A, because we treat each term as an independent variable for the purposes of nonlinear polynomials discovery. 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 off-the-self SMT solver to show the negation of that implication is unsatisfiable. Note that we apply this technique before using the deduction method discussed in Section III-B2.

D. Filtering

Dynamic invariant analysis finds invariants that hold only for specific traces. If additional traces become available, such as by running the program on a different input set, they can be used to check the set of proposed relations in linear time. The obtained relations are verified against the new traces and are removed if they do not satisfy the new data.[4] This step allows us to increase our confidence about the obtained results and to remove those that do not hold for all available traces.

[4]We note this idea of filtering is also used in existing dynamic invariant detectors such as Daikon; however we apply it on our generated results whereas Daikon uses the technique on its predefined templates.

IV. Finding Array Invariants

We take as input the set V of (possibly multi-dimensional) array variables at location l and the associated traces X, and return possible relations among the elements of arrays in V. The *filtering* technique given in Section III-D is also applied to the obtained relations to help remove spurious invariants.[5]

A. Simple Array Relations

Figure 6 outlines the procedure for finding linear simple (non-nested) relations among array elements of the form

$$A_1 + c_2A_2 + \cdots + c_nA_n + c_0 = 0, \quad (3)$$

where A_i are distinct (possibly multi-dimensional) arrays whose elements are real-valued. The array A_1 (with unit coefficient), called the *pivot array*, is privileged in our approach because the coefficients c_i and indices of other arrays $A_2, \ldots, A_n$ are hypothesized as linear expressions ranging over the indices of A_1. An example invariant of this form is $A[i][j] - \frac{1}{2}jB[2i + j] + 2C[7i][3] + 5 = 0$.

```
input  : set $V$ of array variables, set $X$ of traces
output: set $S$ of array relations of the form (3)
$S \leftarrow \emptyset$
// obtain linear relations among array elements
$V' \leftarrow$ genNewVars ($V$)
eqts $\leftarrow$ genInvs_Eqts ($V'$,$X$,$d = 1$)
$Rs \leftarrow$ group(prune(eqts))
if $Rs \neq \emptyset$ then
    foreach $R \in Rs$ do
        pivot $\leftarrow$ genPivot ($R$)
        exps $\leftarrow$ genLinExps (pivot)
        $s \leftarrow$ solve(exps,$R$)
        $S \leftarrow S + \{s\}$
return $S$
```

Figure 6. Procedure for finding simple array relations.

For simplicity, the following explains the procedure for two single-dimensional arrays, i.e., $V = \{A, B\}$, although the method generalizes to multi-dimensional arrays.

Relations Among Array Elements: We first generate a set V' of new variables representing elements of the arrays in V. Next, we find linear equalities (Section III-A) over the variables in V' from the input traces X. The obtained equations represent relations among array elements. Because only relations among different arrays are of interest, we keep only those that express relations among array elements of different arrays. The remaining relations are then grouped based on the arrays whose elements are appearing in those relations; for example, relations among elements of arrays A, B, C belong to one group and relations among elements of arrays D, E, F belong to another group.

[5]The *implication* technique in Section III-C is not necessary because polynomial terms are not used to find array relations.

The complexity of the procedure in Figure 6 is dominated by this step, because we invoke an equation solver to find relations among all array elements. The time complexity of the solver on this problem is $O(n^3)$, where $n = |V'|$, the number of elements of the arrays in V.

Relations Among Array Indices: Among the obtained groups of relations, we only consider ones having the set R of relations of the form:

$$\begin{aligned}
A_0 + b_0 B_{j_0} + c_0 &= 0,\\
A_1 + b_1 B_{j_1} + c_1 &= 0,\\
A_2 + b_2 B_{j_2} + c_2 &= 0,\\
&\vdots\\
A_m + b_m B_{j_m} + c_3 &= 0,
\end{aligned}$$

where m is $|A| - 1$, b_i, c_i are real-valued, j_i are integers, and A_i, B_{j_i} are the variables in V' representing $A[i], B[j_i]$.

In such a group of relations, A is chosen as the pivot array because elements of other arrays are related to elements of A. We hypothesize that the coefficients b_i, c_i and the indices j_i of array B are linear expressions ranging over the indices of A. For instance, we represent the relation between $B[j]$ and $A[i]$ through the parameterized linear expression $j = p_1 i + q_1$, where p_1 and q_1 are unknown coefficients to be solved for. This expression is then instantiated with the information from R: $j = p_1 i + q_1$ instantiated with $i = 0, j = j_0$ (the first relation in R) gives $j_0 = q_1$, with $i = 1, j = j_1$ gives $j_1 = p_1 + q_1$, with $i = 2, j = j_2$ gives $j_2 = 2p_1 + q_1$, and so forth. Any solution of this system of equations gives a relation of the form $A[i] = (p_0 i + q_0)B[p_1 i + q_1] + (p_2 i + q_2)$.

Example: We illustrate the method by finding the relation $A[i] = 7B[2i] + 3i$ between arrays A, B, where $|A| = 3$ and $|B| = 5$, using traces X that exhibit such a relation. An example trace in X contains the values $A = [-546, -641, 34]$ and $B = [-78, 3, -92, -34, 4]$.

We generate a set of eight variables to represent the elements of A and B. Based on the given trace, the set $R = \{A_0 - 7B_0 = 0, A_1 - 7B_2 - 3 = 0, A_2 - 7B_4 - 6 = 0\}$ is obtained. A is designated as the pivot because all of its elements have relations with elements of B. The relation between $B[j]$ and $A[i]$ is expressed as $j = p_1 i + q_1$. We then instantiate $j = p_1 i + q_1$ with the information from R and obtain the set of equations $\{0 = q_1, 2 = p_1 + q_1, 4 = 2p_1 + q_1\}$. The unique solution $\{q_1 = 0, p_1 = 2\}$ of these equations yields $j = 2i$, i.e., $A[i] = b_i B[2i] + c_i$. Similarly, we instantiate the analogous equations for b_i and c_i. After solving these, the array relation $A[i] = 7B[2i] + 3i$ is obtained.

B. Nested Array Relations

Figure 7 outlines the procedure for finding linear nested relations among array elements of the grammar

$$\begin{aligned}
A[i]\cdots[i] &\to e \qquad (4)\\
e &\to c \mid f(i,\ldots,i) \mid B[e]\cdots[e],
\end{aligned}$$

where A, B are distinct (possibly multi-dimensional) arrays, A is the pivot array, c is real-valued, and f is a linear expression ranging over the indices i of A. An example relation of this form is $A[i][j][k] = F[B[2i+3]][C[D[3]][E[4j+k]]]$.

```
input : set $V$ of array variables, set $X$ of traces
output: set $S$ of array relations of the form (4)
$S \leftarrow \emptyset$
// generate nestings
// e.g., $A[i] = B[\ldots], A[i] = C[\ldots], B[i] = C[\ldots], \ldots$
nestings ← genNestings ($V$)
foreach nesting ∈ nestings do
    $R \leftarrow$ analyzeReachability(nesting,$X$)
    if $R \neq \emptyset$ then
        $f \leftarrow$ genFormula ($R$)
        $s \leftarrow$ SMT ($f$)
        if $s \neq \emptyset$ then
            $S \leftarrow S$ + {s}
return $S$
```

Figure 7. Procedure for finding nested array relations.

For simplicity, the following explains the procedure for three single-dimensional arrays, i.e., $V = \{A, B, C\}$, although the method generalizes to multi-dimensional arrays.

Reachability Analysis: When the procedure in Section IV-A cannot find simple relations among the arrays in V, we hypothesize that arrays may be related through some nested structure. For instance, the relation $A[i] = B[C[k]]$ implies that the elements of A are related to the elements of B using the elements of C as indices into B.

For such a relation to hold, the elements of A must be in B, i.e., the contents of A are some subset of B: $\{A[0] = B[j_0], A[1] = B[j_1], \ldots, A[m] = B[j_m]\}$. Moreover, the indices j_i of B occur as elements of C, i.e., the indices j_i of B are some subset of C: $\{j_0 = C[k_0], j_1 = C[k_1], \ldots, j_m = C[k_m]\}$. A set R of relations of the form $\{A[0] = B[C[k_0]], A[1] = B[C[k_1]], \ldots, A[m] = B[C[k_m]]\}$, specifies that elements of A reach elements of C through elements of B. This suggests the possible nested array relation $A[i] = B[C[k]]$.

The complexity of the procedure in Figure 7 is dominated by this step, because if the nesting depth is d and a value is found at n locations at each nesting level, then the number of relations generated by the reachability analysis is n^d. Thus, the time complexity of reachability analysis is exponential in the nesting depth.

Relations Among Array Indices: This step is conceptually similar to that of Section IV-A in which the relation between $C[k]$ and $A[i]$ is represented by the parameterized linear expression $k = pi + q$.

Instantiating $k = pi+q$, where i, k represent the respective indices of A, C, with the information from R, we get the set of equations $\{k_0 = q, k_1 = p + q, \ldots, k_m = mp + q\}$. Any solution of this system of equations gives a relation of the form $A[i] = B[C[pi + q]]$.

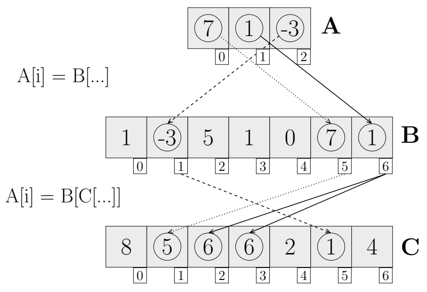

Figure 8. Reachability analysis showing $A[0] = B[C[1]]$ (dotted), $A[1] = B[C[2]]$, $A[1] = B[C[3]]$ (solid), and $A[2] = B[C[5]]$ (dashed).

Example: Figure 8 illustrates the method by finding the relation $A[i] = B[C[2i + 1]]$ from the trace $A = [7, 1, -3], B = [1, -3, 5, 1, 0, 7, 1], C = [8, 5, 6, 6, 2, 1, 4]$.

Nestings such as $B[i] = C[\ldots]$ can be ruled out immediately because the element -3 of B is not in C. Note the use of traces is essential here as it allows us to quickly filter out invalid nestings. Reachability analysis shows that the contents of A are in B, namely $A[0] = B[5], A[1] = B[0] = B[3] = B[6], A[2] = B[1]$, thus suggesting a relation of the form $A[i] = B[\ldots]$. Next, since the index value 5 of B appear in C as $C[1]$, we obtain the relation $A[0] = B[C[1]]$. The index values 0 and 3 of B do not occur in C, however the index value 6 of B appears twice in C as $C[2]$ and $C[3]$. This results in $A[1] = B[C[2]]$ or $A[1] = B[C[3]]$. Finally, the analysis yields $A[2] = B[C[5]]$ because the index value 1 of B appears in C as $C[5]$.

We hypothesize the possibility of the nested relation $A[i] = B[C[k]]$, where k is the parameterized linear expression $k = pi + q$, because elements of A are related to elements of B using elements of C as indices into B as given by either the set of relations $R_1 : \{A[0] = B[C[1]], A[1] = B[C[2]], A[2] = B[C[5]]\}$ or $R_2 : \{A[0] = B[C[1]], A[1] = B[C[3]], A[2] = B[C[5]]\}$.

Instantiating $k = pi + q$ with the information from R_1 gives the set of equations $\{1 = q, 2 = p + q, 3 = 5p + q\}$. This has no solution, thus no relation of the form $A[i] = B[C[k]]$ is derived. Next, we instantiate $k = pi + q$ with the information from R_2 and obtain the set of equations $\{1 = q, 3 = p + q, 5 = 2p + q\}$. The unique solution $\{p = 2, q = 1\}$ for this implies the relation $A[i] = B[C[2i + 1]]$.

We now present techniques to include functions into invariant forms and improve the performance of invariant generation with an SMT solver. The following demonstrates these techniques on nested array relations even though they can also be applied to invariant of other forms.

C. Functions

Array invariants involving user-defined functions, e.g., $A[i] = f(C[i], g(D[i]))$, 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 mod_2 which maps even and odd inputs to 0 and 1 respectively, then the corresponding array G has its indices as elements of A, B, F, e.g., $G[4] = G[28] = 0, G[5] = G[7] = 1$. Currently, we enforce finite depth in nested array relations by disallowing a function to appear in the scope of one of its arguments, e.g., $g(f(g(\ldots), f(\ldots)))$.

The inclusion of functions thus allows us to generate invariants involving functions, such as the nested array invariant $R[i] = T(\mathsf{mod255}(\mathsf{add}(L(A[i]), L(B[i]))))$ required for the $\mathsf{multWord}$ operation in AES.

D. Satisfiability Problem Formulation

Because of the potentially high complexity of reachability analysis, we encode the steps for finding nested array relations as a satisfiability problem, which can be solved efficiently with an SMT solver.[6]

We show how to encode the procedure for finding nested array relations into a CNF formula f. Returning to the example in Section IV-B, once the relation $A[0] = B[C[1]]$ is obtained, we create the atom $1 = q$. Likewise, the disjunctive formula $(2 = p + q \ \vee \ 3 = p + q)$ consisting of two atoms is formed to represent $A[1] = B[C[2]]$ or $A[1] = B[C[3]]$. Similarly, the atom $5 = 2p + q$ is created for $A[2] = B[C[5]]$. Since the relation should hold for all elements (i.e., $\forall i.\ A[i] = B[C[pi + q]]$), we combine all atoms into the final CNF formula $f : (1 = q) \wedge (2 = p + q \vee 3 = p + q) \wedge (5 = 2p + q)$. 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 resulting assignment $\{p = 2, q = 1\}$ yields the relation $A[i] = B[C[2i + 1]]$.

SMT solvers improve the performance of our method considerably, even though the worst-case complexity of the

[6]Satisfiability modulo theories (SMT) solvers determine the satisfiability of a formula (that contains variables ranging over domains such as the reals, the integers, pointers, bit vectors, and so on). In the case of a satisfiable formula, the SMT solver can also produce a satisfying assignment of values to variables in the formula.

problem remains exponential. Our original implementation without the solver took approximately five minutes to find nested array relations of the multWorld function from AES due to the large number of generated relations. By encoding the generated relations directly into an SMT formula, the problem was solved in under five seconds with an off-the-self SMT solver.

V. Experimental Results

We have implemented a prototype of our invariant generation approach in Python using the Sage mathematical environment [20]. The prototype uses built-in Sage functions to solve equations and construct polyhedra. It also uses *qepcad* [21] to remove redundant invariants and *Z3* [22] to check the satisfiability of SMT formulae. The experiments reported here were performed on a Unix-based system with a dual-core 2.3GHz Intel i5 CPU and 8 GB of RAM.

A. Programs

We evaluated our prototype on programs taken from a test suite which we call NLA (nonlinear arithmetic) and an implementation of AES. The details of NLA and AES are given in Tables II and III, respectively.

The NLA test suite consists of 24 programs from various sources collected by Rodríguez-Carbonell and Kapur [23], [24]. These programs implement classic arithmetic algorithms that are widely used in programming, such as mult, div, pow, mod, sqrt, gcd, lcm. 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, the number of variables of interest, and the type of relations among program variables—*not* the size of the program per se. Among the 24 programs from NLA, there are 35 documented nonlinear invariants: 33 are equations and 2 are inequalities.

The second benchmark, AES$'$, is an annotated AES implementation from Yin *et al.* [25]. It exemplifies a real-world security-critical application and contains nontrivial array invariants. To show that functions in the AES$'$ 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 and PVS. The annotated invariants represent the manual effort required to fully functionally verify an AES implementation using axiomatic semantics. AES$'$ contains 868 lines of Ada code organized into 25 functions containing 30 invariants: 8 simple 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 manually instrumented the source code of the programs to trace values of all variables in the scope at each program location containing a known invariant. Our goal is to find invariants at those locations automatically and compare them to the human-documented invariants.

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. From the obtained traces, we randomly chose a set of 200 as traces input to the invariant generator and another set of 50 for filtering as described in Section III-D. Note that our prototype automatically adjusts the number of traces for optimization (e.g., 200 traces are not necessary to solve for 10 unknown coefficients).

B. Result Quality

NLA: Table II lists our experimental results on 24 programs from NLA, averaged over 20 runs. The *Vars* column reports the number of distinct variables in that program's invariants, *Deg* reports the highest polynomial degree in those invariants, *Invs* reports the number of invariants found by our approach and the total number of documented invariants, and *T* reports the average time in seconds to discover the invariants, including the time to refine the results.

We found all 35 documented nonlinear invariants from the NLA test suite. 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, the sqrt1 program has two documented equalities $2a + 1 = t$ and $(a + 1)^2 = s$, our results give $2a + 1 = t$ and $t^2 + 2t + 1 = 4s$, which is equivalent to $(a+1)^2 = s$ by substituting t with $2a+1$. We note that current dynamic analysis approaches cannot find any of these nonlinear relations.

AES[7]*:* Table III lists our experimental results on 25 functions from AES, averaged over 20 runs. The *Arrs* column reports the number of distinct arrays in that program's invariants, *Dim* reports the highest dimension of the arrays in those invariants, *Inv Types* reports the types of invariant: *S*imple, *N*ested, and *O*thers. $N(d)$ specifies that the nesting depth is d. The *driver* functions are composed from other functions in this table.

We found all 17 relations that are expressible in our considered forms. The other 13 invariants do not fall into categories described above and are left for future work. These can be grouped into three categories: Others$_{1-3}$. Others$_1$ includes nested array invariants such as $A[i] = 4B[6C[\ldots]]$. We current do not handle such nested invariants when the elements of A are not exactly nested in B. Others$_2$ includes array invariants such as $A[i] = B[\ldots]$

[7]When there is no ambiguity, we refer to the considered AES$'$ implementation simply as AES.

Table II
EXPERIMENTAL RESULTS ON 24 PROGRAMS FROM NLA.

Program	**Desc**	**Inv Types**	**Vars**	**Deg**	**Invs**	**T (s)**
divbin	div	eq	5	2	1/1	0.5
cohendiv	div	eq, ieq	6	2	2/2	1.3
mannadiv	int div	eq	5	2	1/1	0.3
hard	int div	eq	6	2	1/1	0.9
sqrt1	sqr	eq, ieq	4	2	2/2	0.7
dijkstra	sqr	eq	5	2	1/1	0.5
freire1	sqr	eq	3	2	1/1	0.2
freire2	cubic root	eq	4	3	2/2	3.2
cohencube	cube	eq	5	3	3/3	12.6
euclidex1	gcd	eq	10	2	3/3	6.5
euclidex2	gcd	eq	8	2	2/2	2.5
euclidex3	gcd	eq	12	2	4/4	10.1
lcm1	gcd, lcm	eq	6	2	1/1	0.5
lcm2	gcd, lcm	eq	6	2	1/1	0.6
prodbin	product	eq	5	2	1/1	0.3
prod4br	product	eq	6	3	1/1	8.1
fermat1	divisor	eq	5	2	1/1	0.8
fermat2	divisor	eq	5	2	1/1	0.4
knuth	divisor	eq	8	3	1/1	71.5
geo2	geo series	eq	4	2	1/1	0.2
geo3	geo series	eq	5	3	1/1	3.1
ps2	pow sum	eq	3	2	1/1	0.1
ps3	pow sum	eq	3	3	1/1	0.3
ps4	pow sum	eq	3	4	1/1	0.8
24 programs		**2 types**			**35/35**	**126.0s**

Table III
EXPERIMENTAL RESULTS ON 25 FUNCTIONS FROM AES.

Function	**Desc**	**Inv Types**	**Arrs**	**Dim**	**Invs**	**T (s)**
multWord	mult	N(4)	7	2	1/1	3.6
xor2Word	xor	N(1)	4	2	1/1	0.1
xor3Word	xor	N(1)	5	3	1/1	0.1
subWord	subs	N(1)	3	1	1/1	0.4
rotWord	shift	S	2	1	1/1	0.5
block2State	convert	S	2	2	1/1	2.0
state2Block	convert	S	2	2	1/1	11.7
subBytes	subs	N(1)	3	2	1/1	0.6
invSubByte	subs	N(1)	3	2	1/1	3.8
shiftRows	shift	S	2	2	1/1	12.2
invShiftRow	shift	S	2	2	1/1	8.3
addKey	add	N(1)	4	2	1/1	0.6
mixCol	mult	O_3	4	2	0/1	-
invMixCol	mult	O_3	4	2	0/1	-
keySetEnc4	driver	S,O_2	2	2	1/2	4.5
keySetEnc6	driver	S,O_2	2	2	1/2	6.7
keySetEnc8	driver	S,O_2	2	2	1/2	10.6
keySetEnc	driver	O_3	4	1	0/1	-
keySetDec	driver	O_3	4	2	0/1	-
keySched1	driver	O_1	3	2	0/1	-
keySched2	driver	O_1	3	2	0/1	-
aesKeyEnc	driver	eq,O_3	7	2	1/2	0.1
aesKeyDec	driver	eq,O_3	7	2	1/2	0.1
aesEncrypt	driver	O_3	8	4	0/1	-
aesDecrypt	driver	O_3	8	4	0/1	-
25 functions		**6 types**			**17/30**	**65.9s**

where $i = \{0, 4, 8, 12, \dots\}$ and $A[i'] = B[\dots]$ where $i' = \{1, 2, 3, 5, 6, 7, 9, 10, 11 \dots\}$. We require that generated relations such as $A[i] = B[\dots]$ 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 [25], [26]. Annotating pre- and postconditions and loop invariants has not been solved in general and is known to be a key bottleneck in approaches based on axiomatic semantics [27]. 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.

To summarize, we 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 are beyond the scope of this paper and left for future work. On average, it takes under five seconds to find the invariants for each program. To the best of our knowledge, no other dynamic invariant analysis approaches have analyzed the forms of invariants discussed in this paper.

C. Threats to Validity

Two of the operations, namely reachability analysis and polyhedra construction, have exponential complexity in the worst case. We address these issues by using existing SMT solvers and by deducing new inequalities through loop conditions. Nonetheless, our method may not scale to very large scenarios.

Our method for deducing new inequalities assumes additional inputs from the user. We hypothesize that programmers may have knowledge about the loop entry conditions when they want to analyze more general loop and function invariants. Nonetheless, we emphasize that additional information, while useful, is not required to find inequalities. The use of additional information also allows for hybrid approaches using static analysis. For example, techniques such as *slicing* [28] could be used to find how variables are likely related to one another, reducing the number of variable combinations processed by our method.

Floating point values are subject to roundoff errors that may confound some exact checks. Given an abundance of available traces, we filter out certain traces containing rounded float values. Our tool also allows comparison within ε instead of exact comparisons; e.g., $0.33333 \approx 1/3$ 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.

The effectiveness of our method depends on program traces produced by test inputs. We cannot derive properties that are not exhibited from 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 $x + y \geq 10$. We note the existence of many active research projects on generating 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 [29]–[31].

Spurious invariants, which are common in dynamic invariant analysis, are also present in our work. However, we find few spurious invariants on our benchmarks after filtering: 2 for equalities over polynomials, 0 for inequalities (using the deduction method), and 4 for arrays. One source of spurious invariants is overfitting; i.e. learning behavior from a small set of traces that does not hold for other runs. However, these spurious behaviors are mostly pruned during the filtering step, which checks the candidate invariants against another set of data. Thus, even with the relatively small number of runs in our experiments (200 for data gathering, 50 for filtering), we found the occurrence of spurious invariants to be rare (6, as above).

We acknowledge, however, that the polyhedral method could generate many spurious inequalities if the analysis produces convex polyhedra with multiple overly complicated facets (e.g., $x^2 + 0.123842932 \geq 0$) when run against sample traces representing data points in higher dimensions. Indeed, a key motivation for the deduction method presented in Section III-B2 was to avoid constructing complex polyhedra.

Finally, our work focuses on specialized types of invariants. It is unlikely that it 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.

VI. Related Work

The Daikon system [9] is a very popular invariant detector that uses dynamic analysis to find program invariants. The method is essentially brute-force: Daikon comes with a large list of invariant templates and tests them against program traces. Templates that are violated in any of the test runs are removed and the remainders are presented as the possible invariants. 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. In addition, Daikon can check user-supplied invariants. However, as mentioned earlier, Daikon can only find linear equations and inequalities over a few variables and has limited support for relations among arrays.

Other systems also find invariants using dynamic analysis for specific purposes, such as debugging a certain type of error. The Diduce [32] tool analyzes what happens when an error occurs by looking at the difference between the previous and current values of variables. Statistical Debugging [33] is a fault localization technique that looks for simple relations (e.g., $\{<, =, >\}$) between one variable and another variable or constant. The Spin model checker [34] can also find relations over two variables. In general, these approaches find invariants that are relatively simple compared to those given by Daikon. Thus, they cannot discover the types of invariants considered in this paper.

VII. Conclusion

We present the first dynamic invariant generation technique that can discover nonlinear polynomial and linear array invariants. Our method applies mathematical techniques not previously employed to aid dynamic invariant detection. For nonlinear equality relations, we generate terms representing nonlinear polynomials among variables and use an equation solver to find equality relations among the terms; this yields nonlinear relations among the original variables. For nonlinear inequality relations, we generate terms and then build convex polyhedra, obtaining the desired relations from their facets. 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 simple array relations, we look for relations among individual array elements and extract from those results the possible relations among the array 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 the approach by successfully identifying 100% of the nonlinear invariants in 24 complex 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 Xiang Yin for providing the AES implementation and Matthias Horbach for insightful discussions on early drafts of this work. We gratefully acknowledge the partial support of AFORSR (FA9550-07-1-0532, FA9550-10-1-0277), DARPA (P-1070-113237), DOE (DE-AC02-05CH11231), NSF (SHF-0905236, CCF-0729097, CNS-0905222) and the Santa Fe Institute.

References

[1] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation.* Prentice Hall, 1993.

[2] M. Karr, “Affine relationships among variables of a program,” *Acta Informitica*, vol. 6, pp. 133–151, 1976.

[3] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, 2007.

[4] B. Wegbreit, "The synthesis of loop predicates," *Communication ACM*, vol. 17, no. 2, pp. 102–113, 1974.

[5] S. M. German and B. Wegbreit, "A synthesizer of inductive assertions," *IEEE Transactions Software Engineering*, vol. 1, no. 1, pp. 68–75, 1975.

[6] S. Katz and Z. Manna, "Logical analysis of programs," *Communication ACM*, vol. 19, no. 4, pp. 188–206, 1976.

[7] N. Suzuki and K. Ishihata, "Implementation of an array bound checker," in *Principles of Programming Languages*, 1977, pp. 132–143.

[8] N. Dershowitz and Z. Manna, "Inference rules for program annotation," in *International Conference on Software Engineering*, 1978, pp. 158–167.

[9] M. Ernst, "Dynamically detecting likely program invariants," PhD Thesis, University of Washington, 2000.

[10] Y. Kataoka, M. Ernst, W. Griswold, and D. Notkin, "Automated support for program refactoring using invariants," in *International Conference on Software Maintenance*, 2001, pp. 736–743.

[11] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *Symposium on Operating systems Principles*, 2009, pp. 87–102.

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

[13] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *Programming Languages Design and Implementation*, 2003, pp. 196–207.

[14] M. Roozbehani, E. Feron, and A. Megrestki, "Modeling, optimization and computation for software verification," in *Hybrid Systems: Computation and Control*, 2005, pp. 606–622.

[15] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, "Scalable analysis of linear systems using mathematical programming," in *Verification, Model Checking and Abstract Interpretation*, 2005, pp. 25–41.

[16] C. A. R. Hoare, "Proof of a program: Find," *Communication ACM*, vol. 14, pp. 39–45, January 1971.

[17] R. Bodík, R. Gupta, and V. Sarkar, "ABCD: eliminating array bounds checks on demand," in *Programming Language Design and Implementation*, 2000, pp. 321–333.

[18] E. Cohen, *Programming in the 1990s: an introduction to the calculation of programs*. Springer-Verlag, 1990.

[19] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational geometry: algorithms and applications*. Springer-Verlag, 1997.

[20] W. Stein *et al.*, "Sage Mathematics Software," 2012, http://www.sagemath.org.

[21] C. W. Brown, "Qepcad b: a system for computing with semi-algebraic sets via cylindrical algebraic decomposition," *SIGSAM Bull.*, vol. 38, no. 1, pp. 23–24, 2004.

[22] L. de Moura and N. Bjorner, "The Z3 SMT Solver," 2012, http://research.microsoft.com/en-us/um/redmond/projects/z3/.

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

[24] E. R. Carbonell, "Automatic generation of polynomial invariants for system verification," Ph.D. dissertation, Technical University of Catalonia (UPC), Barcelona, 2006.

[25] X. Yin, J. C. Knight, and W. Weimer, "Exploiting refactoring in formal verification," in *Dependable Systems and Networks*, 2009, pp. 53–62.

[26] X. Yin, J. C. Knight, E. A. Nguyen, and W. Weimer, "Formal verification by reverse synthesis," in *Conference on Computer Safety, Reliability, and Security*, 2008, pp. 305–319.

[27] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," in *Formal Methods for Increasing Software Productivity*, 2001, pp. 500–517.

[28] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Principles of Programming Languages*, San Francisco, California, 1995, pp. 49–61.

[29] N. Gupta and Z. Heidepriem, "A new structural coverage criterion for dynamic detection of program invariants," in *International Conference on Automated Software Engineering*, 2003, pp. 49–58.

[30] M. Harder, J. Mellen, and M. Ernst, "Improving test suites via operational abstraction," in *International Conference on Software Engineering*, 2003, pp. 60–71.

[31] T. Xie and D. Notkin, "Tool-assisted unit test selection based on operational violations," in *International Conference on Automated Software Engineering*, 2003, pp. 40–48.

[32] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *International Conference on Software Engineering*, 2002, pp. 291–301.

[33] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Programming Language Design and Implementation*, 2005, pp. 15–26.

[34] M. Vaziri and G. Holzmann, "Automatic detection of invariants in Spin," in *SPIN Model Checking and Software Verification*, 1998.