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 program factors that can potentially lead to discontinuity and hash the values of such factors during execution in a cost-effective way. The hash values are used to guide the sampling process. Our results show that the technique is very effective for real-world programs. It can achieve the precision of a high sampling rate with the cost of a lower sampling rate.

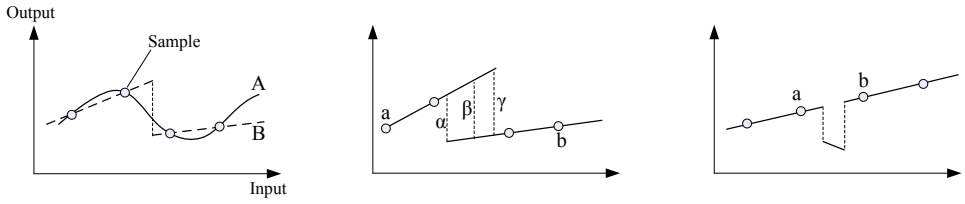
1. Introduction

Uncertainty poses a prominent challenge to data processing programs. Traditional data processing programs handle scalar data values. However in the presence of uncertainty originated from instrument errors and measurement precision limitations, input data have error bounds. It is very important to reason about if a program would behave differently with input errors.

Long term rainfall prediction is often realized by software operating on Sea Surface Temperature (SST) data [24]. Due to the difficulty and cost of deploying sensors, SST contains a lot of interpolated data, which are uncertain. Such uncertainty can lead to different prediction results. In [26], it was shown that a program used to process data from a biology experiment cannot properly model the uncertainty in a parameter provided by human scientists based on their experience such that a protein was mistakenly classified as a cancer indicator. Such mistakes could be highly costly because follow-up wet-bench experiments are usually carried out based on the data processing outcome. In bioinformatics, one of the most widely used sources for protein data is Uniprot [9], 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 [16]. Software facilitating financial decision making is often required to model uncertainty [18].

Traditionally, uncertainty analysis is conducted on the underlying mathematical models [25]. However, modern data processing uses more complex models and relies on computers and programs, rendering mathematical analysis difficult. Recognizing the importance of uncertain data processing, recently, researchers have proposed database techniques to store, maintain and query uncertain data [15, 21]. 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, in uncertain data processing, many properties of interest are dynamic. For instance, whether output is reliable in the presence of uncertainty is dependent on the concrete input error bounds. It demands analyzing program execution instead of the program. More importantly, while static continuity 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 output derivatives can be automatically computed. However, these techniques cannot directly reason about changes within input error bounds; they also have difficulties in handling certain language features, such as control flow related statements.

Monte Carlo (MC) methods provide a simple and effective means of studying uncertainty [5, 15, 23]. 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 often 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 out-



(a) Continuous? (b) Where is the discontiuity? (c) Missing segments?

Figure 1. Three sample problems caused by discontinuity.

put changes according to the variation of an 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 being 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 a cancer biomarker in our experiment.

Traditional black-box MC approaches can be made adaptive by comparing the output values of two samples to determine if additional samples are needed, that is, additional samples will be acquired in between two samples if their corresponding outputs differ substantially. However, such methods are sub-optimal. They have difficulties in handling the case in Fig. 1 (c). Also, unnecessary samples may be acquired for a continuous function if its slope is large.

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 given uncertain inputs can be disclosed with a small number of samples. Discontinuous points break an output curve (i.e the curve representing how output varies according to the uncertain input) 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 functions. Hence, the idea is to monitor MC sample execution, particularly, the value changes of these artifacts, and use the monitored values to direct the MC process to selectively collect more samples where discontinuity is likely to happen.

At a high level, the technique works as follows. 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. If the hash values differ, indicating discontinuity, 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. Our contributions are highlighted as follows.

- We formally define the problem and identify the possible sources of discontinuity in a program, called discrete factors.
- We propose a novel dynamic analysis that hashes the values of discrete factors during an execution. The hash values can be leveraged by MC algorithms to achieve selective sampling. We also propose a more sophisticated version of the analysis that hashes only the discrete factors *relevant* to the selected output. It allows us to avoid considering changes in irrelevant discrete factors to prevent unnecessary samples.
- We propose the basic selective MC algorithm and its two extensions. One extension takes two sampling intervals as configuration, and aims to achieve the precision of the small interval with the cost of the large one. The other extension aims to achieve optimal sampling given a fixed budget (i.e. the number of samples allowed). We also study the safety of the algorithms.
- We observe that in real world programs, there are small code regions in which control flow differences do not cause discontinuity. Such differences are mostly intentional for purposes such as optimization. We develop a profiler to identify such code regions and prove their continuities. Our algorithms can thus avoid hashing these regions.
- We evaluate our technique on a set of SPEC2000 floating-point programs and a biology data processing program. The results show that the proposed white-box sampling technique can identify discontinuity effectively and efficiently.

2. Discontinuity and Discrete Factors

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, ... 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. Program *execution* is hence denoted as a function over the uncertain input $P(x_1, ...x_n)$.

In this paper, we aim to *develop a white-box MC method that can quickly and effectively determine the shape of the output 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 there is only 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 a 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 output discontinuity. We assume uncertain input values are continuous in their error bound. To induce discontinuity on output, these continuous inputs have to go through some discrete factors and result in different discrete values. The basic idea of our technique is hence to monitor the execution of discrete factors to detect 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 integer, leading to the 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 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. In our experience, control flow is the dominant discrete factor.

Consider the following example.

```
1. x=...; //the uncertain input
2. if (x>=2.0)
3.     f=x+3.0;
4. else
5.     f=3.0-x;
```

The predicate at line 2 makes $x = 2.0$ a discontinuous point. On its left ($x < 2.0$), the curve takes on the shape of $f(x) = 3.0 - x$; on the right including the point $x = 2.0$, $f(x) = x + 3.0$.

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 $x = -\frac{\pi}{2}, \frac{\pi}{2}, ...$

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.

There are also other programming language artifacts that do not have correspondence in the mathematical world, such as arrays, pointers, and bit operations. However, these artifacts themselves cannot *initiate* discontinuity such that they are not considered as discrete factors. For example, if the computation of an output involves an array element $A[i]$ which is affected by the uncertain input. Assume the influence is through the array index i, which is of the (discrete) integer type. There must be a preceding discrete factor, such as an explicit or implicit type cast because the uncertain in-

put is of floating-point type. Therefore, as long as we track the value change of that factor, we capture the discontinuity propagated through the array element.

Given the definition of discrete factors, we have the following theorem that serves as the foundation of our technique. Given an execution, we denote the output as a mathematical function of the uncertain input $o(x)$. Here $o(x)$ is limited to be an elementary function and it takes real number inputs only. One can consider the function is constructed by setting the uncertain input as a symbolic variable and then performing symbolic execution concurrently with the concrete execution.

Theorem 1. *Given two sample executions, if all of their discrete factors produce the same discrete values, they must have the same output function $o(x)$.*

Intuitively, if the two executions have the same discrete values, they must have taken the same program paths and all the pointers and array indices must be identical to ensure the same data dependences. As a result, the symbolic output functions must be identical. The formal proof is elided.

Infeasibility of Using Symbolic Analysis. Note that one might formulate the challenge as a dynamic test generation problem that generates uncertain input values to explore the different values of the discrete factors. For example, exploring all the possible program paths within the input error bound may be able to expose the discontinuity caused by control flow. However, we found that this is impractical for real world data processing programs for the following two reasons. (1) Path condition functions are often of high order. We found that 10 out of 11 SPEC CFP 2000 programs we have studied have high order (≥ 2) path conditions [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 choose to 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 [a,b] 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 $t(x) > 0.3$ (line 6), it may take the form $o(x) = 0.3$ (line 7) or $o(x) = 0.75$ (line 9). Function $t(x)$ is of a high order, rendering techniques relying on constraint solving in-applicable. The curves for $t(x)$ and $o(x)$ are depicted in Fig. 2 (c). Our technique aims to leverage program analysis to guide collecting a small set of samples that disclose the shape of the output function $o(x)$, particularly its discontinuity.

Our technique first identifies all the discrete factors in the program. They are places that operate on real values and produce discrete values, and thus 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 the discrete factors encountered. Two sample runs having the same hash value suggests (likely) continuity. Note that the states of the two executions are still largely different despite the identical hash value. For example, the floating point computation that is data dependent on the uncertain input may very likely have different values. If the hash values differ, an additional sample is taken in between the two original samples and the technique continues to inspect the two sub-ranges divided by the new sample, until a threshold is reached.

Assume we start with two samples a and b (step (1) in Fig. 2 (b)). Readers can refer to Fig. 2 (c) for the samples and their corresponding outputs. The order of sampling is denoted by the alphabetical order of the samples. Line 3 has different branch outcomes in the two sample runs, resulting in different hashes. An additional sample c is then taken at the mid-value of a and b, dividing the region into two subregions [a,c] and [c,b]. The technique first considers [a, c] (step (2)) and divides it with an additional sample d. Subregion [a, d] is further divided by e (step (3)). The hash values of a and e are identical. The process ceases to collect more samples in [a, e]. Instead, it collects more in [e, d] until a small sampling interval threshold is reached at Ⓑ, disclosing the discontinuous point at $x = 1.0$. Note that the ranges with the same hash value (and thus no samples needed in between), such as [a, e], denote the savings brought by our technique. A uniform sampling scheme with the threshold as small as the interval of [e, f] (at point Ⓑ) requires a lot more samples.

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. For example in Fig. 2 (a), the discrete factor at line 2 has nothing to do with $o(x)$ and should be excluded from hashing. Similarly, if z is the output variable instead of o, the control flow differences in lines 3-9 should be excluded. We develop a slice-based hashing algorithm that hashes only the discrete factors in the *dynamic slice* of the output variable on the fly.
- 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 if not properly handled.

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

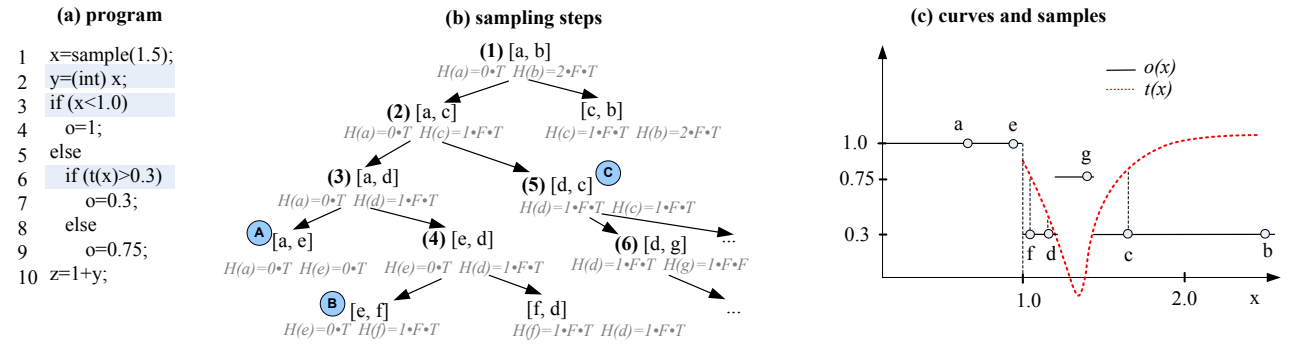


Figure 2. An illustrative example. The dots in (c) represent the samples. The highlighted statements in (a) represent discrete factors. A node in (b) represents a sampling region with the hash values of the lower and upper bounds. The numbers denote the steps. $H(a)$ denotes the hash value of sample a and operator '$\cdot$' denotes hash aggregation (e.g. "$0 \cdot T$" denotes the hash of $y = 0$ and the predicate at line 3 having the *true* value).

We observe that such effects are usually present in small code regions. We develop a profiler to identify regions from the whole code base and prove that they must be continuous. Thus, we can avoid hashing the control flows in these regions.

- It may not be safe to skip sampling when two hashes are identical. Consider the example in Fig. 2. The two hashes of d and c are identical (point Ⓒ in Fig. 2 (b)). If additional samples are not taken in between (e.g. g), we will miss the two discontinuous points in that subregion. Therefore, we need to study the safety of omitting sampling.

The following sections describe the technique in details and our study of the above practical problems.

4. Basic Hashing Semantics

In this section, we discuss the basic hashing semantics that hashes the values of all the discrete factors encountered during program execution. Recall that two identical hashes (of two respective sample runs) indicate that the mathematical forms of the two output functions are identical (Theorem 1), assuming a perfect hashing scheme. The discussion is limited to one uncertain input for simplicity, while our technique supports multiple uncertain inputs.

$$
\begin{array}{lll}
\textit{Program} & P ::= & s \\
\textit{Stmt} & s ::= & s_1; s_2 \mid \textbf{skip} \mid x^{\ell} := e \mid \textbf{while}\ x \bowtie^{\ell} 0\ \textbf{do}\ s \mid \\
 & & \textbf{if}\ x \bowtie^{\ell} 0\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \mid \textbf{exit} \\
\textit{Expr} & e ::= & x \mid v \mid \textbf{sample}(r_1, r_2)^{\ell} \mid e_1\ \textbf{binop}^{\ell}\ e_2 \mid \\
 & & \textbf{discrete}(f, e) \mid x \bowtie^{\ell} 0 \\
\textit{Value} & v ::= & n \mid r \mid b
\end{array}
$$

$$\textit{Var}\ x,\ \textit{Function}\ f \in \textit{Identifier} \quad n \in Z \quad r \in \textit{Real}$$
$$\ell \in \textit{Label} \quad b \in \textit{Boolean}$$

Figure 3. Language

Language. To facilitate formal discussion, we introduce a kernel 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 $y - 1.0$.

We support three kinds of values: integers, real values, and boolean values. Real values could be uncertain. A **sample**(r_1, r_2) expression represents a sample within the error bound of $[r_1, r_2]$. 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.

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

DEFINITION:

$\textit{Store}\ \sigma : \textit{Var} \to \textit{Value} \qquad \textit{Hash}\ \theta \in Z$

$\textit{getSample}(\ell, r_1, r_2)$: sample a value at ℓ within error bound $[r_1, r_2]$

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

$$
\begin{array}{l}
\sigma : x \xrightarrow{e} \bot, \sigma(x) \qquad \sigma : \textbf{sample}(r_1, r_2)^{\ell} \xrightarrow{e} \bot, \textit{getSample}(\ell, r_1, r_2) \\
\sigma : v \bowtie^{\ell} 0 \xrightarrow{e} \ell_T, \mathbf{T}\ \ \textbf{if}\ v \bowtie 0 \qquad \sigma : v \bowtie^{\ell} 0 \xrightarrow{e} \ell_F, \mathbf{F}\ \ \textbf{if}\ \neg(v \bowtie 0) \\
\sigma : v_1\ \textbf{binop}\ v_2 \xrightarrow{e} \bot, v_3 \quad \text{where } v_3 = v_1\ \textbf{binop}\ v_2 \\
\sigma : \textbf{discrete}(f, r) \xrightarrow{e} n, n \quad \text{where } n = f(r)
\end{array}
$$

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

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

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

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

Figure 4. Hashing Semantics

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_1, r_2), and the binary operation v_1 **binop** v_2 are not discrete factors so that their evaluation generates a void hash value, denoted as $\perp$. Observe that we don't hash uncertain input values despite the fact that they are the origin of execution differences. In contrast, the hash for a relational 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. In our implementation, we use addition as the hash operation [2]. For the void hash $\perp$, we have $\theta \triangleleft \perp = \theta$.

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

According to the hashing semantics, we essentially aggregate the branch outcomes of all the predicate instances encountered during execution and the values of all the discrete functions, including those embedded in an expression. It features low runtime overhead as it only entails additions at selected places. This is critical for practicality of white-box sampling because if the technique were heavy-weight, one could simply collect many random samples.

5. Sampling Algorithms

In this section, we discuss a number of sampling algorithms and their safety. The first one is a greedy but unsafe algorithm that aggressively avoids collecting unnecessary samples. The second one is an improved algorithm that provides certain guarantee. The third one works with a fixed sampling budget (i.e. the number of samples is pre-defined). All these algorithms assume single uncertain input for simplicity. Our technique supports multiple uncertain inputs.

5.1 Greedy Algorithm

Algorithm 1 describes a greedy algorithm. It takes two samples as input and generates a sequence of samples, including the two inputs. Ideally, the samples sufficiently expose discontinuity.

Function **sampleDriver**(χ_1, χ_2) represents 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.

[2] Addition is not a perfect hash scheme. However, our experience shows that such a simple scheme is sufficient.

Algorithm 1 Greedy Algorithm.

Input: a pair of sample points χ_1 and χ_2.

```
sampleDriver (χ1,χ2)
 1: θ1 := P(χ1)
 2: θ2 := P(χ2)
 3: return χ1 · sampleInside(χ1, θ1, χ2, θ2) · χ2
```

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

```
sampleInside (χ1, θ1, χ2, θ2)
 4: if θ1 = θ2 ∨ |χ1 − χ2| < τ
 5:     return nil
 6: else {
 7:     χm := (χ1 + χ2)/2
 8:     θm := P(χm)
 9:     return sampleInside(χ1, θ1, χm, θm) · χm ·
10:         sampleInside(χm, θm, χ2, θ2)
11: }
```

In function **sampleInside**(), the algorithm tests if the two provided hashes are the same. If so, it *aggressively* ceases to collect more samples in between the two given samples (line 4). Another termination condition is that even the two hashes are different, if the distance of the two provided samples is less than a pre-defined threshold τ (line 4), no more samples are collected. 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, part of the sampling sequence is $a \cdot e \cdot f \cdot d \cdot c \cdot \ldots$.

Despite its simplicity, the greedy algorithm is not safe. It misses the discontinuity in [d,c].

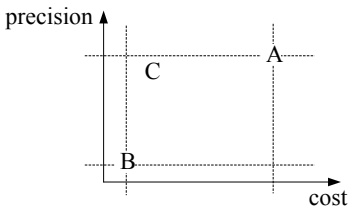


Figure 5. Design space. A is a uniform MC sampling process with a small interval, B a process with a large interval, and C our two-threshold algorithm.

5.2 Two-Threshold Algorithm with Guarantee

Given two sample runs with the same hash values, assuming a perfect hashing scheme, it is unfortunately intractable to determine if it is safe to avoid sampling in between. No matter how small the range delimited by the two samples is, it is always possible to have a conditional statement predicating on an expression that is not monotonic in the region (e.g. line 6 in Fig. 2) such that even though the predicate has the same branch outcome at the two sample points (e.g. d and c in Fig. 2), it may have a different branch outcome somewhere in between (e.g. g in Fig. 2), rendering a sampling decision based on the hash values at the boundaries unsound. Note that the mathematical form of the expression is likely a high order function in practice such that reasoning about its monotonicity analytically is infeasible. Next, we describe an algorithm that provides certain guarantee while retaining the advantage of white-box sampling.

Definition 3. *We say a discontinuous point d (i.e. a value in the range of the uncertain input that causes discontinuity in the output) is detected if and only if there exist two samples χ_i and χ_{i+1} in sequence such that:*
(1) $d \in [\chi_i, \chi_{i+1}]$;
(2) there is not another discontinuous point $d' \in [\chi_i, \chi_{i+1}]$;
(3) their hashes $\theta(\chi_i) \neq \theta(\chi_{i+1})$.

Intuitively, a discontinuous point is detected if its presence is captured by two samples with different hash values and there are no other discontinuous points in between the two samples.

Theorem 2. *A regular MC process that performs uniform sampling with an interval τ guarantees to detect any discontinous point that has a distance $\geq \tau$ to its neighboring discontinous points.*

Proof: Let d be such a discontinuous point and d_p, d_s be its immediate preceding and succeeding discontinuous points, respectively. There must be two neighboring samples χ_p and χ_s such that $\chi_p \in [d_p, d]$ and $\chi_s \in [d, d_s]$. Assuming a perfect hashing scheme and all mathematical library functions are continuous, $\theta(\chi_p) \neq \theta(\chi_s)$, d is detected. □

A uniform MC with a small interval τ_s has strong guarantee but also a high cost, as denoted by point A in the design space in Fig. 5. In contrast, a uniform MC with a large interval τ_l has weak guarantee but a low cost (point B).

We develop a MC algorithm that can achieve the benefits of both A and B (i.e. point C in Fig. 5). The algorithm takes two sampling intervals τ_l and τ_s, denoting the large and small intervals, respectively. It has a low cost close to B and the *practical* precision close to A, that is, it can detect a set of discontinuous points close to A in practice. In the theoretically worst case, it provides the same guarantee as B.

The idea is to continue taking additional samples in between two samples that have the same hash value, if the distance between the two samples is larger than τ_l (i.e. the larger interval provided). It is described in Algorithm 2. It has the same driver as the greedy algorithm. The only difference is at line 4 in **sampleInside** (), which is the termination condition of sampling.

Algorithm 2 Two-Threshold Algorithm.

Definition: τ_l denotes the termination threshold for regions with the same hash;
τ_s denotes the termination threshold for regions with different hashes

sampleInside (χ_1, θ_1, χ_2, θ_2)
4: **if** $(\theta_1 = \theta_2 \wedge |\chi_1 - \chi_2| < \tau_l) \vee (\theta_1 \neq \theta_2 \wedge |\chi_1 - \chi_2| < \tau_s)$
5: **return nil**
6: **else** {
7: /*the same as lines 7-10 in Algo. 1*/
8: }

In the example in Fig. 2, with $\tau_l = 0.5$ and $\tau_s = 0.15$, we get the sequence of samples as described in (b), in which an additional sample g is taken in $[d,c]$ such that all discontinuous points are detected.

5.3 Safety In Practice

While in the worst case, the two-threshold algorithm with thresholds τ_l and τ_s can only provide the same guarantee as a standard uniform sampling algorithm with the large threshold τ_l, in practice, we observe that the algorithm is almost as precise as a standard algorithm with the small threshold τ_s (see Section 8). In this section, we discuss the reasons behind this.

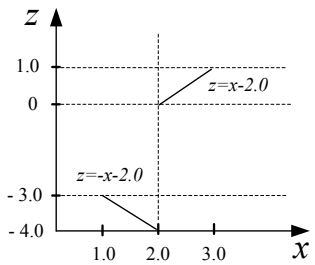
Intuitively, we observe that the majority of the discrete factors are monotonic. Recall that a discrete factor d turns a real value to a discrete value. We denote the real value of a discrete factor as a function $d_r(x)$ of the uncertain input x. Assume through sampling, we observe that two sample runs have the same hash. They must have the same discrete values for all the discrete factors, including that of d. If $d_r(x)$ is monotonic within the range delimited by the two samples, it is easy to infer that the discrete value of d for each input value within the range must be the same [3].

For example, if the value of z in the predicate of a conditional statement "**if** $z \bowtie 0$..." 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 also the continuity within the range.

[3] An implicit assumption is that discrete operations themselves are always monotonic, which is true in practice. For instances, type casts and comparisons are monotonic operations.

```
//x in [1.0,3.0]
1. y=x-2.0;
2. if (y>=0)
3.        f=x+3.0;
4. else
5.        f=3.0-x;
6. z=f-5.0;
7. if (z>0)
8.        o=...;
```

(a)



(b)

Figure 6. Function $z(x)$ of the discrete factor at line 7 has two live ranges: one corresponds to `y>=0` being `true` and the other being `false`. They represent two functions $z(x) = x - 2.0$ and $z(x) = -x - 2.0$.

Formally, the real function of a discrete factor $d_r(x)$ does not need to be monotonic within the entire error bound of x to ensure safety. Instead, we introduce the notion of *live range* of a discrete factor, which is essentially a sub-range in the error bound of x. A discrete factor may have multiple live ranges. As long as all discrete factors are monotonic in each of their live ranges, not necessarily the entire error bound, safety is ensured.

Definition 4. *A sub-range $[\chi_1, \chi_2]$ of the uncertain input error bound is a live range of a discrete factor d if and only if for any $\chi_t \in [\chi_1, \chi_2]$, all the discrete factors encountered during program execution $P(\chi_t)$ before d must have the same discrete value.*

A live range has the following property.

Property 1. *The real function $d_r(x)$ of a discrete factor d must have the same mathematical form in its live range.*

The property can be easily derived from the definition of live range.

Intuitively, within the entire input error bound, there may be inputs that induce different control flow paths leading to a discrete factor and different discrete co-efficients computed along these paths, hence, the real function of the factor may be of different forms. A live range represents an input sub-range that has the same form.

Example. Consider the example in Fig. 6 (a). There are two discrete factors: the comparisons at lines 2 and 7. For line 2, there is only one live range, which is the entire error bound [1.0, 3.0]. The real function has only one form $y(x) = x - 2.0$. For line 7, there are two live ranges, [1.0,2.0] and (2.0,3.0] as the preceding discrete factor at line 2 has different values for the two ranges. The factor hence has two mathematical forms as shown in Fig. 6(b). □

With the above definitions, we have a sufficient condition for safety.

Theorem 3. *If the real functions of all discrete factors are monotonic in their live ranges, it is safe to skip samplings within two samples with the same hash value.*

The theorem can be proved by contradiction. The proof is elided for brevity. The theorem essentially demands that the individual mathematical forms of a discrete factor be monotonic. It substantially lowers the requirement on monotonicity to ensure safety: a discrete factor does not need to be monotonic in the whole sampling range, but rather its individual live ranges. In practice, live ranges could be very small.

For example in Fig. 6, although $z(x)$ is not monotonic in the whole sampling region [1.0,3.0], it is monotonic in the two live ranges. According to Theorem 3, even the greedy algorithm is safe for this program.

Although we have reduced the safety problem to the problem of determining monotonicity of discrete factors in their live ranges, solving the monotonicity problem analytically is still very challenging given the mathematical complexity of the real functions of discrete factors and the resource constraint imposed by the design goal of competing with random sampling. We instead perform empirical study to observe the monotoniciy in practice. We make the following observations. Empirical support can be found in Section 8.

1. The majority (94-100%) of the discrete factors are indeed monotonic in their live ranges. Live ranges could be very small (<10% of the entire sampling region for complex programs). This suggests that we may have very few safety violations in practice. We make the following speculation. Data processing algorithms are developed by humans and humans are usually not good at reasoning and controlling fluctuating functions. Therefore, they use many conditional statements to divide such complex functions into monotonic regions that they can easily reason about. We plan to conduct study of code patterns in the future to deepen our understanding.
2. Some discrete factors have non-monotonic real functions (0-6%), but most of them always produce the same discrete value and thus don't lead to any false negatives in sampling. The majority of the cases are caused by a pattern similar to the following: the non-monotonic function is the addition of a slowly changing function with large values and a fluctuating function with very small values. Hence, despite the non-monotonicity, after discretization, the same value is always yielded.
3. There are some very rare cases (2 in total in our study) that are both non-monotonic and yielding different discrete values. They are the real threats to safety. However, the corresponding program was written in such a way that there exist correlated predicates that guard the non-monotonicity.

Our observations are limited to the benchmarks and the experimental setup. However, even in the worst scenario that the real functions of discrete factors are not monotonic in their live ranges and they yield different values after discretization, the two-threshold algorithm is still able to provide certain level of guarantee.

Ensuring the safety of avoiding sampling in between two runs with the same hash represents only one aspect of the overall soundness. The termination threshold τ_s also indicates that we may not detect segments that are smaller than τ_s. It may be feasible to design a more sophisticated termination condition: the process terminates only when the difference between the sequences of discrete factors of two sample runs is minimal, for instance, their control flow paths differ by only one predicate. The challenge lies in balancing complexity and the entailed overhead as we are competing with the very cheap uniform sampling. We will leave it to our future work.

5.4 Fixed Budget Algorithm

We have also developed a best-effort algorithm that tries to perform optimal sampling given a fixed budget (i.e. a fixed number of samples allowed). The algorithm starts in the same way as the greedy algorithm but employs a priority queue to select the next sample point. When enqueued the subregions are prioritized based on: (1) if they have different hash values; (2) their sizes. At each step it dequeues one subregion to further sample in between from the priority queue, until the budget is used up.

Algorithm 3 Fixed-budget Algorithm.

Input: a pair of sample points χ_1 and χ_2 and a budget B.

sampleDriver (χ_1,χ_2, B)
1: $\theta_1 := P(\chi_1)$
2: $\theta_2 := P(\chi_2)$
3: $Q.enqueue(<\chi_1,\theta_1,\chi_2,\theta_2>)$
4: **return** $\chi_1 \cdot \chi_2 \cdot sampleInside(Q,B)$

Input: a priority queue Q and a budget B;
Output: a sequence of sample points in (χ_1, χ_2);
Definition: τ denotes the termination threshold;

sampleInside (Q,B)
5: $<\chi_1, \theta_1, \chi_2, \theta_2> := Q.dequeue()$
6: **if** $\theta_1 = \theta_2 \vee |\chi_1 - \chi_2| < \tau \vee B.overbudget()$
7: **return nil**
8: **else** {
9: $\chi_m := (\chi_1 + \chi_2)/2$
10: $\theta_m := P(\chi_m)$
11: $Q.enqueue(<\chi_1,\theta_1,\chi_m,\theta_m>)$
12: $Q.enqueue(<\chi_m,\theta_m,\chi_2,\theta_2>)$
13: **return** $\chi_m \cdot sampleInside(Q,B)$
14: }

6. Hashing Slices

In the algorithms discussed in the previous section, a global hash value is computed for an execution. However, this may be too restrict in practice. In some cases, even though two executions have different hash values, 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 output variable. It is worth mentioning that although conceptually we are hashing the discrete factors in output dynamic slices, the computation is peformed on the fly without explicitly computing any dynamic slices.

The semantic rules are presented in Fig. 7. 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 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

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

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

$$\sigma,\Gamma : x \xrightarrow{e} \sigma(x) \quad \textbf{if } \Gamma(x) = \bot \qquad \sigma,\Gamma : x \xrightarrow{e} \langle\sigma(x),\Gamma(x)\rangle \quad \textbf{if } \Gamma(x) \neq \bot$$
$$\sigma,\Gamma : \textbf{sample}(r_1,r_2)^\ell \xrightarrow{e} \langle getSample(\ell,r_1,r_2),\ell\rangle$$
$$\sigma,\Gamma : \langle v,\theta\rangle \bowtie^\ell 0 \xrightarrow{e} \langle \mathbf{T}, \theta \triangleleft \ell_T\rangle \quad \textbf{if } v \bowtie 0$$
$$\sigma,\Gamma : v \bowtie^\ell 0 \xrightarrow{e} \mathbf{T} \quad \textbf{if } v \bowtie 0$$
$$\sigma,\Gamma : \langle v_1,\theta_1\rangle \textbf{ binop } \langle v_2,\theta_2\rangle \xrightarrow{e} \langle v_3, \theta_1 \triangleleft \theta_2\rangle \quad \text{where } v_3 = v_1 \textbf{ binop } v_2$$
$$\sigma,\Gamma : \langle v_1,\theta_1\rangle \textbf{ binop } v_2 \xrightarrow{e} \langle v_3,\theta_1\rangle \quad \text{where } v_3 = v_1 \textbf{ binop } v_2$$
$$\sigma,\Gamma : v_1 \textbf{ binop } v_2 \xrightarrow{e} v_3 \quad \text{where } v_3 = v_1 \textbf{ binop } v_2$$
$$\sigma,\Gamma : \textbf{discrete}(f,\langle r,\theta\rangle) \xrightarrow{e} \langle f(r),\ \theta \triangleleft f(r)\rangle$$
$$\sigma,\Gamma : \textbf{discrete}(f,r) \xrightarrow{e} f(r)$$

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

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

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

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

Figure 7. Slice Hashing Semantics

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. In the subsequent execution, a value is related to the uncertain input if it has a non-void hash.

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 $\sigma,\Gamma,\mathcal{S}: s \xrightarrow{s} \sigma',\Gamma',\mathcal{S}',s'$, 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 own 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, reflecting that evaluations inside a branch of a conditional are control dependent on the predicate of the conditional. 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 a 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 hash inherited from the current last entry of $\mathcal{S}$.

Example. Table 1 presents an example evaluation of the program in Fig. 2(a) with a sample $\chi = 2.20$. Observe that

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

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

```
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 × 10.0;
5      A[i] :=...;
6  o:=A[5];
```
(II)

Figure 8. Examples for safety issues.

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. At line 7, the hash of o is inherited through control dependence. At line 8, the hash of z is inherited from y.

Safety. We have the following sufficient condition for safety.

Theorem 4 (Safety-Slice)**.** *Given two input samples χ_1 and χ_2, assume the slice hashes of the output variable 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 $d_r(x)$ over the uncertain input x. If*
(1) all $d_r(x)$ are monotonic in their live ranges;
(2) all memory addresses remain unchanged within the range,
then the output function must be continuous in $[\chi_1, \chi_2]$.

Condition (1) requires all discrete factors in the execution, not only in the slice, to be monotonic. Consider the example in Fig. 8 (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, without including line 4. As a result, all discrete factors in the slices are monotonic. However, we can easily see that it is unsafe to skip sampling. Condition (1) precludes such cases.

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. 8 (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. How-

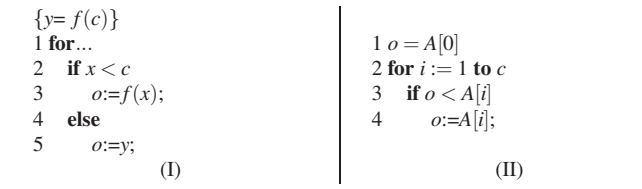


Figure 9. Real continuous core examples from `178.galgel`. Variable o denotes the output; c denotes a compiler time constant; $f(x)$ is a continuous function. The first line in (I) represents the precondition.

ever, 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. According to our experience, the two conditions mostly hold in practice. Note it is currently not affordable to analytically validate them on the fly.

Deciding Global Hashing or Slice Hashing. Slice based hashing is more expensive due to the more complex instrumentation, although it can avoid redundant samples caused by irrelevant discrete factors. Hence, it would only be beneficial when there are enough irrelevant discrete factors to discount its higher instrumentation cost. We use the following method to predict the applicability of slicing based approach. We run the program twice with the same sample input, one with global hashing and the other with slice hashing. If the number of the discrete factors in the global hash and that in the slice hash differ substantially, we will proceed with the slice based approach.

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 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. 9 (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 5. *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, 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. Basically, the profiler first detects predicates that may evaluate to different branch outcomes in their live ranges. For each predicate live range in which the predicate evaluates to both true and false, at its immediate post-dominator (i.e. the joint point of the two branches), the profiler inspects the values of all the variables that are defined inside the conditional and use by others outside the conditional within the live range, to check if they appear continuous. If so, they are continuous core candidates. We elide the details of the profiler for brevity.

Given a continuous core candidate, we then 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. 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. 9 (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. 9 (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 function over the uncertain input $o_1(A[1]) = 3.0$ 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.

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.

program	LOC	pred. order	non-poly funcs in pred.	# of segments
168.wupwise	5K	-2 $\sim$ 2	cos, log, sqrt	1
171.swim	466	0		1
172.mgrid	463	-1 $\sim$ 2	abs, sqrt	1
173.applu	4K	-1 $\sim$ 17	abs, sqrt	1
178.galgel	27K	2	abs, sqrt	100+
183.equake	2K	-1 $\sim$ 3	sin, cos, sqrt	6
187.facerec	3K	2	abs, sin, sqrt	5
188.ammp	14K	2	sin, cos, sqrt	1
191.fma3d	60K	-3 $\sim$ 6	sin, abs, sqrt	1
200.sixtrack	47K	-1 $\sim$ 2	sin, abs, sqrt	1
301.apsi	7K	-1 $\sim$ 14	sin, abs, sqrt	2 - 4
deisotope	2K	2		4

Table 2. Program characteristics.

```
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. Our system supports both C/C++ and Fortran. It is publically available[4].

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 an uncertain input value v, we assume its error bound to be [50%*v, 150%*v].

Table. 2 shows the basic characteristics of the programs. It includes the lines of code, the order of predicate functions (regarding the uncertain input), the non-polynomial mathematical primitives involved in these functions and the number of (continuous) segments in the output curve. We acquire the predicate function orders and the non-polynomial primitives through profiling.

As we can see from the table, a number of programs have high order predicate functions. 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 primitive functions involved, such as trigonometric functions or square root. This supports our earlier discussion about the difficulty of applying symbolic techniques to analyzing the path conditions.

Also observe that 7 out of the 12 benchmarks are continuous. However, it does not mean our technique is not useful for them. Black-box MC approaches will have difficulty in determining if there are small discontinous segments along the output curve (case (c) in Fig. 1).

8.1 Monotonicity of Discrete Factors

In the first experiment, we study the monotonicity of discrete factors. We check the changes of the monotonicity for each live range of every discrete factor. We collect 1000 samples for each program using a regular MC algorithm to conduct our study. Table3 shows the results. Column 2 contains the number of discrete factors in each program. Some of the programs have continuous cores so that their numbers are after excluding the predicates in the cores. The "`mono.`" column shows the percentage of the discrete factors that are monotonic in their live ranges. The "$\neg$ `mono.` $\wedge$ `fixed-val`" column presents the percentage of the discrete factors that are not monotonic but always yield the same value after discretization. The "$\neg$ `mono.` $\wedge$ `diff-val`" column presents the number of those that are neither monotonic nor yielding the same discrete value. These are the cases that could lead to safety issues. The last column shows the length of live ranges as the percentage of the entire error bound.

The discussion of the results can be found in Section 5.3. The results imply that it is highly unlikely for our algorithms to miss samples when they cease to collect samples due to identical hash values. There are only two potentially harmful discrete factors being observed (one in each of `183.equake` and `187.facerec`), both predicates. However, they did not cause any problem because there exists another predicate (after the problematic predicate) that happened to have different branch outcomes at the boundary of the non-monotonic live range of the problematic predicate functions, entailing different hashes and thus more samples in between.

8.2 Runtime Overhead and the Greedy Sampling Algorithm

The second experiment is to evaluate the runtime overhead of the two hashing semantics. The results are shown in Table 4. Columns 3-5 present the results for the basic (global) hashing scheme. Observe that it is highly efficient (an average of 2.67% overhead for each sample run).

Columns 6-8 present the results for the slice-based hashing scheme. Observe that it is more expensive, with an average overhead of 231%. This is due to its more heavyweight instrumentation. Another reason is that we haven't tried hard to optimize our implementation yet. Observe that

[4] http://www.cs.purdue.edu/homes/tbao/smartMC.tar.gz

program	# d-factor	mono.	¬ mono. ∧ fixed-val	¬ mono. ∧ diff-val	avg. live range
168.wupwise	0*	-	-	-	-
171.swim	0*	-	-	-	-
172.mgrid	0	-	-	-	-
173.applu	52	100%	0	0	100%
178.galgel	3219K*	98.68%	1.32%	0	4.37%
183.equake	23043K	99.97%	0.03%	1	9.96%
187.facerec	1891K	94.46%	5.54%	1	7.42%
188.ammp	8070K	100%	0	0	100%
191.fma3d	1521	100%	0	0	100%
200.sixtrack	40364K	100%	0	0	100%
301.apsi	301333K	100%	0	0	2.94%
deisotope	121K	99.99%	0.01%	0	29.25%

*The numbers are after precluding continuous cores.

Table 3. Monotonicity of discrete factors.

program	native	basic			slice		
		time	overhead	# of samples	time	overhead	# of samples
168.wupwise	2.05	2.14	4%	2	6.56	220%	2
171.swim	0.15	0.15	1%	2	0.36	142%	2
172.mgrid	3.31	3.32	0%	2	11.13	236%	2
173.applu	0.06	0.07	6%	2	0.23	271%	2
178.galgel	0.69	0.70	10%	416	3.34	384%	416
183.equake	0.20	0.21	6%	66	0.36	81%	66
187.facerec	1.23	1.25	2%	117	4.12	235%	12
188.ammp	2.72	2.73	0%	2	3.51	29%	2
191.fma3d	0.02	0.02	0%	2	0.06	200%	2
200.sixtrack	2.22	2.27	2%	2	12.60	468%	2
301.apsi	1.75	1.77	1%	167	9.02	415%	24
deisotope	0.02	0.02	0%	55	0.04	90%	20
AVERAGE			2.67%			231%	

Table 4. Efficiency of the basic (global) hashing and the slice based hashing. The results are based on the greedy algorithm

it makes differences on three programs: `187.facerec`, `301.apsi`and `deisotope`. For `187.facerec`and `deisotope`, it reduces the number of samples from 117 to 12and from 55 to 20 respectively, while still precisely exposes all the discontinuous points. For `301.apsi`, the reduction is much larger (from 167 to 24) and easily pays off the extra overhead.

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. Note that here we present the results of the two hashing schemes only for evaluation and comparison purpose. As discussed earlier (Section 6), we have an easy method to predict which hashing scheme should be used and the user is supposed to just apply one approach.

It is also worth mentioning that the overheads are not affected by the sampling algorithms.

We observe that the greedy algorithm is 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. Except `178.galgel`, the discontinuous points of all programs are precisely detected by both the basic and slice hashing schemes (i.e. each continous point is delimited by two samples with different hashes). This is not surprising because according to the study of monotonicity and Theorem 3, we almost never miss discontinuous points when we stop collecting more samples due to identical hash values.

8.3 Effectiveness of the Sampling Algorithms

In the third experiment, we study the effectiveness of the three proposed sampling algorithms. We first collect 10000 uniform samples using a regular MC to acquire a very precise output curve, called the ideal curve. Then we measure

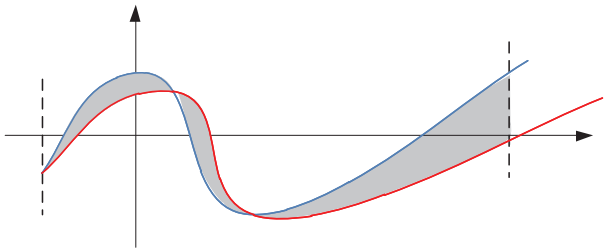


Figure 10. The shaded area between the two curves represents the error.

program	Greedy		2-Threshold		Fix-Budget	
	$\|\chi\|$	err	$\|\chi\|$	err	$\|\chi\|$	err
178.galgel	416	0.58	462	0.58	250	0.76
183.equake	66	0.11	101	0.07	50	0.35
187.facerec	71	0.50	102	0.54	50	0.45
301.apsi	24	0.80	76	0.26	40	0.40
deisotope	55	0.12	93	0.52	45	0.13
AVERAGE		0.42		0.39		0.42

Table 5. Effectiveness of the three sampling algorithms. $|\chi|$ is the number of samples; *err* denotes the ratio between our error and the error of the same number of uniform samples.

the error of the curves generated by the different algorithms regarding the ideal curve as shown in Fig. 10. In this experiment, we focus on the programs that have discontinuity.

Table 5 shows the number of samples needed for each algorithm and the corresponding relative errors. A relative error is the error of the curve acquired by our approach divided by the error of the curve generated by the same number of uniform samples. For example, the relative error of `183.equake` for the greedy algorithm is 0.11 means that the error of the curve with the 66 greedy samples is only 11% of the error of the curve with 66 uniform samples. For the two-threshold algorithm, the two thresholds are $\tau_l = 2\%$ and $\tau_s = 0.1\%$ of the error bound, equivalent to collecting 50 and 1000 samples, respectively. For the fixed budget algorithm, because the curves of different programs are very different, we use roughly half the number of the two-threshold samples as the budget.

From the results, we observe the following. All three algorithms are effectively producing more precise curves compared to those generated by uniform sampling. In some cases, the error of our approach is only 7% of the error of uniform sampling. The relative error of `178.galgel` appears not so impressive as the others. But we will see from our later case study that the regular MC misses many discontinuous points while we don't. It is not reflected in the relative error because the missing segments are so small that their contributions to the error are also small. The relative error of `301.apsi` is larger than others because the continuous segments are curvy and our algorithms avoid collecting samples inside continuous segments. Note that our algorithms anyway capture all the discontinuous points.

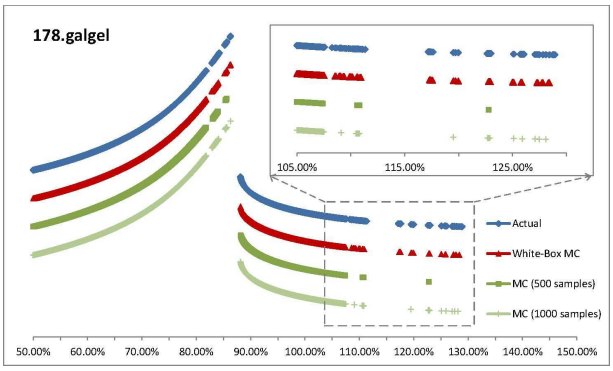


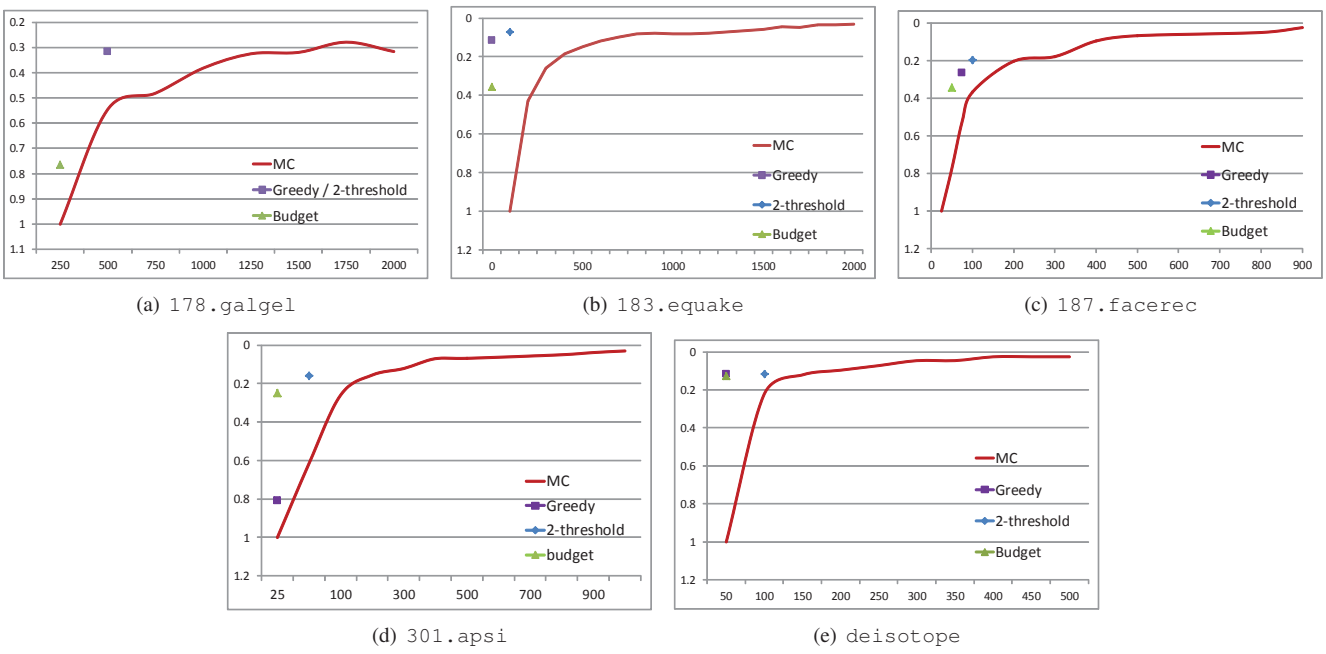
Figure 12. Curves plotted from the samples identified by different methods for program `178.galgel`.

The greedy algorithm requires less number of samples compared to the two-threshold algorithm and the relative errors of these two approaches are comparable. Sometimes, the greedy algorithm has a smaller number of samples but larger relative error (`183.equake`) because it does not collect samples in continuous segments such that its curve has a worse fit. Sometimes, the two-threshold algorithm has a larger relative error even though it collects more samples (`deisotope`). The reason is that the curve is simple enough such that even uniform sampling has low absolute errors.

To better understand the benefit of our algorithms, we gradually increase the number of uniform samples from a number smaller than the samples of our methods (say 50) to 10000 and depict the change of sampling precision (i.e. the error between the approximate curve and the ideal curve) with respect to the number of samples. We then project the results from our algorithms to such figures. The results are shown in Fig. 11. In the figures, the y axis represents the precision with 0 denoting the highest precision (with 10000 samples) and 1 the lowest precision (with 50 samples). We can clearly see *our approaches can achieve the precision of a high sampling rate with the cost of a low sampling rate*. For examples, in `178.galgel`, with around 450 samples (both greedy and 2-threshold), we can achieve the precision of 1600 uniform samples. For `183.equake`, with 60 samples (greedy), we can achieve the precision of 600 uniform samples.

8.4 Effect of Different Uncertain Inputs

So far, our uncertain inputs are randomly selected. In this experiment, we study the effect of selecting different uncertain inputs and observe if our results still hold. We observe that for most of our programs, inputs are uniform, e.g. they are elements an array. Picking a different uncertain input has little effect on the sampling results. For each program, we randomly selected a few inputs and the results were more or less the same. The only exception is `301.apsi`. Its input is not an array, but rather a set of parameters that have different



(a) `178.galgel` (b) `183.equake` (c) `187.facerec`

(d) `301.apsi` (e) `deisotope`

Figure 11. The comparisons between our algorithms and uniform MC.

meanings. Since there are 39 different inputs in `301.apsi`, we randomly pick three of them, the samples generated by our greedy algorithms are 24, 167, 76, and the corresponding relative errors are 0.80, 0.49, 0.26, respectively. Observe that our methods are consistently better than MC approaches. For the majority of the cases, the benefits are substantial. There is one case that the relative error is close to 1 indicating our approach is not that better than regular MC. The reason is that the output is a simple function of that particular uncertain input such that even a regular MC provides good approximation.

8.5 Case Studies

In the last experiment, we present our experience with a few cases.

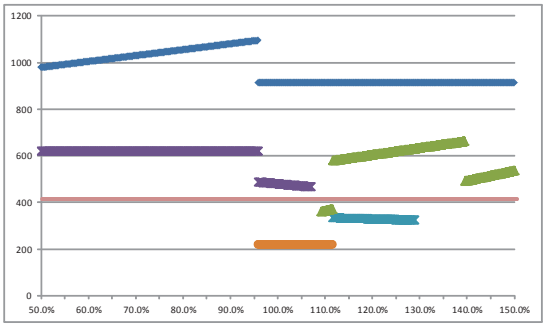
Program `178.galgel` is an interesting case, with which our algorithms generate over 400 samples. Fig 12 shows the output functions computed by different approaches. Let us first focus on the actual curve that is generated by 10000 uniform samples and supposed to be the oracle. The curve has many small missing segments, but 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 no 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 wrong conclusions. These results clearly show the benefit of white-box sampling. Observe from Tables 4 that it only requires 416 samples (greedy) and has only 10% overhead (basic).

Another example comes from the LC-MS (Liquid Chromatography Mass Spectrometry) process [27], which is an effective technique used in real-world 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.

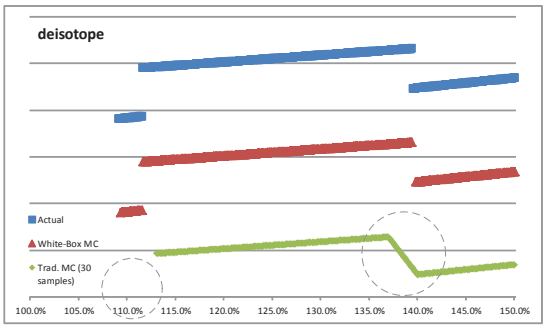
`deisotope` is a program carries out the data processing in LC-MS process. It takes raw data from serum then produces the matching doublets with their intensities.

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. 13(a). The x-axis represents the variation of an input provided by the scientist according to their experience (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 13. 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. 13(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. Observe that on the left, the traditional MC approach misses the small segment. This corresponds to a missing duplet that causes an irrelevant protein being misclassified as the biomarker.

9. Related Work

Uncertainty analysis and MC method. Sampling-based, or Monte Carlo approaches to uncertainty and sensitivity analysis are widely used [13, 17]. Several techniques are proposed to improve the efficiency of MC methods by parallelizing MC trials [2, 4]. In [22], 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. More importantly, they don't focus on analyzing program artifacts to achieve the goal.

Other approaches to uncertainty analysis include model checking [12], automated differentiation [3], and controlled perturbation [14]. They are either too heavy weight or have difficulty handling heterogenous data (certain and uncertain) and program artifacts such as control flow.

MC methods are also used in identifying critical input and code regions [5], and detecting bugs in numerical programs [23]. Our work is complementary to these techniques by reducing the number of needed MC trials.

Static analysis for program continuity and robustness. The technique in [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. For many real-world programs, continuity and robustness cannot be statically proved as they depend on the concrete inputs and input errors. These techniques and ours are synergetic. In fact, we adapt their technique to prove continuity of continuous cores identified by our dynamic analysis.

Others. Taint analysis [8, 20], information flow tracking [19], or lineage tracing [26] are dynamic analysis that track or even quantify information flow during program execution. They can be used to correlate inputs and outputs. However, uncertain data processing requires direct reasoning about how input *changes* lead to output *changes*, which demands reasoning across multiple executions.

White-box fuzzing [10, 11] is an effective technique for dynamic test generation. It analyzes program executions also in a white-box fashion, but based on symbolic execution and constraint solving techniques. In theory, it can be applied to explore the different values of the discrete factors for a given program. However, due to the complexity of high order path conditions and the huge overhead, it is impractical for real world uncertain data processing.

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 analyzing program execution. In particular, it efficiently hashes the values of certain program artifacts called discrete factors during sample execution that are the root causes of discontinuity in output. It then compares the hashes of multiple runs to determine if additional samples are needed. We propose two hashing schemes and three sampling algorithms with different tradeoffs in precision and cost. We also carefully study their safety. For programs in which control flow differences (across multiple sample runs) are intentional and

hence do not affect continuity, we use a profiler to identify such code regions and statically prove that they are continuous so that we don't need to hash their runtime values. Our results show that the technique is very effective for real-world programs.

11. Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments. This research is supported in part by the National Science Foundation (NSF) under grants XXXXXX. Any opinions, findings, and conclu- sions or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

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] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework In *ISSTA*, 2007.

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

[10] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. In *PLDI*, 2008.

[11] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *NDSS*, 2008.

[12] M. P.E. Heimdahl, Y. Choi, and M. W. Whalen. Deviation Analysis Through Model Checking. In *ASE*, 2002.

[13] 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.

[14] Y. C. Ho, M. A. Eyler, and T. T. Chien. A gradient technique for general buffer storage design in a production line. *International Journal of Production Research*,1979.

[15] 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.

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

[17] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 42(1), 2000.

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

[19] S. McCamant and M. Ernst. Quantitative Information Flow as Network Flow Capacity. In *PLDI*, 2007.

[20] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*, 2005.

[21] 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.

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

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

[24] 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.

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

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

[27] 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.

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