Input Range Generation for Compiler-Induced Numerical Inconsistencies

Dolores Miao
University of California, Davis
United States of America
wjmiao@ucdavis.edu

Ignacio Laguna
Lawrence Livermore National Laboratory
United States of America
ilaguna@llnl.gov

Cindy Rubio-González
University of California, Davis
United States of America
crubio@ucdavis.edu

ABSTRACT

Compiler-induced numerical inconsistencies present a significant challenge when testing and verifying numerical software—they can arise in a variety of situations, such as when porting code to a new platform or when using a different compiler or optimization flag. While existing tools can identify the source code location that induce an inconsistency for a specific input, no techniques are available to find input ranges where inputs that trigger these inconsistencies exist. In this paper, we propose a multi-phase approach to detect unknown input ranges that induce such inconsistencies; we call them *inconsistency-inducing inputs*. Our approach combines input-partitioned and coverage-based input sampling, input clustering, and optimization algorithms. We implement our approach in the tool CIGEN, which finds inputs that trigger high compiler-induced inconsistencies in numerical programs and outputs a list of input ranges containing such inconsistency-inducing inputs. Our experimental evaluation show 53.4% improvement over the state of the art in finding inputs that trigger compiler-induced inconsistencies in 175 GNU Scientific Library (GSL) functions. We further examine a subset of the inconsistencies and discuss their characteristics and possible root causes.

CCS CONCEPTS

• **Mathematics of computing** → *Mathematical software*; • **Software and its engineering** → **Software verification and validation**; **Compilers**; **Software reliability**; **Software testing and debugging**.

KEYWORDS

input generation, software testing, floating-point arithmetic, compiler optimizations, numerical reliability, reproducibility, numerical software

ACM Reference Format:

Dolores Miao, Ignacio Laguna, and Cindy Rubio-González. 2024. Input Range Generation for Compiler-Induced Numerical Inconsistencies. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650200.3656618

ICS '24, June 04–07, 2024, Kyoto, Japan

ACM ISBN 979-8-4007-0610-3/24/06
https://doi.org/10.1145/3650200.3656618

1 INTRODUCTION

Testing scientific and high-performance computing (HPC) software involves verifying that numerical calculations are correct and reproducible under various inputs. A significant challenge that arises when testing numerical software is *compiler-induced numerical inconsistencies*. These inconsistencies impact numerical results and manifest themselves in several situations, for example, when code is ported between different platforms such as from CPUs to GPUs or between different GPUs [11], or when using different compilers and/or compiler optimizations [12].

Since compilers have significant freedom when generating floating-point code—particularly when `-ffast-math` or other flags are used—operations can be re-ordered and/or replaced in different ways, which induce such numerical inconsistencies. Such behavior is not limited to programs in native programming languages such as C/C++, and may in fact affect high-level languages and libraries that are built on top of them, as shown in [7]. When such inconsistencies emerge, they decrease programming productivity as they can be tricky to debug and understand. For example, Guo et al. [12] describe a case where scientists at the Lawrence Livermore National Laboratory (LLNL) observed a compiler-induced numerical inconsistency in a real-world hydrodynamics application, Laghos [2]. The inconsistency, which is triggered with the IBM XL C/C++ compiler when using the -O3 optimization flag, took several weeks to be diagnosed.

Previous research studied the question of how to diagnose such inconsistencies for a specific input, and has developed tools to find the code segments in a program that cause inconsistencies. FLiT [23] identifies the CPU function in a program that is impacted by the inconsistency; pLiner [12] can identify the function and line of code in a CPU program; Ciel [17] isolates the function, code line, and expression in a heterogeneous program (CPU and GPU) that is associated with the inconsistency. The main limitation of such tools is that they can only diagnose the inconsistency location for a specific input i out of all the program inputs I—usually i is given by the programmer to the tool and it is confirmed that an inconsistency occurs for i between a specific set of compiler and its optimization flag combinations.

A more general set of questions to ask are: Given a program $\mathcal{P}$, what inputs $j, k, \ldots \in I$, in addition to i, also activate compiler-induced numerical inconsistencies, and how do we find such inputs for $\mathcal{P}$? For all the inputs in I that we know induce inconsistencies, which input induces the largest inconsistency (or error)? As HPC code is ported and tested in different environments, these are crucial questions to answer—having a better understanding of the inputs that cause such inconsistencies and the error they

introduce, helps programmers better address such inputs in testing campaigns, effectively safeguard their programs against these inconsistencies, and produce more accurate code. Additionally, some of these compiler-induced inconsistencies, when triggered by edge cases, may expose unintended or undefined behavior in the hardware/software systems.

Recently, CIV [28] addressed a similar question—finding *an* input which triggers high compiler-induced numerical inconsistencies in programs. However, CIV is limited to finding *one* inconsistency-inducing input, not input ranges, and its requirement of comparing LLVM IR-based instruction traces limits their use when it comes to compilers not built on LLVM.

Main Contributions. In this paper, we present CIGEN (a combination of Compiler-induced Inconsistencies and Input Generation), which is a tool that finds *input ranges* that cause high compiler-induced inconsistencies in numerical programs. Given the observed characteristics of compiler-induced inconsistencies in numerous routines (as discussed in Section 2), we propose a multi-phase approach: first CIGEN performs input-partitioned and coverage-based random sampling to gather an initial set of input points that cause inconsistencies; CIGEN then groups these input points into clusters, and generates non-tight input ranges that cause compiler-induced inconsistencies; then in the last phase, CIGEN employs dense sampling of input values, in addition to running differential evolution (DE) [6] in each input range, so that the inconsistency characteristics in these ranges are discovered.

We evaluate CIGEN on a set of 175 GNU Scientific Library (GSL) [9] functions. We first evaluate the effectiveness and efficiency of CIGEN in finding inputs that *maximize* compiler-induced numerical inconsistencies, which enables a direct comparison with the state of the art in generating inputs that maximize numerical inconsistencies, CIV [28], and Binary Guided Random Testing (BGRT) [3], which is an earlier algorithm, originally designed to find inputs that cause high precision errors. CIGEN compares favorably against CIV and BGRT in terms of the number of functions found with inputs that trigger high compiler-induced inconsistency, with 53.4% more functions than CIV (125 vs. 81) and 525% more than BGRT (125 vs. 20). CIGEN also compares favorably in terms of running time usage and result stability. In addition, CIGEN also finds between 1 to 102 candidate input ranges for these 175 functions in which inputs that cause compiler-induced inconsistencies are found, and we have discussed the characteristics of some of these candidate input ranges. We also use CIGEN and a state-of-the-art compiler-induced inconsistency isolation tool, Ciel [17], to investigate these candidate input ranges and the different source code locations that triggers them.

In summary, the contributions of this paper are as follows:

- A multi-phase approach for finding input ranges with inputs that cause significant compiler-induced numerical inconsistencies in a program, combining input-partitioned and coverage-based random input sampling, input sample clustering, and differential evolution (Section 3).
- A platform and compiler independent implementation of our approach in the tool CIGEN. Users provide as input a program with floating-point inputs and a floating-point output, and CIGEN outputs a number of ranges in the input domain, each with statistics of inputs that cause compiler-induced inconsistencies within the input range.
- An evaluation on 175 functions from the GNU Scientific Library that shows that CIGEN compares favorably against the state of the art in terms of the number of functions found with inputs that trigger high compiler-induced inconsistency, time usage, and result stability against indeterminacy (Sections 4.2 to 4.3).
- An investigation into characteristics of inconsistency-inducing inputs found by CIGEN, and into the difference in source code that causes these compiler-induced numerical inconsistencies (Section 4.4).

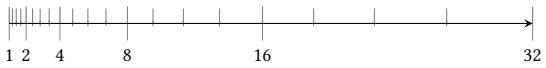

Figure 1: Density of floating-point numbers between 1-32.

2 BACKGROUND AND PROBLEM OVERVIEW

2.1 Floating-Point Types and Arithmetic

Floating-point types split bits allocated for the type into three fixed-point numbers: a sign s which is either 0s or 1, an exponent e which is used as an integer directly, and a mantissa which is then converted to a dyadic fractional number f that is between 0.0 and 1.0. Different floating-point types use a variety of bits for exponent and mantissa. For example, an IEEE 754-2008 compliant 64-bit floating-point number is represented by a 1-bit sign, 11-bit exponent, and 52-bit mantissa. The real value of a 64-bit float number is calculated by the following equation:

$$float_{64}(s,f,e)\begin{cases}(1-2s)(1+f)2^{(e-1024)} & \text{if } 0 < e < 2047\\ (1-2s)f2^{(e-1024)} & \text{if } e = 0\end{cases} \quad (1)$$

The set of possible 64-bit floating-point values are discontinuous values between $-1.79e+308$ and $+1.79e+308$ and is called FP64 in this paper. In general, we will use FP to represent a non-specific set of floating-point numbers in this paper. The equation in (1) varies depending on the exponent; when the exponent is zero, the corresponding floating-point number is subnormal.

An important property of a floating-point type is its logarithmic distribution. It is indicated from Figure 1 that (a) small floating-point numbers are more dense than large numbers; and (b) the amount of unique floating-point numbers between two floating-point numbers are measured in logarithmic terms.

2.2 Calculating Numerical Inconsistency

It is important to establish a standard in how we calculate inconsistencies between outputs from two program variants. Since both outputs are of limited precision, metrics from comparing one limited precision number to an infinite precision counterpart, such as relative error or even unit in last place error (ulp error) does not accurately show the extent of the inconsistencies. Therefore we follow the method used in Herbie [20], originally from STOKE [24], where we simply calculate the amount of floating-point numbers

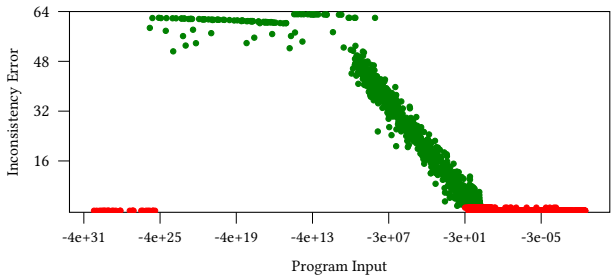

Figure 2: A subset of sampled inputs for the GSL function `gsl_sf_Airy_Ai` and the inconsistency error they have induced. A Red dot is an input that does not trigger inconsistency, while a green dot is an input that does. All negative input values outside of the plot have an inconsistency error of zero and are thus omitted.

Table 1: Inconsistency error when at least one of the values is a special value.

baseline (x)	other (y)	inconsistency
inf or NaN	any	0.0
normal values	+/-inf	$\mathcal{E}$(x, y)
normal values	+/-NaN	$\mathcal{E}$(x, +/-inf)

between the two outputs, then get its base-2 logarithm. We call it *inconsistency error.*

$$\mathcal{E}(x,y) = log_2|\{z \in \mathsf{FP} | min(x,y) \leq z \leq max(x,y)\}| \quad (2)$$

Under this equation, the maximum inconsistency error possible for FP64 is $\mathcal{E}(1.79e+308, -1.79e+308) = 63.99$. Additionally, how inconsistency errors are calculated when either x or y is a special value such as `inf` or `NaN` is shown in Table 1. Note that there is a baseline column in the table, even though unlike precision error calculation in tools such as Herbie, there does not exist a baseline calculated with infinite precision when calculating inconsistency error between two numbers; though in practice, we may still define a certain compiler and optimization flag combination as baseline to determine the degree of an inconsistency.

In this paper we use `clang -O0` as baseline. We choose this specification for two reasons: first, if the baseline result is a special number, the function input domain likely does not include the current input, thus the inconsistency from the current input is not of practical concern; second, `NaN` values usually result from instructions with `inf` as operands, therefore it is fair to treat them as `inf` when calculating inconsistency errors.

2.3 Compiler-Induced Numerical Inconsistency

A compiler-induced numerical inconsistency is an inconsistency shown in output values that come from running programs compiled from the same source code under different compilers and/or optimization flags. An example of a real-world function is the `gsl_sf_Airy_Ai()` GSL function. With input -973569893418508.1, if compiled with `clang -O0`, the return value would be $-6.314246267753519e+77$; but if compiled with `clang -O3` the return value would be 0.00010100276891802863. The inconsistency error introduced here is 63.161, which is very large.

Next, we formally define compiler-induced numerical inconsistency in terms of input. Suppose we have two floating-point representations of the same function under two different compiler/optimization flag combinations, one baseline $y_0 = f_0(x)$, one $y_{opt} = f_{opt}(x)$. Then, we can define an inconsistency function with an input x as follows:

$$Inc_f(x) = \mathcal{E}(f_0(x), f_{opt}(x)) \quad (3)$$

Therefore, to study the characteristics of compiler-induced numerical inconsistencies is to study the characteristics of $Inc_f(x)$ itself; to find an input x that maximizes compiler-induced inconsistency is to find a local or global maximum of $Inc_f(x)$.

During our preliminary research, we found that $Inc_f(x)$ functions may exhibit complex characteristics. As an example, we compile the GSL function `gsl_sf_Airy_Ai()` with `clang -O0` and `clang -O3 -ffast-math` separately into two binaries, run both with 25,000 randomly generated inputs from the $[-\infty, 0]$ input domain, and calculated the inconsistency error triggered by each input. First we observe that for all sampled inputs, except for the input range shown in Figure 2, $Inc_f(x)$ is zero. This makes finding at least *one* input that causes inconsistency a prerequisite of finding input ranges that contain high inconsistency errors. Second, when we focus on inputs in the range [-1e-30, -1e-8] and the inconsistency errors they trigger which is shown in Figure 2, it is evident that the inconsistency function has numerous local minimums and maximums, and thus difficult to express in polynomial terms; simply employing widely used global optimization strategies such as Bayesian Optimization (BO) [18] or Markov Chain Monte Carlo (MCMC) [10] on the whole input domain may not be suitable for functions like this. Lastly, when the input value is around -3e+1, there is no clear boundary where all inputs inside the boundary cause inconsistencies, and all inputs outside the boundary cause no inconsistencies. This means that defining a clear-cut boundary with a limited number of samples is difficult.

To better address the characteristics of inconsistency functions as shown above, we propose a novel, multi-phase method to study the characteristics of compiler-induced inconsistencies which will be discussed in Section 3.

3 TECHNICAL APPROACH

The workflow of CIGEN consists of three phases: first, domain-partitioned sampling and coverage-based random sampling are performed to generate sampling points in the input domain whose inconsistencies are then calculated; second, input clustering is used to generate "candidate input ranges" (CIR) where sample inputs that cause inconsistencies exist; lastly, dense sampling and DE are performed for each CIR to find the statistics of inputs in the CIR, such as the average and maximum inconsistency errors. The visualized workflow of CIGEN can be found in Figure 3.

Because of the logarithmic distribution of floating-point numbers discussed in Section 2.1, all subroutines in this algorithm are either run on the logarithmic scale such as the clustering algorithm and the optimization algorithm, or that exponents and mantissas are treated

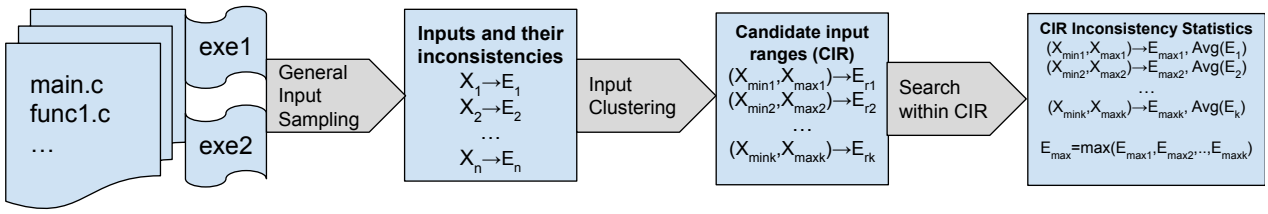

Figure 3: Workflow of CIGEN.

separately such as generating random floating-point numbers from randomized exponent and mantissa. We go into detail in the next subsections.

3.1 General Input Sampling

The first phase in our algorithm is general input sampling, which combines two different methods of generating inputs and testing them to determine whether they trigger compiler-induced inconsistencies: domain-partitioned sampling and coverage-based sampling. The goal of this phase is to find a set of inputs that trigger compiler-induced inconsistency before we can categorize them into groups in the next phase. The algorithm for this phase is shown starting from Line 16 in Algorithm 1.

In this phase, CIGEN generates a set of sample points which are deemed representative of the characteristics of the inconsistency function $Inc_f(x)$ in the input domain. The first step of the algorithm, called domain-partitioned sampling, is to partition the floating-point input domain of the program, FP, into several partitions, based on the exponent part of the floating-point number. The partitions are predefined in the case of FP32 and FP64 in CIGEN, and can be customized depending on the floating-point types used in the program. For FP64 programs which are evaluated in this paper, the partition CIGEN uses is similar to the many-range partition used in XSCOPE [15]:

$$F = N^-_{max}, -2^{1020}, -2^{333}, -2^{32}, -2^3, -1, -2^{-3}, -2^{-32}, -2^{-333}, -2^{-1018}, 0, 2^{-1018}, 2^{-333}, 2^{-32}, 2^{-3}, 1, 2^3, 2^{32}, 2^{333}, 2^{1020}, N^+_{max} \quad (4)$$

$$fp_0 = N^-_{max}, fp_1 = -2^{1020}, \ldots \quad (5)$$

$$\mathsf{FP}_0 = |fp_0, fp_1|, \mathsf{FP}_1 = |fp_1, fp_2|, \ldots \quad (6)$$

CIGEN uses this input partition because the sampling rate is higher in partitions with very small (close to subnormal), close to 1.0, and very large (close to infinite) floating-point numbers. Our assumption is that numbers in these partitions are likely to trigger floating-point exceptions such as underflow or overflow, thus causing compiler-induced numerical inconsistencies. Compared to XSCOPE, CIGEN uses values which are powers of 2 as boundaries of each input partition instead of powers of 10. CIGEN then randomly chooses a particular partition $\mathsf{FP}_i = |fp_i, fp_{i+1}|$ and then generates a random floating-point number within FP_i. Since floating-point numbers are logarithmically distributed, if floating-point numbers are generated in FP_i with uniform distribution, in cases where fp_i is much smaller than fp_{i+1}, for example, if $fp_i = 1$ and $fp_{i+1} = 32$ as shown in Figure 1, the value closest to fp_{i+1} is 16x more likely to be selected. To mitigate this issue, CIGEN instead randomly generates an exponent and a mantissa with uniform distribution, while keeping the resulting floating-point value within FP_i. In this way it is guaranteed that any floating-point number in FP_i has the same probability of being selected, whether the number is closer to fp_i or fp_{i+1}. This step is then repeated K_1 times. The time complexity of this step is $O(2^n)$ where n is the number of input parameters.

Coverage-based Sampling. The next step is coverage-based sampling. CIGEN collects a set of exponent values for each input parameter that are not covered by the inputs generated in the first step. For example, if the exponent value 100 did not appear in previously generated inputs for input parameter 1, then 100 will be added to set 1. Then CIGEN selects an exponent value from set 1, combines this exponent with a random mantissa as input parameter 1, generates other input parameters using domain-partitioned sampling described in the first step, and tests this set of input parameters to see the inconsistency error it triggers. Continue this selection process until all exponents in set 1 has been selected, then repeat the same process in set 2, until all exponents in all sets are selected. The goal of this step is to make sure that for all input parameters, every possible exponent value can be found in at least one input. The time complexity for this step is $O(e \cdot n)$ where e is the number of possible exponents for the floating-point type, and n is the number of input parameters.

The inputs generated by both steps above are tested for compiler-induced numerical inconsistency. If an input triggers an inconsistency, the input is added to S_a, otherwise it is added to S_z. Both sets of inputs, along with the inconsistency errors they trigger, are sent to the next phase where they are used in determining the bounds of input ranges that contain inconsistency-inducing inputs.

3.2 Input Clustering

The input clustering phase is shown starting from Line 34 in Algorithm 1. It takes the two sets of inputs from the previous phase, S_a and S_z, categorizes them into various groups according to the relative proximity between inputs using hierarchical agglomerative clustering, and generate a candidate input range (CIR) for each group.

First, CIGEN categorizes S_a and S_z into multiple subsets according to the orthant each input belongs to. The reason for this is that the clustering algorithm used in this phase calculates the Euclidean distance between points *on the logarithmic scale*, and floating-point

Algorithm 1: The multi-phase algorithm in CIGEN.

```
1  Function RandomFloat(Domain):
2      Sign_j =rand (0, 1);
3      Exp_j =rand (Domain);
4      Mant_j =randmantissa ();
5      return Sign_j, Exp_j, Mant_j
6  Function TestInc(x, S_a, S_z):
7      inc = Inc_f(x);
8      MaxInc =max (inc, MaxInc);
9      if inc > 0 then
10         S_a = S_a ∪ {(x, inc)};
11     else
12         S_z = S_z ∪ {(x, inc)};
13     return inc
14 Function MainAlgorithm(Inc_f, n):
15     // General Input Sampling
16     S_a, S_z, ExpSet_{1...n} = ∅;
17     for i ← 1 to K_1 do
18         for j ← 1 to n do
19             Sign_j, Exp_j, Mant_j =RandomFloat(FP_rand());
20             ExpSet_j = ExpSet_j ∪ {(Sign_j, Exp_j)};
21         x =combine (Sign_{1...n}, Exp_{1...n}, Mant_{1...n});
22         TestInc(x, S_a, S_z);
23     for i ← 1 to n do
24         foreach (Sign, Exp) 8 ExpSet_i do
25             for j ← 1 to n do
26                 if i = j then
27                     Mant_j =randmantissa ();
28                     Sign_j, Exp_j = Sign, Exp;
29                 else
30                     Sign_j, Exp_j, Mant_j =RandomFloat(FP_rand());
31             x =combine (Sign_{1...n}, Exp_{1...n}, Mant_{1...n});
32             TestInc(x, S_a, S_z);
33     // Input Clustering
34     CIR = ∅;
35     for S_ai, S_zi in each orthant do
36         C_i1, ..., C_im =ClusteringByAbsLog (S_ai);
37         foreach C_ij in C_i1, ..., C_im do
38             (BB_j, MaxInc_j, MaxX_j) =MinBoundingBox (C_ij);
39             enlarge BB_j by points in S_zi;
40             CIR = CIR ∪ (BB_j, MaxInc_j, MaxX_j);
41     // Search Within Candidate Input Ranges
42     for CIR_i in CIR do
43         for j ← 1 to K_2 do
44             for k ← 1 to n do
45                 Sign_k, Exp_k, Mant_k =RandomFloat(CIR_i);
46                 ExpSet_j = ExpSet_j ∪ {(Sign_j, Exp_j)};
47             x =combine (Sign_{1...n}, Exp_{1...n}, Mant_{1...n});
48             inc=TestInc(x, S_a, S_z);
49             replace MaxInc_i, MaxX_i if inc > MaxInc_i;
50         DifferentialEvolution (Inc_f, CIR_i, MaxX_i);
```

numbers in different orthants cannot be put on the same logarithmic scale. For example, if $X \in S_a$ & $sign(x) = (+,-)$, then $X \in S_{a(+,-)}$. This means that all inputs in this subset have the same sign configuration: the first input parameter is positive, and the second is negative. We now use the subsets $S_{a(+,-)}$ and $S_{z(+,-)}$ to describe the algorithm in this subsection.

Second, for each subset where all inputs belong to the same orthant, all parameters in $S_{a(+,-)}$ are transformed to the logarithm of the absolute value, then CIGEN runs a clustering algorithm to

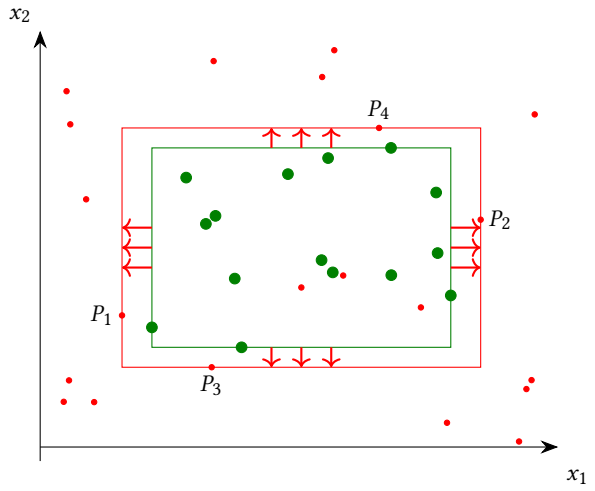

Figure 4: An example showing how to calculate a CIR from a set of input points that cause inconsistencies in larger green dots, and input points that do not cause inconsistencies in smaller red dost. Green box shows the minimum bounding box for the green points, while the red box shows the candidate input range calculated for this set of inconsistency-inducing inputs.

categorize them into different groups. Specifically, CIGEN uses hierarchical agglomerative clustering [19] due to the fact that this algorithm does not require a predefined number of clusters to run, and that CIGEN does not know beforehand how many clusters there are. The end result is a list of groups (subsets) $S_{a(+,-)1}$, $S_{a(+,-)2}$, ..., $S_{a(+,-)r}$.

Lastly, CIGEN calculates a CIR for each $S_{a(+,-)k}$ where $1 \leq k \leq r$. An example is shown in Figure 4. CIGEN first calculates a minimum bounding box (green rectangle in the figure) for the inputs in $S_{a(+,-)k}$ (green dots in the figure), then for each axis (x_1 and x_2) representing each input parameter, CIGEN finds one point in $S_{z(+,-)}$ outside of the minimum bounding box and closest to the lower bound on the current axis (P_1 and P_3 for axes x_1 and x_2 in the figure), then update the lower bound on the current axis to this point; then finds another point in $S_{z(+,-)}$ outside of the minimum bounding box and closest to the upper bound on the current axis (P_2 and P_4 for axes x_1 and x_2 in the figure), then update the upper bound on the current axis to this point. Repeat this process for all axes and the updated bounding box becomes the new CIR. CIGEN then gathers all CIRs and sends them to the next phase.

3.3 Search Within Candidate Input Ranges

The third phase of the CIGEN algorithm identifies characteristics of each CIR sent from the last phase, including average and maximum inconsistency. CIGEN achieves this by running a two-step optimization algorithm on each CIR. The algorithm can be found starting from Line 42 in Algorithm 1.

First, for all CIR, CIGEN runs dense sampling with the same input sampling method as domain-partitioned sampling in Section 3.1 K_2 times to gather data on statistics of inconsistency-inducing inputs in

the CIR. The end results are the average of inconsistencies triggered by sampled points, and a maximum inconsistency along with its triggering input. This information is then sent to an optimization algorithm.

There are many optimization algorithms available for both approximating a function and finding the global maximum/minimum. We have previously shown in Section 2.3 that the inconsistency function $Inc_f(x)$ (a) has numerous local minimums and maximums, and thus it is difficult to express in polynomial terms; and (b) no bound can be inferred where all inputs inside the bound cause inconsistencies, while all inputs outside the boundary cause no inconsistencies. Thus CIGEN uses differential evolution [6] to locate the global maximum of the inconsistency function $Inc_f(x)$ in the CIR. DE is selected because it is a genetic algorithm where current candidates are improved upon by mutations.

CIGEN uses logarithmic scaling for each input range to be optimized. This makes the process of finding maximums more efficient when input ranges are large. Thus the inconsistency function is transformed as:

$$Inc'_f(x) = \mathcal{E}(f_0(2^x), f_{opt}(2^x)) \tag{7}$$

where x is between the logarithm of minimum and maximum values in the input range. The input in the CIR that CIGEN currently reports to trigger maximum inconsistency error is also sent to DE as an initial population, in order to speed up the optimization process. Once all DE processes are finished, the maximum inconsistency between all local maximums found will become the reported global maximum.

4 EXPERIMENTAL EVALUATION

This evaluation answers the following research questions:

- **RQ1** How effective is CIGEN at finding inputs that trigger large compiler-induced inconsistencies in comparison to the state of the art and a general-purpose approach?
- **RQ2** How time-efficient is CIGEN in comparison to the state of the art?
- **RQ3** How many input ranges can CIGEN find with inconsistency-inducing inputs, and what information we can infer from these input ranges?

4.1 Experimental Setup

Benchmarks. We evaluate CIGEN on 175 functions from the GNU Scientific Library (GSL) [8] version 2.7. GSL is a numerical library for C and C++ programmers that provides a wide range of mathematical routines, such as linear algebra, differential equations and special functions. Our evaluation of GSL focuses on all library functions with floating-point input and output, consisting of 175 functions, most of which are special functions such as Bessel and hypergeometric functions. Amongst these functions, 122 are single-parameter functions, while the rest 53 are multi-parameter.

We assume that the input domain of these functions is $\mathsf{FP64}^n$, where n is the number of input parameters, unless the input domain of a function is explicitly stated in the GSL documentation. There are 47 functions out of 175 where there are input domain specifications in the GSL documentation. One example is the input domain for `gsl_sf_bessel_K1(x)` which is $x > 0$. This is a valid assumption because inputs outside of the input domain have no valid meaning in the mathematical sense and may return an `inf` or `NaN` value.

Table 2: Number of GSL functions for which CIGEN, CIV, and BGRT have discovered input that causes an inconsistency error higher than 48, 32, 8, and 0, respectively.

Method	>48	>32	>8	>0
CIGEN	125 (71.4%)	127 (72.6%)	133 (76.0%)	163 (93.1%)
CIV	81 (46.3%)	94 (53.7%)	110 (62.9%)	155 (88.6%)
BGRT	20 (11.4%)	20 (11.4%)	22 (12.6%)	35 (20.0%)

Programs under evaluation are compiled and tested with Clang 16.0.6. Inconsistencies are calculated by comparing results from two binary variants of the same program, one compiled with `-O0` and the other with `-O3 -ffast-math`. This applies to both CIGEN and CIV (both the original paper and our implementation), for a better comparison.

Baselines. As of the writing of this paper, there is no other work that finds input ranges that trigger compiler-induced numerical inconsistencies. The closest work is CIV [28], which for a given program generates an input that triggers a large compiler-induced inconsistency. Thus, we compare CIV and CIGEN in their effectiveness to find large inputs that *maximize* compiler-induced numerical inconsistencies. The source code of CIV is not available publicly, and the CIV authors declined our request for source code. Thus, for better comparison under the same runtime environments, we have implemented[1] the algorithm in CIV, and set up its parameters according to the specifications in the paper.

Since our focus when comparing with the state of the art is in maximizing inconsistencies, we also adapt a general algorithm that finds inputs that cause large errors. Specifically, we use the Binary Guided Random Testing (BGRT) method, implemented in S3FP [3]. This baseline is also used by the CIV authors in their evaluation.

Algorithm Parameters. We specify the number of random samples in both domain-partitioned sampling steps (Section 3.1 and Section 3.3) with $K_1 = K_2 = 256 \cdot 2^n$, where n is the number of input parameters of the function. We use the scikit-learn [21] machine learning library to implement agglomerative clustering (Section 3.2), with parameters as follows: n_clusters=None, distance_threshold=0.2, metric='euclidean', linkage='single'. This means we use the minimum Euclidean distance between input points as the criteria for determining whether two points belong to the same cluster. Lastly, we use SciPy [25] to implement differential evolution (Section 3.3) with parameters: updating='deferred', popsize=20, maxiter=50.

CIGEN and CIV run until termination, thus do not require a budget. For BGRT, we follow the specification given in [28] and set a budget of $T = 60 \cdot 2^n$ seconds, where n is the number of input parameters of the function.

Runtime Environments. We use a PC with an 18-core Intel(R) i9-10980XE processor with 256 GiB RAM running Ubuntu 22.04 OS in our evaluation. While the core algorithms for each tool are run

[1]When needed, we confirmed implementation details with CIV authors. Our implementation of CIV is publicly available in the same code repository as CIGEN: https://github.com/LLNL/CIGEN/.

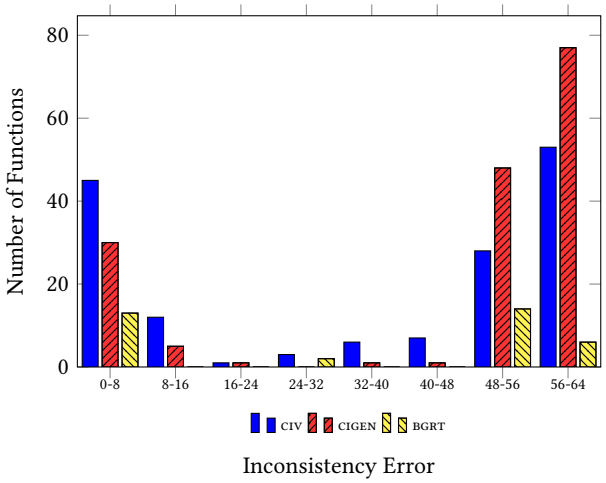

Figure 5: The number of functions with various ranges of maximum inconsistency error discovered by CIGEN, CIV and BGRT. CIGEN finds inputs that trigger high inconsistency error (>48.0) for more functions than CIV and BGRT.

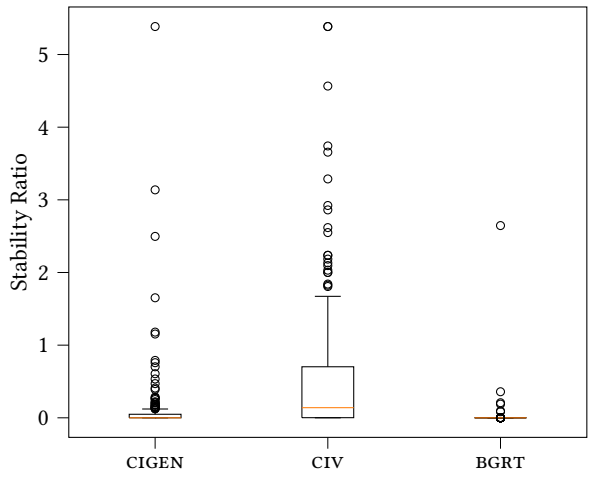

Figure 6: Box plot showing the statistics of stability ratio between the tools for all 175 functions tested. The boxes show the range between the first to the third quartile of results, and the whiskers show the interquartile range. The flier points show stability ratio outside of the interquartile range.

serially, the process of measuring the compiler-induced numerical inconsistencies for every input is run in parallel, utilizing all CPU resources.

4.2 RQ1: Effectiveness in Triggering Large Inconsistencies

We first evaluate the effectiveness of CIGEN in finding inputs that trigger large compiler-induced inconsistencies in comparison to CIV and BGRT. We consider two evaluation metrics: the number of GSL functions for which a compiler-induced inconsistency is triggered, and the average of maximum inconsistency errors discovered for these functions.

Functions with Triggered Inconsistencies. Table 2 shows the number of GSL functions for which the tools can find inconsistency-inducing inputs with error higher than 48, 32, 8, and 0, respectively. CIGEN finds inputs that cause high inconsistency error of 48 or more in 125 functions. This is 54.3% more functions than CIV, which triggers such error for 81 functions. On the other hand, BGRT can only find inputs triggering inconsistency error of 48 or more in 20 functions. Note that CIGEN's 125 functions are a superset of the 81 functions from CIV and the 20 functions from BGRT, which means that all high compiler-induced inconsistencies discovered by CIV and BGRT can also be found by CIGEN. Even when considering smaller inconsistencies, e.g., 8 or higher, CIGEN triggers inconsistencies in a larger number of functions: 133 vs. 110 in CIV and 22 in BGRT.

We further discuss the results based on whether a function is single- or multi-parameter. Out of 122 single-parameter GSL functions tested, CIGEN finds inputs that trigger 32 or higher compiler-induced numerical inconsistency error in 80 of them. In contrast, CIV can only find inputs for 51 functions. CIGEN shows 56.9% improvement over the state of the art. In contrast, BGRT can only find such inputs for 14 functions. On the other hand, among 53 GSL functions with multiple input parameters, CIGEN finds inconsistency-inducing inputs of error 32 or higher in 47 of them, while CIV triggers such errors in 43 functions, a 9.3% improvement. BGRT can only generate inputs for 6 functions. If we consider functions for which an error of 8 or higher is triggered, the improvement of CIGEN over CIV is smaller: CIGEN triggers inconsistencies in 84 single-parameter functions vs. 64 with CIV, a 31.3% improvement; and in 49 multi-parameter functions vs. 46 with CIV, a 6.5% improvement. And again, BGRT performs unfavorably against the other two, with 16 single-parameter functions and 6 multi-parameter functions.

Maximum Average Inconsistency Error. The average inconsistency error triggered on average by CIGEN is 43.40 (38.25 for single-parameter functions and 55.26 for multi-parameter functions) vs. CIV's average of 32.33 (22.51 for single-parameter functions and 51.41 for multi-parameter functions), and BGRT's average of 6.81 (6.72 for single-parameter functions and 7.02 for multi-parameter functions). The statistics of peak compiler-induced inconsistencies of all functions can be found in histogram form in Figure 5. From the histogram, we observe that the maximum inconsistencies of most functions for all tools are distributed in the 0-8 and 48-64 regions, with CIGEN successfully triggering higher inconsistencies for more functions (see the 48-64 region).

From our manual inspection of the results, we can group the compiler-induced inconsistencies CIGEN discover into three categories: (1) smaller (<8) inconsistencies that are accumulated from multiple rounding errors; (2) inconsistencies that are close to 52, where the output from `clang -O3 -ffast-math` is zero while the output from `clang -O0` is subnormal (or close to subnormal); and (3) inconsistencies close to the maximum possible inconsistency error of 64, either from an instruction that generates special numbers such as `inf` or `NaN`, or from sign flips.

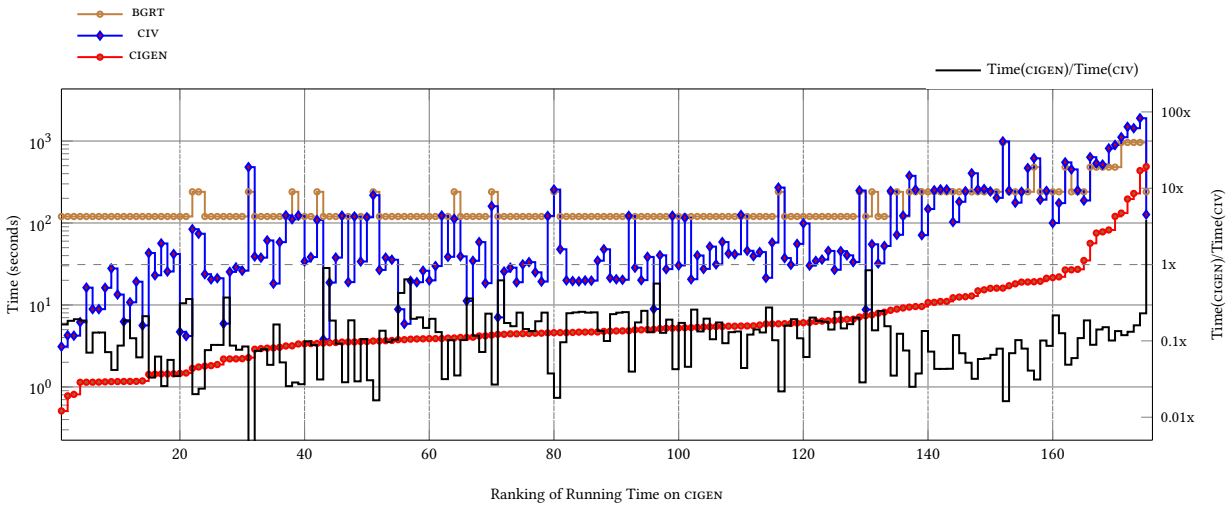

Figure 7: Running time for all 175 functions tested, in ascending order by the running time on CIGEN, along with the running time ratio of Time(CIGEN)/Time(CIV). Running time for BGRT is also displayed for reference.

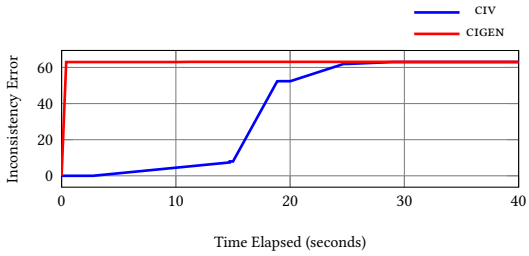

Figure 8: Maximum inconsistency error discovered over time for function `gsl_sf_hyperg_2F1_renorm` for the evaluated tools. Statistics after the first 40 seconds are omitted when the maximum inconsistency error for the tools no longer changes.

Result Stability. Given CIGEN and our baselines utilize random testing, indeterminacy can be a potential issue affecting effectiveness. To evaluate how indeterminacy impacts the maximum inconsistency error triggered by the tools, we use the same method specified in [28] to compare the stability between CIGEN, CIV and BGRT. We run each of the three tools on the same 175 GSL functions 30 times and calculate the stability ratio for each function tested, which is defined as the ratio of the standard deviation ρ to the mean μ: $C_v = \frac{\rho}{\mu}$.

The result is shown in Figure 6. Judging from the box plot, we can first conclude that BGRT performs more stably than both CIGEN and CIV. However, since BGRT performs much less favorably than CIGEN and CIV, the stability shown here simply means it performs consistently worse than the other two. When we compare CIGEN and CIV, we can see that (1) the interquartile range from CIGEN is much smaller than the one from CIV; and (2) in terms of flier points, which indicate functions for which the tools are least stable, CIGEN still performs more stable than CIV. Thus we conclude that CIGEN is more stable than CIV in finding inputs that cause maximum inconsistencies.

Comparison with Original CIV Results. As noted earlier, we did not have access to the original implementation of CIV (Section 4.1), and thus all results discussed in this paper were obtained through our own implementation. Nevertheless, it is important to note that when directly compared to the results presented in [28], CIGEN is even more effective than reported in this paper. From Table 2 we know that CIGEN finds inputs that cause inconsistency error higher than 48 in 71.4% (125 out of 175) of functions, compared to 46.3% (81 out of 175) in our CIV implementation and 30.4% (48 out of 158) reported in the CIV paper. Therefore, we can conclude that CIGEN is superior to both the original and our implementation of CIV in terms of maximum inconsistency error discovered.

Results: CIGEN and CIV perform closely in terms of finding an input that triggers compiler-induced inconsistencies *at all*, even though CIGEN is still slightly ahead; on the other hand, CIGEN is superior than CIV and BGRT in finding inputs that maximize compiler-induced inconsistencies in both single- and multi-parameter functions. Finally, simply adapting a general-purpose approach for maximizing error is not effective for compiler-induced inconsistencies.

4.3 RQ2: Time Efficiency

In this section we evaluate the efficiency of CIGEN and compare it with the state-of-the-art tool CIV.[2] CIGEN finds at least one inconsistency-inducing input for 163 out of 175 functions (see Table 2). For these functions, CIGEN takes an average of 17.95 seconds to generate inputs. For the remaining 12 functions, CIGEN takes an average of 1.31 seconds to conclude that no inconsistency-inducing input is found. On the other hand, our implementation of CIV

[2]BGRT is given a fixed budget to run (see Section 4.1) and thus is not included it in this comparison. Figure 7 includes BGRT only for reference.

takes on average 155.15 seconds to generate inputs that trigger inconsistencies in 155 out of 175 functions. For the remaining 20 functions, it takes 44.73 seconds on average to determine that no inconsistency-inducing input is found. Thus, CIGEN is 8× faster than CIV in finding inconsistency-inducing inputs, and determines that no such input is found 33× faster. Figure 7 plots the running time of each of the 175 functions, regardless of whether an inconsistency is triggered. Functions are sorted in ascending order by the running time of CIGEN.

To investigate how the tools compare when time is limited, we select one of the functions with the longest runtime for both tools: the multi-parameter function `gsl_sf_hyperg_2F1_renorm`. We evaluate how much time it takes for the tools to first discover an input that causes a high inconsistency error. Figure 8 shows the maximum inconsistency error discovered over time. From the figure we can observe that CIGEN finds an input that triggers an inconsistency error of 63 just after 1 second. Meanwhile it takes 28.95 seconds for CIV to achieve the same result. It is possible to use CIGEN to find *an* input that triggers a compiler-induced inconsistency as high as possible within a given budget.

We found only one function, `gsl_sf_multiply`, for which CIGEN takes longer than CIV. From our observation, this is because DE is constantly updating its population with inputs that cause a minuscule amount of increase in inconsistency error, therefore the optimization algorithm takes a long time to finish. However, even when its total running time is higher, CIGEN still finds a large inconsistency-inducing input faster than CIV. Specifically, CIGEN only takes 5 seconds to find an input that causes an inconsistency error higher than 60, while CIV takes 47 seconds to achieve the same result.

Results: CIGEN shows 8× improvement over CIV in running time in finding inputs that trigger compiler-induced inconsistencies. It is 33× faster in determining that no such input is found. It also takes a shorter time than CIV to find *an* input that triggers inconsistencies as high as possible within a specified budget.

4.4 RQ3: Input Ranges With Inconsistency-Inducing Inputs

A unique feature of CIGEN is its ability to find input ranges (candidate input ranges, CIR) in which inputs that cause compiler-induced numerical inconsistencies are found. This section further describes the characteristics of these CIRs in terms of the number of CIRs reported by CIGEN, and the root cause of the compiler-induced inconsistencies triggered by the maximum-error input in each CIR.

Input Range Statistics. We first investigate the CIRs found by CIGEN in single-parameter GSL functions. Amongst 113 functions in which at least one inconsistency-inducing input is found, the number of candidate input ranges are between 1 and 5. A simple example function with only 1 CIR is `gsl_sf_log`, where only positive subnormal input values between [0, 2.225e-308] trigger an inconsistency. A more complex example with 2 CIRs is `gsl_sf_Airy_Ai` in Figure 9. We can see that the inputs we have found to cause compiler-induced inconsistencies can be grouped into two input ranges: one between [-6.63e+26, -1.90], the other between [0.63, 291.13]. For the first input range, among the dense sampling points tested (Section 3.3), 89.1% of sampled inputs trigger inconsistencies, with an average inconsistency error of 46.54. For the second input range, 19.5% of sampled inputs trigger inconsistencies, with an average inconsistency error of 5.49. Developers can utilize this information when they consider improving the accuracy of a function.

In the case of multi-parameter GSL functions, the situation is more complex: among the 50 functions in which we have found inputs that cause compiler-induced inconsistencies, the number of CIRs varies between 1 and 102, with an average of 15.02 CIRs per function. The number of input ranges is especially higher amongst 3 and 4 input parameter functions, likely because there are exponentially more conditions where they satisfy the requirements of introducing a compiler-induced inconsistency. Figure 10 shows an example corresponding to the 2-parameter function `gsl_sf_bessel_Jnu`, for which CIGEN reports 6 CIRs. Each dot in the figure represents a sampled input. Red dots denote inputs that do not trigger inconsistencies. Green dots denote inputs that trigger inconsistencies, and their size is proportional to their inconsistency error. Each CIR is indicated by a blue rectangle.

Root Cause of Triggered Inconsistencies. To further investigate the nature of the compiler-induced inconsistencies captured by each CIR, we use the state-of-the-art tool for isolating compiler-induced inconsistencies, Ciel [17]. Given a program and an input known to trigger a compiler-induced inconsistency in that program, Ciel applies a bisection algorithm and precision enhancement to isolate the expression(s) in the program responsible for causing such inconsistency. We consider all 70 single-parameter functions for which CIGEN reports more than one CIR, and select the input with maximum error from each CIR. We then run Ciel for each program and input. For 21 out of 70 functions, Ciel isolates distinct expressions for different inputs, suggesting that the programs suffer from multiple sources of compiler-induced inconsistency which are captured by the distinct CIRs. An example of these 21 functions is the two inconsistency-inducing inputs in `gsl_sf_Airy_Ai`, in two different input ranges shown in Figure 9: x_1 = 7.663693956591586 and x_2 = −997404629288211.9. For x_1, Ciel isolates a statement in the function `gsl_sf_airy_Ai_e`:

```
double s = exp(-2.0*x32/3.0);
```

Meanwhile for x_2, Ciel isolates a statement in a different function `gsl_sf_cos_e`:

```
z = ((abs_x - y * P1) - y * P2) - y * P3;
```

After investigation at the binary level, it turns out that both inconsistencies are triggered by reassociation of floating-point operations introduced by the `-ffast-math` flag: in the first case, it combines `-2.0/3.0` into one constant; in the second case, it adds `P1,P2,P3` into one constant. This example shows that finding these input ranges can help software developers isolate as much source code that causes compiler-induced numerical inconsistencies as possible, and deal with them accordingly.

On the other hand, Ciel isolated only one source of inconsistency across all CIRs in 28 functions, 25 of which have exactly 2 CIRs. This is still acceptable in common scenarios, for example, positive and negative subnormal numbers trigger an inconsistency from the same source code location, even though the inputs belong to

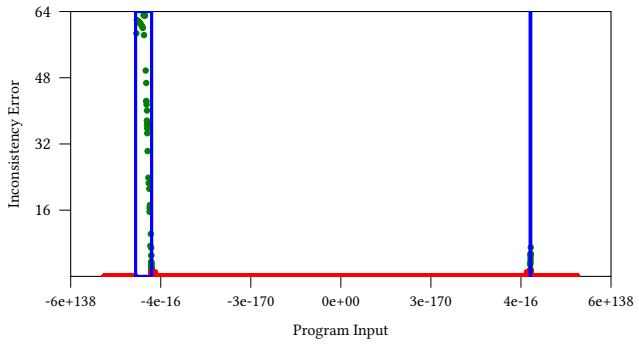

Figure 9: Sampled inputs, the inconsistency error triggered by these points, and detected candidate input ranges of `gsl_sf_Airy_Ai`. Red and green dots indicate sampled inputs, while blue rectangles indicate candidate input ranges.

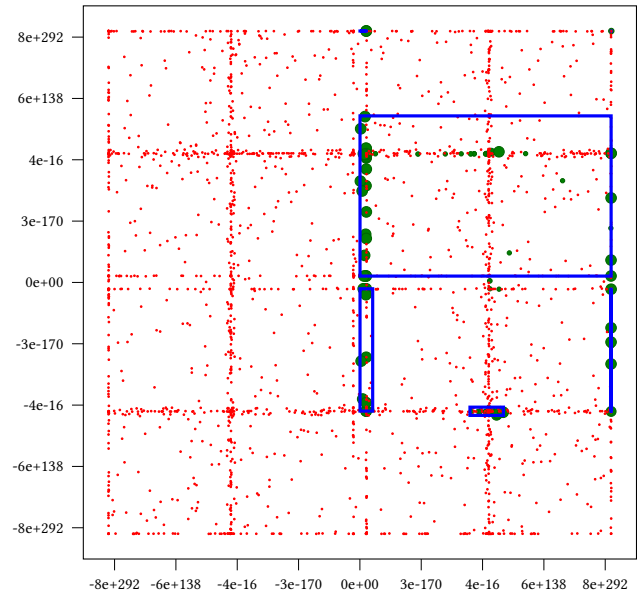

Figure 10: Sampled inputs and detected input ranges of `gsl_sf_bessel_Jnu()`. Red dots denote an input that does not trigger inconsistency, while green dots do. Size of the green dots denotes the inconsistency error triggered. Blue rectangles indicate candidate input ranges.

different CIRs. In such cases is still helpful to know about both CIRs to be able to handle both corner cases. Finally, there were 8 functions for which Ciel only isolated expressions for some inputs, and 13 functions for which Ciel was unable to isolate any expression. This does not invalidate our findings as the inputs generated by CIGEN are used to test the programs and verify the presence of a compiler-induced inconsistency. Rather, these inconsistencies may involve numbers such as ±0 and `NaN`, which remain unchanged regardless of precision and are known sources of false negatives for Ciel.

Results: CIGEN finds between 1 to 102 candidate input ranges for all GSL functions tested. Using the state-of-the-art tool for isolating compiler-induced inconsistencies, Ciel, we find 21 single-parameter functions where multiple sources of compiler-induced inconsistency have been captured by distinct candidate input ranges.

4.5 Limitations

First, our experimental evaluation only considered an assortment of GSL functions with floating-point inputs and output. The number of floating-point inputs is also relatively small, with the maximum being 4 input parameters. In terms of time complexity, the time used for CIGEN to analyze any program is a function of T and I, where T is the execution time of the program and I is the number of program inputs. For a given input $i \in I$, T could be a function of several parameters, such as the number of loops or the number of functions in a program. Thus, for programs with more inputs and longer execution time, CIGEN may still find some inconsistency-inducing inputs under time limits, but it may suffer from scalability limitations mapping CIRs. An improved algorithm that is more aware of the relationship between the source code and the compiled binaries, optimized or not, may be necessary to explore the characteristics of inconsistency-inducing inputs of such programs. Despite the above, our work represents a step forward in automated generation of inputs that trigger compiler-induced numerical inconsistencies, and our evaluation is still representative and on par with the state of the art, considering a large number of real-world functions from the popular GNU Scientific Library.

Second, CIGEN focuses on numerical inconsistencies observed in a deterministic program when compiled with different compilers and/or optimization options. CIGEN does not handle cases where the programs are non-deterministic, or when programs are subject to malicious code injections from outside sources.

Third, as mentioned throughout the paper, we implemented CIV from scratch. In the implementation process we found that many parameters, such as how many partitions the algorithm uses for floating-point input domain (P) or the number of tryouts for each experiment run, are not mentioned in the CIV paper. To minimize the risk of discrepancies between the original implementation and ours, we contacted the authors to request additional information about their implementation. While some questions were answered, there is still room for potential differences. In particular, the results in the CIV paper show a non-exponential relationship between the number of input parameters n and the running time of CIV of a function, but the time complexity of the CIV algorithm should be $O(P^n)$ which is exponential. This in turn means our implementation of CIV may have higher time complexity than the original implementation. The above observations make a direct comparison between CIGEN and the results in the CIV paper impossible. To mitigate this issue, in Section 4.2 we compared the results indirectly, first comparing between our implementations of CIGEN and CIV, then comparing our implementation of CIV against the results in the CIV paper. We also make the source code and data of our work publicly available for further research and validation.

Last, our results reveal an overlap between triggering compiler-induced inconsistencies and triggering floating-point exceptions, such as underflow and overflow. Given this, it may be interesting to see if tools such as XSCOPE [15] can be customized for our purpose.

5 RELATED WORK

Floating-Point Input Generation to Maximize Errors. CIV [28] generates inputs that cause high compiler-induced numerical inconsistencies via input space partition and Markov Chain Monte Carlo (MCMC) sampling. It compares LLVM Intermediate Representation (IR) floating-point execution traces between binary programs compiled with different compiler and/or optimization flags in order to determine if an input is a candidate input for MCMC, but this technique limits its use to compilers based on LLVM. In contrast, CIGEN utilizes the logarithmic distribution of floating-point numbers and the importance of exponents in terms of triggering high compiler-induced numerical inconsistencies while being both architecture and compiler independent.

Other tools related to our work are limited to finding high floating-point precision errors in programs with a limited number of variables. They are referred to in various degrees during the design of CIGEN. S3FP [3] uses binary guided random testing (BGRT) to find inputs that trigger large precision errors in numerical programs. Input ranges for each variable are halved and for every iteration, it picks the upper or lower half of the variable range and creates random test input within these ranges. It then chooses the one range configuration with the largest error; and repeats the random testing process.

Some tools use shadow execution techniques in order to track how precision errors occur and how they are propagated and amplified throughout the program data flow. For example, FPSanitizer [4] and PFPSanitizer [5] run parts of a program in higher precision with posits [14]. The posit version of the program is run in parallel with the same code snippets in original precision, in order to facilitate the debugging of floating-point errors. RAIVE [16] on the other hand uses vectorization to implement shadow execution. Each scalar variable is expanded to a 4-dimension vector, in order to artificially inject errors that are upper bounds of actual errors; RAIVE then detects branch divergence along the execution. We have considered shadow execution during the design of CIGEN. However, we find that optimized and unoptimized versions of the same program can diverge significantly in terms of the order and the types of floating-point instructions which makes shadow execution difficult to implement.

Tools using other dynamic analysis techniques to detect precision errors include ATOMU [31], which uses the condition numbers of each floating-point atomic operation to build an error model to effectively guide the search for large precision errors. FPGen [13] on the other hand treats maximizing floating-point errors as a problem of maximizing code coverage. It uses symbolic execution with KLEE [1] on programs injected with error checks to detect precision loss and catastrophic cancellation. Compared to FPGen, CIGEN uses a black-box approach, and is not limited by the compiler used to implement KLEE—Clang.

Our method of guided random sampling in CIGEN is similar to FPED [26]. It first generates inputs more evenly on the predefined input range, then it partitions the ranges according to error value distribution, and then generates more inputs where precision error is high. Aceso [29] uses an oracle-free approach to estimate floating-point errors. It samples the program and examines the pattern statistics (micro structures) of results of floating-point programs within an input range to estimate the floating-point errors of a program. CIGEN also uses a clustering algorithm to look for patterns in the results of floating-point programs, and also uses an oracle-free approach except when special values are involved.

Use of Optimization Algorithms in Floating-Point Program Analysis. AutoRNP [27] uses differential evolution (DE) and Monte Carlo Markov Chain (MCMC) on the condition number to detect large precision errors, uses a point-to-bound algorithm to expand the point that triggers large precision errors into an input range that triggers large errors. CIGEN also makes use of DE to find inputs that cause compiler-induced numerical inconsistencies.

Similar to CIGEN, genetic algorithm is used in [30] to trigger high precision errors. It uses an example to show that exponents in a floating-point input are important in determining precision errors, and runs genetic algorithm that generates and mutates exponents so that an input value that triggers high precision errors can be found. CIGEN proposes that the exponent is also important in determining compiler-induced numerical inconsistency, and is implemented based on this idea. XSCOPE [15] on the other hand, utilizes an optimization algorithm—Bayesian optimization—to identify inputs that cause floating-point exceptions. Both XSCOPE and CIGEN have the advantage of not requiring the source code of the program to function, and thus not limited to compilers that have source-to-source components.

Input Partition. Herbie [20] is one of the first tools that perform input partition. It first randomly samples program inputs and uses these sample inputs to gauge the extent of precision improvement. and then rewrites numerical expressions under an input range partition scheme (called regime), so that precision can be improved across the whole input domain of an expression. A newer tool, Regina [22], is based on Herbie, and introduces a regime inference algorithm for a multiple of input parameters within respective input ranges. It uses a two-phase, bottom-up then top-down approach to find the most efficient regimes, which are then used in various optimization problems, such as mixed precision tuning, or program rewriting to improve precision.

6 CONCLUSIONS

This paper proposed a multi-phase approach to generate input ranges that trigger compiler-induced numerical inconsistencies by combining input-partitioned and coverage-based input sampling, input clustering, and optimization algorithms. We implemented our approach in the tool CIGEN. Our experimental evaluation showed a 53.4% improvement over the state of the art in detecting inputs triggering high inconsistency errors in 175 GSL functions 8× faster. Further analysis of a subset of inconsistencies revealed their characteristics and possible root causes, which can be helpful for developers to determine the best course to mitigate numerical issues caused by such inconsistencies. Our code and data is publicly available at https://github.com/LLNL/CIGEN/.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-859884), the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under awards DE-SC0022182 and DE-SC0020286, and the National Science Foundation under award CCF-1750983.

REFERENCES

[1] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. USENIX Association, 209–224.

[2] CEED. 2017. CEED/Laghos: High-Order Lagrangian Hydrodynamics Miniapp. https://github.com/CEED/Laghos

[3] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *PPoPP*. ACM, 43–52.

[4] Sangeeta Chowdhary, Jay P. Lim, and Santosh Nagarakatte. 2020. Debugging and detecting numerical errors in computation with posits. In *PLDI*. ACM, 731–746.

[5] Sangeeta Chowdhary and Santosh Nagarakatte. 2021. Parallel shadow execution to accelerate the debugging of numerical errors. In *ESEC/SIGSOFT FSE*. ACM, 615–626.

[6] Swagatam Das and Ponnuthurai Nagaratnam Suganthan. 2011. Differential Evolution: A Survey of the State-of-the-Art. *IEEE Trans. Evol. Comput.* 15, 1 (2011), 4–31.

[7] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *ASE*. IEEE Computer Society, 509–519.

[8] Mark Galassi, Jim Davies, James Theiler, Brian Gough, and Gerard Jungman. 2009. *GNU Scientific Library - Reference Manual, Third Edition, for GSL Version 1.12.* Network Theory Ltd.

[9] Mark Galassi, Jim Davies, James Theiler, Brian Gough, Gerard Jungman, Patrick Alken, Michael Booth, Fabrice Rossi, and Rhys Ulerich. 2002. *GNU scientific library.* Network Theory Limited Godalming.

[10] Charles J Geyer. 1992. Practical markov chain monte carlo. *Statistical science* (1992), 473–483.

[11] Ganesh Gopalakrishnan, Ignacio Laguna, Ang Li, Pavel Panchekha, Cindy Rubio-González, and Zachary Tatlock. 2021. Guarding Numerics Amidst Rising Heterogeneity. In *Correctness@SC*. IEEE, 9–15.

[12] Hui Guo, Ignacio Laguna, and Cindy Rubio-González. 2020. pLiner: isolating lines of floating-point code for compiler-induced variability. In *SC*. IEEE/ACM, 49.

[13] Hui Guo and Cindy Rubio-González. 2020. Efficient generation of error-inducing floating-point inputs via symbolic execution. In *ICSE*. ACM, 1261–1272.

[14] John L. Gustafson and Isaac T. Yonemoto. 2017. Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomput. Front. Innov.* 4, 2 (2017), 71–86.

[15] Ignacio Laguna and Ganesh Gopalakrishnan. 2022. Finding Inputs that Trigger Floating-Point Exceptions in GPUs via Bayesian Optimization. In *SC*. IEEE, 33:1–33:14.

[16] Wen-Chuan Lee, Tao Bao, Yunhui Zheng, Xiangyu Zhang, Keval Vora, and Rajiv Gupta. 2015. RAIVE: runtime assessment of floating-point instability by vectorization. In *OOPSLA*. ACM, 623–638.

[17] Dolores Miao, Ignacio Laguna, and Cindy Rubio-González. 2023. Expression Isolation of Compiler-Induced Numerical Inconsistencies in Heterogeneous Code. In *ISC (Lecture Notes in Computer Science, Vol. 13948)*. Springer, 381–401.

[18] Jonas Mockus. 1994. Application of Bayesian approach to numerical methods of global and stochastic optimization. *J. Glob. Optim.* 4, 4 (1994), 347–365.

[19] Daniel Müllner. 2011. Modern hierarchical, agglomerative clustering algorithms. *CoRR* abs/1109.2378 (2011).

[20] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. In *PLDI*. ACM, 1–11.

[21] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830.

[22] Robert Rabe, Anastasiia Izycheva, and Eva Darulova. 2021. Regime Inference for Sound Floating-Point Optimizations. *ACM Trans. Embed. Comput. Syst.* 20, 5s (2021), 81:1–81:23.

[23] Geoffrey Sawaya, Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, and Dong H. Ahn. 2017. FLiT: Cross-platform floating-point result-consistency tester and workload. In *IISWC*. IEEE Computer Society, 229–238.

[24] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. In *PLDI*. ACM, 53–64.

[25] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, Ilhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy. 2019. SciPy 1.0-Fundamental Algorithms for Scientific Computing in Python. *CoRR* abs/1907.10121 (2019).

[26] Yuanyuan Xia, Shaozhong Guo, Jiangwei Hao, Dan Liu, and Jinchen Xu. 2021. Error detection of arithmetic expressions. *J. Supercomput.* 77, 6 (2021), 5492–5509.

[27] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2019. Efficient automated repair of high floating-point errors in numerical libraries. *Proc. ACM Program. Lang.* 3, POPL (2019), 56:1–56:29.

[28] Hengbiao Yu, Xin Yi, Banghu Yin, Fa Li, Zhenbang Chen, and Chun Huang. 2023. Efficient Generation of Floating-Point Inputs for Compiler-Induced Variability. In *SANER*. IEEE, 224–235.

[29] Daming Zou, Yuchen Gu, Yuanfeng Shi, Mingzhe Wang, Yingfei Xiong, and Zhendong Su. 2022. Oracle-free repair synthesis for floating-point programs. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 957–985.

[30] Daming Zou, Ran Wang, Yingfei Xiong, Lu Zhang, Zhendong Su, and Hong Mei. 2015. A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies. In *ICSE (1)*. IEEE Computer Society, 529–539.

[31] Daming Zou, Muhan Zeng, Yingfei Xiong, Zhoulai Fu, Lu Zhang, and Zhendong Su. 2020. Detecting floating-point errors via atomic conditions. *Proc. ACM Program. Lang.* 4, POPL (2020), 60:1–60:27.