Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

2012

White Box Sampling in Uncertain Data Processing Enabled by Program Analysis

Tao Bao
Purdue University, tbao@cs.purdue.edu

Yunhui Zheng
Purdue University, zheng16@purdue.edu

Xiangyu Zhang
Purdue University, xyzhang@cs.purdue.edu

Report Number:
12-002

White Box Sampling in Uncertain Data Processing Enabled by Program Analysis

Tao Bao, Yunhui Zheng, Xiangyu Zhang

Department of Computer Science, Purdue University
{tbao,zheng16,xyzhang}@cs.purdue.edu

Abstract

Sampling is a very important and low-cost approach to uncertain data processing, in which output variations caused by input errors are sampled. Traditional methods tend to treat a program as a blackbox. In this paper, we show that through program analysis, we can expose the internals of sample executions so that the process can become more selective and focused. In particular, we develop a sampling runtime that can selectively sample in input error bounds to expose discontinuity in output functions. It identifies all the factors that can potentially lead to the discontinuity and hash the values of such factors in a cost-effective way. The hash values are used to guide the sampling process. We overcome a list of practical challenges, and develop techniques to mitigate the safety issue. Our results show that the technique is very effective for real-world programs.

1. Introduction

Uncertain data processing is becoming more and more important. In scientific computation, data are collected through instruments or sensors that may be exposed to rough environmental conditions, leading to errors. Computational processing of these data may hence draw faulty conclusions. For example, it was shown in [18] that a protein was mistakenly classified as a cancer indicator by slightly altering a parameter of the program used to process experimental data. Such parameters are uncertain because they are provided by biologists based on their experience. Such mistakes could be highly costly because expensive follow-up wet-bench experiments could be further conducted based on the faulty protein. Long term rainfall prediction is often realized by the software operating on Sea Surface Temperature (SST) data [16]. Due to the difficulty and the cost of deploying sensors, SST contains a lot of interpolated data, which are uncertain. In bioinformatics, one of the most widely used sources for protein data is Uniprot [8], in which proteins are annotated with functions. The annotations may come from real experiments (accurate) or computation based on protein similarity (uncertain). Software operating on these data has to be aware of the uncertainty issue [11]. Software facilitating financial decision making is often required to model the uncertainty [12].

Traditionally, uncertainty analysis is conducted on the underlying mathematical models [17]. However, modern data processing uses more complex models and relies on computers and programs, rendering mathematical analysis difficult. Realizing the importance of uncertain data processing, recently, researchers have proposed database techniques to store, maintain and query uncertain data [10, 13]. However, more sophisticated data processing is often performed outside a database by programs written in high level languages. Addressing uncertainty from the program analysis perspective becomes natural. Continuity analysis [6] is a static analysis that proves a program always produces continuous output given a set of uncertain inputs. However, the most common case in uncertain data analysis is that given a set of concrete inputs, with some uncertain, scientists want to reason about output variations. It demands analyzing program execution instead of the program itself. More importantly, while the analysis has been shown to be effective on simple programs such as sorting algorithms, real world programs are a lot more complex, involving complex control flows, high order functions, array and pointer manipulations. Automatic derivative computation [3] uses compilers to instrument a program so that the output derivative can be automatically computed. However, these techniques cannot directly reason about changes within an input error bound; they also have difficulties in handling certain language features, such as the control flow.

Monte Carlo (MC) methods provide a simple and effective means of studying the uncertainty [5, 10, 15]. They randomly select input values from predefined distributions and aggregate the computed outputs to yield statistical insights in the output space. While continuous functions are relatively easier to be approximated by MC methods, as data processing is realized by complex programs, outputs are no longer continuous functions of the uncertain inputs. Discontinuity poses significant challenges. Some of the problems are illustrated in Fig 1. These figures show how the output changes according to the variation of the uncertain input. Points represent samples. The first problem in (a) is that from the samples, it is hard to determine if the output is continuous (curve *A*) or discontinuous (*B*). The second problem in (b) is that even though we know that the curve is discontinuous, it is yet difficult to determine where the discontinuity occurs. In this case, it could occur at any locations such as the three shown in the figure, resulting in curves $a\alpha b$, $a\beta b$, and $a\gamma b$, respectively. The third problem in (c) is more problematic as the four samples appear to follow a simple mathematical function (a straight line through *a* and *b*) but indeed there is a discontinuous segment in between *a* and *b*. In many cases, these segments are so small that they are prone to be omitted by random sampling. In fact, we observe that a tiny discontinuous segment along a simple linear function was the root cause for misclassifying an irrelevant protein as the cancer biomarker in our experiment.

In this paper, we develop a *white-box* MC method, powered by program analysis. The technique aims to guide the sampling process through a lightweight dynamic analysis so that output discontinuity for a given input error bound can be disclosed with a small number of samples. The discontinuous points break the output curve into a set of continuous segments, which can be easily approximated with a traditional MC process. Our observation is that for many data processing tasks implemented by programs, discontinuity is mainly caused by language artifacts such as conditional statements and type casting, instead of the intrinsic mathematical function. Hence, the idea is to monitor MC sample execution to detect such possible artifacts and direct the MC process to selectively collect more samples to precisely identify the discontinuous points.

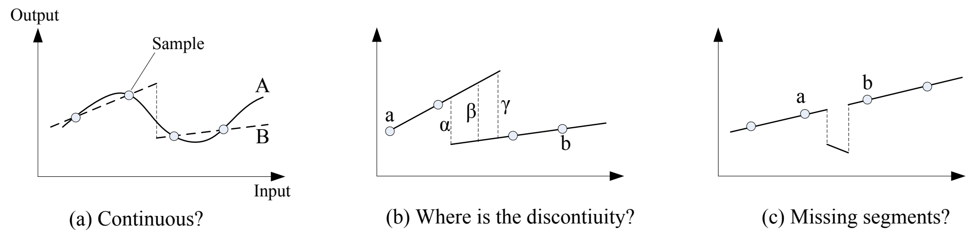


Figure 1. Three sample problems caused by discontinuity.

The basic idea is the following. During a sample execution, the technique generates a hash value that aggregates the execution of the language artifacts that could potentially lead to discontinuity. If two sample runs have the same hash value, implying the same control flow and identical discrete coefficients, the output functions in the two runs have the same mathematical form, suggesting continuity in the range delimited by the two samples. The technique hence avoids collecting more samples in between the two samples. If the hash values differ, an additional sample is taken in between the two original samples. The process continues to inspect the two sub-ranges divided by the new sample, until all sub-ranges become continuous or the discontinuous points are sufficiently narrowed down. In order to make the idea work in practice, we address a list of practical challenges in this paper. Our contributions are highlighted as follows.

- We formally define the problem and identify the possible sources for discontinuity in a program.
- We propose a novel dynamic program analysis driven MC algorithm that preforms selective sampling to expose the discontinuity in the output function.
- We study the safety issue of the algorithm, which is to determine if it is safe to stop further sampling when two sample runs produce the same hash value. We reduce the problem to checking the monotonicity of the internal states of the sample runs. We also propose a vector-based runtime that allows us to efficiently determine the monotonicity.
- We observe that in real world programs, execution differences between sample runs may be irrelevant to the output. We propose a slice-based hashing algorithm that considers only the relevant part of an execution.
- We also observe that in real world programs, there are small code regions in which control flow differences do not cause discontinuity. Such differences are mostly intensional for purposes such as optimization. We develop a profiler to identify such code regions and prove their continuities. The hashing algorithms can thus avoid hashing these regions.
- We evaluate our technique. The results show that the proposed white-box sampling technique can identify the discontinuity effectively and efficiently, and with high confidence.

2. Problem Statement

Given an execution that is derived from a concrete input, we assume part of the input is uncertain. Our ultimate goal is to understand how the program output changes within the input error bound.

We use $x_1,\ x_2, \ldots$ to denote multiple uncertain inputs. An example for such uncertain input is a real number in the input array received from a sensor. We only consider real number inputs unless stated otherwise. The program *execution* is hence abstracted as a function over the uncertain input, denoted as $P(x_1, \ldots x_n)$.

In this paper, we aim to *develop a white-box MC method that can quickly and effectively determines the shape of the function, especially the discontinuity of the function,* as continuous portions can be easily approximated by a regular MC process.

We first precisely define the term continuity. To simplify the discussion, we assume one uncertain input in the definition, even though our technique supports multiple uncertain inputs.

Definition 1. (Continuity) *$P(x)$ is said to be continuous at the point $x = c$ if the following holds: For a value $\varepsilon > 0$, however small, there exists some value $\delta > 0$ such that for any x within the error bound and satisfying $c - \delta < x < c + \delta$, we have $P(c) - \varepsilon < P(x) < P(c) + \varepsilon$.*

*$P(x)$ is **discontinuous** at $x = c$ if the above condition is not satisfied. $P(x)$ is said to be **continuous** if it is continuous throughout the error bound of x.*

Intuitively, if P is continuous regarding the uncertain input x, then any small change to the value of x can only cause a small change to the output value $P(x)$.

Discrete Factors. Our goal is to identify the discontinuity, which cannot be easily exposed by sampling the output. Some of the problems are illustrated by Fig. 1 and discussed in Section 1. Our observation is that the internals of a sample execution provide a lot of hints to the discontinuity of the output function. We define the term *discrete factor* to represent such program artifacts.

Definition 2. *A discrete factor is an operation that has real values as operands and produces a discrete value as result.*

In most cases, discrete factors are the root cause for the output discontinuity. Because for uncertain inputs (real numbers) to induce discontinuity, they have to go through some discrete factors. The basic idea of our technique is to monitor the execution of these discrete factors to detect the discontinuity and guide the sampling process. Next, we discuss the most common discrete factors that we have observed over a set of real world programs.

Type cast. A continuous floating point value can be casted to a discrete type, such as the integer, leading to discontinuity in the final output. Besides explicit casting, implicit casting may also be automatically performed by a compiler when necessary, such as when discrete operations (e.g. `mod`) are applied to floating point values.

Discrete mathematical library functions. Data processing programs usually make heavy use of third-party mathematical library functions. Some of these functions are discrete, such as `SIGN(v)`, which returns 1 if v is positive, -1 if negative, and 0 otherwise. They may eventually lead to disruptions along the output curve.

Control flow. Modern programming languages allow developers to manipulate the control flow through constructs such as `if-then-else` statements and loops. These constructs are the key elements that allow data processing to go beyond the traditional pure mathematical modeling. However, they substantially increase the difficulty of uncertainty analysis by introducing discontinuity. In particular, if a value computed from uncertain inputs is used in a predicate and the predicate guards the following computation leading to the output, there is a good chance discontinuity is introduced. The reason is that the branch outcome may vary depending on the uncertain values, leading to different mathematical forms of the output. An example will be presented in Section 3. In our experience, control flow is the dominant discrete factor.

Besides discrete factors, discontinuity may also arise from the intrinsic mathematical model. For example, $f = \frac{1}{x+1.0}$ is discontinuous at $x = -1.0$; $f = tan(x)$ is discontinuous at $x = -\frac{\pi}{2}, \frac{\pi}{2}, \ldots$.

We observe that in real world programs, such operations are often guarded by predicates or the input domains are specified in such a way that the discontinuous points are excluded. For the above $f = \frac{1}{x+1.0}$ example, a predicate is often used to guard against $x = -1.0$ to avoid runtime exceptions. As a result, the mathematical discontinuity also manifests itself as a control flow discontinuity.

Note that one might formulate the challenge as a dynamic test generation problem that generates uncertain input values exploring the different values of discrete factors. For example, exploring all the possible program paths within the input error bound may be able to expose discontinuity caused by control flow. However, we found that this is impractical for real world scientific programs for the following two reasons. (1) Path condition functions are often of high order. We find that 7 out of 11 SPEC CFP 2000 programs we have studied have high order (≥ 2) path conditions. Some of them have the order as high as a few thousands due to the iterative nature of the computation [1]. More importantly, they are mostly in a complex form, involving functions such as square root, `sin`/`cos`, and fraction. These path conditions go beyond the capability of existing solvers. (2) The entailed symbolic execution is too expensive for our target scenario. The reasoning is as follows: if a technique causes X times slow down, one may just collect X MC samples instead of using the technique.

3. An Illustrative Example

We use an example to illustrate the technique. Fig. 2 (a) shows the program. Variable x is uncertain; its value is within an error bound around the original value 1.5. The output function $o(x)$ may take different forms, depending on the value of x. If $x < 1.0$ (line 3), $o(x) = 1$ (line 4). If $x \geq 1.0$, depending on the comparison $h(x) > 0.3$ (line 6), it may take the form $o(x) = 0.3$ (line 7) or $o(x) = 0.75$ (line 9). Function $h(x)$ is high order, rendering techniques relying on constraint solving in-applicable. The curves for $h(x)$ and $o(x)$ are depicted in Fig. 2 (d). Our technique aims to leverage program analysis to guide collecting a small set of samples that disclose the shape of the output function, particularly the discontinuity.

Phase-1: Basic Sampling. In phase one, our technique first identifies all discrete factors in the program. They are places that operate on real values and produce discrete values, and thus are the root causes of the discontinuity. In this program, lines 3 and 6 are discrete factors as they operate on real values and produce boolean outputs, and the type cast at line 2 is also a discrete factor.

During a sample execution, we generate a hash value that is the aggregation of the values of all discrete factors encountered. If two sample runs have the same hash value, the discrete factors have the same values (assuming a perfect hashing scheme). It indicates that the mathematical formula of the output variable is identical, suggesting continuity in between. If they differ, an additional sample is taken in between the two original samples. The process continues to inspect the two sub-ranges divided by the new sample, until a threshold is reached.

In our example, assume the technique starts with two samples a and b (step (1) in Fig. 2 (c)). Readers can refer to Fig. 2 (d) for the samples and their corresponding outputs. Line 3 has different branch outcomes in the two sample runs, resulting in different hashes. An additional sample c is taken at the mid-value of a and b. Samples c and b lead to the identical hash, the technique stops collecting more samples in between. In contrast, it further divides the sub-region $[a, c]$ (steps (2)-(5)). The process terminates at subregion $[e, f]$ where a threshold is reached. It discloses the discontinuous point at $x = 1.0$. The large intervals without any samples in between, such as that in between c and b, indicate the savings brought by our technique. Observe that a uniform sampling scheme with the threshold as small as the interval of $[e, f]$ requires a lot more samples.

Phase-2: Validation. It may not be safe to skip sampling when two hashes are identical. In Phase-1, although we disclose the discontinuity at $x = 1.0$. We miss the discontinuity in between samples d and c. Therefore, in phase-2, we validate the correctness of our results by checking monotonicity at discrete factors. The intuition is that if the function at a discrete factor, denoted as $f(x)$, is monotonic, as we already know $f(x)$ leads to the the same discrete value in the two sample runs, $f(x)$ must lead to the same discrete value for any samples in between.

To check monotonicity, we divide the sequence of samples into segments such that all samples in a segment have the same hash value. Then we validate that for all samples in a segment, the value at a discrete factor must change monotonically. For example in Fig. 2 (d), samples (f,d,c,b) is a segment, we want to ensure that the three discrete factors in the program have their values change monotonically for these samples.

Observe that checking monotonicity requires comparing values across runs at each dynamic instance of a discrete factor. We cannot afford collecting and comparing traces because with the cost of tracing, one can easily collect many samples. Hence, we develop a vector-based runtime that executes multiple samples at a time. In the vector semantics, a variable related to the uncertain input has a vector of values, each representing the variable value in the corresponding original sample run. An operation on the variable is applied to all values in its vector. Monotonicity check is conducted by comparing values in vectors. Fig 2(c) shows how monotonicity is validated for the sequence (f, d, c, b). It shows the variable values for each executed statement. Initially, at line 1, it loads the three samples to the vector of x. At line 2, the cast operation is applied to the vector. Observe at line 6, the vector does not have monotonicity.

An additional sample g is then collected in the non-monotonic region $[d, c]$ (step (7) in Fig. 2 (c)). The algorithm now is able to detect that the two subregions contain more discontinuous points. The sampling process thus goes on.

Practical Challenges. In order to make the technique work for real world programs, we need to further overcome the following challenges.

- Discrete factors in two sample runs may behave differently. However, such differences may not be relevant to the output variable, and hence they should not cause additional samples. In Fig. 2 (a), if z is the output variable instead of o, the control flow differences in lines 3-9 should be excluded. We develop an online slice-based hashing algorithm that hashes only the discrete factors in the *dynamic slice* of the output variable.
- In reality, developers may write programs in such a way that control flow variations do not lead to discontinuity. It would lead to redundant samples. We observe such effects are usually present in small code regions. We develop a profiler to identify such code regions and prove that they must be continuous. Thus, we can avoid hashing the control flows in these regions.

The following sections discuss the individual components.

4. Basic Algorithm

Given the error bound of an uncertain input, the basic algorithm executes the program on two initial samples, usually the lower and upper bound values. During execution, it hashes the values

[1] We acquire such numbers through profiling, without conducting any mathematical reduction.

(a) program

```
x=sample(1.5);
y=(int) x;
if (x<1.0)
  o=1;
else
  if (h(x)>0.3)
     o=0.3;
  else
     o=0.75;
z=1+y;
```

(b) validation

```
x          =[f,    d,     c,    b ]
y          =[1.0, 1.0,  1.0,  2.0]
x-1.0      =[0.1, 0.25, 0.65,1.5]

h(x)-0.3=[0.40, 0.1, 0.45, 0.8]
```

(c) sampling steps

(1) [a, b]
(2) [a, c],   c=(a+b)/2
(3) [a, d],   d=(a+c)/2
(4) [e, d],   e=(a+d)/2
(5) [e, f],   f=(e+d)/2
(6) (f,d,c,b) non-monotonic.
(7) [d, g],   g=(d+c)/2
(8) …

(d) curves and samples

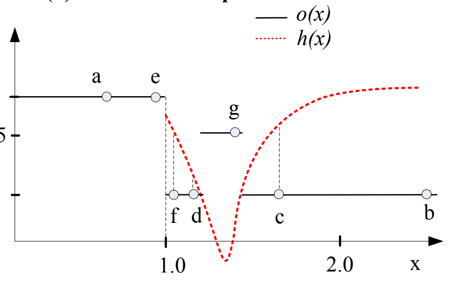


Figure 2. An illustrative example. The dots in (d) represent the samples. The highlighted statements in (a) represent discrete factors.

of all discrete factors. If the hashes of the two sample runs are identical, the algorithm stops taking more samples in the range; otherwise, it takes another sample in the middle of the two original samples and recursively considers the two new sub-ranges, until the discontinuous points are sufficiently located. Our discussion is limited to one uncertain input for simplicity, whereas our technique supports multiple uncertain inputs.

$$
\begin{array}{lll}
\textit{Program} & P ::= & s \\
\textit{Stmt} & s ::= & s_1;\, s_2 \mid \textbf{skip} \mid x^{\ell} := e \mid \textbf{while}\ x \bowtie^{\ell} 0\ \textbf{do}\ s \mid \\
 & & \textbf{if}\ x \bowtie^{\ell} 0\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \mid \textbf{exit} \\
\textit{Expr} & e ::= & x \mid v \mid \textbf{sample}(r)^{\ell} \mid e_1\ \textbf{binop}^{\ell}\ e_2 \mid \textbf{discrete}(f,e) \mid \\
 & & x \bowtie^{\ell} 0 \\
\textit{Value} & v ::= & n \mid r \mid b \\
\multicolumn{3}{c}{\textit{Var}\ x,\ \textit{Function}\ f\ \in \textit{Identifier} \quad n \in Z \quad r \in \textit{Real}} \\
\multicolumn{3}{c}{\ell\ \in\ \textit{Label} \qquad b \in \textit{Boolean}}
\end{array}
$$

Figure 3. Language

Language. To facilitate formal discussion, we introduce a simple language. The syntax is presented in Fig. 3. Note that relational operations are normalized to $x \bowtie 0$, with $\bowtie$ denoting a relational operator. The reason is that we need to study the real values (not the boolean values) of relational expressions. Such values are explicitly denoted by x after normalization. For instance, a conditional statement "**if** $y < 1.0$ ..." is normalized to "$x := y - 1.0$; **if** $x > 0$". It allows us to reason about the value of y.

We support three kinds of values: integers, real values, and boolean values. Real values could be uncertain. A **sample**(r) expression represents a sample within the error bound of r. We explicitly model discrete functions as **discrete**(f,e). The expression denotes the discrete value generated by applying function f to a real value e. Type casts and the `sign`(x) method are examples of such discrete functions.

Operational Semantics. The semantics is presented in Fig. 4. The expression rules have the form of $\boxed{\sigma : e \xrightarrow{e} \theta, e'}$. Given the store σ, an expression e evaluates to a hash value θ and a new expression e'. The variable expression x, sample expression **sample**(r), and the binary operation v_1 **binop** v_2 are not discrete factors so that their evaluation generates a void hash value, denoted as $\perp$. In contrast, the hash for a relation expression $v \bowtie^{\ell} 0$ is a unique integer representing the label of the expression ℓ and the branch outcome. Intuitively, if these relational operations are used in conditional statements or loops, the hash values capture the execution control flow. The hash for a discrete function is the generated discrete value.

Statement rules are standard. The global rules are of the form $\boxed{\sigma, \theta,\ s\ \rightarrow\ \sigma', \theta', s'}$, in which σ is the store and θ the global hash. Rule [H-EXPR] specifies an evaluation step regarding expression e. It aggregates the hash value θ_e generated by the expression evaluation to the global hash θ. Operator $\triangleleft$ denotes the hash operation.

$$E\ ::=\ E;s \mid [\cdot]_s \mid x := [\cdot]_e \mid \textbf{if}\ [\cdot]_e\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \mid [\cdot]_e\ \textbf{binop}\ e \mid v\textbf{binop}\ [\cdot]_e \mid \textbf{discrete}(f, [\cdot]_e) \mid [\cdot]_e \bowtie 0$$

DEFINITION:

Store σ : *Var* $\rightarrow$ *Value*   *Hash* $\theta \in Z$

getSample(ℓ, r) : samples a value at ℓ that is within the error bound of r

EXPRESSION RULES $\boxed{\sigma : e \xrightarrow{e} \theta, e'}$

$$\sigma : x \xrightarrow{e} \perp, \sigma(x) \qquad \sigma:\ \textbf{sample}(r)^{\ell} \xrightarrow{e} \perp,\ \textit{getSample}(\ell, r)$$
$$\sigma:\ v \bowtie^{\ell} 0 \xrightarrow{e} \ell_T,\ \textbf{T}\quad \textbf{if}\ v \bowtie 0 \qquad \sigma:\ v \bowtie^{\ell} 0 \xrightarrow{e} \ell_F,\ \textbf{F}\quad \textbf{if}\ \neg(v \bowtie 0)$$
$$\sigma : v_1\ \textbf{binop}\ v_2 \quad \xrightarrow{e} \quad \perp, v_3 \quad \text{where } v_3 = v_1\ \textbf{binop}\ v_2$$
$$\sigma:\ \textbf{discrete}(f, r) \quad \xrightarrow{e} \quad n,\ n \quad \text{where } n = f(r)$$

STATEMENT RULES $\boxed{\sigma:\ s \xrightarrow{s} \sigma', s'}$

$$\sigma:\ x :=^{\ell} v \xrightarrow{s} \sigma[x \mapsto v],\ \textbf{skip} \qquad \sigma:\ \textbf{skip};s \xrightarrow{s} \sigma,\ s$$
$$\sigma:\ \textbf{if T then}\ s_1\ \textbf{else}\ s_2 \xrightarrow{s} \sigma,\ s_1 \qquad \sigma:\ \textbf{if F then}\ s_1\ \textbf{else}\ s_2 \xrightarrow{s} \sigma,\ s_2$$
$$\sigma:\ \textbf{while}\ e\ \textbf{do}\ s \xrightarrow{s} \sigma,\ \textbf{if}\ e\ \textbf{then}\ s; \textbf{while}\ e\ \textbf{do}\ s\ \textbf{else skip}$$

GLOBAL RULES $\boxed{\sigma, \theta,\ s\ \rightarrow\ \sigma', \theta', s'}$

$$\frac{\sigma : e \xrightarrow{e} \theta_e, e'}{\sigma, \theta,\ E[e]_e\ \rightarrow\ \sigma, \theta \triangleleft \theta_e, E[e']_e}\ \text{[H-EXPR]} \qquad \frac{\sigma : s \xrightarrow{s} \sigma', s'}{\sigma, \theta,\ E[s]_s\ \rightarrow\ \sigma', \theta, E[s']_s}\ \text{[H-STMT]}$$

Figure 4. Hashing Semantics

In our implementation, we use addition as the hash operation. For the void hash $\perp$, we have $\theta \triangleleft \perp = \theta$.

Rule [H-STMT] specifies one step in evaluating a statement.

Sampling Algorithm. The overall process requires multiple sample executions. It is described in Algorithm 1. It takes two samples as input and generates a sequence of samples, including the two inputs. Ideally, the samples sufficiently exposes discontinuity.

Function **sampleDriver**(χ_1, χ_2) presents the overall process. It first executes the two input samples to produce two hash values. It then calls function *sampleInside()*. The function returns the needed samples inside the range (χ_1, χ_2), excluding the two samples themselves. The final output is the resulting sequence from *sampleInside()* prepended with χ_1 and appended with χ_2.

In function **sampleInside**(), the algorithm tests if the two provided hashes are the same, or even they are different, if the distance of the two provided samples is less than a pre-defined threshold ε (line 4). If so, no more samples are needed. Otherwise, it computes another sample representing the mid value of the range (line 7), and then recursively calls *sampleInside*() for the two subregions. The resulting subsequences are concatenated with the mid-sample (lines 9 and 10).

An example can be found in Fig. 2 (c). Given the initial samples a and b. It produces the sequence $a \cdot e \cdot f \cdot d \cdot c \cdot b$.

Safety. Observe in Algorithm 1, no more samples are taken within a range (χ_1, χ_2) if these two samples produce the same hash value. It may not be always safe to do so, meaning there may still be

Algorithm 1 Sampling Driver.

Input: a pair of sample points χ_1 and χ_2.
Output: a sequence of sample points in $[\chi_1, \chi_2]$

```
sampleDriver ($\chi_1$,$\chi_2$)
 1: $\theta_1 := P(\chi_1)$
 2: $\theta_2 := P(\chi_2)$
 3: return $\chi_1 \cdot sampleInside(\chi_1, \theta_1, \chi_2, \theta_2) \cdot \chi_2$
```

Input: the two samples and their hashes;
Output: a sequence of samples points in (χ_1, χ_2);
Definition: ε denotes the termination threshold;

```
sampleInside ($\chi_1$, $\theta_1$, $\chi_2$, $\theta_2$)
 4: if $\theta_1 = \theta_2 \vee |\chi_1 - \chi_2| < \varepsilon$
 5:     return nil
 6: else {
 7:     $\chi_m := (\chi_1 + \chi_2)/2$
 8:     $\theta_m := P(\chi_m)$
 9:     return $sampleInside(\chi_1, \theta_1, \chi_m, \theta_m) \cdot \chi_m \cdot$
10:         $sampleInside(\chi_m, \theta_m, \chi_2, \theta_2)$
11: }
```

discontinuity within the range. Also, depending on the design of the hash operation, two different sample executions may coincidently have the same hash value. As the number of control flow paths is infinite in theory, a perfect hashing scheme is impossible in general. However, since the algorithm each time inspects only a pair of samples in the whole error bound of the uncertain input, it is highly unlikely that a hash conflict happens right at the pair. In our experience, the simple hashing scheme did not cause any conflicts. Hence, hash safety is not our focus.

Our safety claim is defined as follows.

Theorem 1 (Safety). *Given two input samples χ_1 and χ_2, assume they lead to the same hash value. For a discrete factor that generates a discrete value from a real value, let the real value be denoted as a function $f(x)$ over the uncertain input x, called the* **discrete factor function**. *If for each discrete factor during the execution, its function $f(x)$ is monotonic within the range $[\chi_1, \chi_2]$, the output function must be continuous in $[\chi_1, \chi_2]$ and it is safe to skip sampling in between.*

Intuitively, if a discrete factor function is monotonic, and the same discrete value is generated in the two sample runs (i.e. the lower and upper bounds), the same discrete value must be generated for any other samples in between [2]. For example, if the value of x in the predicate of a conditional statement "**if** $x \bowtie 0$..." is monotonic in the range $[\chi_1, \chi_2]$ and it produces the true value in both sample runs, it must consistently produce the same true value for any samples in between. The aggregated effect of all such monotonic predicates is that the same control flow must be taken for all samples inside the range; similarly, all the discrete co-efficients in the output function must also have the same value, ensuring the same mathematical form of the output function and thus continuity within the range. Here, we do not consider mathematical discontinuity, because as mentioned earlier, such discontinuity needs to be explicitly captured by control flow discontinuity otherwise the program is buggy.

Note that we require monotonicity for functions at internal discrete factors during an execution, not the output function. The internal functions are much more regular than the output function because they reflect the programmer's intension to control the program execution and such control usually occurs at a coarse level due to the limited reasoning capability of human developers.

[2] We assume the discrete operations themselves are always monotonic, which is true in practice.

DEFINITION:
VecStore $\mathcal{V}$: $Var \to Value_1 \times ... \times Value_c$
Expr e ::= ... | $v[1,c]$ | **err**

$v[1,c]$ denoting a vector $(v_1, ..., v_c)$; c a constant; $v[i]$ the ith value in the vector; **err** represents a validation error.

$$monoChk(v[1,c]) \;=\; \forall 1 \le i < c\; v[i] - v[i+1] \ge 0 \;\vee \\ \forall 1 \le i < c\; v[i] - v[i+1] \le 0$$

$$getVector(\ell, r) \;=\; (r_1, ..., r_c)$$

r_1, ..., r_c represent the c samples at ℓ to replace r; these samples are within the error bound of r and produce the same hash.

EXPRESSION RULES $\boxed{\sigma, \mathcal{V} : e \overset{e}{\to} e'}$

$$\sigma, \mathcal{V} : x \overset{e}{\to} \sigma(x) \;\; \textbf{if } \mathcal{V}(x) = \perp \qquad \sigma, \mathcal{V} : x \overset{e}{\to} \mathcal{V}(x) \;\; \textbf{if } \mathcal{V}(x) \ne \perp$$
$$\sigma, \mathcal{V} :\; v[1,c] \bowtie 0 \;\overset{e}{\to}\; \mathbf{T} \;\; \textbf{if } monoChk(v[1,c]) \wedge \forall 1 \le i \le c\; v[i] \bowtie 0$$
$$\sigma, \mathcal{V} :\; v[1,c] \bowtie 0 \;\overset{e}{\to}\; \mathbf{F} \;\; \textbf{if } monoChk(v[1,c]) \wedge \forall 1 \le i \le c\; \neg v[i] \bowtie 0$$
$$\sigma, \mathcal{V} :\; v[1,c] \bowtie 0 \;\overset{e}{\to}\; \mathbf{err} \;\; \textbf{if } \neg monoChk(v[1,c]) \vee \exists 1 \le i, j \le c \\ v[i] \bowtie 0 \wedge \neg v[j] \bowtie 0$$
$$\sigma, \mathcal{V} :\; v \bowtie 0 \overset{e}{\to} \mathbf{T} \;\; \textbf{if } v \bowtie 0 \qquad \sigma, \mathcal{V} :\; v \bowtie 0 \overset{e}{\to} \mathbf{F} \;\; \textbf{if } \neg v \bowtie 0$$
$$\sigma, \mathcal{V} :\; \textbf{sample}(r)^{\ell} \;\overset{e}{\to}\; r[1,c] \;\; \text{where } r[1,c] = getVector(\ell, r)$$
$$\sigma, \mathcal{V} : v_1[1,c] \textbf{ binop } v_2[1,c] \;\overset{e}{\to}\; v_3[1,c] \;\; \text{where } \forall 1 \le i \le c \\ v_3[i] = v_1[i] \textbf{ binop } v_2[i]$$
$$\sigma, \mathcal{V} : v_1[1,c] \textbf{ binop } v_2 \;\overset{e}{\to}\; v_3[1,c] \;\; \text{where } \forall 1 \le i \le c \\ v_3[i] = v_1[i] \textbf{ binop } v_2$$
$$\sigma, \mathcal{V} : v_1 \textbf{ binop } v_2 \;\overset{e}{\to}\; v_3 \;\; \text{where } v_3 = v_1 \textbf{ binop } v_2$$
$$\sigma, \mathcal{V} :\; \textbf{discrete}(f, r[1,c]) \;\overset{e}{\to}\; f(r[1]) \;\; \textbf{if } monoChk(r[1,c]) \wedge \\ f(r[1]) = ... = f(r[j])$$
$$\sigma, \mathcal{V} :\; \textbf{discrete}(f, r[1,c]) \;\overset{e}{\to}\; \mathbf{err} \;\; \textbf{if } \neg monoChk(r[1,c]) \vee \\ \exists 1 \le i, j \le c\; f(r[i]) \ne f(r[j])$$
$$\sigma, \mathcal{V} :\; \textbf{discrete}(f, r) \;\overset{e}{\to}\; f(r)$$

STATEMENT RULES $\boxed{\sigma, \mathcal{V} :\; s \overset{s}{\to} \sigma', \mathcal{V}', s'}$

$$\sigma, \mathcal{V} :\; x :=^{\ell} v \;\overset{s}{\to}\; \sigma[x \mapsto v], \mathcal{V}, \textbf{ skip}$$
$$\sigma, \mathcal{V} :\; x :=^{\ell} v[1,c] \;\overset{s}{\to}\; \sigma, \mathcal{V}[x \mapsto v[1,c]], \textbf{ skip}$$
$$\sigma, \mathcal{V} :\; \textbf{if T then } s_1 \textbf{ else } s_2 \;\overset{s}{\to}\; \sigma,\; \mathcal{V},\; s_1$$

GLOBAL RULES $\boxed{\sigma, \mathcal{V}, s \to \sigma', \mathcal{V}', s'}$

$$\frac{\sigma, \mathcal{V} : e \overset{e}{\to} e' \quad e' \ne \mathbf{err}}{\sigma, \mathcal{V}, E[e]_e \to \sigma, \mathcal{V}, E[e']_e} \qquad \frac{\sigma, \mathcal{V} : e \overset{e}{\to} \mathbf{err}}{\sigma, \mathcal{V}, E[e]_e \to \sigma, \mathcal{V}, \mathbf{exit}}$$
$$\frac{\sigma, \mathcal{V} : s \overset{s}{\to} \sigma', \mathcal{V}', s'}{\sigma, \mathcal{V}, E[s]_s \to \sigma', \mathcal{V}', E[s']_s}$$

Figure 5. Validation Semantics

5. Validation

The safety theorem 1 states that as long as all discrete factor functions change monotonically in the range delimited by two samples, and the hash values are identical, it is safe to avoid further sampling in between. Directly determining monotonicity of discrete factor functions based on their mathematical form is infeasible in general, due to the complexity of these functions. Our experience with SPEC CFP 2000 programs is that most of them have high-order functions, involving complex operations such as the square root and sin/cos.

We develop a sample-based validation approach. The idea is to monitor the values of discrete factors for a number of sample runs, validating that the values change in a monotonic way. In order to minimize the number of samples needed, we leverage the samples derived by Algorithm 1. In particular, from the sample sequence generated by the algorithm, we identify all the consecutive subsequences that have the same hash value and validate monotonicity of the discrete factor functions for all the samples in each subsequence. Observe that the intervals between samples in a subsequence are irregular according to the algorithm, allowing us better observe the monotonicity property.

For the example in Fig. 2, after the basic sampling phase, the generated sample sequence is $a \cdot e \cdot f \cdot d \cdot c \cdot b$. The subsequences are hence $a \cdot e$ and $f \cdot d \cdot c \cdot b$. Observe that the samples in the second subsequence have irregular intervals.

Monotonicity across runs is different from monotonicity inside a single run. It requires comparing values in different executions, which is in general challenging because two runs may take different control flow paths, leading to different numbers of occurrences of a variable. It is difficult to determine the alignment of these occurrences across runs. Fortunately, we only need to handle sample runs that have the same hash and thus very likely the same execution path.

We develop a vector semantics to validate monotonicity. Particularly, a value that is related to the sample input is expanded to a vector of values. The size of the vector is exactly the size of the sequence on which we want to validate monotonicity, denoted as c. The ith element of the vector corresponds to the value in the ith sample run. Upon an operation on a variable with a vector value, the operation is performed on each element in the vector. With vectors, it becomes convenient to validate monotonicity for a discrete factor. We only need to ensure the values in its vector are monotonic. For values that are not relevant to the sample value, such as loop indices, we use a single value instead of a vector to avoid redundant operations.

Validation Semantics. The semantics is presented in Fig. 5. We define a vector store $\mathcal{V}$ that maps a variable to a vector with size c. Constant c denotes the number of samples on which we want to validate monotonicity. Besides the original definitions, an expression is extended to include a value vector (of size c) and a special value **err** representing validation failure.

Expression rules are of the form $\boxed{\sigma, \mathcal{V} : e \xrightarrow{e} e'}$. Given the regular store and the vector store $\mathcal{V}$, an expression evaluates to a new expression. A variable expression x evaluates to a vector value if it is associated with a vector, meaning its value is directly/indirectly computed from the sample input. Otherwise, it evaluates to a singleton value. For a relational operation on a vector value, a dynamic check is performed to ensure monotonicity of the values. It also ensures that all vector values lead to the same boolean outcome. This is to validate that the control flow does not change for all sample runs under the validation. Recall that although we assume the perfect hashing in our safety discussion, hash conflicts may occur in practice due to the infinite number of possible control flow paths. The validation allows us to detect any possible hash conflicts (i.e. sample runs have the same hash value but different control flow paths). For sample expression **sample**(r), a vector of input samples is acquired from the driver.

For binary operations, if at least one of the operands is a vector, the result is a vector, otherwise a singleton value indicating that it is irrelevant to the sample input.

For a discrete function application on a vector value, we validate the monotonicity of the vector and the equality of all the resulting discrete values.

Statement rules are in the form of $\boxed{\sigma, \mathcal{V} : \ s \xrightarrow{s} \sigma', \mathcal{V}', s'}$. For an assignment statement, if the *rhs* expression is a vector, it is stored to the vector store, otherwise the regular store. For conditional statements, we do not have rules that handle vectorized predicate outcomes as the dynamic check for a relational expression ensures the same predicate output for all sample runs.

DEFINITION:
HashStore $\Gamma : Var \rightarrow Hash$ *CDStack* $\mathcal{S} ::= \overline{\theta}$
Expr $e ::= \ldots \mid \langle v, \theta \rangle$

EXPRESSION RULES $\boxed{\sigma, \Gamma : e \xrightarrow{e} e'}$

$$\sigma, \Gamma : x \xrightarrow{e} \sigma(x) \quad \textbf{if } \Gamma(x) = \bot \qquad \sigma, \Gamma : x \xrightarrow{e} \langle \sigma(x), \Gamma(x) \rangle \quad \textbf{if } \Gamma(x) \neq \bot$$
$$\sigma, \Gamma : \ \textbf{sample}(r)^{\ell} \xrightarrow{e} \langle getSample(\ell, r), \ell \rangle$$
$$\sigma, \Gamma : \ \langle v, \theta \rangle \bowtie^{\ell} 0 \xrightarrow{e} \langle \mathbf{T}, \theta \triangleleft \ell_T \rangle \quad \textbf{if } v \bowtie 0$$
$$\sigma, \Gamma : \ v \bowtie^{\ell} 0 \xrightarrow{e} \mathbf{T} \quad \textbf{if } v \bowtie 0$$
$$\sigma, \Gamma : \ \langle v_1, \theta_1 \rangle \ \textbf{binop} \ \langle v_2, \theta_2 \rangle \xrightarrow{e} \langle v_3, \theta_1 \triangleleft \theta_2 \rangle \quad \text{where } v_3 = v_1 \ \textbf{binop} \ v_2$$
$$\sigma, \Gamma : \ \langle v_1, \theta_1 \rangle \ \textbf{binop} \ v_2 \xrightarrow{e} \langle v_3, \theta_1 \rangle \quad \text{where } v_3 = v_1 \ \textbf{binop} \ v_2$$
$$\sigma, \Gamma : \ v_1 \ \textbf{binop} \ v_2 \xrightarrow{e} v_3 \quad \text{where } v_3 = v_1 \ \textbf{binop} \ v_2$$
$$\sigma, \Gamma : \ \textbf{discrete}(f, \langle r, \theta \rangle) \xrightarrow{e} \langle f(r), \ \theta \triangleleft f(r) \rangle$$
$$\sigma, \Gamma : \ \textbf{discrete}(f, r) \xrightarrow{e} f(r), \ \bot$$

STATEMENT RULES $\boxed{\sigma, \Gamma, \mathcal{S} : \ s \xrightarrow{s} \sigma', \Gamma', \mathcal{S}', s'}$

$$\sigma, \Gamma, \mathcal{S} : \ x :=^{\ell} \langle v, \theta \rangle \xrightarrow{s} \sigma[x \mapsto v], \ \Gamma[x \mapsto \theta \triangleleft last(\mathcal{S})], \mathcal{S}, \ \textbf{skip}$$
$$\sigma, \Gamma, \mathcal{S} : \ x :=^{\ell} v \xrightarrow{s} \sigma[x \mapsto v], \ \Gamma[x \mapsto last(\mathcal{S})], \mathcal{S}, \ \textbf{skip} \quad \textbf{if } last(\mathcal{S}) \neq \bot$$
$$\sigma, \Gamma, \mathcal{S} : \ x :=^{\ell} v \xrightarrow{s} \sigma[x \mapsto v], \ \Gamma, \mathcal{S}, \ \textbf{skip} \quad \textbf{if } last(\mathcal{S}) = \bot$$
$$\sigma, \Gamma, \mathcal{S} : \ \textbf{if } \langle \mathbf{T}, \theta \rangle \ \textbf{then } s_1 \ \textbf{else } s_2 \xrightarrow{s} \sigma, \ \Gamma, \mathcal{S} \cdot (last(\mathcal{S}) \triangleleft \theta), \ s_1; \textbf{endif}$$
$$\sigma, \Gamma, \mathcal{S} : \ \textbf{if T then } s_1 \ \textbf{else } s_2 \xrightarrow{s} \sigma, \ \Gamma, \mathcal{S}, \ s_1$$
$$\sigma, \Gamma, \mathcal{S} \cdot \theta_t : \ \textbf{endif} \xrightarrow{s} \sigma, \ \Gamma, \mathcal{S}, \ \textbf{skip}$$

GLOBAL RULES $\boxed{\sigma, \Gamma, \mathcal{S}, s \xrightarrow{s} \sigma', \Gamma', \mathcal{S}', s'}$

$$\frac{\sigma, \Gamma : e \xrightarrow{e} e'}{\sigma, \Gamma, \mathcal{S}, E[e]_e \rightarrow \sigma, \Gamma, \mathcal{S}, E[e']_e} \ \text{[S-EXPR]} \qquad \frac{\sigma, \Gamma, \mathcal{S} : s \xrightarrow{s} \sigma', \Gamma', \mathcal{S}', s'}{\sigma, \Gamma, \mathcal{S}, E[s]_s \rightarrow \sigma', \Gamma', \mathcal{S}', E[s']_s} \ \text{[S-STMT]}$$

Figure 6. Slice Hashing Semantics

In the global rules, if an expression evaluates to **err**, the evaluation terminates. The non-monotonic ranges are further divided.

Fig. 2 (b) presents an example for the vector semantics.

Our implementation supports vectorization of most language features in C and Fortran, including arrays, pointers, and library calls. Maintaining singleton values for variables that are not relevant to the uncertain input is key to efficiency.

Observe that the monotonicity check is sample-based, which may not be precise enough. It is unfortunately impractical to analyze the mathematical functions to precisely determine monotonicity. For better precision, one can choose to validate monotonicity on more samples as the vector semantics and the implementation are general. Our experience shows that our current strategy of using samples generated in the basic hashing phase is sufficient.

6. Hashing Slices

In the basic Algorithm 1, a global hash value is computed for an execution. However, this is often too restrict in practice. In many cases, even though two executions have different hash values, e.g. they follow different execution paths, the difference may not be relevant to the output.

In this section, we discuss a more sophisticated hashing algorithm that hashes only the discrete factors relevant to the output. It maintains a hash value for each variable on the fly, denoting the set of discrete factors that have been directly/indirectly used to compute the current value of the variable (i.e. discrete factors in its dynamic slice [1]). When we determine whether a mid-sample is needed, we compare the hash values associated with the corresponding output variables.

The semantic rules are presented in Fig. 6. We introduce a hash store Γ that maps a variable to its hash value. A stack $\mathcal{S}$ is used to propagate hash values through control dependences. Each stack

trace	val	hash	$\mathcal{S}$
1. x=sample(1.5)	2.2	1	$\perp$
2. y=(int) x	2	$1 \lhd 2$	$\perp$
3. **if** x-1.0<0	F	$3_F \lhd 1$	$\perp \cdot (3_F \lhd 1)$
6. **if** h(x)-0.3>0	T	$6_T \lhd 1 \lhd (3_F \lhd 1)$	$\perp \cdot (3_F \lhd 1) \cdot (6_T \lhd 1 \lhd 3_F \lhd 1)$
7. o=0.3	0.3	$6_T \lhd 1 \lhd 3_F \lhd 1$	$\perp \cdot (3_F \lhd 1) \cdot (6_T \lhd 1 \lhd 3_F \lhd 1)$
endif			$\perp \cdot (3_F \lhd 1)$
endif			$\perp$
10. z=1+y	3	$1 \lhd 2$	$\perp$

Table 1. Evaluation of the program in Fig. 2(a) with sample 2.20

entry is the hash value of a predicate. We extend the syntax of expression in Fig. 3 to include a pair consisting of a value and its hash, which is the hash aggregation of all discrete factor values in the slice of the value. This special type of expressions is not part of the source code. They only occur during evaluation.

The expression rules have the form of $\boxed{\sigma, \Gamma : e \xrightarrow{e} e'}$. in which Γ is the hash store. For a variable expression x, if its hash value is void, it evaluates to a regular value; if not, it evaluates to its value and the corresponding hash.

A sampling expression evaluates to a sample value acquired from outside and its label ℓ as the hash. Note that it is the only place that initiates a non-void hash. It is analogous to the introduction of a taint in the taint analysis.

For a relational operation, if the *lhs* value is a pair, meaning that it is relevant to the sample input, the evaluation result is a pair consisting of the comparison outcome and the aggregation of the *lhs* hash and the operation's hash. If the *lhs* value is a singleton, the evaluation produces the singleton comparison outcome. Note that we omit the rules for the false cases for brevity.

For a binary operation, if either value is a pair, the resulting value is also a pair, including the aggregated hash value.

For a discrete function application, if the parameter is a pair, the generated discrete value is aggregated to the hash.

The statement rules have the form of $\boxed{\sigma, \Gamma, \mathcal{S} :\ s \xrightarrow{s} \sigma', \Gamma', \mathcal{S}', s'}$, in which $\mathcal{S}$ is the stack to allow hash computation through control dependence. For an assignment statement, if the *rhs* value is a pair, the evaluation updates both the store and the hash store. The hash of the *lhs* variable is the aggregation of the *rhs* expression hash θ and the hash of its control dependence, which is the last entry in $\mathcal{S}$. If the *rhs* value is a singleton and its control dependence hash is not void, the hash stored is updated with the control dependence hash. It means that although the assigned value is not computed from the sample input, the execution of the assignment is guarded by a predicate relevant to the sample input.

For a conditional statement, if the evaluation of the relational operation yields a pair, the stack is appended with the aggregation of the predicate's control dependence hash, i.e. the last entry of $\mathcal{S}$, and the relational expression hash θ. Since the aggregated hash becomes the new last entry in $\mathcal{S}$, future evaluations occur inside the branch will use it as their control dependence hash. The evaluation also appends a special statement **endif** to the end of the branch. Evaluation of an **endif** statement leads to the removal of the last entry in $\mathcal{S}$, meaning evaluations beyond the end of a branch are no longer control dependent on the predicate.Note that the LIFO nature of the stack captures the nesting effect of the control dependence.

If the evaluation of the relational operation yields a singleton, there is no need to append an new entry to the stack or the **endif** statement to the end of the branch, denoting the irrelevance of the predicate. Note that any statements evaluated inside the branch nonetheless have their control dependence hashes inherited from the current last entry of $\mathcal{S}$.

```
1  x := sample(3.0);
2  y := 0.0;
   /*f(2) = f(4) = 1, f(3) = −1*/
3  if f(x) < 0
4      y:=y + 1.0;
5  o:=x − y;
              (I)
```

```
1  x := sample(0.5);
2  A[5] := ...;
3  if x >= 0.5
4      i:=(int) x × 10;
5      A[i] :=...;
6  o:=A[5];
              (II)
```

Figure 7. Examples for safety issues.

Example. Table 1 presents an example evaluation of the program in Fig. 2(a) with the sample 2.20. Observe that at line 1, the hash is 1, the label of the statement. At line 2, the hash is the aggregation of x's hash and the generated discrete value 2. At line 3, the hash is the aggregation of x's hash and the branch outcome 3_F; it is also appended to $\mathcal{S}$. The entry is removed from $\mathcal{S}$ at the second **endif**. At line 6, the hash is the aggregation of x's hash, the branch outcome 6_T, and the control dependence hash. The others are similarly computed.

Safety. We have the following safety claim.

Theorem 2 (Safety-Slice)**.** *Given two input samples χ_1 and χ_2, assume the slice hashes of the corresponding output variables are identical in the two runs. For a discrete factor that generates a discrete value from a real value, let the real value be denoted as a function $f(x)$ over the uncertain input x. If*
(1) each $f(x)$ is monotonic within the range $[\chi_1, \chi_2]$;
(2) all memory addresses remain unchanged within the range, then the output function must be continuous in [χ_1, χ_2].

Condition (1) requires all discrete factors in the execution, not only the slice, to be monotonic. This is to tolerate implicit dependences [20]. Statements that are related to the output through implicit dependences cannot be captured by the regular data flow and control flow tracking as specified in Fig. 6. The reason is that such dependence is caused by that a statement is not executed (and thus no way to track). Consider the example in Fig. 7 (I), function f is non-monotonic. In the two initial sample runs with $x = 2.0$ and $x = 4.0$, the false branch of line 3 is taken. As a result, statement 4 is not executed. The dynamic slice of o at 5 contains lines 1 and 2 in both runs. However, there is an implicit dependence between line 5 and line 3 because the branch outcome of 3 affects the value of o when $x = 3.0$, causing discontinuity. Condition (1) excludes such cases by requiring $f(x)$ to be monotonic, even though it is not in the slice. Note that condition (1) allows discrete factors to have different values as long as they are monotonic.

Condition (2) is to preclude array indices or pointers differences (note that our discussion here goes beyond the language in Fig. 3). They could cause output discontinuity. Consider the example in Fig. 7 (II). Assume the two initial samples to be $x = 0.4$ and $x = 0.6$. The slice of o at 6 contains only line 2 in both runs. The two discrete factors: the predicate at line 3 and the cast at line 4 are both monotonic. However, the output function is not continuous. As its value is defined at line 5 when $x = 0.5$. Condition (2) excludes such cases by ensuring that memory addresses do not change with different samples.

The proof is omitted. Intuitively, since all predicates are monotonic and all memory addresses do not change, there cannot be any new dependences in the output slice for any sample in between.

The validation process presented in Section 5 is extended to validate not only monotonicity, but also that all values in the vector of an address variable to be the same.

7. Identifying Continuous Cores

A basic assumption of our technique is that if two sample runs produce different hash values, there must be discontinuity between the two samples. However, it may not be the case in practice. Developers can write programs in such a way that the output function is

```
{y= $f(c)$}
1 for...
2   if $x < c$
3       o:=$f(x)$;
4   else
5       o:=y;
                (I)
```

```
1 $o = A[0]$
2 for $i := 1$ to $c$
3   if $o < A[i]$
4       $o$:=$A[i]$;
                (II)
```

Figure 8. Real continuous core examples from `178.galgel`. Variable o denotes the output; c denotes a compiler time constant; $f(x)$ is a continuous function. The first line in (I) represents the precondition.

continuous even though the control flow varies. In this section, we discuss how to detect program regions that have such characteristics and prove that they are continuous despite control flow differences. Then the hashing algorithms can avoid collecting predicate hashes inside these regions.

Consider the example in Fig. 8 (I). It is a coding pattern used a few times in `178.galgel`. The output function is $f(x)$ when $x < c$. It becomes $f(c)$ when $x = c$ and remains that value for $x > c$. The developer hoists the computation of $f(c)$ from the else branch to outside of the loop for better performance. Observe the output function is continuous. However, if the initial two sample runs are for $x = c - 1$ and $x = c + 1$, they have different control flow.

We call such code regions *continuous cores*, which are formally defined as follows.

Definition 3. *A continuous core is a conditional statement s (including its branches) that is modeled as $o = s(I)$ with input I the set of variables used in s and defined outside, and output o the variable computed by s and used later by other statements, and $s(I)$ is a continuous function in the domain of I, despite control flow variations.*

The definition also covers loop statements, which are a special case of conditional statement.

We develop a profiling technique to detect candidates of continuous cores. The profiling semantics is an extension of the validation semantics in Fig. 5. Details are omitted due to the space limitation.

Given a continuous core candidate, we try to prove the continuity in the presence of control flow variation. The technique in [6] tries to prove statically that a program is continuous regarding a given set of variables. We adapt the technique to handle the conditional statements identified by our profiler. The key idea is to first prove the two branches of the conditional statement to be continuous, and then ensure that the two continuous functions have identical output value at the boundary input value at which the branch outcome changes from true to false or vice versa. Intuitively, it means that the two branch functions yield outputs infinitely close to the same value as the input gets infinitely close to the boundary value.

Consider the example in Fig. 8 (I). The statements in the true and false branches are both continuous on their own. And observe that at the bounary point $x = c$, both branches yield the same output value, ensuring continuity.

Other Patterns. There are a few other coding patterns that give rise to continuous cores. Fig. 8 (II) shows another very common core in `178.galgel`. It returns the maximum value of an array. Observe that the program is continuous, i.e. the output changes continuously with the uncertain input. For example, assume an array $A[0-2] = \{1.0, 2.0, 3.0\}$, and $A[1]$ is uncertain and it varies within range $[2.0, 4.0]$. When $A[1]$ changes from 2.0 to 4.0. The output has $o_1(A[1]) = 3.0$ when $A[1]$ changes from 2.0 to 3.0 and then $o_2(A[1]) = A[1]$ when $A[1]$ changes from 3.0 to 4.0. Observe that o_1 and o_2 have the same value at the boundary $A[1] = 3.0$, hence they together denote a continous function.

program	# of pred.	pred. order	non-poly funcs in pred.	# of segments
168.wupwise	8E+8	-2 ∼ 2	cos, log, sqrt	1
171.swim	8E+6	1334		1
172.mgrid	4E+8	-1 ∼ 2	abs, sqrt	1
173.applu	2E+7	-1 ∼ 17	abs, sqrt	1
178.galgel	2E+8	2	abs, sqrt	100+
183.equake	5E+7	-1 ∼ 3	sin, cos, sqrt	6
187.facerec	1E+8	2	abs, sin, sqrt	5
188.ammp	7E+8	2	sin, cos, sqrt	1
191.fma3d	3E+4	-3 ∼ 6	sin, abs, sqrt	1
200.sixtrack	6E+8	-1 ∼ 2	sin, abs, sqrt	1
301.apsi	3E+8	-1 ∼ 14	sin, abs, sqrt	2
deisotope	6E+4	2		4

Table 2. Program characteristics.

To prove the pattern is continuous. We completely unroll the loop. Each unrolled iteration has the following form, with o_i the output defined at the ith iteration.

```
3   if $o_{i-1} < A[i]$
4       $o_i$:=$A[i]$;
    else
        $o_i$:=$o_{i-1}$;
```

Observe that the functions from both branches are continuous by themselves, and they have the same value $o_i = A[i]$ at the boundary $o_{i-1} = A[i]$. We have encountered a few more core patterns. They can be proved similarly.

8. Empirical Evaluation

Our system consists of several components. A modified compiler, to instrument programs to compute the hash values, is built on top of `gcc`. The sampling driver is written in Python. The validation algorithm is also implemented through `gcc`. Our system supports both C/C++ and Fortran.

Our experiments are performed on an Intel i7 2.70GHz machine with 4GB RAM installed. We use SPEC CFP 2000 and one biochemical data processing program (`deisotope`) as the benchmark set. Three programs from SPEC CFP 2000 are excluded. `189.lucas` is a program that identifies prime numbers and hence uncertainty analysis is not applicable. `177.mesa` and `179.art` are excluded as they take discrete inputs. We have totally 12 programs (3 C and 9 Fortran). We randomly select the uncertain inputs. For each uncertain input, we assume its error bound to be [0.5, 1.5]. The termination threshold (for the driver) is 0.001.

Table. 2 shows the basic characteristics of the programs. It includes the number of dynamic predicates, the order of the predicates, the non-polynomial functions involved in the predicate functions and the number of (continuous) segments in the output curve. We acquire the first three pieces of information through profiling. The number of continuous segments is acquired by taking samples at the interval of [0.5, 1.5].

As we can see from the table, a number of programs have high order predicates. Program `171.swim` has an order of over one thousand, due to its iterative computation process. Note that, our profiler works by computing the order of the *lhs* value from the orders of the *rhs* values. Hence, it has to approximate in some situations. Given $h(x) = (1/f(x))(g(x))$ in which $f(x)$ is a function of order 1 and $g(x)$ of order 2, we conservatively assume the result $h(x)$ will have the order of 1. Moreover, most of the programs have non-polynomial functions involved, such as trigonometric functions or square root. This supports our early discussion about the difficulty of any symbolic techniques to analyze the path conditions.

Also observe that 7 out of the 12 benchmarks are completely continuous. However, it does not mean our technique is not useful for them. With black-box MC, no matter how many output samples

program	native	basic algorithm			slice-based algorithm			validation			
		time	overhead	# of samples	time	overhead	# of samples	4-sample	per sample	10-sample	per sample
168.wupwise	2.05	2.14	4%	2	6.56	220%	2	18.81	4.70	30.79	3.08
171.swim	0.15	0.15	1%	2	0.36	142%	2	1.03	0.26	1.07	0.11
172.mgrid	3.31	3.32	0%	2	11.13	236%	2	20.2	5.05	49.5	4.95
173.applu	0.06	0.07	6%	2	0.23	271%	2	0.30	0.08	0.74	0.07
178.galgel	0.69	0.70	10%	416	3.34	384%	416	1304.0	1.63	1353.0	0.68
183.equake	0.20	0.21	6%	48	0.36	81%	48	14.28	0.30	14.28	0.12
187.facerec	1.23	1.25	2%	61	4.12	235%	40	233.4	3.89	270.0	1.80
188.ammp	2.72	2.73	0%	2	3.51	29%	2	15.30	3.83	15.86	1.59
191.fma3d	0.02	0.02	0%	2	0.06	200%	2	0.02	0.01	0.03	0.01
200.sixtrack	2.22	2.27	2%	2	12.60	468%	2	29.15	7.29	29.68	2.97
301.apsi	1.75	1.77	1%	133	9.02	415%	12	22.08	0.92	30.80	0.39
deisotope	0.02	0.02	0%	10	0.04	90%	10	0.32	0.01	0.42	0.01
AVERAGE			2.67%			231%					

Table 3. Results of different algorithms.

are collected, one cannot be confident about the absence of discontinuity.

Table 3 shows the results of the two hashing algorithms and the validation algorithm.

Columns 3-5 present the results for the basic algorithm. Observe that it is highly efficient (an average of 2.67% overhead for each sample run). It is also very effective for most programs. For those that are continuous in the entire input error bound, the algorithm was able to identify that the hashes of the initial two sample runs are identical, it then stops collecting more samples right away.

Columns 6-8 present the results for the slice-based algorithm. Observe that it is more expensive, with an average overhead of 231%. This is due to its much more heavy-weight instrumentation. Another reason is that we haven't tried hard to optimize our implementation yet. Observe that it makes differences on two programs: `187.facerec` and `301.apsi`. For `187.facerec`, it reduces the number of samples from 61 to 40, while still precisely exposes all the discontinuous points. However, the saving does not pay off the extra overhead. In contrast, for `301.apsi`, the reduction is so large that the slice approach is clearly much better.

The reason for not observing more beneficial cases for the slice-based approach is that most of the benchmarks have very cohesive coding structure. They tend to have a very small number of outputs, and most intermediate computation directly/indirectly contributes to these outputs. We speculate for larger scale scientific programs, when more functionalities are integrated into a program, we will have a better chance to observe the benefit. We expect the user of our technique can choose from the two algorithms based on their insights of the programs. In our future work, we will further integrate these two so that when the basic algorithm collects unnecessary samples, we can automatically switch to the other approach.

The last 4 columns show the cost for validation. For many programs, our algorithm stops with two samples. Monotonicity cannot be determined by 2 samples. So we set the minimal samples for each validation run to 4 and 10, respectively. The presented data is the total time of those runs and the time equivalent for a sample execution, i.e. (total time / vector runs $\times$ samples per vector). Most validation runs pass without any violations, indicating those discrete factor functions are very likely monotonic, and thus high confidence of our sampling results. Only `178.galgel` has a few violations. The two configurations do not lead to differences in the validation results.

Although the total time in validation is substantial, the per-sample cost is quite low. For many cases, especially those with the 10-sample configuration, the cost is even lower than the native run. The reason is that we use singletons to denote values irrelevant to the uncertain input, avoiding operations on such values being iterated, causing a kind of saving over a simple aggregation of

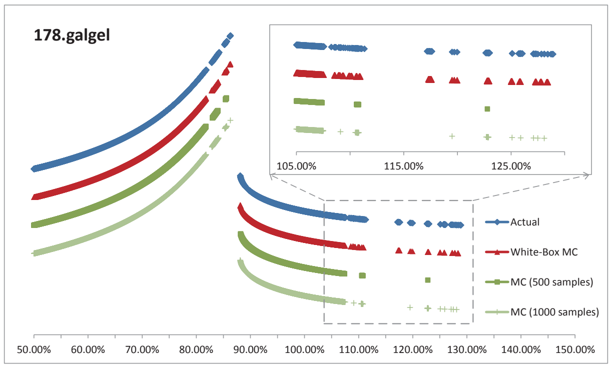


Figure 9. Curves plotted from the samples identified by different methods for program `178.galgel`.

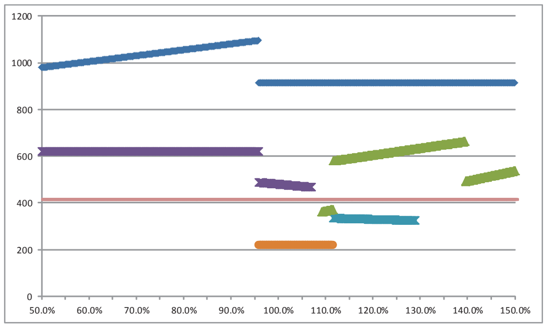
multiple sample runs. Also observe, the per-sample cost decreases as the number of samples per-vector increases. That is because the dominant fixed cost of vector manipulation gets amortized.

Case Studies. Program `178.galgel` is an interesting case, our algorithms generate over 400 samples. Fig 9 shows the output functions computed by different approaches. Lets first focus on the actual curve that is supposed to be the oracle. The curve has many small missing segments, otherwise appears continuous. Observe in the zoom-in view, at around 120%, those overlapping dots are essentially a sequence of tiny continuous segments separated by missing segments. By inspecting the code, we observe that the program is not stable for the inputs falling in missing segments. It fails to converge and produces empty result. Our algorithms are able to closely approximate the real curve. We also collect 500 and 1000 random samples in uniform distribution for comparison. From the zoom-in view, one can observe that a number of points are missing from the lower two curves. However, this is not the worst scenario, when using the traditional MC, people may tend to begin with a small number of samples. Due to the fact that the curve has very large continuous segments and the missing segments are often much smaller. It is very likely that the missing segments are completely missed, leading to the wrong conclusion. These results clearly show the benefit of white-box sampling. Observe from Table 3 that it only requires 416 samples and has only 10% overhead (basic).

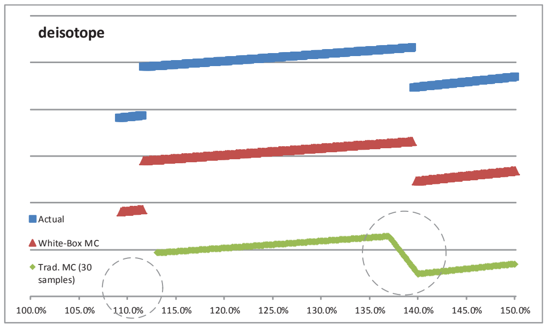
Another example comes from LC-MS (Liquid Chromatography Mass Spectrometry) process [19], which is an effective technique used in cancer biomarker discovery. A biomarker is a protein which undergoes changes in concentration in diseased samples. To detect biomarkers, proteins from cancer patients and normal people are labeled differently and digested into smaller pieces called *peptides*. After the LC-MS process, each peptide would ideally lead to two peaks, or a *doublet*. One of them corresponds to the normal peptide marked with a light label and the other corresponds to the cancer sample marked with a heavy label. The intensity ratio of the doublet indicates the relative concentration of proteins from which the peptides were generated.

`deisotope` is a program carries out the data processing in LC-MS process. It takes raw data from serum then produces the matching doublets with their intensities.

However, this program is highly sensitive to data uncertainty. A tiny variation in the input may lead to different doublets being generated. Sample outputs are shown in Fig. 10(a). The x-axis represents the variation of an input provided by the scientist according to their experience in the area (and thus uncertain) from 50% to 150% of its original value, and the y-axis shows the computed intensity of outputted peaks. We can observe that the intensity of the peaks changes substantially, leading to the potential change of the biomarker. Or it may even disappear, meaning a different set of doublets is generated.



(a) Output variation over the input changes.



(b) Different curves plotted by white-box and traditional MC methods, with the actual curve shown on the top.

Figure 10. Case study of `deisotope`.

Removing false positives caused by uncertainty is very critical since the results determine the subsequent research – typically involving significant effort and expense in wet-bench experiments. Sampling provides a reasonably low-cost method to inspect the effect of uncertainty.

Without loss of generality, we select one of the peaks in the outputted doublets for a close study. Fig. 10(b) shows the change of its intensity, by varying the uncertain input from 100% to 150% of its original value. Observe with 20 samples, our technique is able to precisely model the curve while a traditional MC with 30 samples cannot.

9. Related Work

Uncertainty analysis and MC method. Sampling-based, or Monte Carlo approaches to uncertainty and sensitivity analysis are widely used [9]. Several techniques are proposed to improve the efficiency of the MC method by parallelizing MC trials [2, 4]. In [14], an execution coalescing technique was proposed to pack multiple MC trials in a single run, using vectors. These techniques do not aim at guiding the sampling process to expose critical points using a small number of samples. Our vector semantics is also more simplistic, performing only the single task of validation, which makes it more robust.

Static analysis for program continuity and robustness. [6] shares a similar scenario of analyzing continuity for programs. It uses static analysis to soundly reason about continuity, by proving whether a given program encodes a continuous function. On top of [6], [7] further analyzes and quantifies the robustness of a program to input uncertainty. These techniques are static. In contrast, our work is dynamic. These two are synergetic. In fact, we adapt their technique to prove continuity of continuous cores identified by our dynamic analysis.

10. Conclusion

We develop a white box sampling technique that allows scientists to selectively and efficiently sample discontinuous points in output functions, given input error bounds. It works by hashing the values of discrete factors in sample executions and then compares the hashes of multiple runs to determine if additional samples are needed. Besides the basic algorithm, we develop a validation runtime to improve confidence of our sampling results. We also develop a slice-based hashing scheme to avoid hashing irrelevant discrete factors. For programs in which control flow differences (across multiple sample runs) are intensional and hence do not affect continuity, we develop a profiler to identify such code regions and statically prove that they are continuous. Our results show that the technique is highly effective for the real-world programs.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI '90*.

[2] V. N. Alexandrov, I. T. Dimov, A. Karaivanova, and C. J. K. Tan. Parallel monte carlo algorithms for information retrieval. *Math. Comput. Simul.*, 62(3-6), 2003.

[3] J. Barhen and D. B. Reister. Uncertainty analysis based on sensitivities generated using automatic differentiation. In *ICCSA*, 2003.

[4] I. Beichl, Y. A. Teng, and J. L. Blue. Parallel monte carlo simulation of mbe growth. In *IPPS*, 1995.

[5] M. Carbin and M. C. Rinard. Automatically identifying critical input regions and code in applications. In *ISSTA*, 2010.

[6] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL*, 2010.

[7] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving programs robust. In *ESEC/FSE*, 2011.

[8] U. Consortium. The universal protein resource (uniprot) in 2010. *Nucleic Acids Res*, 38(Database issue), Jan 2010.

[9] J. C. Helton, J. D. Johnson, C. J. Sallaberry, and C. B. Storlie. Survey of sampling-based methods for uncertainty and sensitivity analysis. *Reliability Eng. & Sys. Safety*, 91(10-11), 2006.

[10] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. Mcdb: a monte carlo approach to managing uncertain data. In *SIGMOD*, 2008.

[11] P. D. Karp. What we do not know about sequence analysis and sequence databases. *Bioinformatics*, 14(9), 1998.

[12] M. G. Morgan and M. Henrion. *Uncertainty: A Guide to Dealing with Uncertainty in Quantitative Risk and Policy Analysis*. Cambridge University Press, 1992.

[13] S. Singh, C. Mayfield, R. Shah, S. Prabhakar, S. E. Hambrusch, J. Neville, and R. Cheng. Database support for probabilistic attributes and tuples. In *ICDE*, 2008.

[14] W. N. Sumner, T. Bao, X. Zhang, and S. Prabhakar. Coalescing executions for fast uncertainty analysis. In *ICSE*, 2011.

[15] E. Tang, E. Barr, X. Li, and Z. Su. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In *ISSTA*, 2010.

[16] S. Tripathi and R. S. Govindaraju. Engaging uncertainty in hydrologic data sets using principal component analysis: Banpca algorithm. *Water Resour. Res.*, 44(10), Oct 2008.

[17] B. A. Worley. Deterministic uncertainty analysis. Technical Report ORNL-6428, Oak Ridge National Lab. TN (USA), 1987.

[18] M. Zhang, X. Zhang, X. Zhang, and S. Prabhakar. Tracing lineage beyond relational operators. In *VLDB*, 2007.

[19] X. Zhang, W. Hines, J. Adamec, J. M. Asara, S. Naylor, and F. E. Regnier. An automated method for the analysis of stable isotope labeling data in proteomics. *Journal of the American Society for Mass Spectrometry*, 16(7):1181–1191, July 2005.

[20] X. Zhang, S. Tallam, N. Gupta, and R. Gupta. Towards locating execution omission errors. In *PLDI*, San Diego, CA, 2007.