Learning Randomized Reductions and Program Properties

FERHAT ERATA, Yale University, USA
ORR PARADISE, UC Berkeley, USA
TIMOS ANTONOPOULOS, Yale University, USA
THANHVU NGUYEN, George Mason University, USA
SHAFI GOLDWASSER, UC Berkeley, USA
RUZICA PISKAC, Yale University, USA

The correctness of computations remains a significant challenge in computer science, with traditional approaches relying on automated testing or formal verification. Self-testing/correcting programs introduce an alternative paradigm, allowing a program to verify and correct its own outputs via randomized reductions, a concept that previously required manual derivation. In this paper, we present Bitween, a method and tool for automated learning of randomized (self-)reductions and program properties in numerical programs. Bitween combines symbolic analysis and machine learning, with a surprising finding: polynomial-time linear regression, a basic optimization method, is not only sufficient but also highly effective for deriving complex randomized self-reductions and program invariants, often outperforming sophisticated mixed-integer linear programming solvers. We establish a theoretical framework for learning these reductions and introduce RSR-Bench, a benchmark suite for evaluating Bitween's capabilities on scientific and machine learning functions. Our empirical results show that Bitween surpasses state-of-the-art tools in scalability, stability, and sample efficiency when evaluated on nonlinear invariant benchmarks like NLA-DigBench. Bitween is open-source as a Python package and accessible via a web interface that supports C language programs.

1 INTRODUCTION

The correctness of computations remains a fundamental challenge in computer science. Currently, there are two major approaches to this problem. The first is the classical approach based on automated testing [Claessen and Hughes 2000; Miller and Spooner 1976], that seeks to detect errors by running a program against a diverse set of input scenarios and program properties. The second approach, formal verification [Clarke and Emerson 2008; Hoare 1969], employs mathematical methods to rigorously prove program properties. An alternative paradigm was introduced by Blum et al. [1993], who developed a formal theory of programs that "test and correct themselves" (a notion similar to self-correctors was independently proposed by Lipton [1991]). Self-testing and self-correcting allow one to use a program Π to compute a function f without trusting that Π works correctly for all inputs. The computed function f can be seen as a formal mathematical specification of what the program Π should compute. Two key components of this formalism are: a self-tester and a self-corrector. A *self-tester* for f estimates the fraction of values of x for which indeed it holds $\Pi(x) = f(x)$. A *self-corrector* for f takes a program that is correct on most inputs and turns it into a program that is correct on every input with high probability. Both self-testers and self-correctors have only black-box access to Π and, importantly, do not directly compute f.

Authors' addresses: Ferhat Erata, Yale University, New Haven, CT, USA, ferhat.erata@yale.edu; Orr Paradise, UC Berkeley, Berkley, CA, USA, orrp@eecs.berkeley.edu; Timos Antonopoulos, Yale University, New Haven, CT, USA, timos.antonopoulos@yale.edu; ThanhVu Nguyen, George Mason University, Fairfax, VA, USA, tvn@gmu.edu; Shafi Goldwasser, UC Berkeley, Berkley, CA, USA, shafi.goldwasser@berkeley.edu; Ruzica Piskac, Yale University, New Haven, CT, USA, ruzica.piskac@yale.edu.

Self-correctors exist for any function that is *randomly self-reducible* [Blum et al. 1993]. Informally, when a function f has a randomized self-reduction (RSR) then there is a way to recover the value of f at a given point x by computing f on random correlated points. For example, we can always recover the value of the function $f_c(x) = c \cdot x$ by evaluating f_c on $x - r$ and r, for any value r, and adding the results: $f_c(x) = f_c(x - r) + f_c(x)$. This equation is an RSR for the function f_c. In the context of self-correctors, merely evaluating Π on the given x is not enough, since the program is known to be correct on most inputs, not all. Self-correctness establishes that the correct value of the function at any particular input x can be inferred, even though the program may be incorrect on input x.

Random self-reducibility has been used in several other areas of computer science, including cryptography protocols [Blum and Micali 2019; Goldwasser and Micali 2019], average-case complexity [Feigenbaum and Fortnow 1993], instance hiding schemes [Abadi et al. 1987; Beaver and Feigenbaum 1990; Beaver et al. 1991], result checkers [Blum et al. 1993; Lund et al. 1992] and interactive proof systems [Blum and Kannan 1995; Goldwasser et al. 2019; Lund et al. 1992; Shamir 1992].

Despite their importance and various applications, finding reductions automatically has been a longstanding challenge. In their work on automatic (gadget-based) reduction finding, Trevisan et al. [2000] describe generating such properties as "...[it] has always been a 'black art' with no general methods of construction known". In their seminal papers, Blum et al. [1993], and later Rubinfeld [1999] manually derived randomized reductions through thorough mathematical analysis. We address this challenge and in this paper we present a technique and a tool, called Bitween, which is the first tool for automatically generating self-reductions.

Our tool collects samples by executing the program on a number of randomly chosen values, and then utilizes simple yet powerful machine learning regression techniques, to iteratively converge towards possible self-reductions. At each round, it leverages various measures to eliminate some of the candidates and reduce the dimensions of the problem. Finally, the correctness of the resulting self-reductions is checked against separately generated test-data, as well as formally verified using off-the-shelf tools.

In addition to learning randomized reductions, we also applied the same learning algorithm to infer loop invariants and post-conditions of numerical programs.

We have empirically evaluated Bitween on 40 different programs, including also all the benchmarks previously used in [Blum et al. 1993; Rubinfeld 1999]. We were able to derive all known self-reducible properties, as well as multiple new properties that are not covered in the existing work. Bitween outperforms existing invariant generation tools in scalability, stability, sample efficiency, and execution time. Bitween requires fewer samples, runs faster.

Contributions. This work makes the following contributions:

- We introduce Bitween, a linear regression-based automated learning algorithm that effectively discovers complex nonlinear randomized (self)-reductions and other arbitrary program invariants (e.g., loop invariants and post conditions) in mixed integer and floating-point programs (see Section 4).
- We present a rigorous theoretical framework for learning randomized (self)-reductions, advancing formal foundations in this area (see Section 3).
- We create a benchmark suite, RSR-Bench, to evaluate Bitween's effectiveness in learning randomized reductions. This suite includes a diverse set of mathematical functions commonly used in scientific and machine learning applications (see Table 3). Our evaluation compares Bitween's linear regression backend, which we call Bitween, against a Mixed-Integer Linear Programming

```
1  double Π(double x) {
2      const int trms = 30;
3      double sum = 1.0;
4      double trm = 1.0;
5      double neg_x = -x;
6      for (int n = 1; n < trms; n++) {
7          trm *= neg_x / n;
8          sum += trm;
9      }
10     return 1.0 / (1.0 + sum);
11 }
```
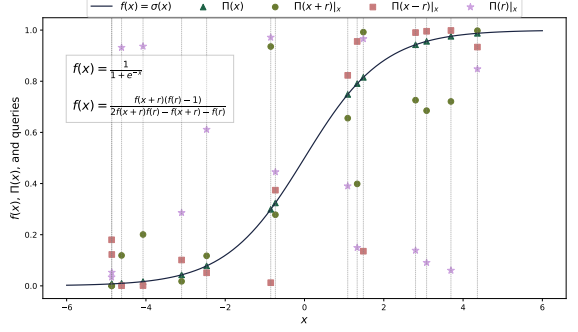

Fig. 1. An implementation of the sigmoid activation function, and a graph representing randomly selected points by Bitween which are then used to compute an RSR $\sigma(x) = \frac{\sigma(x+r)(\sigma(r)-1)}{2\sigma(x+r)\sigma(r)-\sigma(x+r)-\sigma(r)}$

(MILP) backend, which we call Bitween-Milp, showing that linear regression-based learning outperforms MILP in both scalability and stability within this domain (see Section 5).

- We implement Bitween in Python and demonstrate its performance on an extended version of NLA-DigBench [Beyer 2017; Nguyen et al. 2014a], which includes nonlinear loop invariants and post-conditions (see Table 4). Our empirical evaluation shows that Bitween surpasses leading tools Dig [Nguyen et al. 2012, 2014b, 2021] and SymInfer [Nguyen et al. 2017b, 2022] in capability, runtime efficiency, and sampling complexity (see Section 5).

Bitween and RSR-Bench are open source and available as a Python package. They are accessible through an anonymized web application that interfaces with a subset of C language programs at: https://bitween.fun.

2 MOTIVATING EXAMPLE

In this section we illustrate the use of randomized self-reductions (RSR) and how Bitween computes them on the example of the sigmoid function, $\sigma(x) = 1/(1 + e^{-x})$. The sigmoid function is commonly used in neural networks [e.g., Han and Moraga 1995], and in particular in binary classification and logistic regression problems, as it maps input values to a value between 0 and 1. The program $\Pi(x)$ given in Figure 1 approximates the $\sigma(x)$ by using a Taylor series expansion. Here, Π denotes the implementation of sigmoid function σ. Clearly, $\Pi(x)$ computes only an approximate value of $\sigma(x)$.

We invoked Bitween on Π and it derived the following RSR:

$$\sigma(x) = \frac{\sigma(x + r)(\sigma(r) - 1)}{2\sigma(x + r)\sigma(r) - \sigma(x + r) - \sigma(r)}, \tag{1}$$

where r is some random value. We manually verified that indeed the sigmoid function satisfies Equation (1). To the best of our knowledge, this is the first known RSR for the sigmoid function. In this example, Bitween inferred this RSR with 15 independent and random samples of x and r. In the plot in Figure 1, we depict the value of $\Pi(x)$, identified on the graph with ▲, and the values of $\Pi(x+r), \Pi(x-r)$ and $\Pi(r)$, shown on the graph with ●, ■ and ★. Notice that all ▲'s are lying on the line depicting $\sigma(x)$.

Moreover, our derived RSR computes $\sigma(x)$ by using only $\sigma(x+r)$ and $\sigma(r)$, although the algorithm of Bitween is computing Π on random (yet correlated) inputs $x+r, x-r$, and r. The learned RSR in Equation (1) can be used to compute the value of $\Pi(x)$ at any point x by evaluating $\Pi(x+r)$ and

Input

Program Π
```
double Π(double x) {
  const int trms = 30;
  double sum = 1.0;
  double trm = 1.0;
  double neg_x = -x;
  for (int n = 1; n < trms; n++) {
    trm *= neg_x / n;
    sum += trm;
  }
  return 1.0 / (1.0 + sum);
}
```

Output

Learned Set of Properties

Property

$$f(x) = \frac{f(x+r)(f(r)-1)}{2f(x+r)f(r)-f(x+r)-f(r)}$$

Mean Squared Error

$$-6.76542155630955e^{-18}$$

Reduction

Reduced Set of Properties

8 Rank Equivalence Classes of Properties

9 Apply Gröbner Basis Reduction

10 Formal Verification or Property-based Testing

Bitween's Learning Algorithm Λ

1 Construct query class and recovery class $R_k(Q, P)$ where $q_1, \ldots, q_k \in Q$ and $p \in P$

$$u_1 = q_1(x, r) \mapsto x \qquad u_3 = q_3(x, r) \mapsto x + r$$
$$u_2 = q_2(y, r) \mapsto r \qquad u_4 = q_4(x, r) \mapsto x - r \qquad f(x) = p\left(x, r, f(u_1), \ldots, f(u_k)\right)$$

$(x_j, r_j)_{j=1}^m$ for independent and uniformly sampled $x_j \sim X$ and $r_j \sim R$

2 Execute target functions by simulating f with Π: $\Pi(x_j), \Pi(r_j), \Pi(x_j + r_j), \Pi(x_j + r_j)$

	$\Pi(u_1)$	$\Pi(u_2)$	$\Pi(u_3)$	$\Pi(u_4)$	$\Pi(u_1)^2$	$\Pi(u_1)\Pi(u_2)$	$\Pi(u_2)^2$	$\Pi(u_1)\Pi(u_3)$	$\Pi(u_1)\Pi(u_4)$	$\cdots$	1
1	-0.207	1.4017	-4.425	13.971	0.0429	0.2454	-0.775	0.0429	-0.561	$\cdots$	1
2	-1.890	30.384	-26.08	22.398	3.5752	-10.42	8.9486	3.5752	0.9997	$\cdots$	1
...	...	...	...	...	...	...	...	...	...	$\cdots$	...
m	0.1009	4.6261	590.30	2.4527	0.0101	112.39	0.4670	21.400	0.1007	$\cdots$	1

data

3 Formulate an unsupervised learning problem and split data

$c_1 \cdot f(u_1) + c_2 \cdot f(u_2) + c_3 \cdot f(u_3) + c_4 \cdot f(u_4) + c_5 \cdot f(u_1)^2 + c_6 \cdot f(u_1)f(u_2) + c_7 \cdot f(u_2)^2$
$+ c_8 \cdot f(u_1)f(u_3) + c_9 \cdot f(u_1)f(u_4) + c_{10} \cdot f(u_2)f(u_3) + c_{11} \cdot f(u_2)f(u_4) + c_{12} \cdot f(u_3)f(u_4)$
$+ c_{13} \cdot f(u_3)^2 + c_{14} \cdot f(u_4)^2 + \cdots + c_{33} \cdot f(u_1)f(u_3)f(u_4) + c_{34} \cdot f(u_2)f(u_3)f(u_4) + c_{35} =_\delta 0$

training and validation sets

4 Select a target function as the next regressand: $[f(u_1), f(u_2), f(u_3), f(u_4), \ldots, f(u_4)^2]$

test set

refined model   initial model

5 Regression models to find coefficients or reduce dimensionality with cross-validation

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & \cdots & a_{1,35} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & & a_{2,35} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & a_{m,3} & a_{m,4} & \cdots & a_{m,35} \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$$

best model

6 Best Rational Approximation of coefficients

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,35} \\ a_{2,1} & a_{2,2} & & a_{2,35} \\ \vdots & & \ddots & \vdots \\ a_{1,m} & a_{2,m} & \cdots & a_{m,35} \end{bmatrix} \cdot \begin{bmatrix} \hat{c}_1 \\ \hat{c}_2 \\ \vdots \\ \hat{c}_m \end{bmatrix}$$

candidate property

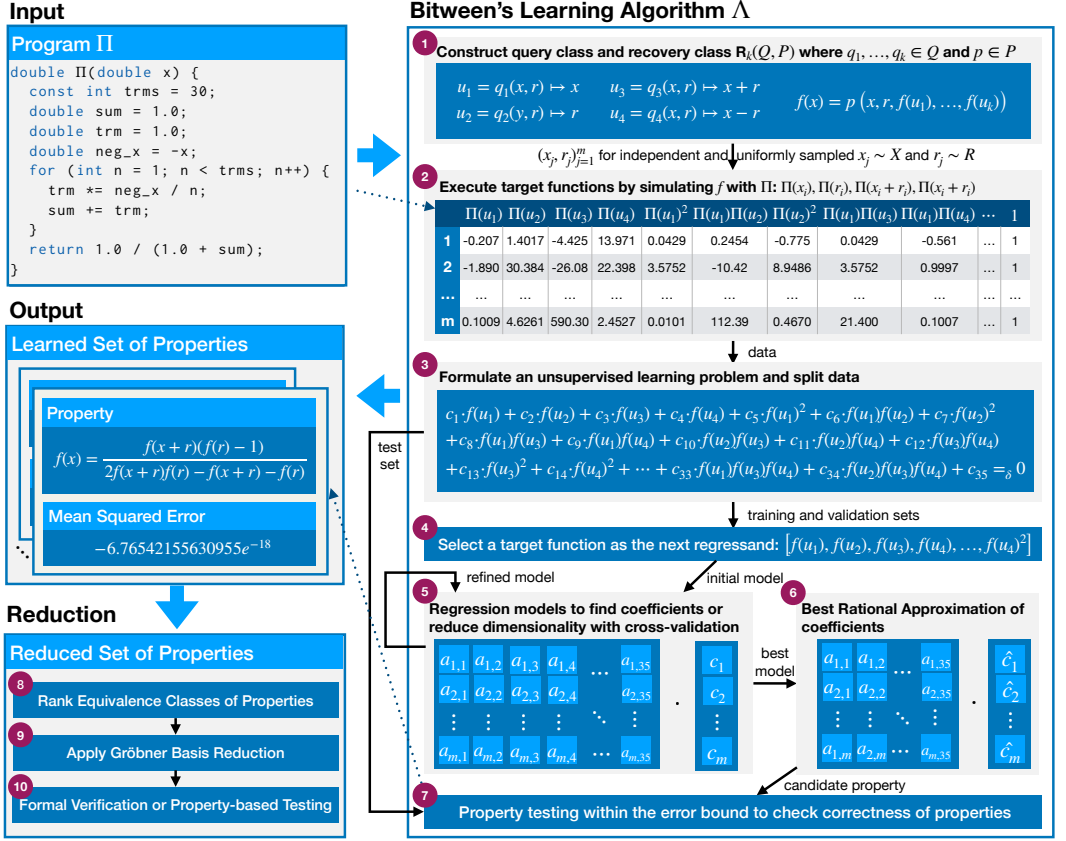**7** Property testing within the error bound to check correctness of properties

Fig. 2. Overview of Bitween.

$\Pi(r)$. Since $\Pi(x) \neq \sigma(x)$ for some values of x, randomized reductions can be used to construct self-correcting programs [Blum et al. 1993; Rubinfeld 1994; Tompa and Woll 1987].

The RSR in Equation (1) can also be used in an instance hiding protocol [Abadi et al. 1987]. As an illustration, if a weak device needs to compute σ on its private input x, it can do that by sending computing requests to two powerful devices that do not communicate with each other: the first powerful device computes $\sigma(r)$ with random r, and the second powerful one computes $\sigma(x+r)$. After receiving their outputs, the weak device can compute $\sigma(x)$ by evaluating Equation (1).

Additionally, in this particular case the derived RSR can be used to reduce the computation costs. If a weak device computes the value of $\sigma(r)$ beforehand and stores it as some constant C, then the computation of $\sigma(x)$ simplifies to $\frac{\sigma(x+r)(C-1)}{2\sigma(x+r)C-\sigma(x+r)-C}$. While x is a double, $x+r$ might require less precision.

We now illustrate how Bitween derived the above RSR.

Bitween applies a series of steps to learn randomized reductions (see Figure 2). Bitween takes a program Π as input and systematically constructs a query class using the input variable x and randomness r, such as $x+r, x-r, r$ (Step **1**). The tool then independently and uniformly samples data using a specified distribution, evaluating target functions $\sigma(x + r), \sigma(x - r), \sigma(r), \ldots$ by simulating σ with Π based on the sampled inputs (Step **2**). Subsequently, Bitween constructs linear models by treating nonlinear terms as constant functions (Step **3**). Since the model is unsupervised, it

instantiates candidate models in parallel, where each model takes a distinct target function as its supervised variable (Step ④). The tool then uses various linear regression models, including Ridge and Lasso with fixed hyperparameters, and the best model is picked based on cross validation. Then it iteratively refines the model by eliminating irrelevant targets from this model (Step ⑤). The coefficients of the final model is further refined using a best rational approximation technique. Finally, BITWEEN validates these properties using the test dataset through property testing (Step ⑥). After validation, BITWEEN outputs the learned randomized self-reductions of the program Π with their corresponding errors.

Optionally, the user can refine the analysis by enabling a set of reduction techniques. First, BITWEEN creates an equivalence class of properties based on structural similarity (Step ⑧). Second, it applies Gröbner Basis [Buchberger 2006] reduction to eliminate redundant properties (Step ⑨). Lastly, BITWEEN uses bounded model-checking [Kroening and Tautschnig 2014] or property-based testing [Goldstein et al. 2024] to ensure correctness of the properties and eliminates any unsound ones (Step ⑩).

3 THEORETICAL FOUNDATIONS

In this section, we give a definitional treatment of learning randomized self-reductions (RSRs). Our goal is to rigorously define the setting in which BITWEEN resides, which may of independent interest for future theoretical work.

Throughout this section, we will say that a set Z is *uniformly-samplable* if it can be equipped with a uniform distribution; we let $z \sim Z$ denote a uniformly random sample from Z.

3.1 Randomized self-reductions

We start from a definition of randomized self-reductions [Goldwasser and Micali 1984]. Our presentation takes after Lipton [1989] and Goldreich [2017], modified for convenience.

DEFINITION 1 (RANDOMIZED SELF-REDUCTION). *Fix a uniformly-samplable input domain X, a range Y, and uniformly-samplable randomness domain R. Let*

$$f \colon X \to Y$$
$$q_1, \ldots, q_k \colon X \times R \to X \qquad \text{(Query functions)}$$
$$p \colon X \times R \times Y^k \to Y \qquad \text{(Recovery function)}$$

such that for all $i \in [k]$ and $x \in X$, $u_i := q_i(x, r)$ is distributed uniformly over X when $r \sim R$ is sampled uniformly at random.[1]

We say that $(q_1, \ldots, q_k, r)$ is a (perfect) randomized self-reduction (RSR) for f if for all $r \in R$, letting $u_i := q_i(x, r)$ for all $i \in [k]$, the following holds:

$$f(x) = p(x, r, f(u_1), \ldots, f(u_k)). \qquad (2)$$

In other words, Equation (2) holds with probability 1 over randomly sampled $r \sim R$.

For errors $\rho, \xi \in (0, 1)$, we say that $(q_1, \ldots, q_k, p)$ is a (ρ, ξ)-approximate randomized self-reduction ((ρ, ξ)-RSR) for f if, for all but a ξ-fraction of $x \in X$, Equation (2) holds with probability $\geq 1 - \rho$ over the random samples $r \sim R$. That is,

$$\Pr_{x \sim X} \left[\Pr_{r \sim R} \left[\begin{array}{l} f(x) = p(x, r, f(u_1), \ldots, f(u_k)) \\ \text{where } \forall i \in [k] \ u_i := q_i(x, r) \end{array} \right] \geq 1 - \rho \right] \geq 1 - \xi.$$

---

[1]Importantly, the u_i's must only satisfy *marginal uniformity*, but may be correlated among themselves.

Given a class of query functions $Q \subseteq X^{X \times R}$ and recovery functions $P \subseteq Y^{X \times R \times Y^k}$, we let $\mathrm{RSR}_k(Q, P)$ denote the class of functions $f \colon X \to Y$ for which there exist $q_1, \ldots, q_k \in Q$ and $p \in P$ such that f is perfectly RSR with $(q_1, \ldots, q_k, p)$. We write $\mathrm{RSR}(Q, P)$ when the number of queries is irrelevant.

In the literature [Goldreich 2017; Lipton 1989], the query functions are defined as randomized functions of the input x to be recovered. That is, as random variables $\tilde{q}_i(x) \sim X$ rather than our deterministic $q_i(x, r) \in X$. The definitions are equivalent; we simply make the randomness in $\widetilde{q_i}$ explicit by giving it r as input. This choice will have two benefits:

(1) It makes explicit the amount of random bits used by the self-reduction, namely, $\log_2 |R|$.
(2) It lets us think about the query functions as deterministic. This could allow one to relate traditional complexity measures (e.g., VC dimensions) of the function classes Q and P to those of the function f.

To elaborate more on the second point, let us restrict the discussion to polynomials over finite fields, i.e., $f \colon \mathbb{F}_n^m \to \mathbb{F}_n$ for some $n \in \mathbb{N}$. Lipton [Lipton 1989] showed that even extremely simple choices of Q and P can be very expressive.

FACT 2 ([LIPTON 1989]). *Any m-variate polynomial $f \colon \mathbb{F}_\ell^m \to \mathbb{F}_\ell$ of degree $d < \ell - 1$ is perfectly randomly self-reducible with $k = d + 1$ queries and randomness domain $R = \mathbb{F}_\ell^m$. Furthermore, the queries $q_1, \ldots, q_{d+1} \colon \mathbb{F}_\ell^{2m} \to \mathbb{F}_\ell^m$ and recovery function $p \colon \mathbb{F}_\ell^{2m+d+1} \to \mathbb{F}_n$ are linear functions.*

3.2 Learning randomized self-reductions

The question of interest is whether, given access to samples from $f \in \mathrm{RSR}_k(Q, P)$, it is possible to learn an (approximate) RSR for f. Before we can continue, we must first specify how these samples are drawn. Typically, learners are either given input-output pairs $(x, f(x))$ where x's are either *independent random samples*, or chosen by the learner herself. Learning RSRs will occur in an intermediate access type, in which x's are drawn in a correlated manner.

We formally define these access types next. Our presentation is based on that of O'Donnell [2014], who, like us, is focused on samples drawn from the uniform distribution. We note that learning from uniformly random samples has been studied extensively in the literature [Golea et al. 1996; Hancock 1993; Jackson et al. 2002; Jackson and Servedio 2006; Klivans et al. 2004; Kucera et al. 1994; Verbeurgt 1990].

DEFINITION 3 (SAMPLE ACCESS). *Fix a uniformly-samplable set X, function $f \colon X \to Y$ and a probabilistic algorithm Λ that takes as input m (labeled) samples from f. We consider three types of sample access to f:*

(1) Independent random samples: Λ is given $(x_j, f(x_j))_{j=1}^m$ for independent and uniformly sampled $x_j \sim X$.
(2) Correlated random samples: Λ is given $(x_j, f(x_j))_{j=1}^m$ drawn from a distribution such that, for each $j \in [m]$, the marginal on x_j is uniformly random over X. However, different x_j's may be correlated.
(3) Oracle queries: During Λ's execution, it can request the value $f(x)$ for any $x \in X$.

The type of sample access will be explicitly stated, unless clear from context.

Remark 4. Facing forward, we note that more restrictive access types will correspond to more challenging settings of learning (formally defined in Definition 5). Intuitively (and soon, formally), if F is learnable from m independent samples, it is also learnable from m correlates samples, and m oracle queries as well.

Learning from correlated samples is one of two main theoretical innovations introduced in this work (the other will appear shortly). We note that PAC learning [Valiant 1984] requires learning

under *any* distribution μ of inputs over X, whereas we consider learning only when samples are drawn from the uniform distribution over X (with possible correlation). Learning from correlated samples could be adapted to arbitrary distributions μ by considering correlated samples $(x_j, f(x_j))_j$ such that the marginal on each x_j is distributed according to μ. This interesting setting is beyond the scope of this work.

Finally, for an input x we will use $n = |x|$ to denote its. This will allow us to place an *efficiency requirement* on the learner (e.g., polynomial time in n). For a class of functions F from inputs X to outputs Y, we will use F_n to denote the class restricted to inputs of length n, and similarly we let Q_n (resp. P_n) denote the restriction of the query (resp. recovery) class.[2]

DEFINITION 5 (LEARNING RSR). *Fix a reduction class* $(Q, P) = (\bigcup_n Q_n, \bigcup_n P_n)$ *and a function class* $F = \bigcup_n F_n$ *where* $F_n \subseteq \mathrm{RSR}_k(Q_n, P_n)$ *for some constant* $k \in \mathbb{N}$.[3] *A* (Q, P, k)-*learner* Λ *for* F *is a probabilistic algorithm that is given inputs* $n \in \mathbb{N}$ *and* m *samples of* $f \in F_n$, *collected in one of the three ways defined in Definition 3.* Λ *outputs query functions* $q_1, \ldots, q_k \in Q_n$ *and a recovery function* $p \in P_n$.

We say F *is* (Q, P)-RSR_k-*learnable if there exists a* (Q, P, k)-*learner* Λ *such that for all* $f \in F$ *and* $\rho, \xi, \delta \in (0, 1)$, *given* $m := m(\rho, \xi, \delta)$ *labeled samples from* f, *with probability* $\geq 1 - \delta$ *over the samples and randomness of* Λ, Λ *outputs* $q_1, \ldots, q_k \in Q$ *and* $p \in P$ *that are* (ρ, ξ)-*RSR for* f.

We say that F *is* efficiently (Q, P)-RSR_k-*learnable if* Λ *runs in time* $\mathrm{poly}(n, 1/\rho, 1/\xi, 1/\delta)$. *The function* $m(\rho, \xi, \delta)$ *is called the* sample complexity *of the learner* Λ. *We will omit the* (Q, P) *prefix when it is clear from context.*

Definition 5 takes after the classic notion of Probably Approximately Correct (PAC) learning [Valiant 1984] in that it allows the learner a δ failure probability, and asks that the learned $q_1, \ldots, q_k, r$ only approximately recover f. The main difference is that in, Definition 5, Λ is required to output an approximate RSR for f, whereas in PAC learning it is required to output a function $\hat{f} \in F$ that approximates f itself; that is, such that $\hat{f}(x) = f(x)$ with high probability over $x \sim X$.

In fact, when samples are drawn uniformly and *independently* (Item 1 in Definition 3), PAC learnability is a strictly stronger form of learning than RSR learnability, as captured by the following two claims:

CLAIM 6. *Fix query class* Q, *recovery class* P, *and function class* $F \subseteq \mathrm{RSR}_k(Q, P)$. *If* F *is (Uniform) PAC-learnable with sample complexity* $m_{\mathrm{PAC}}(\varepsilon, \delta)$, *then it is RSR-learnable with sample complexity*

$$m_{\mathrm{RSR}}(\rho, \xi, \delta) \leq m_{\mathrm{PAC}}(\min(\rho/k, \xi), \delta).$$

However, the RSR-learner may be inefficient.

We note that Claim 6 is trivial when considering a class $F \subseteq \mathrm{RSR}_k(Q, R)$ characterized by a single RSR, i.e., such that there exist $q_1, \ldots, q_k \in Q$ and $p \in P$ such that Equation (2) holds with probability 1 for all $f \in F$. For one, this follows because the RSR-learner does not need any samples and may simply output $q_1, \ldots, q_k, r$.[4] This gives rise to the following claim.

CLAIM 7. *There exist classes* $(Q, P) = (\bigcup_n Q_n, \bigcup_n P_n)$ *and* $F = \bigcup_n F_n \subseteq \mathrm{RSR}(Q, P)$ *such that* F *is efficiently RSR-learnable from* 0 *samples, but for any* $\varepsilon \leq 1/2$, *Uniform PAC-learning* F_n *requires* $m_{\mathrm{PAC}}(\varepsilon, \delta) \geq n$ *oracle queries.*

---

[2]This slight informality will allow us to avoid encumbering the reader with a subscript n throughout the paper.

[3]The requirement that $F_n \subseteq \mathrm{RSR}_k(Q_n, R_n)$ is a *realizability assumption*. We leave the agnostic setting, in which $F_n \nsubseteq \mathrm{RSR}_k(Q_n, R_n)$, to future work.

[4]For a more nuanced reason, note that the sample complexity bound $m_{\mathrm{PAC}}(\rho/k, \delta)$ trivializes: Any class F characterized by a single RSR has *distance* at least $1/k$, meaning that $\Pr_{z \sim X}[\hat{f}(z) \neq f(z)] \geq 1/k$ [Goldreich 2017, Exercise 5.4]. Thus, for any $\varepsilon = \rho/k < 1/k$, f is the only function in F that is ε-close to itself. In other words, learning within accuracy ε amounts to exactly recovering f.

Consequentially, Uniform PAC-learning F_n requires at least n correlated or independent random samples (see Remark 4).

The Fundamental Theorem of Learning states that sample complexity of PAC-learning is tightly characterized by the VC-dimension of the function class F. In the future, it would be interesting to obtain an analogous Fundamental Theorem of RSR-Learning, which relates the sample complexity to "intrinsic" dimensions of Q, R, and $F \subseteq \text{RSR}(Q, R)$.

4 BITWEEN: AN ALGORITHM FOR LEARNING RANDOM SELF-REDUCTIONS

4.1 Algorithm Overview

BITWEEN expects opaque-box access[5] to a program Π which is an alleged implementation of some (unknown) function f. The goal is to learn an RSR for f (see Definition 5). BITWEEN is also given the input domain X,[6] the class of query functions Q and recovery functions P. Note that these classes need not be explicitly given; in fact, throughout the rest of this paper, we will consider recovery functions P to be polynomials of degree at most d (therefore, only the degree d needs to be given to BITWEEN). If the query class Q is not explicitly given, BITWEEN will utilize a pre-constructed collection of query functions, whose compilation has been guided by the many applications of this general approach (see Table 6). Whether the configuration (Q, d) is provided explicitly by the user or fetched from the library of templates, we consider it as input to BITWEEN (Algorithm 1).

At a high level, BITWEEN works as follows: (i) generate all monomials of degree $\leq d$ over symbolic variables $\Pi(q(x, r))$—one variable for each possible query function $q \in Q$; (ii) construct a linear regression problem with a regressand for each monomial; (iii) query $\Pi(x_i)$ and $\Pi(q(x_i, r_i))$ for random $x, r \in X$; (iv) fit the regressands to the samples using sparsifying linear regression, thereby eliminating most monomials. Algorithm 1 is a full description (up to the sparsifying linear regression), and Section 4.2 provides concrete implementation details.

4.2 Implementation Details

Converting an unsupervised problem into supervised learning. Each term in the polynomial expression is treated as a dependent variable (label), while the rest form the vector of regressors (features). Bitween constructs a set of linear regression models for each term in the polynomial expression and runs them in parallel with training data. The correctness of the inferred properties is verified using test data through property testing. For low-degree polynomials, the total number of terms in the polynomial t_{terms} can be considered bounded, so the complexity is $O(t_{\text{terms}} \times n_{\text{samples}} \times n_{\text{features}}^2)$. However, the number of monomials in a polynomial with n terms and degree d grows exponentially, given by the binomial coefficient $\binom{n+d}{d}$. For high-degree polynomials or many variables, the number of terms will grow very fast, leading to an exponential blow-up. Therefore, the general case won't be polynomial due to this exponential factor.

Grid search with cross validation and linear regression. BITWEEN uses multiple linear regression to infer nonlinear properties. The complexity of ordinary linear regression is polynomial: $O(n_{\text{samples}} \cdot n_{\text{features}}^2)$ [Pedregosa et al. 2024]. BITWEEN uses grid search and cross-validation to tune parameters. It applies simple linear regression, Ridge, and Lasso regressions (L1 and L2 regularization) with various hyperparameters, selecting the best model by splitting the data into training, validation, and test sets. With a fixed total number of hyperparameters c_{params}, the complexity is $O(c_{\text{params}} \cdot t_{\text{terms}} \cdot n_{\text{samples}} \cdot n_{\text{features}}^2)$.

---

[5]More precisely, correlated sample access (Definition 3).
[6]BITWEEN is given an input domain X implicitly equipped with the uniform distribution $x \sim X$. For simplicity, the randomness domain R in Definition 1 is taken to be the same as the input domain X.

Algorithm 1: BITWEEN

Input: Program Π, query class Q, recovery function degree bound d, input domain X, sample complexity m.

Output: Randomized self-reduction $(q_1, \ldots, q_k, p)$.

1 For each query function $q \in Q$, initialize a variable v_q.   // Each v_q corresponds to a value $\Pi(q(x, r))$.

2 Let MON be all monomials of degree at most d over the variables $(v_q)_{q \in Q}$.

3 For each monomial $V \in$ MON, initialize the regressand C_V.   // We will fit $\Pi(x) = \sum_V C_V \cdot V(x, r)$

4 For each $i \in [m]$, sample input $x_i \in X$ and randomness $r_i \in X$.

5 Query Π for the values $\Pi(x_i)$ and $\Pi(q(x_i, r_i))$ for each $q \in Q$.

6 Fit the regressands C_V to the equations

$$\Pi(x_1) = \sum_{V \in \text{MON}} C_V \cdot V(x_1, r_1), \quad \ldots \quad \Pi(x_m) = \sum_{V \in \text{MON}} C_V \cdot V(x_m, r_m)$$

using the linear regression variant described in Section 4.2. Let $\widehat{C_V}$ denote the fitted regressands.

7 Initialize an empty set of query functions $\widehat{Q} \leftarrow \emptyset$.

8 **foreach** $V \in MON$ **do**

9   **if** $\widehat{C_V} \neq 0$ **then**

10   Add to $\widehat{Q}$ all queries q such that the variable v_q appears in the monomial V.

11 Let $(\widehat{q_1}, \ldots, \widehat{q_k}) \leftarrow \widehat{Q}$ where $k = |\widehat{Q}|$. For each $\widehat{q_i}$, let $\widehat{v_i}$ denote its corresponding variable defined in Line 1.
Define the recovery function

$$\widehat{p}(x, r, \widehat{v_1}, \ldots, \widehat{v_k}) := \sum_{V : \widehat{C_V} \neq 0} \widehat{C_V} \cdot V(x, r).$$

return *The randomized self-reduction* $(\widehat{q_1}, \ldots, \widehat{q_k}, \widehat{p})$.

The datasets were divided in the following way: 80% of samples in each of the datasets were used for training, whereas the remaining 20% were used for testing. We used grid search to tune the hyperparameters of each method. The optimal setting of the parameters was determined based on 5-fold cross-validation performed on training data only. The setting with the best R^2 score across all folds on the whole training data was picked for the next step of BITWEEN. The performance of the methods was measured on both training and testing datasets on the best model obtained during cross-validation. For linear regression methods, BITWEEN uses scikit-learn [Pedregosa et al. 2011] implementations with tailored hyperparameters for each model:

- *Ordinary Least Squares Regression (Linear Regression):* This model minimizes the sum of squared errors for a linear relationship between inputs and the target variable. By default, the model includes an intercept term, which can be controlled using the `fit_intercept` parameter. Setting `fit_intercept=True` allows the model to estimate an intercept term, which is helpful when target data is not centered around zero. For linear regression, the model's complexity is computed using singular value decomposition of X, where X has shape $(n_{\text{samples}}, n_{\text{features}})$. This results in a cost of $O(n_{\text{samples}} \cdot n_{\text{features}}^2)$, assuming $n_{\text{samples}} \geq n_{\text{features}}$. Hyperparameters: `fit_intercept` (True, False), `positive` (True, False) to enforce non-negative coefficients.

- *Ridge Regression:* Ridge adds L2 regularization to the linear model, penalizing large coefficients and preventing overfitting. Like Linear Regression, Ridge includes the `fit_intercept` parameter to control the inclusion of an intercept term. The primary hyperparameter `alpha` controls the regularization strength and is tested over values [0.001, 0.01, 0.1, 100, 1000]. Ridge has the same complexity as Linear Regression, with a cost of $O(n_{\text{samples}} \cdot n_{\text{features}}^2)$.

- *Lasso Regression:* Lasso introduces L1 regularization, which can reduce certain coefficients to zero, effectively performing feature selection. It includes the `fit_intercept` parameter and

is optimized with `alpha` values $[0.0001, 0.001, 0.01, 0.1, 100, 1000]$. For Lasso, the complexity depends on the number of features K and sample size n. Lasso, as implemented with the LARS algorithm [Efron et al. 2004], has a computational complexity of $O(K^3 + K^2 \cdot n)$. When $K < n$, the computational complexity is $O(K^2 \cdot n)$, and when $K \geq n$, it increases to $O(K^3)$.

These models allow BITWEEN to apply various regularization techniques, improving robustness and feature selection in regression tasks.

Iterative dimensionality reduction. Dimensionality reduction is performed heuristically. If some coefficients approach zero, their corresponding dimensions are removed, and the linear regression models are rerun. The total number of refinement iterations c_{iter} is constant, so the complexity is $O(c_{\text{iter}} \cdot c_{\text{params}} \cdot t_{\text{terms}} \cdot n_{\text{samples}} \cdot n_{\text{features}}^2)$.

Best Rational Approximation. Coefficients are refined using a Best Rational Approximation technique [Khinchin 1964], which involves finding the rational number (a fraction) that most closely approximates a given real number. In our context, the real number is a high-precision floating-point value determined by the linear regression algorithm. The approximation adheres to a constraint on the maximum allowable denominator.

For example, if a coefficient is 0.34, the best rational approximation with a maximum denominator of 10 is $\frac{1}{3}$. This is often more practical in our context than $\frac{34}{100}$. The maximum allowable denominator is defined as 10 times the fractional precision set by the user, with a default value of 1000.

Property testing. In order for BITWEEN to check if a randomized reduction or program property p holds on a program Π, it tests p on a random set of inputs and checks if p is satisfied (Step ⑦). Since the functional equation only contains calls to Π and known functions, it is possible to test p without knowing the actual function f. Here, we define the notion of correctness since we consider program Π almost correct in many cases, meaning that it is correct up to a small error bound ε. Let $\Pr_{x \in X}[\cdot]$ denote the probability of the event in the enclosed expression when x is uniformly chosen from X. Let $\text{error}(f, P, X)$ be the probability that $\Pi(x) \neq f(x)$ when x is randomly chosen from X.

DEFINITION 8 (ε-CORRECTNESS). *A program Π ε-computes f on the domain X if Π is correct on all but an ε fraction of inputs in the domain; that is:* $\Pr_{x \in X}[\Pi(x) \neq f(x)] < \varepsilon$.

When reasoning about properties for programs computing real-valued functions, several issues dealing with the finite precision of the computer must be dealt with. When the program works with finite precision, the properties upon which the equality testers are built will rarely be satisfied, even by a program whose answers are correct up to the required (or hardware maximum) number of digits, since they involve strict equalities. Thus, when testing, one might be willing to pass programs for which the properties are only approximately satisfied.

As an example of a property that does yield a good tester, consider the linearity property $f(x + y) = f(x) + f(y)$, satisfied only by functions for $f : \mathbb{R} \to \mathbb{R}$ of the form $f(x) = cx$, $c \in \mathbb{R}$. If, by random sampling, we conclude that the program Π satisfies this property for most x, y, it can be shown that Π agrees with a linear function g on most inputs.

Equivalence class reduction. After the RSR are learned, user can apply a set of reduction techniques (Step ⑧ in Figure 2). RSRs are first normalized, converting floating-point coefficients to rational numbers and ensuring a consistent lexicographical form. Using a union-find data structure, equations with similar terms are merged into equivalence classes, each representing a unique polynomial identity but accounting for minor numerical discrepancies from the learning process. Within each class, the "best" polynomial is chosen based on a simple scoring system that favors simpler coefficients (e.g., integers, fractions like 0.5, 0.25), which is indirectly controlled by the

Best Rational Approximation technique. The equation with the highest score is selected as the representative for that class.

Gröbner basis reduction. In Step ⑨, the chosen representatives undergo three reduction stages: (1) *Denominator Elimination*, which removes integer denominators to simplify terms; (2) *Removal of Complex Classes*, discarding classes with overly large or complex coefficients based on a predefined threshold; (3) *Groebner Basis Reduction* using SymPy's implementation of the Buchberger algorithm [Buchberger 2006], which minimizes the set of equations while preserving mathematical implications. It is an NP-hard problem with the doubly exponential worst-case time complexity in the number of variables [Bardet et al. 2015; Dubé 1990], but it is often efficient in practice for small systems of equations.

Formal Verification and Property-based Testing. In Step ⑩, the final set of polynomial equations undergoes formal verification and property-based testing to ensure that the inferred properties accurately hold for the functions they represent. This process involves testing the properties on a large number of random inputs and verifying that the equations are satisfied within a specified error margin. When learning program properties, BITWEEN automatically inserts the learned properties into the corresponding code locations as assertions (see the Bresenham Figure 6 and z3sqrt Figure 7 code examples in Appendix). For formal verification, we use CIVL [Siegel et al. 2015a], a symbolic execution tool for C, and perform bounded model checking to validate the properties within the specified intervals.

4.3 Finding Randomized Reductions in Library Setting

The concept of randomized reductions can be extended to what is known as the "library setting" [Blum et al. 1993]. In the library setting, RSRs of certain functions are assumed to be available in a library. This allows for generation of randomized reductions using the library rather than learning RSR properties from scratch. BITWEEN can construct randomized reductions for a given complicated function using randomized self-reductions of elementary functions that BITWEEN learned previously in the library setting.

Let's consider the *Savage loss* [Masnadi-Shirazi and Vasconcelos 2008] function as an example. It is used in classification tasks in machine learning and is defined as: $f(x) = 1/(1 + e^x)^2$. Let $g(x) = e^x$. We aim to derive the RR property for this function using BITWEEN. The RSR property for $g(x)$ is: $g(x+r) = g(x)g(r)$. By applying this property, we can derive the following RR for the Savage loss function: $f(x+r)g(x)^2g(r)^2 + 2f(x+r)g(x)g(r) + f(x+r) - 1 = 0$. We can now express $f(x+r)$ in terms of $g(x)$ and $g(r)$ as follows: $f(x+r) = 1/(g(x)^2g(r)^2 + 2g(x)g(r) + 1)$. This example illustrates how complex functions like the Savage loss can be reduced using RSR properties of elementary functions, leading to a randomized reduction (see Definition 9) that encapsulates the function's behavior under addition using different target functions.

4.4 Learning Program Invariants

Loop invariants are critical aspects of program verification, serving as the foundation for establishing the correctness of loops within algorithms and ensuring the correctness of the program as a whole. A loop invariant is a property that holds true before and after each iteration of the loop, thereby ensuring the loop's correctness under all iterations. Modern formal verification tools such as Dafny [Leino 2010] rely heavily on loop invariants to analyze code having loops. When invariants are provided by the user, these tools can prove properties about programs more efficiently.

The provided C code in Figure 3 efficiently computes the cubes of consecutive integers from 0 up to a specified integer a. This algorithm, attributed to Cohen [2013], leverages a method that avoids direct multiplication, instead using iterative additions to calculate each cube. The algorithm

```
1  int cubes(int a) {
2    int n = 0, x = 0, y = 1, z = 6;
3    while (n <= a) {
4      p₁(a, n, x, y, z);
5      n = n + 1;
6      x = x + y;
7      y = y + z;
8      z = z + 6;
9    }
10   p₂(a, n, x, y, z);
11   return x;
12 }
```

	a	n	x	y	z
	45	0	0	1	6
p_1	45	1	1	7	12
	45	2	8	19	18
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	45	46	97336	6487	282
p_2	30	31	29791	2977	192
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Traces of the cubes function execution

Fig. 3. *Bitween's automated learning of program invariants:* The cubes function computes the sum of the first a cubes. Bitween learns the loop invariant at line 4 (see Table 1) and post condition (see Table 2) at line 10 automatically by observing the function traces. At both locations, Bitween is instructed to learn the function p, using the queries a, n, x, y, and z.

Table 1. Learned Loop Invariants for p_1 (1323 Samples given, Query Functions: {n, x, y, z})

#	Model	Label	d	Invariant/Property	ε-error	m
1	Linear(intercept: T, positive: F)	z	5	$-6n + z - 6 = 0$	0.0	50
2	Linear(intercept: T, positive: T)	y	15	$-nz/2 + y - 1 = 0$	0.0	150
3	Lasso($\alpha : 0.0001$, intercept: T)	z	15	$-6n + z - 6 = 0$	0.0	150
4	Refine(Linear()).1	xz	3	$-6nx + xz - 6x = 0$	0.0	40
5	Lasso($\alpha : 0.0001$, intercept: T)	z	35	$-6n + z - 6 = 0$	0.0	350

is based on the observation that the difference between consecutive cubes n^3 and $(n + 1)^3$ can be expressed as $(n + 1)^3 - n^3 = 3n^2 + 3n + 1$, which allows the computation of each subsequent cube by adding the appropriate increment to the current cube, thereby avoiding direct exponentiation.

In Step ❶ in Figure 2, Bitween does not require query functions to be constructed with specific randomness values r and x to learn loop invariants and post-conditions. Instead, the user provides the program and specifies the locations where invariants and post-conditions should be learned. In this example, the user designates the functions p_1 and p_2 at lines 4 and 10, respectively. Additionally, the user supplies query functions a, n, x, y, and z to guide the learning process. In a side-effect-free program, all these query functions depend on the nondeterministic input a, so there is no need for the user to provide any randomness r to the learning algorithm. The samples are derived from the traces of the function execution, as shown in Figure 3, to feed into Step ❷. Those function executions are triggered by a simple type-directed fuzzer that we developed for C language.

Table 1 and Table 2 show reports that Bitween generates. Different linear regression models were learned by Bitween for the functions p_1 and p_2 with distinct supervised target functions (Step ❹). The last column, m, indicates the number of samples used (sampling complexity) to learn each model.

In Step ❽, a union-find algorithm is applied to merge the learned invariants based on structural similarity: $6nx - xz + 6x = 0 \land nz - 2y + 2 = 0 \land 6n - z + 6 = 0$. After applying Gröbner basis reduction, the learned invariants are simplified to $6n - z + 6 = 0 \land 12y - z^2 + 6z - 12 = 0$ in Step ❾. In these examples, all the learned invariants are formally verified in Step ❿ using a bounded model checker, Civl [Siegel et al. 2015b].

Table 2. Learned Post-Conditions for p_2 (20 Samples given, Query Functions: {a, n, x, y, z})

#	Model	Label	d	Invariant/Property	ε-error	m
1	Linear(intercept: F, positive: F)	a	6	$a - 2n + z/6 = 0$	0.0	20
2	Linear(intercept: F, positive: F)	n	6	$-a/2 + n - z/12 = 0$	0.0	20
3	Lasso(α : 0.0001, intercept: T)	n	21	$-a + n - 1 = 0$	0.0	20
4	Lasso(α : 0.0001, intercept: T)	z	21	$-6a + z - 12 = 0$	0.0	20
5	Refine(Linear()).1	n^2	6	$-az/9 + n^2 - nz/18 + z/18 - 1 = 0$	0.0	16
6	Refine(Linear()).2	nx	11	$ax/14 + ay/11 - az/15 + nx + \cdots = 0$	0.001	16

5 EMPIRICAL EVALUATION

5.1 Benchmark Programs and Properties

5.1.1 RSR-Bench for Randomized Self-Reductions. We create a benchmark, RSR-Bench, for evaluating the learning of randomized self-reductions. RSR-Bench covers many mathematical functions commonly used in scientific and machine learning applications (Table 3). In our evaluation, we introduced a new collection of benchmarks comprising 40 continuous real-valued functions and their RSR (see Definition 1) and RR (see Definition 9), primarily sourced from the self-testing/correctness literature [Blum et al. 1993; Rubinfeld 1999] and mathematical texts on functional equations [Aczél 1966; Kannappan 2009; Kuczma 2009]. We also added some activation and loss functions used in machine learning such as sigmoid, logistics, tanh, and squared loss. For each benchmark, we added the RSR or RR having the smallest query complexity (see Table 3 and Table 5 in Appendix).

All programs implementing these functions are based on the GNU C Library's (glibc) mathematical functions. Trigonometric Functions provide essential calculations like sine, cosine, and tangent, integral to applications in engineering and physics. Exponentiation and Logarithm functions are crucial for algorithms involving growth patterns and financial modeling. Hyperbolic Functions are heavily used in complex analysis and signal processing.

5.1.2 NLA-DigBench for Program Invariants. We use programs from the NLA test suite [Nguyen et al. 2014a] within the SV-COMP benchmark [Beyer 2017]. This test suite comprises 28 programs implementing mathematical functions such as intdiv, gcd, lcm, and power_sum. Although these programs are relatively short (under 50 lines of code), they contain nontrivial structures, including nested loops and nonlinear invariant properties. To the best of our knowledge, NLA is the largest benchmark suite for programs involving nonlinear arithmetic. Many of these programs have also been used to evaluate other numerical invariant systems [Le et al. 2020; Nguyen et al. 2014a, 2022; Rodríguez-Carbonell and Kapur 2007; Sharma et al. 2013]. These programs come with known invariants at various program locations, primarily nonlinear equalities used as loop invariants and post-conditions.

We extended this benchmark with six additional programs that include floating-point operations (e.g., Wensley2, z3sqrt) and non-deterministic behavior (e.g., Readers), bringing the total to 34 benchmarks. For this experiment, we evaluate Bitween by identifying invariants at these specified locations and comparing the results with known invariants and those inferred by DIG [Nguyen et al. 2014a] and SymInfer [Nguyen et al. 2022]. The benchmarks are detailed in Table 4.

5.2 Setup

The benchmarks were executed on a MacBook Pro with 32GB memory and an Apple M1 Pro 10-core CPU processor. For benchmarks on trigonometric, hyperbolic, and exponential functions, Bitween was configured to generate terms up to degree 3, for the rest is set as degree 2. We selected a

uniform sampling distribution in the [-10, 10] interval. We set the error bound at $\delta = 0.001$ for all benchmarks.

Linear regression vs. MILP. Using RSR-BENCH (Section 5.1.1), we compare BITWEEN's linear regression backend to a Mixed-Integer Linear Programming (MILP) variant, which we call BITWEEN-MILP. In the tooling architecture, we integrated an MILP solving into BITWEEN by replacing steps ⑤ and ⑥ with an MILP solver.

Furthermore, we experimented with additional, MILP-specific features in an attempt to make BITWEEN-MILP competitive with its Linear Regression-based counterpart. To drive BITWEEN-MILP to infer simpler properties, we set coefficient bounds of the MILP to 20, as MILP algorithms tend to produce more complex solutions. Adding a timeout for the MILP solver improved efficiency. When the MILP solver struggles with a problem, likely due to numerical errors or approximations in the implementation, BITWEEN-MILP restarts the inference with new samples after a specified timeout, typically set to 3 seconds. This process can be repeated up to three times.

Meanwhile, BITWEEN-MILP uses Gurobi LP solver [Gurobi Optimization 2023] (with an academic license) by default, but it also supports open-source PuLP [Mitchell et al. 2011] and GLPK [Makhorin 2020] solvers.

We observe that MILP solvers tend to find complex solutions. Once the MILP solver finds a solution, we would want to block this solution and check if there exists a new one. However, we cannot efficiently perform blocking evaluations[7] in MILP solvers as it is done in SAT/SMT [De Moura and Bjørner 2008] solvers. As the negation of conjunction of coefficient assignments results in a disjunctive constraint which is not convex [Cococcioni and Fiaschi 2021], and MILP solvers are not good at dealing with them. Therefore, in order to have simpler equations in case BITWEEN does not find any, we incorporated a heuristic: once a property is detected and passes the property test, BITWEEN sequentially blocks the three most computationally expensive terms from the inferred property (here, the 'cost' of a term is defined to be relative to the total number of algebraic operations in it), and try to search for a new optimal solution.

5.3 Results

5.3.1 RSR-BENCH. Our experiment entails executing BITWEEN on 40 input programs; we repeated the experiment 5 times to determine statistical significance, with each experiment taking less than an hour on average. BITWEEN successfully inferred a total of 193 properties, including 46 specific properties we sought after Groebner reductions. Furthermore, they were automatically verified using Sympy's simplification algorithm for the sake of this evaluation [Meurer et al. 2017]. When supplied with the symbolic version of a function, BITWEEN effectively simplifies each property by substituting function calls with their symbolic equivalents. All inferred properties were found to be correct. BITWEEN discovered all random self-reducible properties of univariate and bivariate functions over integer and real fields in the literature. To the best of our knowledge, BITWEEN also discovered an undocumented instance of random self-reducibility for the sigmoid function (see Figure 1).

Our evaluation on RSR-BENCH demonstrates the clear efficiency of the linear regression (LR) approach in BITWEEN compared to the Mixed-Integer Linear Programming (MILP) backend. In terms of sample complexity, LR required only 594 samples, achieving a significant reduction of 46% compared to the 1,095 samples needed by MILP. Similarly, LR outperformed MILP in runtime, completing evaluations in 130.53 seconds, a 30% improvement over MILP's 187.47 seconds. These results highlight LR's superior efficiency in both sample usage and execution time, making it a more scalable and effective approach for learning randomized reductions.

---

[7]https://theory.stanford.edu/~nikolaj/programmingz3.html#sec-blocking-evaluations

In general, Bitween requires fewer samples than Bitween-Milp, as shown in Figure 4d, where all points are below the $y = x$ line. The sampling method we employed incrementally increases the sample size until the method produces a result. We began with 5 random samples, and increased the sample size by increments of 5 up to 15, after which we increased the sample size up to 100 by 10. Across all benchmarks, Bitween consistently solved the benchmarks with fewer samples. Over a total of 40 benchmarks, Bitween-Milp used 1,095 samples, whereas Bitween required almost half as many, totaling 594 samples.

In terms of execution time, Bitween (130.53 seconds) was faster overall than Bitween-Milp (187.47 seconds). This is expected, as solving MILP problems in their general form is NP-complete, while the linear regression method used by Bitween operates in polynomial time for low-degree polynomials. As the size of the query class increases ($|Q|$), Bitween-Milp requires significantly more time to solve the problem, whereas regression remains more stable. This trend is evident in benchmarks marked with [†] in Table 3, such as benchmarks 8, 9, 10, and 27. For instance, Program 27 took 45 seconds for Bitween-Milp, compared to only 7.08 seconds for Bitween.
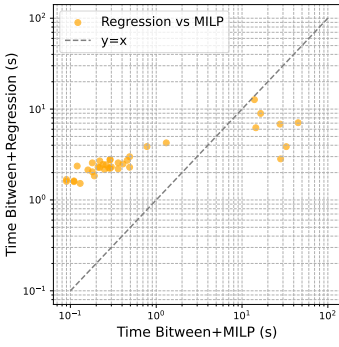
On the other hand, for smaller query classes, Bitween-Milp performs better than Bitween, as can be observed in Table 3. This is further illustrated in Figure 4a, where the data forms two distinct clusters. The cluster on the left represents benchmarks with smaller query classes, while the cluster on the right corresponds to benchmarks with larger query classes.

For Benchmark 4 (marked with [‡]), we were unable to find an RSR property. Therefore, we do not know if an RSR property exists outside the given query class, which has a size of 126. However, when we ran the same program with a library setting (Benchmark 5), both approaches successfully identified the RR property. In this benchmark, MILP solver was able to detect infeasibility very quickly. In contrast, Bitween's regression algorithm must perform all computations to determine whether a model exists within the recovery class. As a result, Bitween-Milp took 92.80 seconds to complete this program. For the remaining benchmarks, the majority of computation times for Bitween were significantly lower, generally fluctuating between 2 and 7 seconds. This demonstrates the efficiency and stability of the regression-based approach for most benchmarks.

Bitween-Milp with different solver backends (Gurobi [Gurobi Optimization 2023], GLPK [Makhorin 2020], and PuLP [Mitchell et al. 2011]) couldn't find the correct coefficients for the Sigmoid and Logistic benchmarks, whereas Bitween was able to successfully learned RSRs for them. Interestingly, the scaled version of the Logistic benchmark was also solvable by Bitween-Milp (program 37). This may be attributed to numerical instability in the MILP solvers while dealing with data having large differences among coefficients.

Our evaluation over RSR-Bench shows that Bitween outperforms Bitween-Milp in scalability, stability, sample efficiency, and execution time, excelling in large query classes where MILP struggles. Bitween requires fewer samples, runs faster, and even discovers novel properties, such as undocumented randomized self-reduction for the sigmoid and logistic functions. While Bitween-Milp performs well in smaller query classes, its numerical instability underscores Bitween's robustness and effectiveness, making it the superior method for learning randomized reductions.
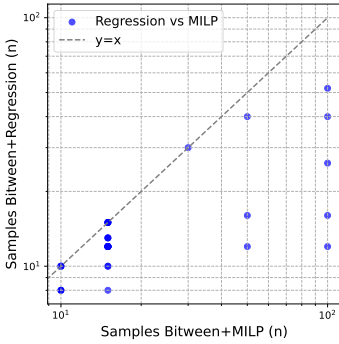
5.3.2 NLA-DigBench. Bitween can generate nonlinear loop invariants for programs, building upon the ideas of predecessor tools such as Daikon [Ernst et al. 1999, 2007], Dig [Nguyen et al. 2014a], NumInv [Nguyen et al. 2017a], and SymInfer [Nguyen et al. 2017b, 2022]. While these tools laid the groundwork for invariant detection, they are limited by their inability to reason about real-valued implementations. Our method fills this gap by generating loop invariants characterized as real-valued linear and nonlinear functional equations, thus providing greater precision and reliability in analysis of loops that process real numbers.

(a) Time: LR vs. Milp   (b) Time: Bitween vs. Dig   (c) Time: Bitween vs. SymInfer

(d) Samples: LR vs. Milp   (e) Samples: Bitween vs. Dig   (f) Samples: Bitween vs. SymInfer

Fig. 4. Comparison of Time and Samples for Bitween-Milp vs. Bitween on RSR-Bench and Bitween vs. Dig, and Bitween vs. SymInfer on NLA-DigBench.

Our evaluation on NLA-DigBench highlights the advantages of Bitween over existing tools such as Dig and SymInfer in terms of sample efficiency, despite differences in runtime. Bitween demonstrated exceptional sample efficiency, requiring only 12,399 samples compared to 67,981 for Dig and 39,789 for SymInfer, achieving a significant reduction of 82% and 69%, respectively. This indicates Bitween's ability to infer properties with far fewer samples, making it more data-efficient. In terms of runtime, Dig was the fastest at 90.69 seconds, followed by Bitween at 180.57 seconds, while SymInfer required 446.83 seconds. While Bitween's runtime was slightly higher than Dig, it strikes a better balance between sample efficiency and execution time, outperforming SymInfer in both metrics. These results emphasize Bitween's capability to maintain high inference accuracy while optimizing resource usage. While Bitween successfully identifies all program properties, Dig failed to solve 8 properties, primarily due to its inability to handle floating-point types, and SymInfer failed to solve 11 properties, mainly due to its reliance on symbolic execution and limitations in handling floating-point operations.

6 DISCUSSION

What is the difference between our approach and the existing symbolic regression methods? We compared the results of Bitween with the symbolic regression backends integrated into our framework. Traditional genetic programming (GP)-based symbolic regression methods, such as

Table 3. RSR-Bench: Comparison of Bitween and Bitween-Milp on RSR-Bench (Random Self-Reducible Benchmarks). Lib.: Selected as a Library function. $|Q|$: Size of the Query Class. Sample: Sample Complexity.

#	Program	Specification	L	$|Q|$	Bitween-Milp			Bitween		
					Sample	Time	RSR	Sample	Time	RSR
1	Linear	$f(x) = cx + a$	Y	5	10/10	0.13	✓	10/10	1.52	✓
2	Exp	$h(x) = e^{cx}$	Y	15	10/10	0.21	✓	8/10	2.29	✓
3	Exp_min_one	$f(x) = e^{cx} - 1$	Y	15	10/10	0.41	✓	8/10	2.49	✓
4	Exp_divby_x	$f(x) = e^x/x$	-	126†	None	0.48†	✗‡	None	92.80†	✗‡
5	Exp_divby_x1¶	$f(x) = e^x/x$	-	15	10/10	0.29	✓^{RR}	8/10	2.80	✓^{RR}
6	Sum	$g(x, y) = x + y$	Y	6	15/15	0.19	✓	10/15	1.84	✓
7	Mean	$f(x, yz) = (x+y+z)/3$	-	8	15/15	0.18	✓	10/15	2.04	✓
8	Tan	$f(x) = \tan cx$	Y	35†	15/15	32.77†	✓	8/10	3.86†	✓
9	Cot	$f(x) = \cot cx$	Y	35†	15/15	28.16†	✓	12/15	2.83†	✓
10	Squared Diff.	$f(x, y) = x^2 - y^2$	-	45†	15/15	45.05†	✓	13/15	7.08†	✓
11	Inv. Squared	$f(x) = 1/x^2$	-	15	15/15	0.22	✓	12/15	2.28	✓
12	Inverse	$f(x) = c/x$	-	15	15/15	0.28	✓	15/15	2.25	✓
13	Inv_add	$f(x) = 1/(x + 1)$	-	15	15/15	0.18	✓	12/15	2.56	✓
14	Inv_Cot_Add	$f(x) = 1/(1 + \cot cx)$	-	20	15/15	0.78	✓	12/15	3.87	✓
15	Inv_Tan_Add	$f(x) = 1/(1 + \tan cx)$	-	20	30/40	0.46	✓	30/40	2.73	✓
16	Inv_Sub	$f(x) = cx/(1 - cx)$	-	15	15/15	0.30	✓	12/15	2.31	✓
17	Inv_Sub2	$f(x) = -cx/(1 - cx)$	-	15	15/15	0.29	✓	12/15	2.75	✓
18	Cos	$f(x) = \cos(cx)$	Y	15	15/15	0.29	✓	12/15	2.23	✓
19	Cosh	$f(x) = \cosh(cx)$	Y	15	15/15	0.25	✓	12/15	2.16	✓
20	Squared	$f(x) = cx^2$	Y	15	15/15	0.49	✓	13/15	2.29	✓
21	Sin	$\alpha(x) = k \sin cx$	Y	15	15/15	0.22	✓	12/15	2.69	✓
22	Sinh	$f(x) = k \sinh cx$	Y	15	15/15	0.16	✓	12/15	2.14	✓
23	Cube	$f(x) = x^3$	-	15	15/15	0.26	✓	13/15	2.44	✓
24	Log	$f(x) = \log x$	Y	4	10/10	0.09	✓	10/10	1.60	✓
25	Sec	$f(x) = \sec x$	Y	35	50/50	1.31	✓	40/50	4.25	✓
26	Csc	$f(x) = \csc x$	Y	70†	100/100	13.94†	✓	52/100	12.70	✓
27	Sinc¶	$f(x) = \sin x/x$	-	56†	50/50	16.48†	✓^{RR}	12/15	8.95	✓^{RR}
28	Sinc2	$f(x) = \sin x/x$	-	10	15/15	0.12	✓	15/15	2.36	✓
29	Mod	$f(x, R) = x \mod R$	-	4	10/10	0.09	✓	10/10	1.68	✓
30	Modular Mult.	$f(x, y, R) = (x \cdot_R y)$	-	6	10/10	0.11	✓	10/10	1.62	✓
31	Integer Mult.	$f(x, y) = x \cdot y$	-	6	15/15	0.11	✓	15/15	1.59	✓
32	Tanh	$f(x) = C \tanh cx$	Y	35†	100/100	27.72	✓	16/20	6.83	✓
33	Sigmoid**	$f(x) = 1/(1 + e^{-x})$	-	35†**	100/100	36.02†	✗**	26/30	6.70	✓
34	Softmax_1	$f(x, y) = e^x/(e^x + e^y)$	-	15	15/15	0.36	✓	15/15	2.56	✓
35	Softmax_2	$f(x, y) = e^y/(e^x + e^y)$	-	15	15/15	0.36	✓	15/15	2.20	✓
36	Logistic**	$f(x) = 1/(1 + e^{-2x})$	-	35**	100/100	34.09†	✗**	12/15	2.33	✓
37	Scaled Logistic	$f(x) = 3/(1 + e^{-2x})$	-	35**	100/100	24.31†	✓	25/30	6.23	✓
38	Square Loss	$f(x) = (1 - x)^2$	-	15	15/15	0.49	✓	12/15	2.99	✓
39	Savage Loss¶**	$f(x) = 1/(1 + e^x)^2$	-	35**	50/50	26.51†	✗**	16/20	4.03	✓^{RR}
40	Savage Loss¶	$f(x) = 1/(1 + e^x)^2$	-	56†	100/100	14.48†	✓^{RR}	40/60	6.23	✓^{RR}

‡ The MILP solver can quickly detect infeasibility, while the regression-based approach requires full computation to determine if a model exists within the recovery class, resulting in a longer runtime.

† MILP, in its general form, is an NP-complete problem, whereas regression operates in polynomial time. Consequently, for benchmarks with high query complexity, regression tends to be more efficient than MILP.

¶ In these benchmarks, the library setting (see Section 4.3) was used.

** Bitween-Milp with different solver backends (Gurobi [Gurobi Optimization 2023], GLPK [Makhorin 2020], and PuLP [Mitchell et al. 2011]) couldn't find the correct coefficients for the Sigmoid and Logistic benchmarks, whereas Bitween was able to successfully learned RSRs for them.

^{RR} Randomized Reductions: Indicates benchmarks where Randomized Self-Reductions could not find a solution within the given recovery class. Instead, the Library Setting (see Section 4.3) was used.

Table 4. NLA-DigBench: Comparison with Dig [Nguyen et al. 2014a] and SymInfer [Nguyen et al. 2022] on NLA-DigBench (Invariant Generation for Integer Benchmarks). L: Trace locations (vtrace). V: Trace variables. D: Max degree of invariants. T: Type of program (i: integer, d: double, f: float, u: unsigned).

#	Program	Benchmark					Bitween				Dig				SymInfer	
		L	V	D	T			sample	time			sample	time			sample	time
1	Bresenham	2	5	2	i		✓	2112	2.19		✓	10712	3.05		✓	3836	16.71
2	Cohencu	2	5	3	i		✓	1556	3.69		✓	3760	3.08		✓	8881	10.75
3	CohenDiv	3	6	2	i		✓	96	3.73		✓	634	0.98		✓	255	19.04
4	Dijkstra	3	5	3	i		✓	208	6.95		✓	301	2.85		✓	513	9.57
5	Divbin	3	5	2	i		✓	90	2.02		✓	679	0.92		✓	212	6.41
6	Egcd	2	8	2	i		✓	299	6.59		✓	4580	7.81		✓	3564	48.91
7	Egcd2	3	10	3	i		✓	629	9.21		✓	4342	43.50		✓	11635	72.07
8	Egcd3	4	12	2	i		✓	562	16.84		✓§	2457	3.63		✓§	1693	15.05
9	Fermat1	4	5	2	d		✓	965	4.94		✓	9578	1.46		✗**	–	6.09
10	Fermat2	2	5	2	d		✓	336	5.59		✓	2805	1.29		✓	157	15.74
11	Freire1_int	1	3	2	i		✓	175	1.36		✓	417	0.64		✓	223	5.57
12	Freire1	1	3	2	d		✓	111	1.42		✗‡	–	0.45		✗‡	–	2.09
13	Freire2	1	3	3	d		✓	39	1.22		✗‡	–	0.45		✗‡	–	4.71
14	Geo1	2	4	2	i		✓	54	2.58		✓	140	0.76		✓	114	7.05
15	Geo2	2	4	2	i		✓	60	2.66		✓	140	0.75		✓	115	6.69
16	Geo3	2	5	3	i		✓	166	5.77		✗	159	0.49		✗	438	10.78
17	Hard	3	6	2	i		✓	44	3.35		✓	659	1.07		✓	264	7.01
18	Knuth*	2	8	3	u		✓	144	21.99		✗¶	–	1.16		✗†	–	5.43
19	Lcm1	3	6	2	i		✓	238	3.41		✗	4253	1.16		✗†	–	5.26
20	Lcm2	2	6	2	i		✓	153	2.38		✓	4881	1.84		✗†	1719	43.40
21	Mannadiv	2	5	2	i		✓	1230	2.42		✓	10586	1.59		✓	2916	7.20
22	Prod4br	2	6	2	i		✓	127	2.26		✓	810	5.22		✓	1056	49.30
23	Prodbin	2	5	2	d		✓	95	2.21		✓	790	0.94		✓	256	23.52
24	Ps1	1	3	1	i		✓	1128	1.39		✓	3950	0.68		✗†	–	5.19
25	Ps2	2	3	2	i		✓	35	1.88		✓	228	0.62		✓	324	6.67
26	Ps3	2	3	3	i		✓	164	2.56		✓	228	0.77		✓	324	6.67
27	Ps4	2	3	4	i		✓	223	2.63		✓	228	1.07		✓	468	7.44
28	Ps5	2	3	5	i		✓‖	154	11.36		✓	154	1.25		✓	251	7.19
29	Ps6	2	3	6	i		✓‖	154	22.65		✓	229	2.57		✓	350	8.05
30	Readers‡‡	2	6	2	i		✓	559	4.40		✗	?	?		✗††	–	5.19
31	Sqrt	2	4	2	i		✓	146	2.81		✓	281	0.70		✓	225	6.08
32	Wensley2	2	7	2	f		✓	208	7.54		✗‡	–	0.45		✗‡	–	2.09
33	z3sqrt	2	4	2	f		✓	27	3.86		✗‡	–	0.45		✗‡	–	2.09
34	isqrt	2	4	2	i		✓	112	4.71		✓	301	4.59		✓	324	8.76

‡ DIG could not find invariants due to floating-point operations in the benchmark.
† SymInfer could not handle external function calls, as it requires analyzing symbolic states.
¶ DIG could not call the external sqrt function because it returns floating-point results.
§ DIG and SymInfer did not find all target invariants.
* New invariants for Knuth's post-conditions were found; these are not reported in the literature.
** SymInfer could not reach the end for these benchmarks.
‡‡ Benchmark includes non-deterministic choices.
†† Symbolic execution failed due to non-determinism.
‖ Due to numerical instability, advanced scaling techniques were applied for Ps5 and Ps6.

PySR [Cranmer et al. 2023] and GPlearn [Arno 2020], face scalability challenges and often fail to capture certain invariants, particularly when dealing with a high-dimensional feature space or integer-valued invariants. Symbolic regression methods are designed to explore a wide range of term compositions, which makes them more general but less efficient for learning randomized self-reductions and program invariants. In contrast, Bitween focuses exclusively on polynomial
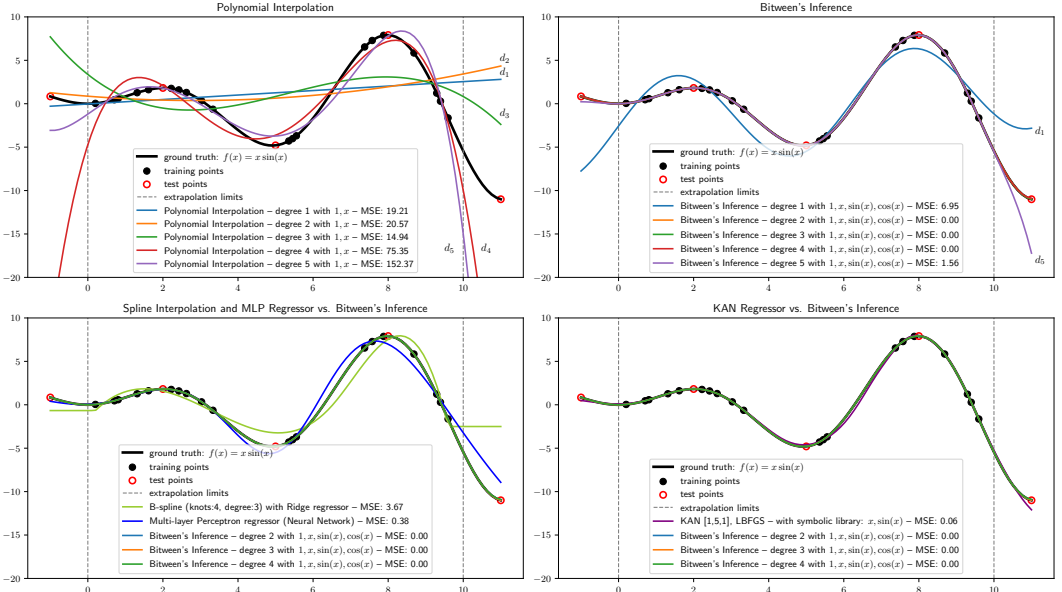
Fig. 5. Comparison with Polynomial Interpolation, Spline Interpolation, Multi-Layered Perceptron (MLP) Regressor, and Kolmogorov-Arnold Networks (KAN) [Liu et al. 2024]. MSE: Mean Squared Error.

interactions of terms, as these are representative of the benchmark suites it targets. While Bitween has the capacity to handle deeper levels of function composition, this aspect has not been explicitly evaluated. To provide a fair comparison, we configured symbolic regression methods to include only multiplication, subtraction, and addition operations. Our experiments demonstrate that the results obtained using symbolic regression methods are generally less accurate and approximately 100 times slower compared to Bitween.

What is the difference between our approach and existing ML-based regression methods? We compared our results with those obtained from Polynomial Interpolation, Spline Interpolation, Multi-Layered Perceptron (MLP) Regressor, and Kolmogorov-Arnold Networks (KAN) [Liu et al. 2024]. Figure 5 illustrates the performance of these methods alongside Bitween in regressing the function $f(x) = x \sin x$. The MLP Regressor, while accurate, lacks interpretability and requires a large dataset to fit the model effectively. Polynomial regression, though interpretable, is less accurate and prone to instability when using high-degree polynomials, particularly for extrapolation beyond the fitted data range. B-spline interpolation offers improved accuracy with an increased number of knots but remains uninterpretable. KAN provides both interpretability and accuracy for fitting data; however, its symbolic interpretation requires manual analysis and lacks scalability. We integrated the KAN regressor into Bitween by removing steps ⑤ and ⑥ and attaching the KAN regressor, but its performance was slow, and the symbolic interpretation was only partially accurate. Generally, ML methods yield black-box models with high predictive power but limited interpretability. In contrast, our method produces a symbolic model that is both interpretable and capable of elucidating the underlying relationships within the data.

How do we handle extrapolation? Is the model robust to unseen data? We focus primarily on continuous functions rather than piecewise functions. If the model can predict without errors, it remains robust to unseen data. We expect users to have prior domain knowledge about the function.

7 RELATED WORK

BITWEEN uses dynamic analysis to discover challenging functional equations for programs. This line of dynamic invariant generation work is pioneered by the Daikon system [Ernst et al. 2007], which demonstrated that program properties can be inferred by observing concrete program states. Daikon used a template-based approach to define candidate invariants and, to mitigate cost, a rather modest set of templates is used that do not capture nonlinear properties.

The most related dynamic invariant generation tool to BITWEEN is the DIG [Nguyen et al. 2012, 2014b, 2021] approach that focuses on numerical invariants, which include nonlinear equations, min-max, and octagonal inequalities. Many researchers have built on DIG to infer invariants for various purposes. For example, Dynamite [Le et al. 2020] uses DIG to infer ranking functions and recurrent sets to prove/disprove termination and Dynaplex [Ishimwe et al. 2021] extends DIG's numerical templates to find recurrent relations.

BITWEEN is inspired by Daikon and DIG's dynamic and template approach, but significantly extends these templates and develops new linear regression based algorithms to learn randomized reductions and program properties. The resulting properties discovered by BITWEEN are beyond the capabilities of existing invariant tools, e.g., both Daikon and DIG cannot generate such complex functional equations.

In contrast to dynamic inference, static analyses based on the classical abstract interpretation framework [Cousot 1996; Cousot and Cousot 1992; Miné 2006] generate sound invariants under abstract domains (e.g., interval, octagonal, and polyhedra domains) to overapproximate program behaviors to prove the absence of errors [Cousot et al. 2005]. Trade-offs occur between the efficiency and expressiveness of the considered domains. The work in [Rodríguez-Carbonell and Kapur 2004, 2007] uses the domain of nonlinear polynomial equalities and Gröbner basis to generate equality invariants. This approach is limited to programs with assignments and loop guards expressible as polynomial equalities and requires user-supplied bounds on the degrees of the polynomials to ensure termination. Similarly, Sankaranarayanan et al. [2004] reduces the nonlinear invariant generation problem to a parametric linear constraint-solving problem using Gröbner bases, but the method is manually demonstrated only on a few examples. The complexity of Gröbner basis computation in the worst case is doubly exponential in the number of variables [Bardet et al. 2015; Dubé 1990]. Various algorithms like Faugère's F4/F5 [Faugere 1999, 2002] and M4GB [Makarim and Stevens 2017] have been proposed to obtain Gröbner bases in better runtime. Recently, Kera et al. [2023] introduced a new direction by employing Transformer-based models to compute Gröbner bases. This machine learning approach approximates reduction steps in Gröbner basis computation. While the outputs may be incorrect or incomplete, it is an interesting direction to revisit Gröbner basis-based approaches in settings where verification of generated properties is feasible.

Many static analyses generate invariants to prove given program specifications, and thus can exploit the given specifications to guide the invariant inference process. PIE [Padhi et al. 2016] and ICE [Garg et al. 2016] use CEGIR approach to learn invariants to prove given assertions. For efficiency, they focus on octagonal predicates and only search for invariants that are boolean combinations of octagonal relations (thus do not infer nonlinear invariants in this paper). The data-driven approach G-CLN [Yao et al. 2020] uses gated continuous neural networks to learn numerical loop invariants from program traces and uses Z3 to check inductive loop invariants. Due to their purpose, these approaches (PIE, ICE, G-CLN) cannot generate strong invariants unless they are provided with sufficiently strong goals (e.g., given postconditions), G-CLN also relies on substantial problem-specific customizations to generate invariants.

All these approaches are not capable of learning randomized (self-)reductions and program invariants in mixed integer and floating-point programs.

8 CONCLUSION

In this paper, we present Bitween, a novel approach to automating the discovery of complex program properties through a linear regression-based learning method. First, Bitween leverages a polynomial time optimization method, linear regression to effectively infer randomized self-reductions and diverse program invariants, such as loop invariants and postconditions, in mixed-integer and floating-point programs. Furthermore, we establish a comprehensive theoretical framework for learning these reductions, contributing to the formal understanding of randomized self-reducibility. Additionally, we introduce the RSR-Bench benchmark suite, featuring a range of mathematical functions used in scientific and machine learning contexts, allowing us to systematically evaluate Bitween's capabilities. Our empirical analysis demonstrates Bitween outperforms state-of-the-art tools, such as Dig [Nguyen et al. 2014a] and SymInfer [Nguyen et al. 2022] in terms of capability, runtime, and sample efficiency. Lastly, Bitween, along with all benchmark data, is available as an open-source package, accessible via a user-friendly web interface that supports C language program inputs (see https://bitween.fun).

REFERENCES

Martin Abadi, Joan Feigenbaum, and Joe Kilian. 1987. On hiding information from an oracle. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 195–203.

János Aczél. 1966. *Lectures on functional equations and their applications*. Academic press.

Trevor K. Arno. 2020. gplearn: Genetic programming for symbolic regression in Python. https://gplearn.readthedocs.io/.

Magali Bardet, Jean-Charles Faugère, and Bruno Salvy. 2015. On the complexity of the F5 Gröbner basis algorithm. *Journal of Symbolic Computation* 70 (2015), 49–70.

Donald Beaver and Joan Feigenbaum. 1990. Hiding instances in multioracle queries. In *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 37–48.

Donald Beaver, Joan Feigenbaum, Joe Kilian, and Phillip Rogaway. 1991. Security with low communication overhead. In *Advances in Cryptology-CRYPTO'90: Proceedings 10*. Springer, 62–76.

Dirk Beyer. 2017. Software Verification with Validation of Results: (Report on SV-COMP 2017). In *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 331–349.

Manuel Blum and Sampath Kannan. 1995. Designing programs that check their work. *Journal of the ACM (JACM)* 42, 1 (1995), 269–291.

Manuel Blum, Michael Luby, and Ronitt Rubinfeld. 1993. Self-Testing/Correcting with Applications to Numerical Problems. *J. Comput. Syst. Sci.* 47, 3 (1993), 549–595. https://doi.org/10.1016/0022-0000(93)90044-W

Manuel Blum and Silvio Micali. 2019. How to generate cryptographically strong sequences of pseudo random bits. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 227–240.

Bruno Buchberger. 2006. Gröbner Bases: A Short Introduction for Systems Theorists. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*. ACM, 1–19.

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279.

Edmund M Clarke and E Allen Emerson. 2008. Design and synthesis of synchronization skeletons using branching time temporal logic. *25 Years of Model Checking: History, Achievements, Perspectives* (2008), 196–215.

Marco Cococcioni and Lorenzo Fiaschi. 2021. The Big-M method with the numerical infinite M. *Optimization Letters* 15, 7 (2021), 2455–2468.

Henri Cohen. 2013. *A course in computational algebraic number theory*. Vol. 138. Springer Science & Business Media.

Patrick Cousot. 1996. Abstract interpretation. *ACM Computing Surveys (CSUR)* 28, 2 (1996), 324–328.

Patrick Cousot and Radhia Cousot. 1992. Abstract interpretation frameworks. *Journal of logic and computation* 2, 4 (1992), 511–547.

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The Astrée analyzer. In *European Symposium on Programming*. Springer, 21–30.

Miles Cranmer, Muhammad Farhan Kasim, and Brian Nord. 2023. PySR: Fast & parallelized symbolic regression in Python/Julia. *SoftwareX* 18 (2023), 101090.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

Thomas W Dubé. 1990. The structure of polynomial ideals and Gröbner bases. *SIAM J. Comput.* 19, 4 (1990), 750–773.

Bradley Efron, Trevor Hastie, Iain Johnstone, and Robert Tibshirani. 2004. Least angle regression. (2004).

Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 1999. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*. 213–224.

Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.

Jean-Charles Faugere. 1999. A new efficient algorithm for computing Gröbner bases (F4). *Journal of pure and applied algebra* 139, 1-3 (1999), 61–88.

Jean Charles Faugere. 2002. A new efficient algorithm for computing Gröbner bases without reduction to zero (F 5). In *Proceedings of the 2002 international symposium on Symbolic and algebraic computation*. 75–83.

Joan Feigenbaum and Lance Fortnow. 1993. Random-self-reducibility of complete sets. *SIAM J. Comput.* 22, 5 (1993), 994–1005.

Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices* 51, 1 (2016), 499–512.

Oded Goldreich. 2017. *Introduction to Property Testing*. Cambridge University Press. https://doi.org/10.1017/9781108135252

Harrison Goldstein, Joseph W Cutler, Daniel Dickstein, Benjamin C Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

Shafi Goldwasser and Silvio Micali. 1984. Probabilistic Encryption. *J. Comput. Syst. Sci.* 28, 2 (1984), 270–299. https://doi.org/10.1016/0022-0000(84)90070-9

Shafi Goldwasser and Silvio Micali. 2019. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Providing sound foundations for cryptography: on the work of Shafi Goldwasser and Silvio Micali*. 173–201.

Shafi Goldwasser, Silvio Micali, and Chales Rackoff. 2019. The knowledge complexity of interactive proof-systems. In *Providing sound foundations for cryptography: On the work of shafi goldwasser and silvio micali*. 203–225.

Mostefa Golea, Mario Marchand, and Thomas R. Hancock. 1996. On learning ?-perceptron networks on the uniform distribution. *Neural Networks* 9, 1 (1996), 67–82. https://doi.org/10.1016/0893-6080(95)00009-7

LLC Gurobi Optimization. 2023. Gurobi Optimizer. https://www.gurobi.com/

Jun Han and Claudio Moraga. 1995. The Influence of the Sigmoid Function Parameters on the Speed of Backpropagation Learning. In *From Natural to Artificial Neural Computation, International Workshop on Artificial Neural Networks, IWANN '95, Malaga-Torremolinos, Spain, June 7-9, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 930)*, José Mira and Francisco Sandoval Hernández (Eds.). Springer, 195–201. https://doi.org/10.1007/3-540-59497-3_175

Thomas R. Hancock. 1993. Learning $k\mu$ Decision Trees on the Uniform Distribution. In *Proceedings of the Sixth Annual ACM Conference on Computational Learning Theory, COLT 1993, Santa Cruz, CA, USA, July 26-28, 1993*, Lenny Pitt (Ed.). ACM, 352–360. https://doi.org/10.1145/168304.168374

Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

Didier Ishimwe, KimHao Nguyen, and ThanhVu Nguyen. 2021. Dynaplex: analyzing program complexity using dynamically inferred recurrence relations. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–23.

Jeffrey C. Jackson, Adam R. Klivans, and Rocco A. Servedio. 2002. Learnability beyond AC0. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, John H. Reif (Ed.). ACM, 776–784. https://doi.org/10.1145/509907.510018

Jeffrey C. Jackson and Rocco A. Servedio. 2006. On Learning Random DNF Formulas Under the Uniform Distribution. *Theory Comput.* 2, 8 (2006), 147–172. https://doi.org/10.4086/TOC.2006.V002A008

Palaniappan Kannappan. 2009. *Functional equations and inequalities with applications*. Springer Science & Business Media.

Hiroshi Kera, Yuki Ishihara, Yuta Kambe, Tristan Vaccon, and Kazuhiro Yokoyama. 2023. Learning to Compute Gr\" obner Bases. *arXiv preprint arXiv:2311.12904* (2023).

Aleksandr Ya. Khinchin. 1964. *Continued Fractions*. University of Chicago Press.

Adam R. Klivans, Ryan O'Donnell, and Rocco A. Servedio. 2004. Learning intersections and thresholds of halfspaces. *J. Comput. Syst. Sci.* 68, 4 (2004), 808–840. https://doi.org/10.1016/J.JCSS.2003.11.002

Daniel Kroening and Michael Tautschnig. 2014. CBMC–C Bounded Model Checker: (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*. Springer, 389–391.

Ludek Kucera, Alberto Marchetti-Spaccamela, and Marco Protasi. 1994. On Learning Monotone DNF Formulae under Uniform Distributions. *Inf. Comput.* 110, 1 (1994), 84–95. https://doi.org/10.1006/INCO.1994.1024

Marek Kuczma. 2009. *An Introduction to the Theory of Functional Equations and Inequalities: Cauchy's Equation and Jensen's Inequality*. Birkhäuser Basel, Basel.

Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. 2020. DynamiTe: dynamic termination and non-termination proofs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*. Springer, 348–370.

Richard Lipton. 1991. New directions in testing. *Distributed computing and cryptography* 2 (1991), 191–202.

Richard J. Lipton. 1989. New Directions In Testing. In *Distributed Computing And Cryptography, Proceedings of a DIMACS Workshop, Princeton, New Jersey, USA, October 4-6, 1989 (DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 2)*, Joan Feigenbaum and Michael Merritt (Eds.). DIMACS/AMS, 191–202. https://doi.org/10.1090/DIMACS/002/13

Ziming Liu, Yixuan Wang, Sachin Vaidya, Fabian Ruehle, James Halverson, Marin Soljačić, Thomas Y Hou, and Max Tegmark. 2024. Kan: Kolmogorov-arnold networks. *ArXiv preprint* abs/2404.19756 (2024). https://arxiv.org/abs/2404.19756

Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. 1992. Algebraic methods for interactive proof systems. *Journal of the ACM (JACM)* 39, 4 (1992), 859–868.

Rusydi H Makarim and Marc Stevens. 2017. M4GB: an efficient Gröbner-basis algorithm. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*. 293–300.

Andrew Makhorin. 2020. *GNU Linear Programming Kit*. GNU Project. https://www.gnu.org/software/glpk/ Accessed: 2024-11-12.

Hamed Masnadi-Shirazi and Nuno Vasconcelos. 2008. On the Design of Loss Functions for Classification: theory, robustness to outliers, and SavageBoost. In *Advances in Neural Information Processing Systems 21, Proceedings of the Twenty-Second Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 8-11, 2008*, Daphne Koller, Dale Schuurmans, Yoshua Bengio, and Léon Bottou (Eds.). Curran Associates, Inc., 1049–1056. https://proceedings.neurips.cc/paper/2008/hash/f5deaeeae1538fb6c45901d524ee2f98-Abstract.html

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (Jan. 2017), e103. https://doi.org/10.7717/peerj-cs.103

Webb Miller and David L. Spooner. 1976. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering* 3 (1976), 223–226.

Antoine Miné. 2006. The octagon abstract domain. *Higher-order and symbolic computation* 19, 1 (2006), 31–100.

Stuart Mitchell, Michael OSullivan, and Iain Dunning. 2011. PuLP: a linear programming toolkit for python. The University of Auckland, Auckland, New Zealand.

ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017a. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 605–615.

ThanhVu Nguyen, Matthew B Dwyer, and Willem Visser. 2017b. Symlnfer: Inferring program invariants using symbolic states. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, IEEE, 804–814.

ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using dynamic analysis to discover polynomial and array invariants. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 683–693.

Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014a. Dig: A dynamic invariant generator for polynomial and array invariants. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (2014), 1–30.

ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014b. Using dynamic analysis to generate disjunctive invariants. In *Proceedings of the 36th International Conference on Software Engineering*. 608–619.

ThanhVu Nguyen, KimHao Nguyen, and Hai Duong. 2022. SymInfer: inferring numerical invariants using symbolic states. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 197–201.

Thanhvu Nguyen, KimHao Nguyen, and Matthew B Dwyer. 2021. Using symbolic states to infer numerical invariants. *IEEE Transactions on Software Engineering* 48, 10 (2021), 3877–3899.

Ryan O'Donnell. 2014. *Analysis of Boolean Functions*. Cambridge University Press. http://www.cambridge.org/de/academic/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/analysis-boolean-functions

Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven Precondition Inference with Learned Features. In *Programming Language Design and Implementation*. ACM, 42–56.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2024. *1.1. Linear Models*. scikit-learn developers. https://scikit-learn.org/stable/modules/linear_model.html#ordinary-least-squares-complexity Accessed:

2024-11-13.

Enric Rodríguez-Carbonell and Deepak Kapur. 2004. Automatic generation of polynomial loop invariants: Algebraic foundations. In *International symposium on Symbolic and algebraic computation*. 266–273.

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

Ronitt Rubinfeld. 1994. *Robust functional equations with applications to self-testing/correcting*. Technical Report. Cornell University.

Ronitt Rubinfeld. 1999. On the robustness of functional equations. *SIAM J. Comput.* 28, 6 (1999), 1972–1997.

Sriram Sankaranarayanan, Henny B Sipma, and Zohar Manna. 2004. Non-linear loop invariant generation using Gröbner bases. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 318–329.

Adi Shamir. 1992. Ip= pspace. *Journal of the ACM (JACM)* 39, 4 (1992), 869–877.

Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V Nori. 2013. A data driven approach for algebraic loop invariants. In *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*. Springer, 574–592.

Stephen F Siegel, Manchun Zheng, Ziqing Luo, Timothy K Zirkel, Andre V Marianiello, John G Edenhofner, Matthew B Dwyer, and Michael S Rogers. 2015a. CIVL: the concurrency intermediate verification language. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. 2015b. CIVL: The Concurrency Intermediate Verification Language. In *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis, Proceedings (SC '15)*. IEEE Press, Piscataway, NJ, USA, 61:1–61:12.

Martin Tompa and Heather Woll. 1987. Random self-reducibility and zero knowledge interactive proofs of possession of information. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*. IEEE, 472–482.

Luca Trevisan, Gregory B. Sorkin, Madhu Sudan, and David P. Williamson. 2000. Gadgets, Approximation, and Linear Programming. *SIAM J. Comput.* 29, 6 (2000), 2074–2097. https://doi.org/10.1137/S0097539797328847

Leslie G. Valiant. 1984. A Theory of the Learnable. *Commun. ACM* 27, 11 (1984), 1134–1142. https://doi.org/10.1145/1968.1972

Karsten A. Verbeurgt. 1990. Learning DNF Under the Uniform Distribution in Quasi-Polynomial Time. In *Proceedings of the Third Annual Workshop on Computational Learning Theory, COLT 1990, University of Rochester, Rochester, NY, USA, August 6-8, 1990*, Mark A. Fulk and John Case (Eds.). Morgan Kaufmann, 314–326. http://dl.acm.org/citation.cfm?id=92659

Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning nonlinear loop invariants with gated continuous logic networks. In *Programming Language Design and Implementation*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 106–120.

A PROOFS

Proof of Claim 6. Suppose $F \subseteq \mathrm{RSR}_k(Q, P)$ is PAC-learnable with sample complexity $m_{\mathrm{PAC}}(\varepsilon, \delta)$ and learner Λ_{PAC}. To RSR-learn F with errors (ρ, ξ, δ), we let $\varepsilon = \min(\rho/k, \xi)$ and draw $m = m_{\mathrm{PAC}}(\varepsilon, \delta)$ labeled samples $(x_i, f(x_i))_{i=1}^m$. The RSR-learner is described in Algorithm 2. For simplicity of notation, we will omit the input length n; following this proof is a brief discussion of Algorithm 2's inefficiency with respect to n.

At a high level, the learner invokes Λ_{PAC} to obtain a hypothesis $\hat{f} \in F$ that, is ε-close to the ground truth function f (with probability $\geq 1 - \delta$ over the samples). It then uses $\hat{f}$ to exhaustively search through possible query functions $\hat{q}_1, \ldots, \hat{q}_k \in Q$ and recovery functions $\hat{p} \in P$, until it finds those that are a *perfect* RSR for $\hat{f}$.

To conclude the proof, we will show that, because $\hat{f}$ is ε-close to f, then $(\hat{q}_1, \ldots, \hat{q}_k, \hat{p})$ is a (ρ, ξ)-RSR for f.

We say that $x \in X$ is *good* if $\hat{f}(x) = f(x)$. By choice of ε, we know that there are at least $(1 - \varepsilon)|X| \geq (1 - \xi)|X|$. It therefore suffices to show that for all good x,

$$\Pr_{r \sim R} \left[\begin{array}{l} f(x) = \hat{p}(x, rf(u_1), \ldots, f(u_k)) \\ \text{where } \forall i \in [k] \ u_i \coloneqq \hat{q}_i(x, r) \end{array} \right] \geq 1 - \varepsilon \cdot k \geq 1 - \rho.$$

Algorithm 2: RSR-learning via PAC-learning.

Input: Query class Q, recovery class P, and hypothesis class $F \subseteq \mathrm{RSR}_k(Q, P)$. Query complexity k and randomness domain R. Uniform PAC-learner Λ_{PAC} for F. Labeled samples $(x_i, y_i)_{i=1}^m$.

Output: Query functions $\hat{q}_1, \ldots, \hat{q}_k \in Q$ and recovery function $\hat{p} \in P$.

1 Invoke Λ_{PAC} on samples $(x_i, y_i)_{i=1}^m$ to obtain a hypothesis $\hat{f} \in F$.

2 foreach *Query functions $(q_1, \ldots, q_k) \in Q^k$ and recovery function $p \in P$* **do**

3 $\quad$ **foreach** $x \in X$ *and* $r \in R$ **do**

4 $\quad\quad$ Compute $u_i := q_i(x, r)$ for each $i \in [k]$.

5 $\quad\quad$ **if** $\hat{f}(x) \neq p(x, r, \hat{f}(u_1), \ldots, \hat{f}(u_k))$ **then**

6 $\quad\quad\quad$ Go to line 2. $\qquad\qquad\qquad\qquad$ `# Continue to the next` $q_1, \ldots, q_k, p.$

7 $\quad$ Output $(\hat{q}_1, \ldots, \hat{q}_k, p) := (q_1, \ldots, q_k, p)$.

8 Output $\bot$.

The right inequality is by choice of $\varepsilon \leq \rho/k$. For the left inequality,

$$\Pr_{r \sim R} \left[\begin{array}{l} f(x) = \hat{p}(x, rf(u_1), \ldots, f(u_k)) \\ \text{where } \forall i \in [k] \ u_i := \hat{q}_i(x, r) \end{array} \right] \geq$$

$$\Pr_{r \sim R} \left[\begin{array}{l} f(u_1) = \hat{f}(u_1), \ldots, f(u_k) = \hat{f}(u_k) \\ \text{where } \forall i \in [k] \ u_i := \hat{q}_i(x, r) \end{array} \right] \geq$$

$$1 - k \cdot \Pr_{x \sim X} \left[f(x) \neq \hat{f}(x) \right] \geq 1 - k \cdot \varepsilon.$$

Here, the first inequality is because $(\hat{q}_1, \ldots, \hat{q}_k, \hat{p})$ is a perfect RSR for $\hat{f}$, the second is by a union bound and the fact that each u_i is distributed uniformly in X (Definition 1), and the last is because $\hat{f}$ is ε-close to f.

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

Note that, as mentioned in Claim 6, the running time of the RSR-learner is not bounded by a polynomial in the number of samples m. In more detail, we denote

- $T_{\mathrm{PAC}}(m)$: an upper-bound on the running time of the PAC learner Λ_{PAC} as a function of the number of samples $m = m_{\mathrm{PAC}}(\min(\rho/k, \xi), \delta)$.
- $T_Q(n)$ (resp. $T_P(n)$): an upper-bound on the running time of query functions $q \in Q$ (resp. recovery function $p \in P$) as a function of the input length $n = |x|$.

Then the running time of the RSR-learner is

$$O\left(T_{\mathrm{PAC}}(m) + |Q_n|^k \cdot |P_n| \cdot |X_n| \cdot |R_n| \cdot (k \cdot T_Q(n) + T_P(n)) \right).$$

In particular, $|X_n|$ typically grows exponentially in n. Therefore, even if the PAC learner were efficient, the RSR-learner will not be efficient.

PROOF SKETCH OF CLAIM 7. We will show a setting in which an RSR is "learnable" without any samples ($m \equiv 0$). However, without samples it will not be possible to PAC learn a hypothesis for f.

Consider the RSR that captures the so-called BLR relation $g(z) = g(z + \tilde{x}) - g(\tilde{x})$ for Boolean functions $g \colon \mathbb{F}_2^\ell \to \mathbb{F}_2$ [Blum et al. 1993]. Indeed, this relation characterizes ℓ-variate linear functions over $\mathbb{F}_2$. In a nutshell, we will choose our reduction class (Q, P) to consist only of the reduction specified by the BLR relation, and the hypothesis class F to consist of all linear functions. Then, a

(Q, P)-RSR$_2$ learner for F will always output the BLR relation, but on the other hand, PAC learning F requires a superconstant number of samples. Details follow.

We choose the input and randomness domains to be $X_n := R_n := \mathbb{F}_2^n$, and the range $Y_n = \mathbb{F}_2$. The query class Q_n consists of just two query functions

$$Q_n = \{q_n, q'_n\} \quad \text{where } q_n, q'_n \colon \mathbb{F}_2^n \times \mathbb{F}_2^n \to \mathbb{F}_2^n,$$
$$q_n(x, r) := x + r,$$
$$q'_n(x, r) := r.$$

The recovery class P_n is the singleton $P_n = \{p_n\}$ where $p_n(x, r, y, y') := y - y'$. Indeed, the trivial algorithm that takes no samples and outputs (q, q', p) is a (Q, P)-RSR$_2$ learner for any linear function $f \colon \mathbb{F}_2^n \to \mathbb{F}_2$.

On the other hand, fix $\varepsilon < 1/2$, $\delta < 1/2$ and number of samples $m < n$. Given m labeled samples $(x_i, f(x_i))_{i=1}^m$, there exists another linear function $f' \neq f$ such that $f'(z_i) = f(z_i)$ for all $i \in [m]$. The learner cannot do better than guess between f and f' uniformly at random , in which case it outputs f' with probability $1/2$. Lastly, we note that any two different linear functions agree on *exactly* $1/2$ of the inputs in $\mathbb{F}_2^n$, therefore

$$\Pr_{x \sim X}[f(x) = f'(x)] \geq 1 - \varepsilon \iff f = f'.$$

All in all, we have

$$\Pr_{\substack{x_1, \dots, x_m \sim X \\ \hat{f} \leftarrow \Lambda(x_1, f(x_1), \dots, x_m, f(x_m))}} \left[\Pr_{x \sim X}[\hat{f}(x) = f(x)] \geq 1 - \varepsilon \right] = \Pr_{\substack{x_1, \dots, x_m \sim X \\ \hat{f} \leftarrow \Lambda(x_1, f(x_1), \dots, x_m, f(x_m))}} \left[\hat{f} = f \right] \leq 1/2 < 1 - \delta.$$

$\square$

B   GENERAL DEFINITIONS

Definition 9 (Randomized reduction). *Let*

$$f \colon X \to Y \qquad\qquad\qquad \textit{(Source function)}$$
$$g_1, \dots, g_k \colon U \to V \qquad\qquad \textit{(Target functions)}$$
$$q_1, \dots, q_k \colon X \times R \to U \qquad \textit{(Query functions)}$$
$$p \colon X \times R \times V^k \to Y \qquad\quad \textit{(Recovery function)}$$

such that U and V are uniformly-samplable, and for all $i \in [k]$ and $x \in X$, $u_i := q_i(x, r)$ is distributed uniformly over U when $r \sim R$ is sampled uniformly at random.

We say that $(q_1, \dots, q_k, p)$ is a perfect randomized reduction (RR) from f to $(g_1, \dots, g_k)$ with k queries and $\log_2 |R|$ random bits if for all $x \in X$ and $r \in R$, letting $u_i := q_i(x, r)$ for all $i \in [k]$, the following holds:

$$f(z) = p(z, r, g_1(u_1), \dots, g_k(u_k)). \tag{3}$$

In other words, if Equation (3) holds with probability 1 over randomly sampled $r \in R$.

For errors $\rho, \xi \in (0, 1)$, we say that $(q_1, \dots, q_k, r)$ is a (ρ, ξ)-approximate randomized reduction $((\rho, \xi)$-RR) from f to $(g_1, \dots, g_k)$ if, for all but a ξ-fraction of $x \in X$, Equation (3) holds with probability $\geq 1 - \rho$ over the random samples $r \sim R$. That is,

$$\Pr_{x \sim X} \left[\Pr_{r \sim R} \left[\begin{array}{l} f(x) = p(x, r, g_1(u_1), \dots, g_k(u_k)) \\ \text{where } \forall i \in [k]\ u_i := q_i(x, r) \end{array} \right] \geq 1 - \rho \right] \geq 1 - \xi.$$

Definition 1 is derived from Definition 9 by making the following restrictions:

- Letting $X = U$, $Y = V$, and $f = g_1 = \cdots = g_k$. This is called a *randomized self-reduction (RSR)* for f.
- Considering a *randomness-oblivious recovery function* which take as input the queries $u_1, \ldots, u_k$ instead of the randomness r used to generate these queries. That is, letting $p \colon X \times (X \times Y)^k \to Y$.[8]
- Letting the randomness domain R consist of n uniformly random samples from X, i.e., $R = X^n$.

Table 5. RSR-Bench: Specifications and Learned Randomized (Self)-Reductions

#	Specification	RSR Properties
1	$f(x) = cx + a$	$f(x+r) = f(x) + f(r)$
2	$h(x) = e^{cx}$	$h(x+r) = h(x)h(r)$ and $h(x+r)h(x-r) = h(x)^2$
3	$f(x) = e^{cx} - 1$	$f(x+r) = f(x) + f(r) + f(x)f(r)$
4	$f(x) = e^x/x$	*RSR not found*
5	$f(x) = e^x/x$	$f(x+r) = h(x)h(r)/g(x,r)$
6	$g(x, y) = x + y$	$g(x+r, y+s) = g(x, y) + g(r, s)$
7	$f(x, yz) = (x+y+z)/3$	$f(x_1 + x_2 + x_3, y_1 + y_2 + y_3, z_1 + z_2 + z_3) - f(x_1, y_1, z_1) - f(x_2, y_2, z_2) - f(x_3, y_3, z_3)$
8	$f(x) = \tan cx$	$f(x+r) = (f(x) + f(r))/(1 - f(x)f(r))$
9	$f(x) = \cot cx$	$f(x+r) = (f(x)f(r) - 1)/(f(x) + f(r))$
10	$f(x, y) = x^2 - y^2$	$f(x-r, y-s) = f(r+x, y-s) - 2f(r+x, y) - f(x, y-r) - f(x, r+y) + 4f(x, y)$
11	$f(x) = 1/x^2$	$f(x+r) = (2f(x)f(r) - f(x)f(x-r) - f(r)f(x-r))/(f(x) + f(r) - 8f(x-r))$
12	$f(x) = c/x$	$f(x)f(x+r) - f(x)f(r) + f(x+r)f(r) = 0$
13	$f(x) = 1/(x+1)$	$f(x+r) = f(x)f(x-r)/(-f(x) + 2f(x-r))$
14	$f(x) = 1/(1 + \cot cx)$	$f(x+r) = (f(x) + f(r) - 2f(x)f(r))/(1 - 2f(x)f(r))$
15	$f(x) = 1/(1 + \tan cx)$	$f(x+r) = (f(x) + f(r) - 1)/(2f(x) + 2f(r) - 2f(x)f(r) - 1)$
16	$f(x) = cx/(1 - cx)$	$f(x+r) = (f(x) + f(r) + 2f(x)f(r))/(1 - f(x)f(r))$
17	$f(x) = -cx/(1 - cx)$	$f(x+r) = (f(x) + f(r) - 2f(x)f(r))/(1 - f(x)f(r))$
18	$f(x) = \cos(cx)$	$f(x+r) + f(x-r) = 2f(x)f(r); \ f(x+r) = f(x)f(r) - \sqrt{1 - f(x)^2}\sqrt{1 - f(r)^2}$
19	$f(x) = \cosh(cx)$	$f(x+r) + f(x-r) = 2f(x)f(r); \ f(x+r) = f(x)f(r) + \sqrt{f(x)^2 - 1}\sqrt{f(r)^2 - 1}$
20	$f(x) = cx^2$	$f(x+r) + f(x-r) = 2f(x) + 2f(r)$
21	$\alpha(x) = k \sin cx$	$\alpha(x+r) = (\alpha(x)^2 - \alpha(r)^2)/\alpha(x-r)$
22	$f(x) = k \sinh cx$	$f(x+r) = (f(x)^2 - f(r)^2)/f(x-r)$
23	$f(x) = x^3$	$f(x+r) = (4f(x)^2 - f(x-r)(f(x) + f(r)) - 4f(r)^2)/(f(x) - f(r) + 2f(x-r))$
24	$f(x) = \log x$	$f(xr) = f(x) + f(r)$
25	$f(x) = \sec x$	$f(x)f(x+r)f(r) + f(x)f(x-r)f(r) - 2f(x+r)f(x-r) = 0$
26	$f(x) = \csc x$	$f(x)^2 f(x+r)f(x-r) + f(x)^2 f(r)^2 - f(x+r)f(-r)f(r)^2 = 0$
27	$f(x) = \sin x/x$	$f(x+r) = (\alpha(x)^2 - \alpha(r)^2)/(g(x,r)\alpha(x-r))$
28	$f(x) = \sin x/x$	$f(x+r) = \alpha(x,r)/g(x,r)$
29	$f(x, R) = x \mod R$	$f(x+r) = f(x) +_R f(r)$
30	$f(x, y, R) = (x \cdot_R y)$	$f(x+r, y+s) = f(r,s) +_R f(r, y) +_R f(x, s) +_R f(x, y)$
31	$f(x, y) = x \cdot y$	$f(x+r, y+s) = f(r, s) + f(r, y) + f(x, s) + f(x, y)$
32	$f(x) = C \tanh cx$	$f(x+r) = f(x) + f(r)/(1 + (f(x)f(r)/C^2))$
33	$f(x) = 1/(1 + e^{-x})$	$f(x-r) = f(x)(f(r) - 1)/(2f(x)f(r) - f(x) - f(r))$
34	$f(x, y) = e^x/(e^x + e^y)$	$f(x, y) = f(r+x, r+y)$
35	$f(x, y) = e^y/(e^x + e^y)$	$f(x, z) = f(x + y, y + z)$
36	$f(x) = 1/(1 + e^{-2x})$	$f(x+r) = (-f(x)f(x-r) + f(x) - f(r)f(x-r))/(f(x) - f(r) - 2f(x-r) + 1)$
37	$f(x) = 3/(1 + e^{-2x})$	$f(x+r) = (-f(x)f(x-r) + 3f(x) - f(r)f(x-r))/(f(x) - f(r) - 2f(x-r) + 3)$
38	$f(x) = (1 - x)^2$	$f(x+r) = \pm 4\sqrt{f(x)f(x-r)} + 4f(x) + f(x-r)$
39	$f(x) = 1/(1 + e^x)^2$	$f(x+r) = 1/(h(x,r)^2 + 2h(x,r) + 1)$
40	$f(x) = 1/(1 + e^x)^2$	$f(x+r) = 1/(h(x)^2 h(r)^2 + 2h(x)h(r) + 1)$

---

[8] *Tedious comment:* For simplicity of notation, we also rearrange the inputs of p from $X \times X^k \times Y^k$ to $X \times (X \times Y)^k$.

$F[x, y] = F[y, x]$	(Commutativity)	$F[F(x, u), v] = F(x, u + v)$		(Translation)
$F[k \cdot x, k \cdot y] = k \cdot F[x, y]$	(Multiplicative)	$F[F(x, y), z] = F[x, F(y, z)]$		(Associativity)
$F[x, x] = x$	(Idempotency)	$F[F(x, z), F(y, z)] = F(x, y)$		(Transitivy)
$F[x] = F[-x]$	(Even)	$F[F(x, y), F(u, v)] = F[F(x, u), F(y, v)]$		(Bisymmetry)
$F[x] = -F[-x]$	(Odd)	$F[x, F(y, z)] = F[F(x, y), F(x, z)]$		(Autodistributive I)
$F[x + nP] = F[x]$	(Periodic)	$F[F(x, y), z] = F[F(x, z), F(y, z)]$		(Autodistributive II)
$F[x + y] = F[x] + F[y]$	(Additive)	$F[F[x]] = x$		(Involution)
$F[x + y] = F[x]F[y]$	(Exponential)	$F[F[x]] = G[x]$		(Functional Square Root)
$F[xy] = F[x] + F[y]$	(Logarithmic)	$F[x + y] = G[x] + H[y]$		(Pexider)
$F[xy] = F[x]F[y]$	(Multiplicative)	$F[x + y] = G[x]H[y]$		(Pexider)
$F[x, z] = F[x, y] + F[y, z]$	(Cyclic)	$F[xy] = G[x] + H[y]$		(Pexider)
		$F[xy] = G[x]H[y]$		(Pexider)

Table 6. Some Functional Equations   Table 7. Some Composite Functional Equations

C EXAMPLES FOR LEARNING PROGRAM INVARIANTS IN C

Bitween combines machine learning with formal methods to automatically infer randomized reductions and program invariants. Through random fuzzing and static analysis, it identifies assertions, deriving loop invariants and post-conditions for C programs. Users instrument their code by adding functions prefixed with vtrace at key locations, such as after entering loops, to monitor terms likely to appear in invariants.

For instance, in Figure 6, Bitween aims for learning a loop invariant, by using vtrace1 at line 8 as $F[X, Y, x, y, v] = 0$, where F is an unknown algebraic function. Bitween aims to interpret F in the hypothesis class of polynomials constructed by the terms passed to F and bounded up to a given degree. Additionally, users can include assume statements to refine the analysis, focusing on specific cases of interest. To infer post-conditions, users can place vtrace calls before function exits or within function calls, ensuring the invariants are inferred at key points.

For more sophisticated input generation, users can define intervals using the vdistr function to guide the fuzzing process with uniform distributions as shown in Figure 7a at line 2 and 3.

Although Bitween currently supports only a subset of C, ongoing development will extend it to handle arrays, pointers, and structs.

In the example programs in Figure 6a and Figure 7a, Bitween starts by using 20 random inputs to find invariants about the variables. Here's an overview of two specific examples:

(1) **Bresenham's Line Drawing Algorithm**: This algorithm is used to draw lines on a grid, selecting the best-fitting pixels between two points. In the first version (Figure 6a), the program is instrumented with vtrace calls to track variables. Bitween processes these traces and infers invariants about the relationships between variables involved in the line drawing process. The output (Figure 6b) replaces the vtrace calls with assertions that represent the inferred invariants, ensuring that the program correctly draws lines under different conditions.

(2) **Square Root Computation by Zuse**: This example computes square roots using an iterative method. Although the calculation is approximate, based on a fixed error ('err = 0.0001'), Bitween mathematically finds the correct relationships between the variables. The program is first instrumented with vtrace calls to track key variables (Figure 7a). Using 20 random inputs, Bitween analyzes the program and infers the necessary relationships (invariants) between these variables during computation. The output (Figure 7b) contains the inferred assertions in

```
1  int bresenham(int X, int Y) {
2    vdistr(X, 0, 1000);
3    vdistr(Y, 0, 1000);
4    assume(X >= 0);
5
6    int v = 2 * Y - X, x = 0, y = 0;
7    while (1) {
8      vtrace1(X, Y, x, y, v);
9
10     if (!(x <= X)) break;
11     if (v < 0) {
12       v = v + 2 * Y;
13     } else {
14       v = v + 2 * (Y - X); y++;
15     }
16     x++;
17   }
18
19   vtrace2(X, Y, x, y, v);
20   return v;
21 }
```

```
int bresenham(int X, int Y) {
  vdistr(X, 0, 1000);
  vdistr(Y, 0, 1000);
  assume(X >= 0);

  int v = 2 * Y - X, x = 0, y = 0;
  while (1) {
    assert(2*X*y + X - 2*Y*x - 2*Y + v == 0);

    if (!(x <= X)) break;
    if (v < 0) {
      v = v + 2 * Y;
    } else {
      v = v + 2 * (Y - X); y++;
    }
    x++;
  }
  assert(X - x + 1 == 0);
  assert(2*Y*x +2*Y -v -2*x*y -x +2*y +1==0);
  return v;
}
```

(a) Input program instrumented with vtrace1 and vtrace2.

(b) Output program with vtrace locations replaced by inferred invariants.

Fig. 6. Input and output for Bresenham's line drawing algorithm, based on Srivastava et al.'s *From Program Verification to Program Synthesis*, POPL '10.

```
1  float z3sqrt(float a) {
2    assume(a >= 1);
3
4    float x = a;
5    float q = 0.5 * (x + a / x);
6    float err = 0.0001;
7
8    while(x - q > err or q - x > err){
9      vtrace1(a, x, err, q);
10
11     x = q;
12     q = 0.5 * (x + a / x);
13   }
14
15   return q;
16 }
```

```
float z3sqrt(float a) {
  assume(a >= 1);

  float x = a;
  float q = 0.5 * (x + a / x);
  float err = 0.0001;

  while(x - q > err or q - x > err){
    assert(a - 2*q*x + pow(x, 2) == 0);
    assert(err == 0);
    x = q;
    q = 0.5 * (x + a / x);
  }

  return q;
}
```

(a) Instrumented input program for computing square roots, by Zuse.

(b) Output program with vtrace replaced by inferred invariants.

Fig. 7. Input and output programs for computing square roots, based on Zuse's implementation https://www.cs.upc.edu/~erodri/webpage/polynomial_invariants/z3sqrt.htm.

place of the vtrace calls, ensuring the correctness of the square root calculation, despite the approximate nature of the iterative method.