Discrete Choice in the Presence of Numerical Uncertainties

Debasmita Lohar, *Student Member, IEEE,* Eva Darulova, Sylvie Putot, and Eric Goubault

***Abstract*—Numerical uncertainties from noisy inputs or finite-precision roundoff errors are unavoidable on resource-constrained systems. While techniques exist to compute worst-case bounds on these errors for arithmetic operations, these approaches do not generalize to programs which take *discrete* decisions. In this case, the more interesting quantity is the *probability* of the program making the wrong decision. In this paper, we study two approaches to compute a guaranteed bound on this probability: exact probabilistic inference and probabilistic static analysis. By themselves, they provide accuracy and scalability, respectively, but unfortunately not at the same time. We propose an extension to the latter approach which allows us to bound the probability tightly and fully automatically while scaling to small but interesting embedded examples.**

***Index Terms*—uncertainty, floating-point, fixed-point, probability, static analysis.**

I. Introduction

For many computations in embedded resource-constrained systems numerical uncertainties are unavoidable. They are usually a combination of input uncertainties, e.g. from noisy sensors, and finite-precision roundoff errors committed during the computation. Especially for safety-critical applications it is important to show that the uncertainties remain small and the results accurate enough. Furthermore, such programs are often first developed and validated in a higher precision, before being deployed in low-precision on the actual hardware, raising additional questions about the correctness of the computed results. Accurately propagating and bounding input and finite-precision roundoff errors is challenging to achieve manually and thus automated techniques and tools are imperative.

Today's tools compute guaranteed upper bounds for floating-point and fixed-point arithmetic roundoff errors in multivariate straight-line programs [1], [2], [3], and some techniques even offer some support for conditional branches and loops [3], [4], [5]. Unfortunately, none of the present techniques considers the case where the result of the numerical computation (`res`) is the basis for a discrete decision, as in `res = ...; if(res < const) true else false`.

Such programs are important whenever a system must decide, e.g., whether to raise an alarm or not, which of several discrete control signals to sent, etc. Furthermore, machine learning classifiers are increasingly deployed in resource-constrained systems where they may decide whether a heartbeat signal is normal or not [6].

Debasmita Lohar and Eva Darulova are with MPI-SWS.

Sylvie Putot and Eric Goubault are with LIX, CNRS and École Polytechnique.

This article was presented in the International Conference on Embedded Software 2018 and appears as part of the ESWEEK-TCAD special issue.

Previous work on bounding numerical errors has focused on worst-case analysis, i.e. at each step of a computation the analysis computes the largest possible absolute error that can appear for a given input domain. Unfortunately, this analysis cannot be simply extended to a program which makes discrete decisions. For instance, if the program outputs either '0' or '1' (representing true and false), then the worst-case error is exactly 1. In general, some uncertainty is unavoidable and thus the computation will necessarily output a different result than the ideal reference, at least for some inputs.

Such a trivial result is clearly not very useful. Ideally, a program with uncertainties should compute the correct answer, resp. decision, for most inputs, i.e. the uncertainties should be small enough that the answers are mostly correct. A more useful measure of correctness is thus the *probability* that the discrete result will be correct. For this, we clearly have to take into account the *distributions* of inputs and propagate them through the computation.

In this paper, we consider two approaches to propagating input distributions through numerical computations: exact symbolic probabilistic inference [7] and sound probabilistic static analysis [8]. We show that while both approaches seemingly solve the problem, the former produces good results, but only for tiny programs and the latter, while it scales significantly better, overapproximates the probability of wrong results too much to be useful in practice.

We extend the static analysis approach with interval subdivision and reachability checking to soundly determine which part of the input domain is actually relevant for computing the probability of wrong results. The probabilistic analysis is then only run where needed and due to a smaller input domain suffers from less overapproximation. Overall, our analysis produces *tight* probability bounds, while being *sound* and scalable enough for small, yet realistic embedded programs. This analysis is *fully automated* and works on general numerical programs with arithmetic and elementary operations, and can consider different probability distributions on the inputs, including uncertain ones.

We have implemented this analysis in a prototype tool, which we will release as open source. We integrated it with an existing tool for bounding numerical errors due to input uncertainties and roundoff errors in floating-point as well as fixed-point arithmetic [9] and thus obtain a fully automated end-to-end solution. We evaluate it on several embedded controllers as well as machine learning classifiers and obtain encouraging results.

a) Contributions: To summarize, in this paper:

- we provide a precise problem definition and present, to the best of our knowledge, the first sound analysis of the effects of numerical uncertainties on discrete decisions,

- we show that while symbolic probabilistic reasoning provides exact results, it does not scale well,
- we extend an existing probabilistic static analysis with interval subdivision and reachability checks,
- we evaluate our analysis and the various design decisions that we make on a set of representative embedded examples
- we implement our analysis in a prototype tool and will release it as open source.

II. Problem Definition and Overview

Before diving into the details, we first specify the problem we are considering more precisely, introduce terminology that we will use in this paper, and provide an overview of our approach.

a) Input Programs: In this work, we focus on the core issue and consider programs which perform a finite-precision numerical computation and then make a discrete decision based on the computed result. For example, based on the control output calculated by a nonlinear controller [10], an alarm may be raised:

```
res = -x1*x2 - 2*x2*x3 - x1 - x3
if(res <= 0.0) raise_alarm()
```

Here, `x1, x2, x3` are the controller inputs and may carry some initial uncertainties. Abstracting away the effects of the decision, the programs that we consider are defined as a function $f : \mathbb{R}^n \rightarrow \mathbb{B}$ from a possibly multivariate real-valued input to a boolean output. This function specifies the ideal baseline computation.[1] The function executing on actual hardware is implemented in finite-precision: $\tilde{f} : \mathbb{F}^n \rightarrow \mathbb{B}$. Due to input uncertainties and roundoff errors, f and $\tilde{f}$ may return different results for the same ideal inputs (note that the inputs have to be rounded as well).

The goal of this paper is to automatically compute a *guaranteed bound* on the probability that f and $\tilde{f}$ return different results. We will call this probability the *wrong path probability* (WPP). In general, we cannot expect this probability to be zero, but we want to be able to show that it is small enough (for a particular application).

While we presently only consider programs with the relatively simple structure above, our approach straight-forwardly generalizes to nested conditionals and to other discrete return types by considering each path and each return value separately. The numerical computation in our input is a straight-line expression, consisting of arithmetic operations $(+, -, *, /, \sqrt{})$ as well as transcendental functions $(\sin, \cos, \exp, \log)$. Conditionals or loops in the numerical portion of the program are out of scope of this paper; for these constructs, even sound roundoff error computation is challenging and only limited techniques exist so far [5], [4].

[1]In many applications, the baseline may be a higher-precision computation, for which we use the real-valued one as a proxy.

```
x1 := uniform(-15.0, 15.0);
x2 := uniform(-15.0, 15.0);
x3 := uniform(-15.0, 15.0);

res := -x1*x2 - 2*x2*x3 - x1 - x3;
error := 0.2042266; // computed externally
assert(0.0 - error <= res && res <= 0.0 + error);
```

Fig. 1. Probabilistic program encoding the wrong path probability for the example of a nonlinear controller

b) Bounding Numerical Uncertainties: A numerical program may take the wrong path due to uncertainties in the inputs as well as accumulated roundoff errors in the numerical computation. Thus, the first step in computing an upper bound on the wrong path probability for some f is to compute a sound upper bound on the numerical errors, for which we use established techniques [9].

c) Critical Intervals: For each input program, we have one *threshold* value t, which determines the decision boundary, i.e. the boundary around which the ideal and uncertain computations may diverge. In our running example, this is `0.0`.

The numerical error (e) we computed in the previous step tells us how far away from this threshold we can possibly observe a divergence. This defines the *critical interval*: $[t-e, t+e]$. If the result of the ideal numerical computation falls into this interval, then the uncertain, finite-precision computation may compute a different discrete result. The probability that the ideal computation lies in $[t-e, t+e]$ is thus a sound bound on the wrong path probability.

d) Propagating Probabilities: Given probability distributions on the inputs, we want to compute a sound bound on the probability that the result of the numerical computation falls within the critical interval. We consider only the forward approach here, i.e. we propagate the input distributions through the numerical computation.

We can express the wrong path probability precisely as a probabilistic program as in Figure 1, where we use the syntax of the PSI solver [7]. In this particular example, we consider as the numerical uncertainty the roundoff error of a 16-bit fixed-point arithmetic implementation and we choose uniform input distributions.

We consider two quite different approaches for propagating probability distributions. First, we use exact symbolic probabilistic inference implemented in PSI. We provide more details in section III, but in short our experiments show that while PSI can indeed be applied to our problem, it unfortunately does not scale to larger programs. For instance, for our running example, we do not obtain a result within 20 minutes. An alternative to using exact reasoning is abstraction. For this, we have reimplemented a static analysis which propagates uncertain distributions [11]. It computes a sound over- and underapproximation of the probability distribution, and thus scales much better. Unfortunately, for many of our example programs, the overapproximation is too large. For instance for our running example, we compute the wrong path probability in 0.63s but the probability computed is 1.

e) Our Targeted Analysis: The reason why the above approaches do not work well is that they compute general probability distributions, but in our case we are interested in the probability of the result being inside the fairly small critical interval. A large portion of the input space never comes close to this critical interval, but which inputs *are* relevant is not immediately obvious. Ideally we could perform a backwards reachability analysis, starting from the critical interval to obtain only the relevant input domain. Unfortunately, for multivariate programs, computing such a sound input domain is nontrivial.

We combine probabilistic static analysis with interval subdivision and reachability checks to narrow down the input domain in a forward manner. Subdivision is a standard technique to reduce overapproximations in static analysis. While it can improve the probability bounds by itself, we further use a nonlinear decision procedure to completely remove parts of the subdomain which definitely cannot reach the critical interval. For our running example, our method computes a wrong path probability of 0.07060 in 155s.

In summary, we obtain a technique which can compute fairly tight probability bounds (judging from the few examples where PSI computes a result), and which is nonetheless scalable enough to handle programs which appear in embedded systems in practice and for which guaranteed results are of importance.

III. Using Symbolic Inference

We can express the wrong path probability as a probabilistic program as shown in Figure 1, which we can directly evaluate using a probabilistic programming inference tool.

Several approaches for probabilistic inference exist. As exact probabilistic inference is a hard problem [12], most algorithms compute numerical approximations using sampling [13], [14], [15]. Such methods, however, do not provide accuracy guarantees. The recently developed PSI (Probabilistic Symbolic Inference) solver [7] performs *symbolic* inference and generates a compact representation of the *exact* final distribution. This distribution is in general still quite complex and large, but it can be numerically evaluated using Mathematica [16], for which we can specify an output precision.

A. Experimental Setup

We empirically evaluate to which extent exact probabilistic inference is suitable for computing wrong path probabilities. For these experiments we consider roundoff errors as the only uncertainty, i.e. we assume that the inputs are otherwise exact. For this, we combine three off-the-shelf tools.

Roundoff error analysis with Daisy In the first step, we use the open source tool Daisy to compute a sound absolute roundoff error bound on the numerical part of each benchmark. Daisy performs forward dataflow analysis and supports both floating-point and fixed-point arithmetic and can also be used to propagate input errors [9].

We compute roundoff errors for 16 bit fixed-point arithmetic as well as 32 bit (single precision) floating-point arithmetic and record the roundoff error as a constant in the probabilistic program. The computation of roundoff errors for all benchmarks for one precision took under 30 seconds.

Symbolic probabilistic inference with PSI In the second step, we run PSI on the probabilistic programs, each of the same form as in Figure 1. We choose all inputs to be either uniformly or normally distributed on the same support for which we have computed roundoff errors. All inputs are independent. PSI computes the output and the probability distribution of the assertion failing; we record the latter for our experiment. We set a timeout of 10 minutes (we did not observe different results with longer timeouts).

Numerical evaluation with Mathematica PSI returns the exact simplified expression of the wrong path probability in the input format of Mathematica. This expression can then be numerically evaluated using the `N[...]` command. We set the output precision to be 5 decimal digits (Mathematica adjusts the internal precision automatically), and a timeout of 10 minutes.

We used PSI downloaded on 22 March 2018, Mathematica version 10.4.0.0 and Daisy's version from 6 December 2017. We ran all experiments on a Debian desktop with 3.3 GHz and 32 GB RAM.

a) Benchmarks: As no standard benchmark set exists, we retrofit a number of existing benchmarks from the area of finite-precision roundoff error estimation [1], [5]. Each of these benchmarks comes with lower and upper bounds on input variables. These benchmarks cover various scenarios: sine, sineOrder3 and sqroot are polynomial approximations of functions which are often costly to compute precisely, B-Splines are used in embedded image applications [17], traincar and rigidBody are linear and nonlinear controller [10], respectively, and doppler and turbine are physics expressions. Table II shows the number of arithmetic operations and variables of each benchmark.

We consider all input variables to be either all uniformly or all normally distributed within the input ranges. For variable x which is uniformly distributed on $[a, b]$, we declare it as `x := uniform(a, b)`. For a normally distributed variable, we choose one standard deviation from the mean and declare it as

```
tmp := gauss(a, b);
observe(-1.0 <= tmp && tmp <= 1.0);
x := mid + radius * tmp;
```

with mid $= (a + b)/2$ and radius $= |(a - b)/2|$.

For each benchmark, we select two thresholds: one where we expect the wrong path probability to be low and one where we expect it to be high. We generate these thresholds by simulation. From a plot of the recorded results we choose one threshold from the more and one from the less likely result region.

B. Results

Table I shows the wrong path probabilities for benchmarks where both PSI and Mathematica finished within the time limit. For doppler, rigidBody*, turbine* and traincar*, either PSI or Mathematica timed out, so that we do not show them in the table.

Table II shows the real time (measured by the bash time command) taken by PSI to compute the exact probability distribution. We also measured the time taken by Mathematica (using the AbsoluteTiming command), however we observed that either Mathematica reported a time below 1ms, or it timed

benchmark	precision	16-bit fixed-point				32-bit floating-point			
	input distribution	normal		uniform		normal		uniform	
	threshold	high	low	high	low	high	low	high	low
sine		8.12e-4	4.55e-4	9.55e-4	2.98e-4	6.47e-7	3.63e-7	7.61e-7	2.38e-7
sineOrder3		2.14e-3	5.24e-4	2.47e-3	4.49e-4	1.64e-6	4.03e-7	1.90e-6	3.44e-7
sqroot		3.07e-2	9.59e-4	2.79e-2	1.10e-3	9.62e-6	3.03e-7	8.74e-6	3.49e-7
bspline0		1.73e-2	6.05e-4	1.82e-2	7.23e-4	1.00e-5	3.53e-7	1.05e-5	4.21e-7
bspline1		3.26e-3	1.48e-3	3.46e-3	1.59e-3	2.39e-6	1.08e-6	2.54e-6	1.16e-6
bspline2		2.78e-3	1.26e-3	2.95e-3	1.32e-3	2.09e-6	9.46e-7	2.22e-6	9.95e-7
bspline3		2.78e-3	3.58e-4	2.30e-3	4.27e-4	1.70e-6	2.32e-7	1.49e-6	2.77e-7

TABLE I
WRONG PATH PROBABILITY COMPUTED BY PSI (NOT SHOWN BENCHMARKS DID NOT COMPLETE WITHIN 20MIN)

benchmark	#ops-#vars	normal distrib.	uniform distrib.
doppler	8 - 3	*	*
sine	18 - 1	1s 301ms	688ms
sineOrder3	5 - 1	437ms	523ms
sqroot	14 - 1	565ms	767ms
bspline0	6 - 1	729ms	354ms
bspline1	8 - 1	690ms	303ms
bspline2	10 - 1	729ms	356ms
bspline3	4 - 1	716ms	994ms
rigidBody1	7 - 3	2m 1s 510ms	-
rigidBody2	14 - 3	-	-
turbine1	14 - 3	-	-
turbine2	10 - 3	-	1m 46s 348ms
turbine3	14 - 3	-	-
traincar1	6 - 3	15s 553ms	15s 330ms
traincar2	10 - 5	5m 53s 4ms	5m 52s 947ms
traincar3	14 - 7	5m 55s 616ms	5m 53s 52ms
traincar4	18 - 9	-	-

TABLE II
AVERAGE ANALYSIS TIME OF PSI (- : TIMEOUT, * : SEGFAULT)

out after 10 minutes. For this reason, we only report PSI's running times. For the benchmarks where PSI successfully computes a results (as reported in Table II) and which do not appear in Table I, Mathematica timed out.

Where PSI and Mathematica are able to compute wrong path probabilities, they indeed compute a smaller probability for lower thresholds than for higher ones. Similarly, for smaller uncertainties, i.e. for 32-bit floating-point roundoff errors, the probabilities are also significantly smaller, as expected. While the values for uniform and normal input distributions are not directly comparable, we have included them as different distributions pose different difficulties for inference. This can be seen in Table II where PSI can compute a result for rigidBody1 for normal and for turbine2 for uniform input distributions only.

IV. PROBABILISTIC STATIC ANALYSIS

Clearly the results obtained by exact symbolic probabilistic inference are unsatisfactory. We now turn to our second approach—static analysis. Before we explain our extension (section V), we review necessary background on probabilistic static analysis.

A. Uncertain Probabilities

Interval arithmetic (IA) [18] is an efficient choice for range estimation, which computes a bounding interval for each basic operation as $x \circ y = [\min(x \circ y), \max(x \circ y)], \circ \in \{+, -, *, /\}$ and analogously for square root. Interval arithmetic cannot track correlations between variables (e.g. $x - x \neq 0$), and thus can introduce significant over-approximations of the true ranges. Furthermore, interval arithmetic only captures nondeterminism, i.e. $x \in [a, b]$ expresses that variable x can take any value between a and b, but it does not provide any more information about its distribution.

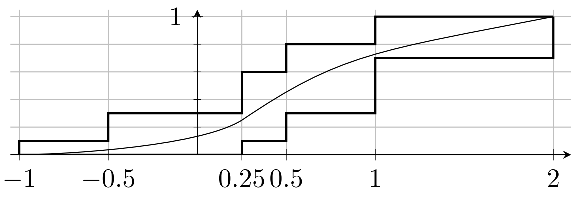
Interval Dempster-Shafer (DSI) structures [19] associate weights with intervals and thus allow to capture uncertain probabilities. We provide the basics needed to follow the remaining part of the paper here, for more details see [19], [20].

Formally, a DSI structure consists of a finite set of so-called focal elements: $d = \{\langle \mathbf{x_1}, w_1\rangle, \langle \mathbf{x_2}, w_2\rangle, \cdots, \langle \mathbf{x_n}, w_n\rangle\}$ where $\mathbf{x_i} \in I$ is a closed non-empty interval and $w_i \in]0, 1]$ is the associated probability and $\sum_{k=1}^{n} w_k = 1$. The focal elements express that the value of a variable represented by d is within $\mathbf{x_i}$ with probability w_i, but the variable can (non-deterministically) take any value within the interval $\mathbf{x_i}$.

a) Example: The DSI

$$d = \{\langle[-1, 0.25], 0.1\rangle, \langle[-0.5, 0.5], 0.2\rangle, \langle[0.25, 1], 0.3\rangle,$$
$$\langle[0.5, 1], 0.1\rangle, \langle[0.5, 2], 0.1\rangle, \langle[1, 2], 0.2\rangle\}$$

represents the set of probability distributions with support $[-1, 2]$, where the probability of picking a value between -1 and 0.25 is 0.1, the probability of picking a value between -0.5 and 0.5 is 0.2 etc. The figure below shows the set of probability distributions graphically.



b) Arithmetic Operations: Arithmetic operations over DSIs,
$X \odot Y, \odot \in \{+, -, \times, \div\}$, distinguish the cases where X and Y are independent, or dependent with unknown dependency.

With $d_X = \{\langle \mathbf{x_i}, w_i\rangle \mid i \in [1,n]\}$ and $d_Y = \{\langle \mathbf{y_j}, v_j\rangle \mid j \in [1,m]\}$, obtaining the DSI structure for $Z = X \odot Y$ for *independent* X and Y is straightforward: $d_Z = \{\langle \mathbf{z_{i,j}}, r_{i,j}\rangle \mid i \in [1,n], j \in [1,m]\}$ with $\mathbf{z_{i,j}} = \mathbf{x_i} \odot \mathbf{y_j}$ and $r_{i,j} = w_i \times v_j$.

The *dependent* case is more involved as we have to consider *any* dependency between X and Y to compute a sound over- and under- approximation of the probability. We use the method of [21], which first computes the solution in an alternative representation of DSI's, so-called *discrete p-boxes* [20], and then transforms the P-box back to a DSI. A discrete P-box $[\underline{P}, \overline{P}]$ is a pair of two non-decreasing step-functions functions $\underline{P}$ and $\overline{P}$ such that $\underline{P}$ is left-continuous, $\overline{P}$ is right-continuous and $\forall x.\ \underline{P}(x) \leq \overline{P}(x)$. A P-box encloses all probability distributions whose cumulative distribution function (CDF) P satisfy $\forall x. \underline{P}(x) \leq P(x) \leq \overline{P}(x)$.

To compute the solution $Z = X \odot Y$ as a P-box $[\underline{F_Z}, \overline{F_Z}]$, we first compute the arithmetic operation on the intervals of all pairs of focal elements (as in the independent case), obtaining a matrix of intervals: $\boldsymbol{x_i} \odot \boldsymbol{y_j} = [\underline{z_{i,j}}, \overline{z_{i,j}}]$. However, it no longer holds that $r_{i,j} = w_i \times v_j$. Rather, we only know the following constraints, corresponding to the rows and columns of this matrix:

$$\forall i \in [1,n],\ \sum_{j=1}^{m} r_{i,j} = w_i \qquad \forall j \in [1,m],\ \sum_{i=1}^{n} r_{i,j} = v_j$$

Intuitively, we want to compute an upper and a lower bound on the cumulative distribution function at every point in the domain. Since the P-boxes are step functions, we effectively only need to perform this computation at the end points of the intervals in the interval matrix. The maximization, resp. minimization, of the cumulative distribution function can be phrased as a linear program:

$$\begin{array}{rcll} \underline{F_Z}(z) & = & \textbf{minimize} & \sum_{\overline{z_{i,j}} \leq z} r_{i,j} \\ & & \textbf{such that} & \forall i \in [1,n],\ \sum\limits_{j=1}^{m} r_{i,j} = w_i \\ & & & \forall j \in [1,m],\ \sum\limits_{i=1}^{n} r_{i,j} = v_j \end{array}$$

This computes the lower P-box. The constraint $\sum_{\overline{z_{i,j}} \leq z} r_{i,j}$ expresses that all $r_{i,j}$'s have to be taken into account, whose corresponding intervals are definitely below z. The formula for $\overline{F_Z}$ is analogous.

We can then convert the resulting discrete P-Box $[\underline{F_Z}, \overline{F_Z}]$ to the DSI d_Z. The intervals $\boldsymbol{x_i}$ are obtained by matching the lower and upper P-Box and the weights w_i from the height of the steps.

Dependent arithmetic is clearly costly, and in addition may lose some precision. It is thus important to keep track of dependencies between variables in order to be able to apply independent arithmetic operations as much as possible. We do this by using affine arithmetic, which we describe next.

B. Probabilistic Affine Arithmetic

Affine arithmetic (AA) [22] is an extension of IA which tracks linear correlations between variables. Each quantity is represented as an affine combination of *noise symbols* ε_i: $\hat{x} := x_0 + \sum_{i=1}^{n} x_i \varepsilon_i,\ \varepsilon_i \in [-1,1]$. The same noise symbol can be shared by several affine forms, capturing correlations. An affine form $\hat{x}$ represents a set of values (an interval): $\big[x_0 - \sum_{i=1}^{n} |x_i|,\ x_0 + \sum_{i=1}^{n} |x_i|\big]$.

Linear arithmetic operations are computed term-wise:

$$\alpha\hat{x} + \beta\hat{y} + \zeta = (\alpha x_0 + \beta y_0 + \zeta) + \sum_{i=1}^{n} (\alpha x_i + \beta y_i)\varepsilon_i$$

Nonlinear operations are approximated by linearization and a fresh noise symbol for the remainder term, see e.g. [22] for more details.

Bouissou et.al. [11] introduced probabilistic affine forms which combine affine arithmetic with DSI structures: DSIs compute probability distributions and affine arithmetic tracks the (linear) dependency information between them. Concretely, a probabilistic affine form for a variable x is given by $\hat{x} = x_0 + \sum_{i=1}^{n} x_i \eta_i$ where each noise symbol η_i is equipped with a DSI d_{η_i} with support $[-1,1]$. Thus, noise symbols in standard affine arithmetic track nondeterministic uncertainty, while probabilistic affine forms can capture more detailed probabilistic information.

The presentation in [11] explicitly distinguishes between two kinds of noise terms, independent ones and those with an undefined dependency. In our present implementation, we do not make this distinction. Instead, each noise symbol keeps a set of indices on which it has a potential dependency.

As a standard affine arithmetic form represents an interval, a probabilistic affine form represents a probability distribution which is computed by summing the weighted DSIs associated with each noise symbol: $x_0 + \sum_{i=1}^{n} x_i d_{\eta_i}$. Depending on whether a d_{η_i} has a dependency on the current running sum, the addition operation is dependent or independent.

Linear and unary arithmetic operations over probabilistic affine forms work exactly the same as standard affine form operations. For non-linear operations, like multiplication, we compute the magnitude of the remainder term as in e.g. [11].

V. Implementation and Extension

While the previous section covers the high-level algorithm of probabilistic affine forms, in practice the implementation relies on several additional design decisions. Unfortunately, the previous implementation [11] was not available, nor were many of the details. In this section, we first provide the most important design decisions we took before turning to our extension.

```
1  def reduce(DSI d, threshold T):
2    for $e_i$ : ($x_i$, $w_i$) ← d:
3      if $w_i$ < 1e-5 && overlaps($x_i$, $x_{i-1}$):
4        merge($e_i$, $e_{i-1}$) // merging low weights
5
6    avg = avgWidth(d), sd = stdDevWidth(d)
7    if sd > 1e-3: // variation in widths
8      for $e_i$: ($x_i$, $w_i$) ← d:
9        if width($x_i$) < (avg - sd) && overlaps($x_i$, $x_{i-1}$):
10         merge($e_i$, $e_{i-1}$)
11
12   avg = avgWeight(d), sd = stdDevWeight(d)
13   if sd > 1e-4: // variation in weights
14     for $e_i$ : ($x_i$, $w_i$) ← d:
15       if $w_i$ < (avg - sd) && overlaps($x_i$, $x_{i-1}$):
16         merge($e_i$, $e_{i-1}$)
17
18   if length(d) > T:
19     reduceAlternateOverlapping(d)
20 return d
```

Fig. 2. Reduction algorithm

A. Implementation

In order to use probabilistic AA to propagate input distributions, these have to be first transformed into DSIs. This happens through discretization and our implementation provides convenience methods for uniform and normal distributions. The overall accuracy of the computed distributions depends on the amount of discretization: a finer discretization into more focal elements increases the accuracy of the computed probability distributions, but it also naturally increases the running time.

The results furthermore depend on the algorithm used for reducing the number of focal elements of the DSI's, and the translation of probabilistic AA back into a DSI, which we describe next. We also discuss our choice of LP solver and soundness of internal arithmetic computations in subsubsection V-A3.

1) Reduction: With each arithmetic operation, the number of DSI focal elements grows exponentially and can soon become a performance bottleneck. We thus implement a reduction method which keeps the length of DSIs bounded, but which retains as much information as possible. Figure 2 shows the high-level algorithm. It is based on the idea of repeatedly merging two focal elements $e_i = \langle [a_i, b_i], w_i \rangle$ and $e_j = \langle [a_j, b_j], w_j \rangle$, producing one focal element with a wider interval and larger weight:

$$e_{ij} = \langle [\min(a_i, a_j), \max(b_i, b_j)], w_i + w_j \rangle$$

The decision whether or not two focal elements should be merged depends on the weights and interval widths. Furthermore, our algorithm only merges two focal elements when their intervals overlap in order to reduce loss of accuracy.

Our algorithm first merges focal elements whose weight is less than a threshold (line 2 - 4). When this does not reduce the number of focal elements sufficiently, we merge all focal elements whose widths or weights are less than one standard deviation from the average (lines 6-10 and 12-16). Additionally, this merging is only applied when the standard deviation is sufficiently large. Without this check, should all focal elements have identical width (resp. weight), they would all be merged into a single focal element. We have determined all numerical thresholds empirically. Should all these methods not be sufficient to reduce the length of the DSI, we merge every two overlapping focal elements, irrespective of weights and interval widths. We repeatedly apply the algorithm in Figure 2 until the size of the DSI is sufficiently reduced.

2) Converting AA to DSI: After performing all arithmetic operations, the resulting probabilistic affine form needs to be converted to a DSI, i.e. the affine terms need to be summed up, using the dependency information collected by the affine form. For this conversion, we implement a greedy strategy which maximizes independent arithmetic operations. Our algorithm first partitions the affine terms into sets where each set contains mutually independent terms. These terms are added using independent addition. The resulting mutually dependent partial sums are then added (in any order) using the costlier dependent addition.

3) Floating-point Arithmetic: Our prototype is implemented using rationals (using infinite-precision integers) and thus does not suffer from internal roundoff errors. This is important to ensure soundness, but it naturally increases the running time of our analysis. For operations which cannot be computed in rationals, e.g. square root, we use the MPFR arbitrary-precision arithmetic library [23] and ensure that the result is rounded correctly to ensure sound, i.e. overapproximated bounds.

We use the GLPK [24] solver for the linear programs, which is implemented with floating-point arithmetic internally and thus does not compute guaranteed results. GLPK is generally efficient and precise, but does occasionally generate solutions with extraneous focal elements with weights on the order of double floating-point roundoff error. A fully satisfactory solution would use a guaranteed LP-solver, such as Lurupa [25] or LPex [26], but the effect on performance is uncertain. Given that the probabilities that we compute are many orders of magnitude larger than double precision floating-point roundoff errors, we have kept GLPK as our solver.

B. Evaluation of Probabilistic AA

Using only probabilistic AA, we can compute a sound overapproximation of the wrong path probability (WPP). However, our experimental results show that for many of our benchmarks, the WPP computed is 1 or close to 1 (see the column marked 'A' in Table III). For this experiment, we discretized each of the initial distributions into 25 pieces, and we have not observed a large impact on the results if we discretize further. The reason behind this high overapproximation is that the intervals of the focal elements tend to be wide and thus too many intersect with the critical interval.

C. Interval Subdivision + Reachability Checks

To reduce the overapproximation, we propose to subdivide the 'outer' input interval before they are passed to the probabilistic static analysis and to combine the subdivision with reachability checks. The overall algorithm is shown in Figure 3.

The algorithm takes as input an arithmetic expression `e`, an environment `E` mapping variables to intervals, the assumed

	wrong path probability					analysis time (in seconds)				
setting	A	B	C	D	E	A	B	C	D	E
# DSI subdiv.	25	0	2	4	4	25	0	2	4	4
# outer subdiv.	0	32000	16000	8000	8000 (deriv.)	0	32000	16000	8000	8000 (deriv.)
doppler	1	4.683e-2	3.064e-2	2.165e-2	**2.157e-2**	14.35	222.08	137.75	184.34	189.30
sine	0.3240	**3.125e-5**	5.762e-5	6.445e-5	6.445e-5	3.54	279.76	141.48	75.05	74.63
sineOrder3	0.3584	**6.250e-5**	1.172e-4	1.230e-4	1.230e-4	0.27	196.14	97.90	49.38	49.20
sqroot	1	**3.125e-5**	6.250e-5	9.375e-5	9.375e-5	80.51	213.92	107.28	54.35	53.88
bspline0	1	**3.125e-5**	5.469e-5	6.055e-5	6.055e-5	0.76	207.97	104.77	51.88	51.86
bspline1	0.9597	6.250e-5	3.125e-5	**1.953e-5**	**1.953e-5**	3.95	195.92	98.78	49.16	49.08
bspline2	0.9161	**6.250e-5**	1.230e-4	1.719e-4	1.719e-4	3.02	197.29	98.62	49.60	49.52
bspline3	1	3.125e-5	3.125e-5	**7.813e-6**	**7.813e-6**	0.76	190.35	94.85	47.51	47.47
rigidBody1	1	7.992e-2	7.105e-2	7.060e-2	**7.052e-2**	0.65	190.26	116.32	154.54	147.83
rigidBody2	1	1.022e-1	0.1048	**9.229e-2**	0.1069	1.26	235.95	145.07	238.85	242.38
turbine1	1	5.747e-2	5.767e-2	**4.820e-2**	5.252e-2	65.24	250.59	323.07	376.17	425.58
turbine2	1	5.280e-2	5.441e-2	**4.641e-2**	4.905e-2	17.81	223.60	215.74	259.42	250.76
turbine3	1	6.965e-2	6.387e-2	**5.389e-2**	5.731e-2	49.15	238.16	356.71	447.97	524.22
traincar1	0.0983	4.807e-2	2.982e-2	**1.863e-2**	2.465e-2	0.21	184.95	99.15	58.29	59.78
traincar2	0.1142	0.2958	0.1741	**9.173e-2**	0.1232	1.24	112.28	87.43	302.02	384.86
traincar3	**0.1328**	0.5372	0.3663	0.1971	0.1329	1.23	109.72	57.82	186.31	460.53
traincar4	**0.2043**	0.8545	0.7132	0.4200	0.2826	1.32	132.27	21.32	73.90	650.28

TABLE III
WRONG PATH PROBABILITIES COMPUTED WITH DIFFERENT SETTINGS FOR 32 BIT FLOATING-POINT ROUNDOFF ERRORS AS UNCERTAINTY, UNIFORM INPUT DISTRIBUTIONS AND A HIGH THRESHOLD DEFINING THE CRITICAL INTERVAL, AS WELL AS ANALYSIS TIMES (AVERAGED OVER 3 RUNS)

```
1  def wrongPathProb(expr e, env E, distribution dist,
2     divLimit L, discretLimit X, criticalInterval cI):
3    numDiv = pow(L, 1/len(E)) // distributing L
4    subDomains = carthesianSubdiv(E, numDiv, dist)
5    $\rho = 0$ // total WPP
6    for (dom, $\rho_{\text{dom}}$) $\leftarrow$ subDomains:
7      if (cheackReachable(e, dom, cI)):
8        outputDSI = probAnalysis(e, dom, X)
9        $\rho_{\text{loc}}$ = intersect(cI, outputDSI)
10     else:
11       $\rho_{\text{loc}} = 0$ // critical interval not reachable
12     $\rho = \rho + \rho_{\text{dom}} \times \rho_{\text{loc}}$ // add local WPP
13  return $\rho$
```

Fig. 3. Interval subdivision with reachability check

distribution of inputs `dist`, a limit on outer subdivisions `L`, a limit on the discretization of input distributions `X` and the critical interval `cI`.

The algorithm first distributes the outer subdivision limit `L` among the input variables (line 3), and then subdivides each input variable's interval `numDiv` times such that we obtain overall at most `L` subdomains (line 4).

For each subdomain `dom`, the algorithm then checks whether the critical interval is reachable (line 7). This check is performed by encoding reachability as an SMT query and discharged using Z3 [27]. We set a timeout of 1s; if the SMT solver times out, we assume the critical interval is reachable, thus ensuring soundness. If the critical interval is potentially reachable, we perform probabilistic analysis, obtaining a sound overapproximation of the output distribution as an DSI (line 8). From this, we obtain the wrong path probability ρ_{loc} for the subdomain `dom` by summing up the probabilities of the `outputDSI`'s focal elements which intersect with the critical interval (line 9). If the SMT solver determines that the critical interval is not reachable from `dom`, the wrong path probability is $\rho_{\text{loc}} = 0$.

Finally, the algorithm returns the overall WPP as the weighted sum $\rho = \sum \rho_{\text{dom, i}} \times \rho_{\text{loc, i}}$, where the weights ρ_{dom} depend on the input distribution. Our implementation currently supports normal and uniform distributions; e.g. for the latter, all ρ_{dom}'s are the same.

Recall our running example from Figure 1 which takes three inputs $x1, x2, x3$ uniformly distributed in $[-15.0, 15.0]$. Choosing $L = 8000$, our algorithm subdivides each input range into 20 pieces, each with equal weight. For this example, the reachability checks determine that for 87% of the subdomains the critical interval is unreachable and thus for these the probability analysis is skipped. For the remaining 13%, probabilistic analysis computes the total WPP as 0.07060. The WPP using only reachability checks and no probabilistic analysis would be 0.07992.

D. Evaluation of Parameter Settings

The algorithm in Figure 3 is conceptually simple, but very effective for computing tight wrong path probabilities. Nonetheless, it has several parameters which need to be specified. In particular, it performs two kinds of subdivisions: we can discretize the DSIs in the probabilistic analysis, and we can subdivide the input intervals. We determined suitable parameters with a systematic empirical exploration, of which we present a subset here. For our evaluation in section VI, we choose one such setting for which the analysis is then fully automated.

Table III summarizes our experiments, where we compare the wrong path probabilities computed and the analysis time taken for the following different subdivision strategies, which we denote with the letters A-E. The row '# DSI subdiv' gives the number of focal elements of each input variable distribution, and '# outer subdivision' the maximum value of L in Figure 3. To ensure a fair comparison, we choose the number of outer and DSI subdivisions such that the total number of divisions is at most 32000.

A, probabilistic analysis only: In this experiment, we do not subdivide the outer interval, i.e. L = 1. The probabilistic analysis discretizes each initial variable distribution into 25 focal elements.

B, non-probabilistic analysis: Here, we only subdivide the outer interval and do not apply any probabilistic analysis. If the critical interval is reachable, as determined using the SMT solver, we assign 1 to the local probability and 0 otherwise. In general, this analysis already provides much better results, i.e. smaller wrong path probabilities, and for some of our univariate benchmarks, even the best results. On the other hand, the analysis is on the more expensive side.

C, outer subdivision and probabilistic analysis with coarse discretization: In this setting, we apply both discretization and outer subdivision, but choose the focus on the latter, i.e. we choose a large $L = 16000$ and a small number of focal elements (2 per variable) for the initial discretization. This setting provides overall decent, but not best, results.

D, outer subdivision and probabilistic analysis with larger discretization As in setting C, we use both kinds of subdivisions, but use a smaller $L = 8000$, but 4 initial focal elements (per variable). We found this setting to provide overall the best results, at an acceptable analysis time.

E: derivative guided outer subdivision For settings B - D, we subdivide each input variable equally (outer subdiv.). One could argue that some variables may influence the result more or less. Based on this idea, we use the magnitude of the derivative w.r.t. each input variable to decide which variable's interval should be subdivided more: variables with larger derivatives are subdivided more. Somewhat surprisingly, this provides a (small) benefit only for two benchmarks, when compared to setting D.

The running times are below 10mins, except for traincar4 where the analysis takes 10.838mins (650.28s), which is comparable to the 10min timeout for PSI. We note that increasing the running time for PSI beyond 10mins did not lead to any different results and that Mathematica's running time needs to be accounted for as well.

Overall, we observe that for univariate functions outer subdivision alone without a probabilistic analysis is sufficient to provide reasonable wrong path probabilities and for the linear traincar controllers, DSI discretization alone is enough without outer subdivisions. For other benchmarks, a combination with outer and DSI subdivision provides the best results, with setting D being better overall.

benchmark	high threshold	low threshold
doppler	2.16e-2	4.93e-3
sine	6.45e-5	6.25e-5
sineOrder3	1.23e-4	6.25e-5
sqroot	9.38e-5	7.62e-5
bspline0	6.05e-5	6.64e-5
bspline1	1.95e-5	6.05e-5
bspline2	1.72e-4	5.66e-5
bspline3	7.81e-6	6.64e-5
rigidBody1	7.06e-2	6.28e-3
rigidBody2	9.23e-2	6.10e-2
turbine1	4.82e-2	4.47e-3
turbine2	4.64e-2	3.35e-3
turbine3	5.39e-2	2.08e-3
traincar1	1.86e-2	2.61e-3
traincar2	9.17e-2	3.51e-3
traincar3	1.97e-1	7.29e-4
traincar4	4.20e-1	7.19e-3

TABLE IV
WPP FOR HIGH AND LOW THRESHOLDS (SETTING D)

We note that for those benchmarks, where PSI computes a probability *using exact inference*, the results with our static analysis method are quite close, confirming that the overapproximations due to static analysis are acceptable.

1) Low vs High Thresholds: Finally, we also compare our analysis results using setting D for low and high thresholds, see Table IV. We observe that our static analysis method is not able to detect the low-probability threshold as consistently as PSI, which is due to the fact that our method computes an overapproximation.

E. Future Improvements

While our prototype already computes useful wrong path probabilities, the performance of our implementation could be improved in several ways. First, the probabilistic analysis on each subdomain is trivially parallelizable, but we currently compute it sequentially as the GLPK solver is not threadsafe (but has a convenient interface for a prototype implementation). Secondly, a smarter subdivision similar to branch-and-bound methods could reduce the number of subdomains checked by SMT. Next, instead of DSIs, we could employ a more sophisticated method using concentration of measure inequalities [28]. This method does not necessarily provide tighter probability bounds, but is more efficient in general (but also more complex to implement and the implementation from [28] is not available). Finally, our implementation using rationals is costly and can be improved either by using high-precision floating-point arithmetic internally—at the expense of some guarantees—or by using interval arithmetic with floating-point bounds and outwards rounding—at the expense of complexity of implementation.

VI. Further Evaluation

In this section, we choose the overall most successful setting found in subsection V-C (setting D) and perform additional experiments, comparing the results computed by our method for different sized uncertainties, for uniform and normal distributions, as well as independent and dependent inputs. We furthermore choose a number of additional benchmarks where studying discrete choices is particularly relevant.

`filter`: We unrol this 2nd order numerical filter [8], which appears frequently in embedded software, three and four times and obtain benchmarks with 15 and 25 arithmetic operations and 3 and 4 input variables each.

`solveCubic`: This benchmark, obtained from GNU Scientific Library, has been used in previous work [29] in the context of test case generation for floating-point programs with branches. The benchmark has 14 operations and 3 variables.

`classID`: We trained a Linear Support Vector Classifier using the Python sklearn library on the Iris standard data set which comes with sklearn, and extracted source code from the classifier using sklearn-porter [30]. For our benchmark, we have inlined the weights computed and the three versions of this benchmark correspond to the three decision variables, resp. classes, in the data set. They each have 15 arithmetic operations and 4 input variables.

`neuron`: This is a simplified version of the DNN which is supposed to have learned the 'AND' operator [31]. This neural network has one hidden layer of size 2, and six weights defining them. We consider here a one-input version, keeping one of the inputs constant. This benchmark uses the exponential function and has 22 arithmetic operations.

a) Different Sized Uncertainties: In this first experiment we compare the wrong path probabilities computed using our method for different-sized critical intervals: obtained by computing roundoff errors for 16-bit fixed-point and 32-bit floating-point arithmetic without input uncertainties, as well as assuming an input uncertainty of 0.001 (and 32-bit floating-points). All errors have been computed using the Daisy tool. The four missing entries in the fixed-point arithmetic column are due to overflows in the 16-bit implementation, and for the neuron benchmarks, an input error of 0.001 leads to a potential division by zero.

Table V shows the results. We observe that while the wrong path probabilities are all relatively small, our method is not able to distinguish in a notable way, except for the bspline benchmark, between different sized critical intervals. This is, as before with the low and high thresholds, a fundamental limitation of our abstraction-based method, which we show here for completeness.

b) Uniform vs Gaussian Inputs: In this section, we use our analysis to compare how the wrong path probability changes with the input distributions. For this we run our analysis with all inputs uniformly and normally distributed, but keeping the thresholds, i.e. critical intervals, constant.

Table VI shows the results. As expected, the wrong path probabilities can differ substantially, which shows the importance of taking input distributions into account. We also show the running times, as we have observed differences. We suspect that these are due to the reduction method, which will have different effects for different input distributions.

benchmark	no uncertainty		input uncert.: 0.001
	16-bit fixed	32-bit float	32-bit float
bspline2	1.38e-3	5.66e-5	6.08e-3
rigidBody1	6.35e-3	6.28e-3	6.32e-3
rigidBody2	-	5.05e-2	5.22e-2
traincar1	2.72e-3	2.61e-3	2.66e-3
traincar2	3.53e-3	3.51e-3	3.53e-3
traincar3	-	7.29e-4	7.36e-4
traincar4	-	7.19e-3	7.19e-3
filter3	3.51e-3	3.23e-3	3.31e-3
filter4	6.81e-4	6.72e-4	6.74e-4
solveCubic	3.32e-5	3.13e-5	3.32e-5
classID2	9.50e-2	9.37e-2	9.44e-2
classID1	5.79e-2	5.70e-2	5.74e-2
classID0	5.79e-2	5.70e-2	5.76e-2
neuron	-	1.38e-1	-

TABLE V
WPP for different critical intervals

benchmark	wrong path probability		time (in s)	
	uniform	normal	uniform	normal
bspline2	5.66e-5	6.43e-5	53.30	59.86
rigidBody1	6.28e-3	4.39e-3	69.20	153.81
rigidBody2	5.05e-2	5.10e-2	195.46	595.88
traincar1	2.61e-3	1.69e-3	52.02	51.89
traincar2	3.51e-3	2.10e-3	68.07	74.67
traincar3	7.29e-4	4.22e-4	25.01	35.00
traincar4	7.19e-3	5.49e-3	20.70	42.13
filter3	3.23e-3	1.97e-3	54.54	54.46
filter4	6.72e-4	3.17e-4	47.14	48.94
solveCubic	3.13e-5	1.30e-5	58.00	147.38
classID2	9.37e-2	0.100	383.73	661.24
classID1	5.70e-2	6.49e-2	293.04	532.12
classID0	5.70e-2	6.37e-2	331.13	609.52
neuron	0.138	0.160	16.34	16.31

TABLE VI
Uniformly vs. normally distributed inputs

c) Dependent Inputs: Until now, we have always assumed that all inputs were independent and dependencies were only introduced by arithmetic operations. In this experiment, we look at the wrong path probability computed with all inputs being either independent or dependent (with unknown dependency).

Table VII shows the results. We observe that in general the wrong path probabilities computed are greater for the dependent case. This is to be expected as we have to consider all possible dependencies. The difference are, however, not too large and the results still useful.

Curiously, we also observe that the running times of our method are *smaller* for the dependent case, even though in this case we are calling the LP solver for every arithmetic operation. But this is also the reason for the smaller running times; the GLPK solver we use is implemented in floating-

benchmark	wrong path probability		time (in s)	
	indep.	dep.	indep.	dep.
bspline2	5.66e-5	6.25e-5	53.30	54.27
rigidBody1	6.28e-3	1.37e-2	69.20	53.53
rigidBody2	5.05e-2	7.68e-2	195.46	77.91
traincar1	2.61e-3	5.22e-3	52.02	52.32
traincar2	3.51e-3	9.48e-3	68.07	56.88
traincar3	7.29e-4	7.20e-3	25.01	19.08
traincar4	7.19e-3	8.89e-2	20.70	9.32
filter3	3.23e-3	3.78e-3	54.54	54.20
filter4	6.72e-4	5.33e-4	47.14	47.36
solveCubic	3.13e-5	2.50e-4	58.00	56.43
classID2	9.37e-2	0.158	383.73	62.57
classID1	5.70e-2	8.66e-2	293.04	57.92
classID0	5.70e-2	0.119	331.13	60.38
neuron	0.138	0.138	16.34	6.82

TABLE VII
INDEPENDENT VS. DEPENDENT INPUTS (WITH UNIFORM INPUT DISTRIBUTIONS)

point arithmetic, whose results we translate to rationals at the interface to our implementation. Calling the solver often effectively does not allow our rationals to grow too large and thus keeps the running time low.

VII. Related Work

a) Probabilistic Inference: In section III we have chosen the tool PSI for our experiments which performs exact symbolic inference. Probabilistic inference is an active field, and there are several other alternative approaches. Among many others, R2 uses MCMC sampling in combination with program analysis to improve efficiency [15], Stan's Hamiltonian sampling generates samples from high-probability regions [14]. Infer.NET implements several algorithms including expectation and belief propagation and Gibbs sampling [13], and it achieves efficiency by compiling the graphical models into executable code. MAYHAP [32] statically builds a distribution from a probabilistic program simplifies the Bayesian network it constructs before evaluating it via sampling. While these approaches provide efficient solutions, they fundamentally are all sampling based and thus cannot provide guaranteed bounds.

b) Probabilistic Program Analysis: Our approach is linked to probabilistic static analysis, most notably to the approach of [28], based on the propagation of uncertain probability distributions, represented as P-boxes or Dempster-Shafer structures. More broadly related is the work by Sankaranarayanan et.al. [33], which verifies probabilistic properties for programs with many paths by examining only finitely many. A combination of symbolic execution and volume computation provides bounds on the probability of the property holding. This work only considers linear programs and does consider the effects of finite-precision, while handling more general programs and properties. Uncertain<T> [34] is a programming language construct which makes uncertainties explicit and which is lazily evaluated by sampling. While the input programs are floating-point programs, roundoff errors are not taken into account.

c) Finite-Precision Roundoff Error Analysis: Alternative tools to Daisy for computing roundoff error bounds are for instance FPTaylor [1] and Fluctuat [3]. The former's analysis technique does not support fixed-point arithmetic and the latter is similar to Daisy in the analysis used, but the code is not available as open source. The results computed by these tools have been shown to be largely comparable for our purpose [1], [9]. Daumas et.al. [35] compute bounds on the probability that accumulated floating-point roundoff errors exceed a given threshold, using known inequalities on sums of probability distributions. This approach has also been used for value analysis [28] and could be combined with ours, in the future. Constraint-based approaches [36] have been recently proposed which heuristically generate test cases where floating-point and real-valued results differ, and in particular when the control flow differs [29]. This work is complementary to our approach as the generated test cases can be used for debugging, for instance once our approach computes a wrong path probability which is too large.

d) Finite-precision in Classification: While a classifier can be trained in low precision [37], the common way is to train in some higher precision and then translate to a lower precision [38], [6]. For example, Lin et. al. [39] translate a neural network trained in floating- point arithmetic to fixed-point arithmetic using a dynamic analysis to determine the dynamic range, mean and standard deviation of all weights and activations. The error analysis is different from ours in that computes the signal-to-noise ratio (SNR). It is also only approximate and the evaluation shows that the predicted and observed SNR do not match exactly. Our proposed method computes, in contrast, a guaranteed bound on the misclassification probability, albeit for classifiers of limited size which are nonetheless important in safety-critical systems [6].

Reluplex [40] uses SMT-solving to prove input-output properties of deep-neural networks (DNNs) with ReLus. In particular, it can test for adversarial robustness, i.e. perturbing a particular input up to some small quantity does not change the classification result. Our approach is not specific to DNNs, but can be used to *quantify* the potential misclassification rate.

VIII. Conclusion

We have considered several guaranteed analyses for bounding the effects of numerical uncertainties on discrete decisions in terms of the probability of making the wrong decision. We conclude that existing exact and static analysis approaches have shortcomings in terms of scalability and accuracy, but that those can be overcome with a combination with a non-probabilistic reachability analysis. Our presented method computes wrong path probabilities which are tight enough to be useful in practice for small, but interesting embedded programs. We view this work as a first important step, with more work especially on improving the performance to follow.

References

[1] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan, "Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions," in *FM*, 2015.

[2] M. Daumas and G. Melquiond, “Certification of Bounds on Expressions Involving Rounded Operators,” *ACM Trans. Math. Softw.*, vol. 37, no. 1, pp. 2:1–2:20, 2010.
[3] E. Goubault and S. Putot, “Static Analysis of Finite Precision Computations,” in *VMCAI*, 2011.
[4] ——, “Robustness Analysis of Finite Precision Implementations,” in *APLAS*, 2013, pp. 50–57.
[5] E. Darulova and V. Kuncak, “Towards a Compiler for Reals,” *ACM TOPLAS*, vol. 39, no. 2, 2017.
[6] R. Braojos, G. Ansaloni, and D. Atienza, “A Methodology for Embedded Classification of Heartbeats using Random Projections,” in *DATE*, 2013.
[7] S. Misailovic, M. Vechev, and T. Gehr, “PSI: Exact Symbolic Inference for Probabilistic Programs ,” in *CAV*, 2016.
[8] A. Adje, O. Bouissou, J. Goubault-Larrecq, E. Goubault, and S. Putot, “Static Analysis of Programs with Imprecise Probabilistic Inputs,” in *VSTTE*, 2013.
[9] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian, “Daisy - Framework for Analysis and Optimization of Numerical Programs,” in *TACAS*, 2018. [Online]. Available: https://github.com/malyzajko/daisy
[10] A. Anta and P. Tabuada, “To Sample or not to Sample: Self-Triggered Control for Nonlinear Systems,” *IEEE Transactions on Automatic Control*, vol. 55, no. 9, pp. 2030–2042, 2010.
[11] O. Bouissou, E. Goubault, J. Goubault-Larrecq, and S. Putot, “A Generalization of P-boxes to Affine Arithmetic,” *Computing*, vol. 94, no. 2-4, pp. 189–201, 2012.
[12] G. F. Cooper, “The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks (Research Note),” *Artif. Intell.*, vol. 42, no. 2-3, pp. 393–405, 1990.
[13] T. Minka, J. Winn, J. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill, “Infer.NET 2.6,” 2014, Microsoft Research Cambridge. http://research.microsoft.com/infernet.
[14] B. Carpenter, A. Gelman, M. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell, “Stan: A Probabilistic Programming Language,” *Journal of Statistical Software, Articles*, vol. 76, no. 1, pp. 1–32, 2017.
[15] A. V. Nori, C.-K. Hur, S. K. Rajamani, and S. Samuel, “R2: An Efficient MCMC Sampler for Probabilistic Programs,” in *AAAI*, 2014.
[16] W. R. Inc., “Mathematica, Version 10.4,” https://www.wolfram.com/mathematica/, 2018, champaign, IL.
[17] D. U. Lee, A. A. Gaffar, R. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, “Accuracy-Guaranteed Bit-Width Optimization,” *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 25, no. 10, pp. 1990–2000, 2006.
[18] R. Moore, *Interval Analysis*. Prentice-Hall, 1966.
[19] G. Shafer, *A Mathematical Theory of Evidence*. Princeton university press, 1976, vol. 42.
[20] S. Ferson, V. Kreinovich, L. Ginzburg, D. S. Myers, and K. Sentz, “Constructing Probability Boxes and Dempster-Shafer Structures,” Technical report, Sandia National Laboratories, Tech. Rep., 2003.
[21] D. Berleant and C. Goodman-Strauss, “Bounding the Results of Arithmetic Operations on Random Variables of Unknown Dependency Using Intervals,” *Reliable Computing*, vol. 4, no. 2, pp. 147–165, May 1998.
[22] L. H. de Figueiredo and J. Stolfi, “Affine Arithmetic: Concepts and Applications,” *Numerical Algorithms*, vol. 37, no. 1-4, 2004.
[23] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, “MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding,” *ACM Trans. Math. Softw.*, vol. 33, no. 2, 2007.
[24] “GLPK (GNU Linear Programming Kit),” https://www.gnu.org/software/glpk/, 2012.
[25] C. Keil, “Lurupa - Rigorous Error Bounds in Linear Programming,” in *Algebraic and Numerical Algorithms and Computer-assisted Proofs*, ser. Dagstuhl Seminar Proceedings, no. 05391, 2006. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2006/445
[26] M. Dhiflaoui, S. Funke, C. Kwappik, K. Mehlhorn, M. Seel, E. Schömer, R. Schulte, and D. Weber, “Certifying and Repairing Solutions to Large LPs. How Good are LP-Solvers?” in *SODA*, 2003, pp. 255–256.
[27] L. De Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” in *TACAS*, 2008.
[28] O. Bouissou, E. Goubault, S. Putot, A. Chakarov, and S. Sankaranarayanan, “Uncertainty Propagation using Probabilistic Affine Forms and Concentration of Measure Inequalities,” in *TACAS*, 2016.
[29] H. Zitoun, C. Michel, M. Rueher, and L. Michel, “Search Strategies for Floating Point Constraint Systems,” in *CP*, 2017.
[30] “Project Sklearn-porter,” https://github.com/nok/sklearn-porter, 2018.
[31] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, “The Limitations of Deep Learning in Adversarial Settings,” in *EuroS&P*, 2016.
[32] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze, “Expressing and Verifying Probabilistic Assertions,” in *PLDI*, 2014.
[33] S. Sankaranarayanan, A. Chakarov, and S. Gulwani, “Static Analysis for Probabilistic Programs: Inferring Whole Program Properties from Finitely Many Paths,” in *PLDI*, 2013.
[34] J. Bornholt, T. Mytkowicz, and K. S. McKinley, “Uncertain<T>: A First-order Type for Uncertain Data,” in *ASPLOS*, 2014.
[35] M. Daumas, D. Lester, E. Martin-Dorel, and A. Truffert, “Improved bound for stochastic formal correctness of numerical algorithms,” *Innovations in Systems and Software Engineering*, vol. 6, no. 3, pp. 173–179, 2010.
[36] H. Collavizza, C. Michel, and M. Rueher, “Searching Critical Values for Floating-Point Programs,” in *ICTSS*, 2016.
[37] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations,” *arXiv e-prints*, vol. abs/1609.07061, 2016.
[38] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, “Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights,” in *ICLR*, 2017.
[39] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, “Fixed Point Quantization of Deep Convolutional Networks,” in *ICML*, 2016.
[40] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, “Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks,” in *CAV*, 2017.

Debasmita Lohar received her Bachelor of Technology degree in Computer Science and Engineering from the Heritage Institute of Technology, Kolkata, India in 2013. She completed her Master of Science in Computer Science and Engineering from the Indian Institute of Technology Kharagpur, India in 2017. Currently she is a doctoral student at the Max Planck Institute for Software Systems, Saarbrücken, Germany. Her research interests include programming languages and verification.

Eva Darulova obtained her PhD from École Polytechnique Fédérale de Lausanne, Switzerland, in 2014. Since 2015, she is a tenure-track faculty at the Max Planck Institute for Software Systems in Germany. Her research interests include programming languages, software verification and approximate computing.

Sylvie Putot is Professor of Computer Science at Ecole Polytechnique. Her research focuses on verification methods, from program analysis by abstract interpretation to the analysis of hybrid and cyber-physical systems.

Eric Goubault is a professor at Ecole Polytechnique, currently serving as head of the computer science department. He has written about 90 papers in semantics and static analysis of programs, concurrent and distributed systems, and hybrid systems. His main focus now is on modeling and analysis of Cyber-Physical Systems, using and developing a variety of methods and theories among which are abstract interpretation, model-checking, set-based, topological and algebraic methods.