Scalable and Actionable Roundoff Error Analysis for Branching Floating-Point Programs

ANONYMOUS AUTHOR(S)

Reasoning about floating-point roundoff errors is hard for both developers and automated tools as the finite-precision program and its real-valued specification compute different results. The task of bounding their difference is especially challenging in the presence of branches; due to accumulated roundoff errors they may take different paths through the program. Existing static analysis tools that compute bounds on the error due to divergence either compute accurate bounds but do not scale beyond small programs, or scale better but often compute bounds too loose to be useful. Separately, dynamic analysis tools search for a single divergence-triggering input, so they neither cover all branches of a program nor indicate how large the error can be.

This paper presents a novel static analysis for bounding floating-point errors in programs with branches that is both scalable and accurate. The main idea is to first compute a cheap overapproximated bound that is refined on demand. We also integrate dynamic and static analysis. Using static results, the dynamic analysis can find inputs causing divergence for any branch in the code, even if branching is deeply nested. The dynamic results provide a lower bound on the true error and thus indicate the tightness of the static results. We implement both analyses in the Stainless deductive verifier for Scala programs, and empirically evaluate them on existing (smaller) benchmarks and (larger) real-world code, demonstrating scalability and effectiveness.

1 Introduction

Reasoning about floating-point arithmetic is challenging for both developers [25, 26] and automated verification tools, as evidenced by recent work in this area [1, 30, 37, 52]. Because floating-point numbers have finite precision, arithmetic operations round their results. When the resulting roundoff errors accumulate as a program executes, its result can differ from the one one would have obtained if all arithmetic was real-valued and thus error-free. A comparison against a real-valued specification is important in many domains, where, for instance, correctness proofs—automated or human on paper—may assume real-valued arithmetic. Computing the true roundoff error exactly is, however, not practical for a range of inputs.

Static analysis tools [20, 33, 52] take programs written in a (small) subset of, *e.g.,* C, Scala or PVS and aim to automatically compute a sound upper bound on the *worst-case absolute roundoff error*, over all inputs satisfying some precondition. In a complementary way, dynamic analysis tools [14, 34] search for concrete inputs that induce a large roundoff error. Static and dynamic analyses thus yield an upper and a lower bound on the true roundoff errors, respectively.

One of the main open challenges when analysing floating-point code is to efficiently and effectively handle branching. When a conditional involves floating-point values, their accumulated roundoff error can send the floating-point execution down a different path in the program than the real-valued execution would take, potentially resulting in a completely different result. This problem is far from marginal: a study of floating-point code on GitHub [31] shows that half of the functions found contain conditionals and are thus potentially exposed to such divergence. Analysis tools should thus not only consider roundoff errors due to arithmetic, but also these additional *divergence errors* due to diverging control-flow.

Accurately bounding these divergence errors is challenging. Divergence is caused by the typically small roundoff errors due to arithmetic operations. The inputs on which divergence occurs therefore often form very small regions of the input space, sometimes only a few representable numbers wide. Additionally, computing bounds on divergence errors requires comparing two potentially very different paths of the program over these small regions, which is non-trivial to do both accurately and efficiently. Existing approaches trade scalability [33] for accuracy [20, 52], or vice versa. As a result, they often yield loose bounds that can lead to false positives or time out for even modestly sized programs.

Moreover, an error bound alone is not actionable: it leaves the user without the information needed to repair the program, such as which inputs trigger a divergence and where in the code it occurs. Dynamic analyses can find such inputs, but existing approaches [13] search for a single divergence-triggering input, regardless of the error it causes; they thus neither cover the whole program nor indicate how large the error can be.

This paper brings scalable, accurate, and actionable analysis of divergence errors to a general-purpose deductive verifier. Our two core technical contributions are 1) a novel static analysis that computes tight sound upper bounds on the error while scaling to heavily branching programs, and 2) a dynamic analysis that identifies inputs that trigger divergence throughout the program. The dynamic analysis complements the static bound in two ways: by computing a lower bound on the true error, it shows how tight the static upper bound is, and the concrete inputs it reports for each divergence give the user the information that can help to repair the program.

The key insights that make our static analysis scale are twofold: instead of comparing every pair of paths, our analysis computes divergence errors per branching point, requiring only a linear number of expensive optimisation queries in the number of paths. To recover accuracy, we introduce a refinement step that improves the bounds of the largest divergence errors found, empirically matching the accuracy of pairwise comparison with far fewer queries.

The main idea of our dynamic analysis is to use the solutions of the optimisation problems from the static analysis to focus the search around each condition that may diverge. This lets it find divergences throughout the program, including deeply nested ones that unguided techniques overlook.

We implement both analyses in Stainless [30], a general-purpose deductive verifier for Scala that already supported floating-point arithmetic but had no roundoff error analysis. To our knowledge, this makes Stainless the first deductive verifier that automatically infers roundoff error bounds that account for control-flow divergence. Our implementation supports language features such as function calls and algebraic data types, so users can bound the roundoff error of programs much closer to real-world code than the numerical kernels most existing tools accept. We further expect our approach to be portable to other deductive verifiers.

We evaluate both analyses on a diverse set of benchmarks: the standard floating-point benchmark suite FPBench [17], Scala code mined from GitHub, and highly branching stress tests. Ablation studies confirm the contributions of each component. Comparing with three existing static analysers shows that Stainless scales to programs with significantly more branches, while matching or improving on the accuracy of the computed error bounds. The dynamic analysis finds divergence-triggering inputs for all benchmarks and for at least some it is able to confirm that static bounds are tight. We upload benchmarks, experimental results and an appendix as supplementary material.

In summary, this paper makes the following contributions:

- A scalable static analysis that computes tight error bounds for branching floating-point programs with an algorithm that grows linearly with the number of paths (Sec. 3).
- A guided dynamic analysis that finds divergence-triggering inputs throughout the program, rather than only for the most likely divergences (Sec. 4).
- An implementation in Stainless, a deductive verifier for Scala, that supports language features such as function calls, compound conditions, and algebraic data types (Sec. 5).
- An evaluation on a diverse set of benchmarks showing that Stainless matches the accuracy of pairwise comparison while scaling to programs on which existing tools time out (Sec. 6).

This paper addresses the challenge of branching floating-point code. It does not aim to handle other open challenges, such as analysing roundoff errors in loops and recursive functions [20, 52], analysing roundoff errors with a modular analysis [1, 52], or bounding relative errors [36]. These issues have so far only seen initial and partial solutions, and are orthogonal to this paper.

```
1  val dt = 0.1     // time step (s)
2  val brake = 4.0  // emergency braking
3
4  @roundoff @fuzz
5  def newAcc(d: Double, s: Double): Double =
6    require(5.0 <= d && d <= 15.0)
7    require(0.0 <= s && s <= 10.0)
8    val acc = net(d / 10 - 1, s / 5 - 1)
9    val newS = s + dt * acc
10   val newD = d - dt * newS
11   if newS * newS < (2 * brake) * (newD - 1)
12   then acc else -brake
13 def clamp(v: Double): Double =
14   if      v < -1.0 then -1.0
15   else if v > 1.0  then 1.0
16   else                  v
17
18 def net(x1: Double, x2: Double): Double =
19   val h1 = clamp(-3.3 * x1 + 3.2 * x2 - 1.1)
20   val h2 = clamp(-3.3 * x1 + 0.6)
21   val g1 = clamp(3.2 * h1 - 2.1 * h2 - 0.1)
22   val g2 = clamp(3.3 * h1 - 2.2 * h2 + 0.1)
23   -1.55 * g1 - 1.45 * g2 - 1.0
```

Fig. 1. Example of code with conditional statements: controller of a rover

2 Overview

We illustrate our approach on a controller for a one-dimensional version of the rover of Phan et al. [46, Sec. 2]; Fig. 1 shows its Scala implementation. Given the rover's distance to an obstacle (`d`) and its current speed (`s`) obtained from sensors, a neural network (`net`) proposes a new acceleration. A safety check then simulates the rover for one time step (lines 9 and 10) and, if it cannot stop in time before the obstacle, replaces the proposal with full braking (line 12). Assuming real-valued arithmetic computations, this check provably keeps the rover safe[1]. In the floating-point implementation, however, roundoff errors can flip its outcome: on a small set of inputs, the controller skips the emergency brake and applies the network's proposal instead, putting the rover at risk of collision.

Our static analysis—triggered by the `@roundoff` annotation—soundly bounds the worst-case absolute difference between the result of the floating-point and the corresponding real-valued execution, including all roundoff and divergence errors, when the executions start from the same valid inputs as specified in the `require` clauses. The dynamic analysis—triggered by `@fuzz`—finds concrete inputs on which divergence happens.

Analysing this example and branching real-world floating-point code in general poses two main challenges: the analysis needs to *scale* to relatively many branches and it needs to *accurately* bound the divergence errors to avoid false positives. In our example, to analyse `newAcc` Stainless first inlines all function calls (in the absence of an effective modular analysis). Every activation in the network (`clamp`) is a conditional, as is the safety check (line 11), and the floating-point execution may diverge at any of them. Most of these divergences are benign: the network computes a continuous function, so, as roundoff errors due to arithmetic are small here, taking the other branch of a `clamp` only slightly changes its output and the analysis should reflect this. The divergence at the safety check, however, replaces full braking with the network's proposal, with a potentially large resulting difference.

Existing approaches trade scalability for accuracy, or vice versa. Fluctuat [33] uses coarser abstractions and bounds the error of our controller's output by 80.6, although its output provably lies in $[-4, 2]$, so no two outputs can differ by more than 6. Tools using symbolic reasoning such as Rosa [20] and PRECiSA [52] compare every pair of paths that the two executions may take (162 in our example). Neither tool produces a result within 20 minutes.

Our approach, in contrast, bounds the total error of the controller by 0.150 in 30 seconds, and finds an input showing that this bound is tight (here d = 5.00000000000172, s = 5.656028409962584). This error is due entirely to divergence: an analysis that ignored it would report a bound of 7e−14, underestimating the actual error by twelve orders of magnitude.

[1]Safety verification of controllers is typically performed assuming real arithmetic for feasibility reasons.

Static Analysis. Our static analysis builds on two ideas: comparing *sets* of paths of the program instead of pairs of paths, and refining the comparisons when they lead to large divergence errors. The first idea rests on the observation that when real-valued and floating-point executions diverge, they part ways at a *single condition*. In our example, every input first reaches the condition $-3.3x_1+3.2x_2-1.1 < -1$ (line 14 called from line 19): if the two executions diverge there, one continues into one of the 54 paths of the `then` branch, the other into one of the 108 of the `else` branch. Rather than comparing these paths pair by pair, we compare the two sets of paths as a whole, bounding the difference between the largest value one set can produce and the smallest value the other can (and vice versa). Since the executions can diverge at any conditional, our approach solves one optimisation problem per conditional—161 in total in our example—to compute the overall divergence error, whereas existing techniques formulate one per unordered pair of paths, which in our example are 13 041.

This efficiency, however, sacrifices accuracy. A set comparison ignores the conditions needed to reach each path, so it also pairs paths that can never be taken on the same input, and on our network, it yields a loose bound of 40.4. For instance, it may pair a path where a neuron is clamped at -1 in floating-point arithmetic with one where the same neuron is clamped at 1 in real arithmetic, although this difference cannot actually occur due to roundoff errors being provably too small.

Our second idea is to recover accuracy through refinement. We take the pair of path sets that yields the largest bound, since it determines the overall bound, and split one of the two path sets at its first condition. The same paths are still covered, but in two smaller groups, each under the additional constraint of the condition we split on. Consider again the first neuron (line 19), and inputs on which one execution clamps it at -1.0 while the other does not. Before refinement, the analysis does not know whether this other execution leaves the neuron unclamped or clamps it at 1, so it must account for both. Refinement separates these two cases.

In our evaluation, a number of refinement steps linear in the number of conditions suffices to obtain bounds close to those of pairwise comparison. On this example, 322 refinement steps, twice the number of conditions, bring the bound down from 40.4 to 0.150.

If a bound seems too loose, the user can increase the timeout, giving Stainless more time to tighten it. An upper bound alone, however, does not tell the user how close it is to the ground truth, and thus whether more time would help. Since this bound is an overapproximation, it may also warn of divergences that cannot actually occur. Furthermore, static analysis provides no input that triggers a divergence and possibly causes a large error.

Dynamic Analysis. To make the bound actionable, we complement the static analysis with a dynamic analysis that identifies program inputs that trigger divergence and runs the program on them in both floating-point and high-precision arithmetic (as a proxy for real-valued execution [16]) to compute the total error on the result. The largest error it observes is an empirical lower bound on the true worst-case error, which shows the user how far the static bound may be from the truth.

As for the static analysis, the main challenge is that divergence-triggering inputs are extremely sparse, so unguided search, such as random sampling, rarely finds any. Finding a large error in heavily branching code moreover requires exercising as many divergences as possible, including those at deeply nested branches.

Our key idea is to let the static analysis guide the search and to use floating-point specific fuzzing to maximise divergence errors. For each condition, the optimiser returns not only a bound but also an interval containing the input at which this bound is reached. Since the bound is an upper bound, the interval itself may not contain divergence-triggering inputs, so our dynamic analysis performs a bisection search in the vicinity of this interval. Once it finds such a divergence-triggering input, the analysis mutates it randomly to explore its neighbourhood, keeping the new inputs that also trigger divergence, and repeats this process until the user's time budget is exhausted.

On our network, the dynamic analysis finds 74 754 unique divergence-triggering inputs in 30 seconds, with at least one in every interval returned by the static analysis. Together, they trigger divergence at every condition of the network where divergence can actually occur, and yield a lower bound of 0.150, thus confirming that the static analysis bound is tight.

Stainless reports upper and lower bounds on errors not only for the global result of the function, but also for each individual conditional. It also writes all divergence-triggering inputs it finds to a separate file, grouped by condition and by pair of paths. The aim is to provide as much information as possible to help the user to repair their program, if needed. In our example, the user may then add a safety margin to the check on line 11, or limit the acceleration returned by the network.

3 Static Analysis

We first recall how existing work bounds the error of a single and nested conditional [51], introducing our notation along the way, before explaining our approach. We focus on functional programs, since Stainless translates imperative code into functional code.

3.1 Background and Notation: Comparing Every Pair of Paths

Consider an expression

$$t(x) := \texttt{if } c_1(x) \le c_2(x) \texttt{ then } f(x) \texttt{ else } g(x)$$

where x is a vector of inputs ranging over a domain D, and c_1, c_2, f, and g are straight-line arithmetic expressions. Writing $[\![\cdot]\!]_{\mathbb{R}}$ and $[\![\cdot]\!]_{\mathbb{F}}$ for the real-valued and floating-point semantics, we aim to bound

$$\max_{x \in D} \; err_t(x), \quad \text{where } err_t(x) := \left| [\![t(x)]\!]_{\mathbb{R}} - [\![t(x)]\!]_{\mathbb{F}} \right|. \tag{1}$$

Evaluating c_1 and c_2 in floating-point arithmetic incurs rounding, so when $c_1(x)$ and $c_2(x)$ are close, $[\![t(x)]\!]_{\mathbb{F}}$ and $[\![t(x)]\!]_{\mathbb{R}}$ may take different branches on the same input. For a given input x, the error $err_t(x)$ thus splits into four cases:

$$err_t(x) = \begin{cases} \left|[\![f(x)]\!]_{\mathbb{R}} - [\![f(x)]\!]_{\mathbb{F}}\right| = err_f(x) & \text{when } [\![c_1(x) \le c_2(x)]\!]_{\mathbb{R},\mathbb{F}} \\ \left|[\![g(x)]\!]_{\mathbb{R}} - [\![g(x)]\!]_{\mathbb{F}}\right| = err_g(x) & \text{when } [\![c_1(x) > c_2(x)]\!]_{\mathbb{R},\mathbb{F}} \\ \left|[\![f(x)]\!]_{\mathbb{R}} - [\![g(x)]\!]_{\mathbb{F}}\right| & \text{when } [\![c_1(x) \le c_2(x)]\!]_{\mathbb{R}} \text{ and } [\![c_1(x) > c_2(x)]\!]_{\mathbb{F}} \\ \left|[\![g(x)]\!]_{\mathbb{R}} - [\![f(x)]\!]_{\mathbb{F}}\right| & \text{when } [\![c_1(x) > c_2(x)]\!]_{\mathbb{R}} \text{ and } [\![c_1(x) \le c_2(x)]\!]_{\mathbb{F}} \end{cases} \tag{2}$$

The first two cases are roundoff errors of straight-line code, for which tools such as Daisy [18] and FPTaylor [51] compute tight bounds. We write ε_f and ε_g for these bounds, *i.e.*, $err_f(x) \le \varepsilon_f$ and $err_g(x) \le \varepsilon_g$ for all $x \in D$.[2] The last two cases are divergence errors: they compare *different* expressions in different semantics, which these tools cannot handle.

For the third case, with the triangle inequality we can separate the real-valued difference between the two branches from the roundoff error of the branch taken in floating point semantics:

$$\left|[\![f(x)]\!]_{\mathbb{R}} - [\![g(x)]\!]_{\mathbb{F}}\right| \le \left|[\![f(x)]\!]_{\mathbb{R}} - [\![g(x)]\!]_{\mathbb{R}}\right| + \left|[\![g(x)]\!]_{\mathbb{R}} - [\![g(x)]\!]_{\mathbb{F}}\right| \tag{3}$$

$$\le \left|[\![f(x) - g(x)]\!]_{\mathbb{R}}\right| + \varepsilon_g. \tag{4}$$

The second term of Eq. (3) is bounded by ε_g, while the first term is real-valued, but its path constraint $[\![c_1(x) > c_2(x)]\!]_{\mathbb{F}}$ still involves floating-point semantics. Let ε_{c_1} and ε_{c_2} be the roundoff error bounds of c_1 and c_2. The floating-point value of c_1 differs from its real value by at most ε_{c_1}, and likewise for c_2. The floating-point condition $[\![c_1(x) > c_2(x)]\!]_{\mathbb{F}}$ can therefore only hold if the real values of both

[2] Note that these tools report any possible floating-point special value (Infinity or Not a Number) as an error, and hence (in their absence), comparison operators, on the same operands give the same result in $\mathbb{R}$ and $\mathbb{F}$.

sides are within $\varepsilon_{c_1} + \varepsilon_{c_2}$ of each other, i.e., if $[\![c_2(x) - c_1(x)]\!]_{\mathbb{R}} < \varepsilon_{c_1} + \varepsilon_{c_2}$. We call this constraint a *relaxed condition*: it is a real-valued overapproximation of the floating-point condition, i.e., it holds on every input where the floating-point condition may hold. Combining it with the real-valued path condition $[\![c_1(x) \le c_2(x)]\!]_{\mathbb{R}}$, which we rewrite as $0 \le [\![c_2(x) - c_1(x)]\!]_{\mathbb{R}}$, yields a purely real-valued optimisation problem for the third case:

$$\max_{x \in D} \quad \left|[\![f(x) - g(x)]\!]_{\mathbb{R}}\right| \quad \text{s.t.} \qquad 0 \le [\![c_2(x) - c_1(x)]\!]_{\mathbb{R}} < \varepsilon_{c_1} + \varepsilon_{c_2}. \tag{5}$$

The fourth case is symmetric, with ε_f in place of ε_g, and the same optimisation objective. Merging the two problems gives us the optimisation problem for the divergence error δ of t:

$$\delta = \max_{x \in D} \quad \left|[\![f(x) - g(x)]\!]_{\mathbb{R}}\right| \quad \text{s.t.} \qquad -(\varepsilon_{c_1} + \varepsilon_{c_2}) \le [\![c_2(x) - c_1(x)]\!]_{\mathbb{R}} < \varepsilon_{c_1} + \varepsilon_{c_2}. \tag{6}$$

Note that since the constraint confines x to the thin divergence region and the objective is a single expression in x, this problem preserves the dependencies between variables that abstract-interpretation-based techniques, such as Fluctuat, lose.

Overall, bounding the error of t requires solving five problems: four roundoff error bounds ($\varepsilon_f, \varepsilon_g, \varepsilon_{c_1}, \varepsilon_{c_2}$), and one optimisation problem for divergence (δ). The first two cases of Eq. (2) are then bounded by ε_f and ε_g, and the last two by $\delta + \varepsilon_g$ and $\delta + \varepsilon_f$, so for all inputs, the error of t is bounded by

$$\delta + \max(\varepsilon_f, \varepsilon_g). \tag{7}$$

This approach generalises directly to nested conditionals. We view the control flow of a function as a tree whose inner nodes are conditions and whose leaves are straight-line expressions, see an illustration in Fig. 3a. Each path p from the root to a leaf has a leaf expression f_p and a path condition φ_p, the conjunction of the guards along it.

Consider the purposefully simple example in Fig. 2. The program has four paths, two on line 3 and two on line 5: -1 when $y \le -1$, y when $-1 < y \le 0$, y^2 when $0 < y \le 1$, and y when $y > 1$. On a given input, the real-valued and floating-point executions follow paths p and q. When $p = q$, the error is the roundoff error of f_p, which we bound by ε_p as before. When $p \neq q$, the triangle inequality again separates the real-valued difference between the two leaves from the roundoff error of the floating-point one:

```
1 val y = x * x - 2.0
2 if (y <= 0) {
3   if (y <= -1) -1.0 else y
4 } else {
5   if (y <= 1) y * y else y
6 }
```

Fig. 2. A continuous piecewise-defined function.

$$\left|[\![f_p(x)]\!]_{\mathbb{R}} - [\![f_q(x)]\!]_{\mathbb{F}}\right| \le \left|[\![f_p(x) - f_q(x)]\!]_{\mathbb{R}}\right| + \varepsilon_q.$$

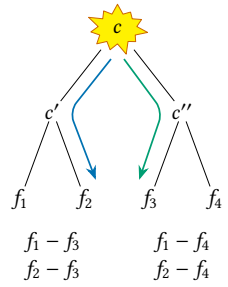
Bounding the first term requires one optimisation problem per unordered pair of paths:

$$\delta_{\{p,q\}} = \max_{x \in D} \quad \left|[\![f_p(x) - f_q(x)]\!]_{\mathbb{R}}\right| \quad \text{s.t.} \qquad \tilde{\varphi}_p(x) \wedge \tilde{\varphi}_q(x) \tag{8}$$

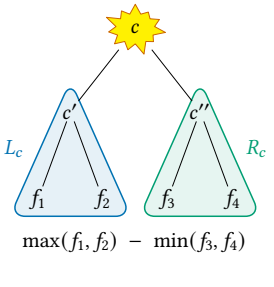
where $\tilde{\varphi}_p$ relaxes each guard of φ_p by its error bound, as in Eq. (6). Generalising Eq. (7), the error of the function is bounded by

$$\max_{p \neq q} \delta_{\{p,q\}} + \max_p \varepsilon_p.$$

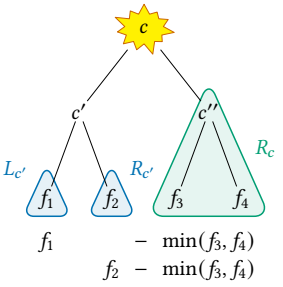
For a function with n paths, this requires n roundoff error bounds, but $n(n-1)/2$ optimisation problems for divergence, one per unordered pair of paths (as in the approach of Rosa and PRECiSA). On the example, pairwise comparison solves six problems. Comparing -1 and y on line 3 confines y to a thin band around -1, where the function is continuous, so the bound is small. Comparing -1 (line 3) and y^2 (line 5) requires both $y \le -1 + \varepsilon_y$ and $y > -\varepsilon_y$, which contradict each other, so this problem is infeasible and the pair does not contribute to the bound. The remaining pairs behave analogously.



(a) Pairwise comparison: maximise the difference between every pair of paths; requires $O(n^2)$ optimisation problems for n paths.



(b) Subtree comparison: for each condition maximise the difference between its two subtrees; requires $O(n)$ optimisation problems.



(c) Refined subtree comparison: requires $O(n+r)$ optimisation problems for r refinement steps.

Fig. 3. Three ways to bound the divergence error.

3.2 Our Approach: Subtree Comparisons

Comparing pairs of paths is affordable for small functions, but does not work well already for code as small as our example in Fig. 1. One contributing factor is also that to obtain tight bounds, analysers inline function calls. The error of a callee depends on the ranges and errors of its arguments, which differ at each call site, and inlining preserves the correlations between variables that a separate analysis would lose [1]. Every inlined conditional thus multiplies the number of paths.

To scale to such programs, we compare subtrees instead of pairs of paths, which requires a number of optimisation problems linear, rather than quadratic, in the number of paths. Fig. 3b illustrates this idea.

Our approach relies on the observation that whenever the real-valued and floating-point executions diverge, they part ways at a single condition c, the lowest common ancestor of their paths in the tree (Fig. 3b). Writing L_c and R_c for the sets of paths through the left and right subtrees of c, one execution follows a path $p \in L_c$ in the left subtree, and the other a path $q \in R_c$, in the right subtree. For any input x in the input domain, the real-valued difference $\left[\!\left[f_p(x) - f_q(x) \right]\!\right]_{\mathbb{R}}$ is at most the difference between the worst possible path in the left subtree for that input $\max_{p' \in L_c} \left[\!\left[f_{p'}(x) \right]\!\right]_{\mathbb{R}}$ and the worst possible path in the right subtree $\min_{q' \in R_c} \left[\!\left[f_{q'}(x) \right]\!\right]_{\mathbb{R}}$. Taking the absolute value of that difference gives:

$$\begin{aligned} \left| \left[\!\left[f_p(x) - f_q(x) \right]\!\right]_{\mathbb{R}} \right| &= \max \left\{ \left[\!\left[f_p(x) \right]\!\right]_{\mathbb{R}} - \left[\!\left[f_q(x) \right]\!\right]_{\mathbb{R}} \;,\; \left[\!\left[f_q(x) \right]\!\right]_{\mathbb{R}} - \left[\!\left[f_p(x) \right]\!\right]_{\mathbb{R}} \right\} \\ &\leq \max \left\{ \max_{p' \in L_c} \left[\!\left[f_{p'}(x) \right]\!\right]_{\mathbb{R}} - \min_{q' \in R_c} \left[\!\left[f_{q'}(x) \right]\!\right]_{\mathbb{R}}, \max_{q' \in R_c} \left[\!\left[f_{q'}(x) \right]\!\right]_{\mathbb{R}} - \min_{p' \in L_c} \left[\!\left[f_{p'}(x) \right]\!\right]_{\mathbb{R}} \right\} \end{aligned} \tag{9}$$

In other words, we are comparing subtrees instead of individual paths. This bound has two advantages. First, it depends only on the condition c where the paths part ways, not on the paths p and q themselves. A single bound on the divergence error per condition therefore covers every pair of paths diverging there. A tree with n paths has $n-1$ conditions, so we need only $n-1$ bounds, instead of one per pair of paths, which is quadratic in n.

Second, although this bound contains more terms, the total number of terms across all bounds never exceeds that of all pairwise comparisons. Writing n for the number of paths in the tree and h for its height, subtree comparisons produce $O(n \cdot h)$ terms in total, compared to $O(n^2)$ for pairwise comparisons.

Returning to the example in Fig. 2, consider a divergence at $y \leq 0$ (line 2). The left subtree has leaf expressions -1 and y (line 3), and the right subtree y^2 and y (line 5). Instead of comparing the four pairs of paths, we bound the differences $\max(-1, y) - \min(y^2, y)$ and $\max(y^2, y) - \min(-1, y)$ and take the max of those.

Comparing subtrees comes however at a cost in precision. When comparing two paths p and q in the pairwise approach, Eq. (8) constrains x to both relaxed path conditions $\tilde{\varphi}_p(x)$ and $\tilde{\varphi}_q(x)$. A subtree comparison, in contrast, ranges over all paths of each subtree at once, so it cannot add the conditions below c in the constraints of the optimisation problem. In Fig. 2, this means that the nested conditions $y \leq -1$ (line 3) and $y \leq 1$ (line 5) cannot be used as constraints when maximising the bound.

Putting the pieces together, for each condition c in the program, we solve two optimisation problems (δ_c and δ'_c), one per argument of the right-hand side of Eq. (9). Both problems are constrained by (i) the path condition $\varphi_c(x)$ leading to c, and (ii) the constraints on which inputs can possibly diverge. For a condition c of the form $c_1(x) \leq c_2(x)$, we thus solve

$$\begin{aligned} \delta_c = \max_{x \in D} \quad & \max_{p \in L_c} [\![f_p(x)]\!]_{\mathbb{R}} - \min_{q \in R_c} [\![f_q(x)]\!]_{\mathbb{R}} \qquad \delta'_c = \max_{x \in D} \quad \max_{q \in R_c} [\![f_q(x)]\!]_{\mathbb{R}} - \min_{p \in L_c} [\![f_p(x)]\!]_{\mathbb{R}} \\ & \text{s.t.} \quad \varphi_c(x) \wedge -(\varepsilon_{c_1} + \varepsilon_{c_2}) \leq [\![c_2(x) - c_1(x)]\!]_{\mathbb{R}} < \varepsilon_{c_1} + \varepsilon_{c_2} \end{aligned} \tag{10}$$

and the error of the function is bounded by

$$\max_c \max(\delta_c, \delta'_c) + \max_p \varepsilon_p.$$

3.3 Refining the Bounds

Subtree comparison drops the path conditions below the diverging condition, which can make the bound much looser. In Fig. 2, to bound the divergence error at the condition $y \leq 0$ (line 2), we maximise $\max(-1, y) - \min(y^2, y)$ and $\max(y^2, y) - \min(-1, y)$ within the band $-\varepsilon_y < y \leq \varepsilon_y$, *i.e.*, $y \approx 0$. The first problem yields a very small bound, since both sides of the subtraction are close to 0. The second, however, yields a bound close to 1. Since the function is continuous at $y = 0$, the actual roundoff error caused by a divergence there is tiny, so this bound is far from tight.

This looseness comes from the leaf `-1.0` that is only reachable when $y \leq -1$. This condition is incompatible with the band around 0, but because we drop path conditions below the diverging condition, the bound still includes this leaf.

To recover the tightness lost by subtree comparison, we refine the largest bounds it produces. We illustrate this process in Fig. 3c. Suppose the largest bound is at condition c, and let c' be the condition of its left subtree, which splits L_c into $L_{c'}$ and $R_{c'}$. Each path in L_c continues into either $L_{c'}$ or $R_{c'}$, so comparing L_c with R_c is equivalent to comparing (i) $L_{c'}$ with R_c and (ii) $R_{c'}$ with R_c. On its own, this split does not change the bound. It does, however, let us constrain the first comparison by the relaxation of c', and the second by the relaxation of its negation, recovering part of the path conditions that subtree comparison dropped.

On the example in Fig. 2, a single refinement step eliminates the pessimistic bound. We split $\max(y^2, y) - \min(-1, y)$ into two subproblems. The first, $\max(y^2, y) - (-1)$, is constrained by both the diverging region $y \approx 0$ **and** the relaxed condition of line 3, $y \leq -1 + \varepsilon_y$. These constraints contradict each other, so the problem is infeasible and does not contribute to the overall bound. The second, $\max(y^2, y) - y$, is constrained by $y \approx 0$ and the relaxed negation of the same condition, $y > -1 - \varepsilon_y$, and yields a bound very close to 0. A divergence at $y \leq 0$ (line 2) thus contributes only a small error, matching the bound of pairwise comparison.

Our approach applies refinements on the largest bound, and proceeds in decreasing order. In the limit, continuing to apply refinements until there are no more bounds to be refined results exactly in a pairwise path comparison, with a quadratic number of optimisation problems in the

worst case. In practice, a small number of refinement steps suffices, for two reasons. First, many subtree comparisons are infeasible, because no input near the diverging condition can reach both subtrees (Fig. 5 in the appendix). A single infeasible problem then rules out every pair of paths between the two subtrees at once, where pairwise comparison checks each pair separately, and such bounds never need refinement. Second, we have observed empirically that refinement has diminishing returns: refining the few largest bounds already brings the result close to that of pairwise comparison (Fig. 6 in the appendix). Our implementation therefore allows a number of refinement steps linear in the number of paths, so that the total number of optimisation problems remains linear as well. In our evaluation (Sec. 6), we use one refinement step per condition, and our ablation study shows that, with the same total time, refinement then matches the bounds of pairwise comparison on all benchmarks, whenever it finishes.

3.4 Solving the Optimisation Problems

Solving these optimisation problems requires care: most optimisers compute in floating point, so their own roundoff errors could make the bounds unsound. Two kinds of solvers avoid this: SMT solvers, which reason exactly over the reals, and branch-and-bound solvers, which rely on interval arithmetic. SMT solvers such as Z3 [24] and cvc5 [5] are precise and are what Rosa uses, but in our experience they often time out on the constraints that encode our divergence regions. Branch-and-bound solvers split the search space into intervals, bound the objective on each interval with outward-rounded interval arithmetic, and discard the intervals that cannot satisfy the constraints or cannot contain the maximum. The result is an interval guaranteed to contain the maximum.

Our analysis is sound with any solver that returns a rigorous upper bound on the maximum; in our implementation, we use the branch-and-bound solver Ibex [44]. Ibex is based on contractor programming [10]: contractors shrink each interval to the part that may satisfy the constraints, so the search reaches the constraint set quickly, even when it is as thin as our divergence regions. We also tried Kodiak [50] and Gelpia [51], the branch-and-bound solvers used by PRECiSA and Seesaw [22] (an unsound tool computing divergence errors), but in preliminary tests neither matched Ibex on our problems. Ibex also degrades gracefully: when it runs out of time, it still returns a sound interval around the maximum, namely the tightest one it has found so far. A timeout thus makes our bounds looser, but never unsound.

Our implementation uses two optimisations, neither of which affects soundness. First, before optimising a problem, we check whether its constraints are satisfiable, which quickly rules out divergences that cannot occur. Second, when Ibex solves an optimisation problem, we also give it the largest bound computed so far that can no longer be refined. Ibex can then stop as soon as it proves that the maximum of the current problem does not exceed this bound. Since this largest bound is final, the overall error bound is at least as large, so stopping early does not change it.

Finally, besides the bound, Ibex returns a small interval around the input with the largest objective value it found. This interval is where our dynamic analysis starts its search (Sec. 4).

4 Dynamic Analysis

While static analysers provide guarantees by computing a sound upper bound on the worst-case error, they do not make this bound actionable. In particular, the user cannot tell whether a divergence reported by the analyser is a false alarm, how far the bound is from the true worst-case error, or which inputs trigger the divergence. We therefore design a dynamic analysis that finds inputs triggering divergence and derives an empirical lower bound on the program's worst-case error.

Our analysis builds on shadow execution [16] for estimating the roundoff error of floating-point arithmetic. Shadow execution runs the program in floating-point and a higher-precision arithmetic side by side, and considers the difference in the result to be an estimate of the true roundoff error on

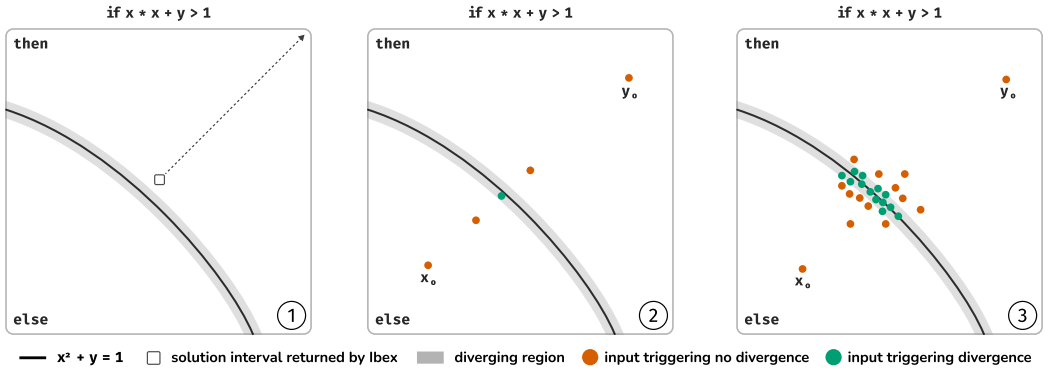


Fig. 4. Dynamic analysis overview: ① widen the solution interval returned by Ibex, ② bisect the widened interval to find a divergence-triggering input, and ③ apply mutation-based fuzzing to find more inputs.

the particular input. The higher-precision arithmetic execution replaces exact real-valued execution, since the latter can be prohibitively expensive (e.g. when implemented with rationals) or infeasible in the case of transcendental functions. Vanilla shadow execution [16] typically assumes that both the floating-point and the higher-precision version of the program take the same path.

Extending this analysis to cover divergence errors faces two challenges. First, divergence-triggering inputs are rare: they typically lie in narrow regions of the input space close to branch boundaries, so unguided random sampling is unlikely to find them within a practical time budget. Second, nested conditionals give rise to several possible divergences, each of which can induce a different error magnitude; since the lower bound is only as tight as the worst divergence found, the analysis must exercise as many of them as possible.

Our core idea to find divergence-triggering inputs is to use the results of the static analysis to narrow down the regions where they may lie. For each possible divergence, when Ibex maximises the difference between two subtrees or two paths, it also returns an interval containing the input at which this maximum is reached. Since one of the constraints given to Ibex overapproximates the divergence region, this interval is likely to lie inside or close to the actual divergence region. However, because the constraint is only an overapproximation, the interval may fall entirely outside the actual region, so sampling within it alone is not enough.

We therefore widen each interval returned by Ibex as illustrated in Fig. 4 ①. Widening makes it more likely that the interval contains divergence-triggering inputs, but also makes it too large to search exhaustively. To find a divergence-triggering input efficiently, we use a bisection search to approach the boundary between the `then` and and`else` branch of the condition, since divergence-triggering inputs necessarily lie close to this boundary (Fig. 4 ②).

Our goal, however, is not only to find one divergence-triggering input but to maximise the resulting error, which requires finding many such inputs. Rather than repeating the bisection, we exploit the observation that divergence-triggering inputs tend to lie close to one another by randomly mutating the input found by bisection to discover neighbouring divergence-triggering inputs (Fig. 4 ③). We detail the bisection and mutation steps below.

② **Bisect towards the branch boundary.** Inspired by the approach of Chiang et al. [13], we uniformly sample up to N inputs in the widened interval until we find two inputs x_0 and y_0 such that the original program takes different paths on them. If x_0 or y_0 triggers divergence, that is, if the shadow program takes a different path on one of these inputs than the original program, we have found a divergence-triggering input. Otherwise, we check whether the midpoint of x_0 and

y_0 triggers divergence and, if not, use it to replace whichever of x_0 and y_0 takes the same path in the original program. Repeating this process ends either with a divergence-triggering input or with x_0 and y_0 being adjacent floating-point values. The latter can happen if the conditional cannot diverge, or if the widened interval still lies entirely outside the divergence region. In that case, or if no such pair is found among the N samples, we return to step ① with a larger widening factor. We abandon the search for this conditional when the sampling or time budget is exhausted, or when the widened interval covers the entire input domain.

③ **Mutate divergence-triggering inputs.** The neighbourhood of a divergence-triggering input is likely to contain more of them, so we explore it with random local mutations. We flip each bit of the input's binary representation with a probability that decreases geometrically from the least significant bit: $1/2$ for the last bit, $1/4$ for the penultimate one, and so on. Small mutations are thus the most likely, which keeps the search close to known divergence-triggering inputs, while occasional larger ones let it escape regions where such inputs are sparse. If a mutated input triggers divergence, we keep it and continue the search from it.

Since Ibex takes the function's precondition into account, the inputs it returns satisfy it. Widening and mutation can still produce inputs that do not, which we discard.

Without the intervals returned by Ibex, bisection alone finds only the most likely divergences and may miss deeply nested ones. This problem grows with nesting depth: reaching a divergence in an inner conditional also requires satisfying the enclosing conditions, so each divergence becomes less likely to be triggered. Sampling within the intervals returned by Ibex addresses this, because the static analysis identifies all possible divergence regions regardless of how likely they are to be hit. Restricting the search to these intervals rather than the whole input domain also reduces the number of samples and bisection steps needed to find divergence-triggering inputs, which matters for programs that are expensive to run. Mutation, in turn, increases the number of divergence-triggering inputs found in each region, bringing the empirical lower bound closer to the true worst-case error. Sec. 6.6 evaluates the contribution of each step in an ablation study.

5 Implementation

We implement our approach in Stainless, a general-purpose deductive verifier for Scala. Due to prior advancements introduced in [30], its floating-point support can already handle checking that a function's output is finite (not infinite or Not a Number) via the SMT-LIB floating-point theory.

Stainless first normalises the program into a control-flow tree. The inner nodes of the tree are comparisons between straight-line arithmetic expressions and leaves are straight-line arithmetic expressions. Stainless also inlines function calls and let bindings, and hoists conditional expressions, among other transformations. All roundoff error computations of straight-line expressions are then discharged in a single batch via Daisy [18]. We use Daisy because it is reasonably fast, but other straight-line analysers, such as FPTaylor [51], could be used instead. Stainless supports both `Float` and `Double` types and all Scala standard library elementary functions that Daisy supports. Since the static and dynamic analyses are both included in Stainless, users can verify the roundoff error with other functional properties, and Stainless reports all results together.

Analysis Timeout. The user sets a single timeout for the entire analysis. The static analysis returns a sound bound within it, since Ibex degrades gracefully (Sec. 3.4). Having an adjustable timeout per optimisation problem would make the total running time unpredictable as the number of problems is program-dependent. Stainless reports a timeout in only two cases: (i) on programs with very many paths when normalisation exceeds the timeout; and (ii) when the remaining time per optimisation problem is less than 1 ms, because it is less than the overhead of starting Ibex.

Once Daisy has computed the straight-line roundoff bounds, Stainless splits the remaining time equally between subtree comparisons and refinement. Within each half, each problem receives the remaining time divided by the number of problems left; unused time passes to the next problem, and leftover time from the first half goes to refinement. Stainless performs a fixed number of refinement steps per condition (one by default) and stops early if the largest bound can no longer be refined; on harder problems, the user can increase this number to obtain tighter bounds. The dynamic analysis receives the same timeout as the static analysis, and searches for inputs until it runs out.

Dynamic Analysis. We implement our approach for the dynamic analysis as a Scala compiler plugin. The plugin generates a shadow copy of the program that replaces 32- and 64-bit floating-point operations with high-precision ones, using the MPFR library [27] with 128-bit precision to approximate real arithmetic, and instruments both programs to record every branch they take. We compute the error of an execution as the absolute difference between the outputs of the two programs. This difference also includes the shadow program's own roundoff errors, but at 128-bit precision these are much smaller than the 32- and 64-bit errors we measure.[3]

At the end of this process, the dynamic analyser reports all the divergence-triggering inputs it found, grouped by the condition at which divergence occurs, together with the paths that the original and shadow programs take on each input. This lets the user see which inputs trigger each divergence and focus on the ones they want to address.

Supporting Main Features of Scala. Stainless splits conditions that combine comparisons with Boolean operators into nested conditionals. For example, it rewrites `if (c1 && c2) t else e` into `if (c1) { if (c2) t else e } else e`.

Algebraic data types and case classes (similar to structs in C) are Scala's building blocks for organising floating-point data, such as 2D and 3D vectors. When a function parameter is a case class, we introduce a variable for each of its fields, recursively for nested case classes, and substitute these variables for the corresponding field accesses in the function body. Since function calls are inlined beforehand, we can then statically resolve the remaining field accesses on case class values constructed in the body (*e.g.,* `Vec(a, b).x` becomes `a`), so that the analysed expressions contain only floating-point variables and operations. As PRECiSA 4 does for records [52], when a function returns a case class, Stainless analyses and reports the error of each field separately, while exploiting their shared control flow (*e.g.,* when pruning unreachable paths).

We support sum types (enums) such as `Option`, and treat Booleans as a sum type with two constructors. When a function returns a sum type, we first generate a verification condition that checks whether divergence can cause the real and floating-point executions to return different constructors (*e.g.,* `Some` and `None`). We then generate a verification condition for each constructor with floating-point fields, and analyse only the paths that return it. When a function takes a sum type as input, we run the static analysis once per constructor. The dynamic analysis handles them analogously: it reports its results per field for case classes and, for sum types, the number of inputs on which the two executions return different constructors.

While some analysers for straight-line code support mixed integer and floating-point arithmetic, rigorous real-valued constraint solvers do not. We therefore treat integer inputs involved in floating-point computations as reals. Assuming no overflow, this soundly, if loosely, overapproximates the divergence error they induce. Stainless already checks for integer overflows via SMT queries. We do not yet support integer division, which rounds its result. For conditions that do not depend on floating-point values, we skip the divergence analysis, since no divergence can occur.

[3] These errors can be made arbitrarily small by increasing the precision further. In our case, the Java Native Interface crashes for any precision higher than 128 bits.

6 Evaluation

We evaluate our enhanced implementation of Stainless, with Daisy [18] as the backend for straight-line code and Ibex for constrained optimisation. We answer four research questions.

RQ1 On branching floating-point programs, how do the error bounds of Stainless compare with the state-of-the-art tools PRECiSA, Fluctuat, and Rosa?
RQ2 How does Stainless scale as the amount of branching increases?
RQ3 To what extent can Stainless bound roundoff errors in real-world floating-point programs?
RQ4 How effectively does Stainless find inputs that trigger divergence?

We consider only benchmarks with direct or indirect branching. Without it, the evaluation would reduce to comparing Daisy with other straight-line backends, which can be found elsewhere [18].

6.1 Benchmarks

FPBench. We select all 5 benchmarks from the standard benchmark suite for comparing floating-point analysers FPBench [17] that are loop-free and contain conditional statements. However, none of them calls auxiliary functions and only one contains a nested conditional.

Real-world user benchmarks mined from GitHub. We sample 300 loop-free functions with at least one conditional statement from a recent data set of real-world floating-point code from GitHub [31]. The vast majority of the functions rely on external libraries that are unsupported by Stainless (108), unbounded data structures or recursion (116), or features that it could not parse prior to our changes (57, *e.g.,* macros, concurrency, IO). Excluding these leaves 19 benchmarks, and adding their callers from the same repositories yields 6 more, for a total of 25 benchmarks.

Case study. The two suites above focus on isolated functions from disjoint contexts. To assess whether Stainless can analyse entire modules of a code base, we conduct a case study on the statistics module of Algebird [53]: an abstract algebra library developed at Twitter (now X) and one of the most popular Scala repositories on GitHub (2.3k stars as of Oct 2026). The module computes statistics over large datasets in a distributed setting. Each machine computes partial statistics on its share of the data and these statistics are then merged. Merging is order-independent over the reals but not in floating point, so the module's tests only check that merging in different orders gives the same result, up to a hand-picked tolerance. We analyse these merge functions, and benchmarks inspired by Algebird's test suite, and bound the roundoff error. We translate each benchmark into the input language of each other tool when that language can express it, and report unsupported otherwise.

Stress tests. Finally, to obtain heavily branching code, we use four families of stress tests whose number of paths grows with the number of unrolled loop iterations or recursive calls of general algorithms. `mlp` are neural networks with clipping activations, which we trained with PyTorch v2.14.1 [45] to classify points on a 2D circle [49], `bouncingBall` is a physics simulation, `rayMarching` a ray-tracing algorithm, and `stepResponseArea` computes the area under a controller's step response.

6.2 Experimental Setup

We compare Stainless with PRECiSA v4.2.0 [52], Fluctuat [33][4], and Rosa [20].[5] For a fair comparison, we consider Fluctuat in two configurations: without interval subdivision (fast, but loose bounds) and with 10k subdivisions split in input variables (slow, but tighter bounds). We found no option in the other tools with a comparable effect on the tradeoff between accuracy and running time. We run

[4] Fluctuat is closed source; we used the version the authors sent us.
[5] https://github.com/malyzajko/rosa/tree/68e58b8c2f334ddde0f7647dc2d9c9ecc30309af

PRECiSA without the `--assume-stability` flag[6] and Rosa with default settings. All configurations account for divergence. To show that the effectiveness of our approach does not depend on the choice of backend, we report both the total error and the roundoff error ignoring divergence. All reported bounds are absolute. We run Stainless with timeouts of (i) 5 seconds, to show what it outputs under time pressure; (ii) 30 seconds, representing a typical user scenario; and (iii) 600 seconds, to assess capabilities under a generous time budget. Since the other tools do not return intermediate results when interrupted, we give them a 10-minute timeout, and 20 minutes on the stress tests, and report t/o when it expires. We execute each benchmark 5 times. We report the execution with the median bound to avoid outliers (degenerate bounds, timeouts, or infinities) from skewing the results. We report the analysis time measured by each tool to exclude startup and parsing overhead.

Comparison Exclusions. Real2Float [40] claims to support conditional statements, but in our experiments, it raises a runtime error on every such program. Seesaw [22] also supports conditionals, but it builds on Satire [21], which trades soundness for scalability by discarding part of the error term. We also found that its implementation ignores the roundoff error of the inputs. A fair comparison with sound tools is therefore difficult. Despite these underapproximations, Seesaw times out on roughly the same benchmarks as Rosa (cf. Table 5 in the appendix). For these reasons, we exclude Real2Float and Seesaw.

Machine Environment. We run Stainless with OpenJDK 25.0.4.1 and Scala 3.10.0 on a machine with a 12-core AMD Ryzen 9 5900X CPU and 128 GB of RAM, running Ubuntu 24.04. Unless stated otherwise, we run all experiments sequentially with BenchExec [7], pinning each run to a single core and limiting it to 32 GB of memory.

6.3 RQ1: Comparison with State-of-the-Art

Table 1 shows the error bounds computed by each tool on FPBench, the case study functions and tests, and the stress tests, for the benchmarks on which at least one tool returns a finite bound. Tables 6 to 9 in the appendix show more detailed results, including roundoff errors for straight-line code without divergence errors. All tools often bound the roundoff error along individual paths more tightly than Daisy, yet compute much looser total bounds, so Stainless's advantage comes from how it bounds the divergence error. On 9 of the case-study benchmarks, the roundoff error along individual paths contributes even more to Stainless's bound than the divergence error.

On FPBench, Stainless's bounds are on par with Rosa's, except on `smartRoot`, where Daisy reports a possible square root of a negative number, and Stainless returns a possible NaN. Stainless's finiteness check shows that this alarm is spurious when given more than 30 seconds. On the case study, Stainless computes bounds at least 8 orders of magnitude tighter than existing tools on all benchmarks except `corrScale.m1x`, where all tools reach the same bound. These functions operate on statistical moments, whose ranges can grow very large, and both Fluctuat and PRECiSA amplify the error over such ranges because they rely on abstract interpretation.[7] Rosa first bounds ranges by solving optimisation problems with Z3; when Z3 fails to return a bound, it falls back to abstract interpretation, and its bounds become equally pessimistic.

On the stress tests and the case-study tests, Rosa and PRECiSA time out or run out of memory on most benchmarks. Fluctuat computes bounds within one or two orders of magnitude of Stainless's on 8 of 18 stress tests. On `stepResponseArea3` and `rayMarching2`, Fluctuat computes a tighter bound than Stainless, because Ibex does not have enough time to tighten all problems. Fluctuat's default mode

[6]The flag assumes that the floating-point execution will always take the same conditional branches as real execution.
[7]Although PRECiSA is designed to be tighter than Fluctuat, its authors confirmed to us, on a representative example, that it cannot compute significantly tighter bounds on these benchmarks.

Table 1. Absolute total error bounds, assuming rounded inputs. Stainless runs with timeouts of 5, 30 and 600 s. Analysis times are in brackets. Fluctuat runs without (fast) and with (tight) 10k interval subdivisions. The tightest bound of each row is in bold, t/o denotes a timeout and error a crash or out of memory.

Benchmark	#paths	Stainless			Fluctuat		PRECiSA	Rosa
		5s	30s	600s	fast	tight		
FPBench (10 min timeout)								
cav10	2	2.9	2.9	**2.9**	1.0e2 (<10 ms)	**2.9** (3.0 s)	1.0e2 (0.9 ms)	**2.9** (6.2 s)
smartRoot	4	∞/NaN	∞/NaN	∞/NaN	NaN	NaN	NaN	**5.3e−15** (25 s)
squareRoot3Invalid	2	1.2e−9	1.2e−9	**1.2e−9**	0.43 (<10 ms)	1.9e−8 (2.0 s)	2.7 (0.7 ms)	**1.2e−9** (5.8 s)
squareRoot3	2	1.3e−11	1.3e−11	**1.3e−11**	0.43 (<10 ms)	3.0e−8 (2.0 s)	2.7 (0.8 ms)	**1.3e−11** (5.7 s)
Case study (10 min timeout)								
avgGroupPlus.value	8	2.5e2	29	**1.3**	1.9e9 (<10 ms)	1.7e8 (12 s)	3.1e8 (290 s)	unsupported
corrScale.c2	2	8.6e−5	8.6e−5	**8.6e−5**	2.5e11 (<10 ms)	1.0e10 (16 s)	2.5e11 (2.2 ms)	3.8e6 (1.4 s)
corrScale.m0	2	3.4e−10	3.4e−10	**3.4e−10**	2.5e5 (<10 ms)	2.1e4 (16 s)	1.0e6 (2.2 ms)	15 (1.0 s)
corrScale.m1x	2	1.0e3	1.0e3	**1.0e3**	**1.0e3** (<10 ms)	**1.0e3** (16 s)	**1.0e3** (2.2 ms)	**1.0e3** (735 ms)
corrScale.m2x	2	8.6e−5	8.6e−5	**8.6e−5**	6.3e10 (<10 ms)	5.2e9 (16 s)	2.5e11 (2.2 ms)	3.8e6 (734 ms)
combinedMean	8	6.6e2	33	**1.7**	1.9e9 (<10 ms)	1.7e8 (11 s)	3.1e8 (171 s)	t/o
momMonoidPlus.m1	9	8.8e2	43	**2.0**	1.9e9 (10 ms)	5.7e8 (76 s)	t/o	t/o
momPlus.m1	8	0.029	0.029	**0.029**	5.0e8 (<10 ms)	1.0e8 (32 s)	2.3e7 (275 s)	t/o
momScale.m0D	2	3.4e−10	3.4e−10	**3.4e−10**	5.0e5 (<10 ms)	2.0e4 (10 s)	1.0e6 (2.0 ms)	8.0e6 (1.3 s)
momScale.m2	2	8.6e−5	8.6e−5	**8.6e−5**	1.2e11 (<10 ms)	5.0e9 (10 s)	2.5e11 (2.0 ms)	2.0e12 (918 ms)
momScale.m3	2	0.038	0.038	**0.038**	1.0e14 (<10 ms)	5.0e12 (10 s)	1.0e14 (2.0 ms)	8.0e14 (836 ms)
momScale.m4	2	38	38	**38**	5.0e16 (<10 ms)	2.5e15 (10 s)	1.0e17 (2.0 ms)	8.0e17 (666 ms)
Case-study tests (10 min timeout)								
avgAssoc3Lhs.value	64	t/o	2.7e3	**2.9e3**	1.9e7 (10 ms)	7.7e5 (31 s)	t/o	unsupported
avgAssoc3Rhs.value	64	t/o	2.3e3	**1.1e3**	1.5e7 (10 ms)	7.4e5 (32 s)	t/o	unsupported
avgAssoc4Lhs.value	512	t/o	t/o	**5.4e3**	2.9e7 (10 ms)	2.9e6 (73 s)	t/o	unsupported
avgAssoc4Rhs.value	512	t/o	t/o	**8.4e3**	1.9e9 (10 ms)	5.0e7 (67 s)	t/o	unsupported
corrDistrLhs.m1x	130	t/o	54	**0.63**	1.9e5 (10 ms)	8.4e4 (111 s)	error	error
corrDistrLhs.m1y	130	t/o	1.3e3	**2.2e2**	1.9e5 (10 ms)	8.4e4 (111 s)	error	error
corrDistrRhs.m1x	260	t/o	t/o	**88**	∞	∞	t/o	t/o
corrDistrRhs.m1y	260	t/o	t/o	**1.2e3**	∞	∞	t/o	t/o
corrMonoidPlus.m1x	65	t/o	1.5e2	**4.1**	1.9e9 (10 ms)	8.4e8 (113 s)	t/o	t/o
corrMonoidPlus.m1y	65	t/o	1.3e3	**8.8e2**	1.9e9 (10 ms)	8.4e8 (113 s)	t/o	t/o
momAssocLhs.m1	81	t/o	t/o	**6.1e2**	1.9e7 (20 ms)	4.2e6 (271 s)	error	error
momAssocLhs.m2	81	t/o	t/o	**1.1e8**	6.0e14 (20 ms)	3.4e13 (271 s)	error	error
momAssocLhs.m3	81	t/o	t/o	**1.8e15**	1.5e22 (20 ms)	2.1e20 (271 s)	error	error
momAssocLhs.m4	81	t/o	t/o	**7.5e18**	3.6e29 (20 ms)	1.2e27 (271 s)	error	error
momAssocRhs.m1	81	t/o	t/o	**6.1e2**	1.6e7 (20 ms)	3.6e6 (268 s)	error	error
momAssocRhs.m2	81	t/o	t/o	**2.2e8**	6.0e14 (20 ms)	3.4e13 (268 s)	error	error
momAssocRhs.m3	81	t/o	t/o	**1.7e15**	1.5e22 (20 ms)	2.1e20 (268 s)	error	error
momDistrLhs.m1	18	1.6e2	7.5	**0.29**	1.9e5 (10 ms)	7.3e4 (78 s)	error	error
momDistrRhs.m1	36	t/o	3.3e2	**57**	∞	error	t/o	error
Stress tests (20 min timeout)								
bouncingBall1	8	0.22	0.22	**0.22**	5.2 (<10 ms)	2.2 (7.4 s)	38 (25 ms)	7.8 (15 s)
bouncingBall2	32	2.1	0.40	**0.40**	7.3 (<10 ms)	4.4 (13 s)	t/o	9.4 (1149 s)
bouncingBall3	128	t/o	0.57	**0.57**	9.7 (<10 ms)	6.7 (20 s)	error	t/o
bouncingBall4	512	t/o	10	**0.75**	12 (10 ms)	9.2 (28 s)	error	t/o
bouncingBall5	2.0k	t/o	t/o	**0.93**	15 (10 ms)	12 (38 s)	error	t/o
bouncingBall6	8.2k	t/o	t/o	t/o	18 (10 ms)	**15** (49 s)	error	t/o
rayMarching1	256	6.2	2.2	**0.020**	4.8 (<10 ms)	2.3 (20 s)	t/o	t/o
rayMarching2	33.0k	t/o	t/o	20	**14** (10 ms)	error	t/o	error
rayMarching3	4.2M	t/o	t/o	t/o	**27** (10 ms)	error	error	error
stepResponseArea1	4	2.1e−5	2.0e−5	**2.0e−5**	3.4 (10 ms)	2.4e−4 (57 s)	1.4 (109 s)	unsupported
stepResponseArea2	34	t/o	0.18	**2.4e−4**	3.4 (10 ms)	0.023 (149 s)	error	unsupported
stepResponseArea3	2.3k	t/o	t/o	2.1	3.4 (30 ms)	**0.042** (345 s)	error	unsupported
stepResponseArea4	10.7M	t/o	t/o	t/o	**3.4** (70 ms)	3.6 (778 s)	t/o	unsupported
mlp1	9	9.8e−15	9.8e−15	**9.8e−15**	3.4 (<10 ms)	0.067 (9.2 s)	13 (252 ms)	13 (15 s)
mlp2	81	20	3.0e−13	**3.0e−13**	4.4e2 (<10 ms)	3.7 (18 s)	t/o	t/o
mlp3	729	t/o	6.4e−14	**6.4e−14**	40 (<10 ms)	0.41 (24 s)	error	t/o
mlp4	6.6k	t/o	t/o	**1.5e−12**	1.1e3 (10 ms)	error	error	t/o
mlp5	59.0k	t/o	t/o	t/o	**1.6e3** (10 ms)	error	error	error

Table 2. Ablation study on the stress tests. Absolute total error bounds computed by Stainless on rounded inputs for pairwise path comparison (Pairwise) and subtree comparison with and without refinements. #p is the number of paths. The tightest bound of each row is in bold.

Benchmark	#p	Pairwise			Without refinements			With refinements		
		5s	30s	600s	5s	30s	600s	5s	30s	600s
mlp1	9	9.8e−15	9.8e−15	**9.8e−15**	8.7	9.8e−15	**9.8e−15**	9.8e−15	9.8e−15	**9.8e−15**
mlp2	81	t/o	3.0e−13	**3.0e−13**	1.6e2	2.0e2	4.1	20	3.0e−13	**3.0e−13**
mlp3	729	t/o	t/o	t/o	t/o	15	23	t/o	6.4e−14	**6.4e−14**
mlp4	6.6k	t/o	t/o	t/o	t/o	t/o	13	t/o	t/o	**1.5e−12**
mlp5	59.0k	t/o	t/o	t/o	t/o	t/o	t/o	t/o	t/o	t/o
rayMarching1	256	t/o	t/o	0.022	6.2	6.0	6.0	6.2	2.2	**0.020**
rayMarching2	33.0k	t/o	t/o	t/o	t/o	t/o	29	t/o	t/o	**20**
rayMarching3	4.2M	t/o	t/o	t/o	t/o	t/o	t/o	t/o	t/o	t/o
bouncingBall1	8	0.22	0.22	**0.22**	2.2	0.22	**0.22**	0.22	0.22	**0.22**
bouncingBall2	32	0.40	0.40	**0.40**	4.6	4.6	**0.40**	2.1	0.40	**0.40**
bouncingBall3	128	t/o	t/o	**0.57**	t/o	5.3	5.8	t/o	0.57	**0.57**
bouncingBall4	512	t/o	t/o	t/o	t/o	8.7	10	t/o	10	**0.75**
bouncingBall5	2.0k	t/o	t/o	t/o	t/o	t/o	7.7	t/o	t/o	**0.93**
bouncingBall6	8.2k	t/o	t/o	t/o	t/o	t/o	t/o	t/o	t/o	t/o
stepResponseArea1	4	2.0e−5	2.0e−5	**2.0e−5**	2.0e−5	2.0e−5	**2.0e−5**	2.1e−5	2.0e−5	**2.0e−5**
stepResponseArea2	34	t/o	t/o	**2.4e−4**	t/o	0.67	0.086	t/o	0.18	**2.4e−4**
stepResponseArea3	2.3k	t/o	t/o	t/o	t/o	t/o	**2.1**	t/o	t/o	**2.1**
stepResponseArea4	10.7M	t/o	t/o	t/o	t/o	t/o	t/o	t/o	t/o	t/o

scales extremely well with the number of paths: it returns a bound on almost every benchmark on which Stainless does, and on four more on which Stainless times out. Its interval-subdivision mode is significantly slower, but still returns a bound on the same benchmarks except four, where it crashes.

Across 5 runs, the bounds of the competing tools are identical, as are Stainless's on 33 of 52 benchmarks. Stainless bounds vary more only on the four `avgAssoc` outputs and `rayMarching2` (factor 2.7 to 6.1), and differ by at most 6 % on the remaining benchmarks.

> **RQ1 finding:** Stainless computes the tightest bounds on almost all benchmarks, often by several orders of magnitude. It also returns a bound on almost all benchmarks, whereas Rosa and PRECiSA time out or run out of memory on most programs with more than two paths.

6.4 RQ2: Scalability of Stainless as Branching Increases

To measure how our approach affects the scalability of Stainless, we run an ablation study on the stress tests and the case-study tests. For each benchmark, we measure the bound computed by Stainless in three configurations: comparing every pair of paths, which simulates existing techniques but uses the same solver; comparing subtrees without refinement; and comparing subtrees with as many refinement steps as there are conditions in the program. Table 2 shows the results on the stress tests, and Table 10 in the appendix those on the case-study tests.

On the stress tests, pairwise comparison produces too many optimisation problems and times out on all functions with more than 256 paths, even with a 10-minute budget. Subtree comparison, with or without refinement, solves within 30 seconds every benchmark that pairwise comparison solves in 10 minutes, and with a 10-minute budget, it solves almost twice as many. With or without refinement, it times out only on benchmarks with 8k paths or more.

Refinement also significantly tightens the bounds on the stress tests, matching those of pairwise comparison whenever the latter finishes. On `rayMarching1`, it even computes tighter bounds than pairwise comparison: since it solves fewer problems, each problem receives a larger share of the time budget, so the solver can tighten it further.

Table 3. Absolute total error bounds computed by Stainless on the GitHub benchmarks, assuming rounded inputs, with timeouts of 5, 30 and 600 s. #p is the number of paths. `cases` rows indicate whether divergence can make the floating-point and real-valued executions return different constructors (*e.g.,* `Some` and `None`).

Benchmark	#p	5s	30s	600s	Benchmark	#p	5s	30s	600s
removeDatum.Some.v.average	3	NaN	NaN	NaN	recall	2	NaN	NaN	NaN
removeDatum.Some.v.count	3	0	0	0	valueExprs.DoubleConst.value	4	NaN	NaN	NaN
removeDatum.cases	3	no	no	no	valueExprs.SqlTypedNull.precOpt	4	0	0	0
findClosestDivision	4	unsupported			valueExprs.cases	4	yes	yes	yes
standardRemapping	16	unsupported			samplesFraction	4	7.5e−9	7.5e−9	7.5e−9
initialize	6	0.10	0.10	0.10	Tweedie.deviance	12	4.1e4	1.9e4	3.0e3
project	3	1.1e−13	1.1e−13	1.1e−13	Binomial.deviance	4	NaN	NaN	NaN
fixLineSpacing	2	1.0	1.0	1.0	convertToTimeUnit	7	t/o	7.3e−12	7.3e−12
bone._1	3	1.8e−15	1.8e−15	1.8e−15	jValueDecode	5	unsupported		
bone._2	3	2.2e−15	2.2e−15	2.2e−15	ylogy	2	NaN	NaN	NaN
bone._3	3	2.0e−15	2.0e−15	2.0e−15	Poisson.deviance	2	NaN	NaN	NaN
hot._1	3	8.5e−16	8.5e−16	8.5e−16	heal	4	3.8e−6	3.8e−6	3.8e−6
hot._2	3	1.7e−15	1.7e−15	1.7e−15	plus.x	4	1.0e5	1.0	2.9e−11
hot._3	3	1.2e−15	1.2e−15	1.2e−15	plus.y	4	1.0e5	1.0e5	1.0e5
pink._1	3	9.8e−14	9.8e−14	9.8e−14	update._2	4	63	16	0.49
pink._2	3	4.0e−14	4.0e−14	4.0e−14	update.cases	4	yes	yes	yes
pink._3	3	4.0e−14	4.0e−14	4.0e−14	Circular.easeInOut	2	NaN	NaN	NaN
logLogistic	2	unsupported			Exponential.easeInOut	4	4.9e−4	4.9e−4	4.9e−4
f1	8	NaN	NaN	NaN					

On the case-study tests, the trend is the same. There, however, the bottleneck is the complexity of each problem rather than their number, so the bound depends mostly on the time each problem receives. As a result, on 6 of 20 benchmarks, subtree comparison computes tighter bounds without refinement than with it, since each initial problem then receives twice as much time.

> **RQ2 finding:** Subtree comparison scales far beyond pairwise comparison: it solves within 30 seconds every benchmark that pairwise comparison solves in 10 minutes, and almost twice as many within 10 minutes, including programs with tens of thousands of paths. Refinement preserves this scalability and matches the bounds of pairwise comparison whenever the latter finishes.

6.5 RQ3: Usability in Real-World Code Verification

Table 3 shows the results on the functions mined from GitHub. Since most use Scala features other tools do not support, we instead evaluate how much real-world code Stainless supports, and identify the research and engineering challenges that remain towards full support.

Together, the benchmarks exercise every feature described in Sec. 5: of the 25 functions, 16 use composite types (algebraic data types or case classes), 7 integers, 3 compound conditions, and 4 explicit NaN or infinity. Stainless does not support 4 functions: `logLogistic` calls `log1p`, which neither Daisy nor Ibex supports; `jValueDecode` converts a `Double` to a `Float`; and two functions convert floating-point values to integers. Conversions to integers introduce a source of divergence that real-valued optimisers cannot encode directly; we leave them for future work.

Of the 22 outputs on which Stainless computes a finite bound, 15 are continuous at every condition; 3 have a real-valued discontinuity (`initialize`, `fixLineSpacing`, and `Exponential.easeInOut`) and 4 do not diverge. Stainless computes small bounds on 12 of the continuous outputs, and on all 3 discontinuous ones, a bound very close to the real-valued jump. On `Tweedie.deviance` and `update._2`, Ibex does not exploit that the two branches share most of their computation. On most outputs, the bound is already final at 5 seconds; on the others, more time tightens it, *e.g.,* from 1e5 to 2.9e−11 on `plus.x`. In two functions, `valueExprs` and `update`, Stainless detects that divergence can make the floating-point and real-valued executions return different constructors.

Of the 8 outputs Stainless flags as possibly non finite, 4 are genuine and 4 are false alarms. Except for `Binomial.deviance`, Stainless's finiteness check distinguishes genuine from spurious alarms and reports an input that triggers each genuine one. On `Binomial.deviance`, the check also raises a false alarm, because of how Stainless abstracts the specification of `log` [30].

> **RQ3 finding:** Real-world code relies on features that the input languages of existing tools often cannot express, such as composite types, integers, and special values; Stainless supports several of them. It computes small bounds on most continuous functions and bounds close to the real-valued jump on discontinuous ones, and its finiteness check separates genuine NaN alarms from spurious ones. Conversions to integers remain the main open challenge.

6.6 RQ4: Dynamic Analysis Effectiveness

To answer RQ4, we evaluate the dynamic analysis in an ablation study. This is necessary because no existing tool searches for divergence-triggering inputs across all program conditions (Sec. 7).

We perform the study following Klees et al. [38]. As benchmarks, we consider every stress test and case study output for which the static analysis reports a possible divergence. We consider five configurations: (**All**) the full pipeline, with static intervals, widening, bisection, and random mutations; (**−M**) the pipeline without random mutations; (**−I**) the pipeline on the entire input domain, instead of the static intervals; (**−B**) the pipeline with random sampling within the static intervals, instead of bisection, to find the first divergence-triggering input; and (**−IM**) bisection alone on the entire input domain. Configurations that use static intervals take them from the median run of the static analysis, measured over 5 runs and with a 600s timeout. To match the dynamic analysis, which evaluates both semantics on the same floating-point inputs, we run the static analysis assuming that the inputs carry no initial rounding error. We do not ablate widening as the intervals returned by Ibex are too small to contain divergence-triggering inputs and the dynamic analysis finds none. We run each configuration with 30 randomly generated seeds. We report the median of three metrics: (i) the largest error found, which is a lower bound on the worst-case error; (ii) the number of distinct divergence-triggering inputs; and (iii) the number of conditions at which divergence is triggered. To check that a difference is not just noise, we use the Mann–Whitney U test [3] and consider $p < 0.05$ significant. To quantify a difference, we report the Vargha–Delaney effect size $\hat{A}_{12}$: the probability that a random run of the full pipeline beats a random run of the variant [54]. We use the setup as before, but run experiments in parallel on 10 cores. Since parallel runs can interfere in timings, we limit the runs to 1M inputs rather than a time limit.

Table 4 reports the results on all stress tests and on a representative subset of the case study, namely every other output sorted by upper bound; Table 11 in the appendix reports all of them. We mark statistically significant results with respect to **All** in blue and orange$^{\uparrow}$.

On the stress tests, the dynamic analysis finds errors (first column) close to the static upper bound (`mlp3`: 1.5e−15 against 6.4e−14), and on 7 benchmarks, the `bouncingBall` and `stepResponseArea` families, it matches it. On the case-study tests, the largest error found is several orders of magnitude below the static bound, which suggests that the bounds returned by the static analysis are loose.

The full pipeline generally finds larger errors than the other configurations on the stress tests, while on the case study, it outperforms all configurations except **−I**, which searches the whole input domain. We believe that the static bounds are very loose there, so the static intervals passed to the dynamic analysis lie far from the diverging region, widening cannot compensate, and the search explores the wrong region. We conclude that static intervals are most effective when the static analysis computes tight bounds. Mutation also increases the largest error on some benchmarks, *e.g.,* on `combinedMean` from 1.8e−13 to 2.2e−13, and never decreases it significantly.

Table 4. Ablation study of the dynamic analysis. *Upper bound* is the static error bound, assuming exact inputs. Shaded cells differ significantly from **All** ($p < 0.05$) with a medium ($\hat{A}_{12} \geq 0.64$, light) or large ($\hat{A}_{12} \geq 0.71$, dark) effect: blue when **All** is better, orange$^{\uparrow}$ otherwise ($\hat{A}_{12} \leq 0.36$ or ≤ 0.29); white cells show no statistically significant difference. #p is the number of paths. Full results are in Table 11 in the appendix.

		Upper	Largest roundoff error					Diverging conditions					Unique diverging inputs				
Benchmark	#p	bound	All	−M	−I	−B	−IM	All	−M	−I	−B	−IM	All	−M	−I	−B	−IM
Stress tests																	
bouncingBall1	8	0.22	0.22	0.22	0.22	0.22	0.22	2	2	2	2	2	14k	15k$^{\uparrow}$	9.8k	171.5	8.1k
bouncingBall2	32	0.40	0.40	0.40	0.39	0.40	0.39	3	3	3	3	3	17k	18k	13k	200.5	10k
bouncingBall3	128	0.57	0.57	0.57	0.57	0.57	0.57	5	5	5	5	5	27k	27k	18k	1.4k	13k
bouncingBall4	512	0.75	0.75	0.75	0.75	0.75	0.75	7	7	7	7	7	40k	39k	23k	2.6k	16k
bouncingBall5	2.0k	0.93	0.93	0.93	0.92	0.93	0.92	10	10	10	10	10	39k	38k	25k	2.2k	17k
rayMarching1	256	0.011	1.0e−4	1.0e−4	0	1.0e−4	0	1	1	0	1	0	24k	15k	0	1.2k	0
rayMarching2	33.0k	20	2.1e−4	2.1e−4	1.0e−4	2.1e−4	1.0e−4	2	2	1	1	1	102k	24k	89k	445.5	8.3k
stepResponseArea1	4	2.0e−5	2.0e−5	2.0e−5	2.0e−5	2.0e−5	2.0e−5	1	1	1	1	1	50k	30k	39k	30k	1.5k
stepResponseArea2	34	2.4e−4	2.4e−4	2.4e−4	2.4e−4	2.4e−4	2.4e−4	3	3	3	3	3	88k	65k	49k	18k	11k
mlp1	9	9.8e−15	4.5e−16	4.5e−16	4.2e−16	4.5e−16	4.2e−16	3	3	3	2	3	74k	40k	115k$^{\uparrow}$	34k	41k
mlp2	81	3.0e−13	9.9e−15	1.0e−14	9.0e−15	9.7e−15	8.8e−15	8	8	8	8	8	81k	66k	62k	22k	33k
mlp3	729	6.4e−14	1.5e−15	1.5e−15	1.9e−15$^{\uparrow}$	1.3e−15	1.8e−15$^{\uparrow}$	14	14	14	14	14	48k	42k	51k$^{\uparrow}$	3.5k	24k
mlp4	6.6k	1.5e−12	3.5e−14	3.2e−14	2.8e−14	3.1e−14	2.6e−14	20	20	20	19	20	83k	61k	73k	23k	30k
Case study																	
avgAssoc4Lhs.value	512	5.4e3	2.2e−13	2.1e−13	2.4e−13$^{\uparrow}$	2.1e−13	2.3e−13$^{\uparrow}$	27	29	19	20	20	136k	63k	104k	284k$^{\uparrow}$	33k
corrDistrLhs.m1x	130	0.63	2.4e−13	2.3e−13	2.3e−13	1.1e−13	2.0e−13	2	2	2	1	2	184k	109k	87k	3.2k	2.5k
corrDistrRhs.m1x	260	88	2.5e−13	2.5e−13	2.4e−13	1.3e−13	2.2e−13	3	3	3	1	3	161k	90k	85k	13k	4.7k
corrDistrRhs.m1y	260	1.2e3	2.5e−13	2.5e−13	2.4e−13	1.3e−13	2.3e−13	3	3	3	1	3	161k	90k	85k	13k	4.7k
corrMonoidPlus.m1x	65	3.7	2.2e−13	2.1e−13	2.3e−13	0	2.0e−13	2	2	2	0	2	131k	52k	86k	0	3.3k
corrMonoidPlus.m1y	65	8.8e2	2.2e−13	2.2e−13	2.3e−13$^{\uparrow}$	0	2.1e−13	2	2	2	0	2	131k	52k	86k	0	3.3k
combinedMean	8	1.3	2.2e−13	1.8e−13	2.4e−13$^{\uparrow}$	0	2.0e−13	2	2	2	0	2	76k	32k	46k	0	2.4k
momAssocLhs.m2	81	1.1e8	8.9e−5	8.9e−5	8.9e−5	4.6e−5	8.8e−5	14	14	12	4	12	167k	92k	90k	5.3k	8.2k
momAssocRhs.m1	81	8.3e2	2.7e−13	2.5e−13	2.9e−13$^{\uparrow}$	1.4e−13	2.6e−13	14	14	12	4	12	167k	91k	90k	5.8k	8.2k
momAssocRhs.m3	81	1.7e15	0.18	0.17	0.17	0.020	0.15	14	14	12	4	12	167k	91k	90k	5.8k	8.2k
momAssocRhs.m4	81	7.5e18	3.5e2	3.3e2	4.1e2$^{\uparrow}$	40	3.5e2	14	14	12	4	12	167k	91k	90k	5.8k	8.2k
momPlus.m1	8	0.029	1.4e−13	3.4e−14	1.6e−13	0	8.5e−14	1	1	1	0	1	455k	416k	74k	0	9

For the number of diverging conditions and of unique diverging inputs, the full pipeline consistently matches or outperforms the other configurations. Both bisection and static intervals help reach diverging conditions: on `avgAssoc4Lhs`, removing either significantly reduces the number of diverging conditions found (27 against 19 and 20). As expected, mutation does not help reach new diverging conditions, but exploits those already found. Finally, the full pipeline finds the most unique diverging inputs on 31 of 37 benchmarks, even when compared with the best other configuration on each benchmark. Removing any of static intervals, bisection, or mutation reduces this number, so no single component is responsible for the result, but rather their combination.

> **RQ4 finding:** Stainless finds divergence-triggering inputs on all benchmarks. Static intervals and bisection let it reach diverging conditions that it misses without them, and mutation exploits the conditions they reach to find more inputs and larger errors. Combined, the components find the most unique diverging inputs, and tight static bounds help find larger errors, while loose ones tend to mislead the search.

7 Related Work

Several static analysers compute error bounds for floating-point programs. Many specialise in computing tight bounds for straight-line code [18, 21, 23, 29, 32, 41, 51]. Rather than competing with these analyses, we build on them: we use any such analyser as a black box for the straight-line error bounds and add the divergence error on top. NumFuzz's type system is sound for

conditionals only when the real and floating-point executions provably take the same branch [37]. Fluctuat [33] bounds the divergence error with zonotopes that overapproximates the diverging region. PRECiSA [52], Seesaw [22], Real2Float [40] and Rosa [19, 20] solve at least one optimisation problem per pair of paths in the program, a number that grows quadratically and quickly becomes intractable. Our approach instead solves a linear number of optimisation problems, by considering sets of paths with refinement. The static analysers above that support conditionals warn the user about a potential divergence of the control flow, but offer no way of checking that warning against concrete inputs, and hence no way of telling a false alarm apart from a genuine defect in the code.

Like Stainless before our modifications [30], KeY can verify floating-point properties of programs through SMT-solving, but cannot reason about roundoff errors [2]. Frama-C can reason automatically about roundoff errors in straight-line code via Gappa [9]. It can verify error bounds supplied by the programmer for conditionals interactively [8], but cannot automatically infer them.

Several lines of work study how to prove continuity of real-valued programs [11], or design languages in which continuity holds by construction [48]. Continuity alone does not provide enough information to derive an error bound in the presence of a conditional, since we also need quantitative information about the behaviour of the program near the boundary of the branch condition. Continuity is nonetheless useful information for the programmer, and is complementary to our work. Other work analyses Lipschitz continuity or performs sensitivity analyses [12, 42]. In practice, many common functions, such as the square root, the logarithm and the inverse, are not globally Lipschitz continuous, and discontinuities must still be bounded separately.

Among the static analysers described above, only FErA reports concrete inputs alongside the error bound it computes [29]. FErA does not support conditionals, which require a different approach, such as the one introduced in this paper. Closer to our work, the divergence detection of S3FP [13] returns a single divergence-triggering input, found by bisection, which our dynamic analysis builds on. Bao and Zhang [4] detect unstable branches only on inputs provided by the user. Weak-distance minimisation [28] finds inputs on branch boundaries, *i.e.,* that satisfy a condition with equality in floating-point arithmetic, but such inputs do not necessarily trigger divergence. To our knowledge, no existing tool searches for divergence-triggering inputs across all conditions of a program.

Other approaches generate inputs that maximise the error rather than trigger divergence. The search can be guided by coverage [34], condition number [56, 58], heuristics [14, 55] or random mutations [57], often measuring the error with shadow execution [16]. Floating-point debuggers [15, 35, 47] have a somewhat different goal in that they aim to dynamically identify specific operations that cause numerical issues such as large roundoff errors. These approaches are complementary to ours: they find large error along a fixed path, whereas we find inputs for which the path itself differs from the real-valued one. Similar techniques find inputs that trigger runtime errors, overflows or NaN results [6, 39, 43], which Stainless can already check via SMT.

AI Declaration

Parts of the implementation were generated by Claude Opus 5.5 from detailed design prompts, with every implementation choice validated by us. We edited, reviewed and unit-tested all AI-generated code. For the case study, Claude suggested popular Scala repositories with loop-free branching floating-point code, from which we chose the one with the most challenging functions to analyse.

Data-Availability Statement

We provide the results of the analysis of the example in Sec. 2, the four benchmark suites and the raw results of all runs as supplementary material to this submission. We will submit a full artifact to Artifact Evaluation and archive it permanently on Zenodo. The artifact will contain:

- **Implementation.** The source code of our extension of Stainless (Sec. 5), which implements the static and dynamic analyses. Like Stainless, we release it under the Apache 2.0 license.
- **Benchmarks.** The four benchmark suites of Sec. 6 and our translations of them into the input languages of the other tools.
- **Tools and scripts.** PRECiSA, Rosa, and Seesaw, and scripts that rerun every experiment of Sec. 6. Fluctuat is closed source, so for it we include only its input files and the outputs it produced.
- **Results.** The raw results of all runs, and the scripts that generate the paper's tables and figures from them.

References

[1] Rosa Abbasi and Eva Darulova. 2023. Modular Optimization-Based Roundoff Error Analysis of Floating-Point Programs. In *International Static Analysis Symposium (SAS)*. doi:10.1007/978-3-031-44245-2_4

[2] Rosa Abbasi, Jonas Schiffl, Eva Darulova, Mattias Ulbrich, and Wolfgang Ahrendt. 2023. Combining Rule- and SMT-based Reasoning for Verifying Floating-Point Java Programs in KeY. *International Journal on Software Tools for Technology Transfer* 25, 2 (2023), 185–204. doi:10.1007/s10009-022-00691-x

[3] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/1985793.1985795

[4] Tao Bao and Xiangyu Zhang. 2013. On-the-fly detection of instability problems in floating-point program execution. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 817–832. doi:10.1145/2509136.2509526

[5] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 415–442.

[6] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic detection of floating-point exceptions. In *Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. 549–560. doi:10.1145/2429069.2429133

[7] Dirk Beyer, Stefan Löwe, and Philipp Wendler. 2017. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer* 21, 1 (Nov. 2017), 1–29. doi:10.1007/s10009-017-0469-y

[8] Sylvie Boldo and Claude Marché. 2011. Formal Verification of Numerical Programs: From C Annotated Programs to Mechanical Proofs. *Mathematics in Computer Science* 5, 4 (2011), 377–393. doi:10.1007/s11786-011-0099-9

[9] Sylvie Boldo and Guillaume Melquiond. 2021. Some Formal Tools for Computer Arithmetic: Flocq and Gappa. In *28th IEEE Symposium on Computer Arithmetic, ARITH 2021, Lyngby, Denmark, June 14-16, 2021*. IEEE, 111–114. doi:10.1109/ARITH51176.2021.00031

[10] Gilles Chabert and Luc Jaulin. 2009. Contractor programming. *Artificial Intelligence* 173, 11 (2009), 1079–1100. doi:10.1016/j.artint.2009.03.002

[11] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2010. Continuity analysis of programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 57–70. doi:10.1145/1706299.1706308

[12] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. 2011. Proving programs robust. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 102–112. doi:10.1145/2025113.2025131

[13] Wei-Fan Chiang, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2015. Practical Floating-Point Divergence Detection. In *Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing - Volume 9519* (Raleigh, NC, USA) *(LCPC 2015)*. Springer-Verlag, Berlin, Heidelberg, 271–286. doi:10.1007/978-3-319-29778-1_17

[14] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 43–52. doi:10.1145/2555243.2555265

[15] Sangeeta Chowdhary and Santosh Nagarakatte. 2022. Fast shadow execution for debugging numerical errors using error free transformations. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 190 (Oct. 2022), 28 pages. doi:10.1145/3563353

[16] Clement Courbet. 2021. NSan: a Floating-Point Numerical Sanitizer. In *Proceedings of the International Conference on Compiler Construction (CC)*. doi:10.1145/3446804.3446848

[17] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. 2017. Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In *Numerical Software Verification*, Sergiy Bogomolov, Matthieu Martel, and Pavithra Prabhakar (Eds.). Springer International Publishing, Cham, 63–77.

[18] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. 2018. Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 270–287. doi:10.1007/978-3-319-89960-2_15

[19] Eva Darulova and Viktor Kuncak. 2014. Sound Compilation of Reals. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. 235–248. doi:10.1145/2535838.2535874

[20] Eva Darulova and Viktor Kuncak. 2017. Towards a Compiler for Reals. *ACM Trans. Program. Lang. Syst.* 39, 2 (2017), 8:1–8:28. doi:10.1145/3014426

[21] Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, Sriram Krishnamoorthy, and Pavel Panchekha. 2020. Scalable yet Rigorous Floating-Point Error Analysis. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. doi:10.1109/SC41405.2020.00055

[22] Arnab Das, Tanmay Tirpankar, Ganesh Gopalakrishnan, and Sriram Krishnamoorthy. 2021. Robustness Analysis of Loop-Free Floating-Point Programs via Symbolic Automatic Differentiation. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 481–491. doi:10.1109/Cluster48925.2021.00055

[23] Marc Daumas and Guillaume Melquiond. 2010. Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Softw.* 37, 1 (2010), 2:1–2:20. doi:10.1145/1644001.1644003

[24] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[25] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 509–519. doi:10.1109/ASE.2017.8115662

[26] Peter Dinda and Conor Hetland. 2018. Do Developers Understand IEEE Floating Point?. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 589–598. doi:10.1109/IPDPS.2018.00068

[27] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.* 33, 2 (June 2007), 15 pages. doi:10.1145/1236463.1236468

[28] Zhoulai Fu and Zhendong Su. 2019. Effective floating-point analysis via weak-distance minimization. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 439–452. doi:10.1145/3314221.3314632

[29] Rémy Garcia, Claude Michel, and Michel Rueher. 2020. Rigorous Enclosure of Round-Off Errors in Floating-Point Computations. In *Software Verification*, Maria Christakis, Nadia Polikarpova, Parasara Sridhar Duggirala, and Peter Schrammel (Eds.). Springer International Publishing, Cham, 196–212.

[30] Andrea Gilot, Axel Bergström, and Eva Darulova. 2026. Verifying Floating-Point Programs in Stainless. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[31] Andrea Gilot, Tobias Wrigstad, and Eva Darulova. 2026. Floating-Point Usage on GitHub: A Large-Scale Study of Statically Typed Languages. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 95 (April 2026), 28 pages. doi:10.1145/3798203

[32] Eric Goubault and Sylvie Putot. 2011. Static Analysis of Finite Precision Computations. In *Verification, Model Checking, and Abstract Interpretation*, Ranjit Jhala and David Schmidt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 232–247. doi:10.1007/978-3-642-18275-4_17

[33] Eric Goubault and Sylvie Putot. 2013. Robustness Analysis of Finite Precision Implementations. In *Programming Languages and Systems*, Chung-chieh Shan (Ed.). Springer International Publishing, Cham, 50–57.

[34] Hui Guo and Cindy Rubio-González. 2020. Efficient Generation of Error-Inducing Floating-Point Inputs via Symbolic Execution. In *Proceedings of the International Conference on Software Engineering (ICSE)*. doi:10.1145/3377811.3380359

[35] Yumeng He and Pavel Panchekha. 2026. Accurate Residues for Floating-Point Debugging. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 394 (Oct. 2026), 23 pages. doi:10.1145/3839526

[36] Anastasiia Izycheva and Eva Darulova. 2017. On sound relative error bounds for floating-point arithmetic. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, Daryl Stewart and Georg Weissenbacher (Eds.). IEEE, 15–22. doi:10.23919/FMCAD.2017.8102236

[37] Ariel E. Kellison and Justin Hsu. 2024. Numerical Fuzz: A Type System for Rounding Error Analysis. *Proc. ACM Program. Lang.* 8, PLDI, Article 226 (June 2024), 25 pages. doi:10.1145/3656456

[38] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. doi:10.1145/3243734.3243804

[39] Ignacio Laguna and Ganesh Gopalakrishnan. 2022. Finding inputs that trigger floating-point exceptions in GPUs via bayesian optimization. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 33, 14 pages.

[40] Victor Magron, George Constantinides, and Alastair Donaldson. 2017. Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Trans. Math. Softw.* 43, 4 (2017), 34:1–34:31. doi:10.1145/3015465

[41] Victor Magron, Alexandre Rocca, and Thao Dang. 2019. Certified Roundoff Error Bounds Using Bernstein Expansions and Sparse Krivine-Stengle Representations. *IEEE Trans. Comput.* 68, 7 (2019), 953–966. doi:10.1109/TC.2018.2851235

[42] Rupak Majumdar and Indranil Saha. 2009. Symbolic Robustness Analysis. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium (RTSS '09)*. IEEE Computer Society, USA, 355–363. doi:10.1109/RTSS.2009.17

[43] Dolores Miao, Ignacio Laguna, and Cindy Rubio-González. 2025. FloatGuard: Efficient Whole-Program Detection of Floating-Point Exceptions in AMD GPUs. In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing* (University of Notre Dame Conference Facilities, Notre Dame, IN, USA) *(HPDC '25)*. Association for Computing Machinery, New York, NY, USA, Article 2, 12 pages. doi:10.1145/3731545.3731586

[44] Jordan Ninin. 2016. Global Optimization Based on Contractor Programming: An Overview of the IBEX Library. In *Mathematical Aspects of Computer and Information Sciences*, Ilias S. Kotsireas, Siegfried M. Rump, and Chee K. Yap (Eds.). Springer International Publishing, Cham, 555–559.

[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG] https://arxiv.org/abs/1912.01703

[46] Dung T. Phan, Radu Grosu, Nils Jansen, Nicola Paoletti, Scott A. Smolka, and Scott D. Stoller. 2020. Neural Simplex Architecture. In *NASA Formal Methods: 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11–15, 2020, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 97–114. doi:10.1007/978-3-030-55754-6_6

[47] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. 2018. Finding root causes of floating point error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 256–269. doi:10.1145/3192366.3192411

[48] Benjamin Sherman, Jesse Michel, and Michael Carbin. 2019. Sound and robust solid modeling via exact real arithmetic and continuity. *Proc. ACM Program. Lang.* 3, ICFP, Article 99 (July 2019), 29 pages. doi:10.1145/3341703

[49] Daniel Smilkov, Shan Carter, D. Sculley, Fernanda B. Viégas, and Martin Wattenberg. 2017. Direct-Manipulation Visualization of Deep Networks. arXiv:1708.03788 [cs.LG] https://arxiv.org/abs/1708.03788

[50] Andrew P. Smith, Cesar A. Munoz, Anthony J. Narkawicz, and Mantas Markevicius. 2015. A Rigorous Generic Branch and Bound Solver for Nonlinear Problems. In *Proceedings of the 2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC) (SYNASC '15)*. IEEE Computer Society, USA, 71–78. doi:10.1109/SYNASC.2015.20

[51] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Trans. Program. Lang. Syst.* 41, 1 (2018), 2:1–2:39. doi:10.1145/3230733

[52] Laura Titolo, Mariano Moscato, Marco A. Feliu, Paolo Masci, and César A. Muñoz. 2025. Rigorous Floating-Point Round-Off Error Analysis in PRECiSA 4.0. In *Formal Methods (FM)*. doi:10.1007/978-3-031-71177-0_2

[53] Twitter. 2026. *algebird: Abstract Algebra for Scala.* Retrieved October 2026 from https://github.com/twitter/algebird/tree/464917d6a3f2b1c9f2048e3dd38df5dd97edc965

[54] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. arXiv:https://doi.org/10.3102/10769986025002101 doi:10.3102/10769986025002101

[55] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2017. Efficient Global Search for Inputs Triggering High Floating-Point Inaccuracies. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 11–20. doi:10.1109/APSEC.2017.7

[56] Xin Yi, Hengbiao Yu, Liqian Chen, Xiaoguang Mao, and Ji Wang. 2024. FPCC: Detecting Floating-Point Errors via Chain Conditions. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 324 (Oct. 2024), 28 pages. doi:10.1145/3689764

[57] Daming Zou, Ran Wang, Yingfei Xiong, Lu Zhang, Zhendong Su, and Hong Mei. 2015. A genetic algorithm for detecting significant floating-point inaccuracies. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 529–539.

[58] Daming Zou, Muhan Zeng, Yingfei Xiong, Zhoulai Fu, Lu Zhang, and Zhendong Su. 2019. Detecting floating-point errors via atomic conditions. *Proc. ACM Program. Lang.* 4, POPL, Article 60 (Dec. 2019), 27 pages. doi:10.1145/3371128

Appendix

Table 5. Absolute error bounds computed by Stainless assuming rounded inputs (600 s timeout) and by Seesaw (commit 336c4c4) on all four benchmark sets. For each output, *tot* is the total error bound and *rnd* the roundoff error bound assuming no divergence. Seesaw's analysis time is in brackets.

Benchmark	#p		Stainless 600s	Seesaw	Benchmark	#p		Stainless 600s	Seesaw
FPBench (10 min timeout)									
cav10	2	tot rnd	2.9 3.2e−14	2.9 (65 s) 4.4e−16	squareRoot3	2	tot rnd	1.3e−11 1.1e−15	5.5e−16 (13 s) 5.5e−16
smartRoot	4	tot rnd	∞/NaN	6.8e−5 (58 s) 1.3e−15	triangleSorted	2	tot rnd	∞/NaN	∞
squareRoot3Invalid	2	tot rnd	1.2e−9 1.1e−15	5.5e−16 (13 s) 5.5e−16					
Case study functions (10 min timeout)									
avgGroupPlus.value	8	tot rnd	1.3 0.058	unsupported	decayedScale.value	4	tot rnd	∞/NaN	unsupported
corrScale.c2	2	tot rnd	8.6e−5 5.8e−5	2.8e−5 (89 s) 2.8e−5	decayedValueAsOf	8	tot rnd	∞/NaN	unsupported
corrScale.m0	2	tot rnd	3.4e−10 2.3e−10	1.1e−10 (89 s) 1.1e−10	combinedMean	8	tot rnd	1.7 0.12	3.3e8 (600 s) 1.1e−7
corrScale.m1x	2	tot rnd	1.0e3 5.7e−14	1.2e−29 (89 s) 1.2e−29	momMonoidPlus.m1	9	tot rnd	2.0 0.12	t/o
corrScale.m1y	2	tot rnd	1.0e3 5.7e−14	1.2e−29 (89 s) 1.2e−29	momPlus.m1	8	tot rnd	0.029 0.029	2.6e5 (383 s) 1.4e−10
corrScale.m2x	2	tot rnd	8.6e−5 5.8e−5	2.8e−5 (89 s) 2.8e−5	momScale.m0D	2	tot rnd	3.4e−10 2.3e−10	1.1e−10 (88 s) 1.1e−10
corrScale.m2y	2	tot rnd	8.6e−5 5.8e−5	2.8e−5 (89 s) 2.8e−5	momScale.m1	2	tot rnd	1.0e3 5.7e−14	1.2e−29 (88 s) 1.2e−29
decayedAverageFrom	5	tot rnd	∞/NaN	unsupported	momScale.m2	2	tot rnd	8.6e−5 5.8e−5	2.8e−5 (88 s) 2.8e−5
decayedPlus.scaledTime	8	tot rnd	∞/NaN	unsupported	momScale.m3	2	tot rnd	0.038 0.027	0.011 (88 s) 0.011
decayedPlus.value	8	tot rnd	∞/NaN	unsupported	momScale.m4	2	tot rnd	38 27	11 (88 s) 11
decayedScale.scaledTime	4	tot rnd	∞/NaN	unsupported					
Case-study tests (10 min timeout)									
avgAssoc3Lhs.value	64	tot rnd	2.9e3 1.1e−7	unsupported	momAssocLhs.m1	81	tot rnd	6.1e2 2.4e−7	t/o
avgAssoc3Rhs.value	64	tot rnd	1.1e3 1.1e−7	unsupported	momAssocLhs.m2	81	tot rnd	1.1e8 3.4	t/o
avgAssoc4Lhs.value	512	tot rnd	5.4e3 1.5e−7	unsupported	momAssocLhs.m3	81	tot rnd	1.8e15 3.9e7	t/o
avgAssoc4Rhs.value	512	tot rnd	8.4e3 1.2e−5	unsupported	momAssocLhs.m4	81	tot rnd	7.5e18 4.7e14	t/o
corrDistrLhs.m1x	130	tot rnd	0.63 1.5e−9	t/o	momAssocRhs.m1	81	tot rnd	6.1e2 2.4e−7	t/o
corrDistrLhs.m1y	130	tot rnd	2.2e2 1.5e−9	t/o	momAssocRhs.m2	81	tot rnd	2.2e8 3.4	t/o
corrDistrRhs.m1x	260	tot rnd	88 1.3e−8	t/o	momAssocRhs.m3	81	tot rnd	1.7e15 3.9e7	t/o
corrDistrRhs.m1y	260	tot rnd	1.2e3 1.3e−8	t/o	momAssocRhs.m4	81	tot rnd	7.5e18 4.7e14	t/o
corrMonoidPlus.m1x	65	tot rnd	4.1 0.12	t/o	momDistrLhs.m1	18	tot rnd	0.29 1.5e−9	t/o
corrMonoidPlus.m1y	65	tot rnd	8.8e2 0.12	t/o	momDistrRhs.m1	36	tot rnd	57 1.3e−8	t/o
Stress tests (20 min timeout)									
bouncingBall1	8	tot rnd	0.22 1.2e−15	t/o	stepResponseArea1	4	tot rnd	2.0e−5 5.2e−15	error
bouncingBall2	32	tot rnd	0.40 2.0e−15	t/o	stepResponseArea2	34	tot rnd	2.4e−4 9.9e−15	error

Benchmark	#p		Stainless 600s	Seesaw	Benchmark	#p		Stainless 600s	Seesaw
bouncingBall3	128	tot rnd	0.57 3.0e−15	t/o	stepResponseArea3	2.3k	tot rnd	2.1 2.0e−14	error
bouncingBall4	512	tot rnd	0.75 4.0e−15	t/o	stepResponseArea4	10.7M	tot rnd	t/o	error
bouncingBall5	2.0k	tot rnd	0.93 5.5e−15	t/o	mlp1	9	tot rnd	9.8e−15 3.1e−15	8.7 (387 s) 1.1e−15
bouncingBall6	8.2k	tot rnd	t/o	t/o	mlp2	81	tot rnd	3.0e−13 1.3e−13	t/o
rayMarching1	256	tot rnd	0.020 1.2e−15	t/o	mlp3	729	tot rnd	6.4e−14 2.3e−14	t/o
rayMarching2	33.0k	tot rnd	20 5.0e−15	t/o	mlp4	6.6k	tot rnd	1.5e−12 7.1e−13	t/o
rayMarching3	4.2M	tot rnd	t/o	t/o	mlp5	59.0k	tot rnd	t/o	t/o

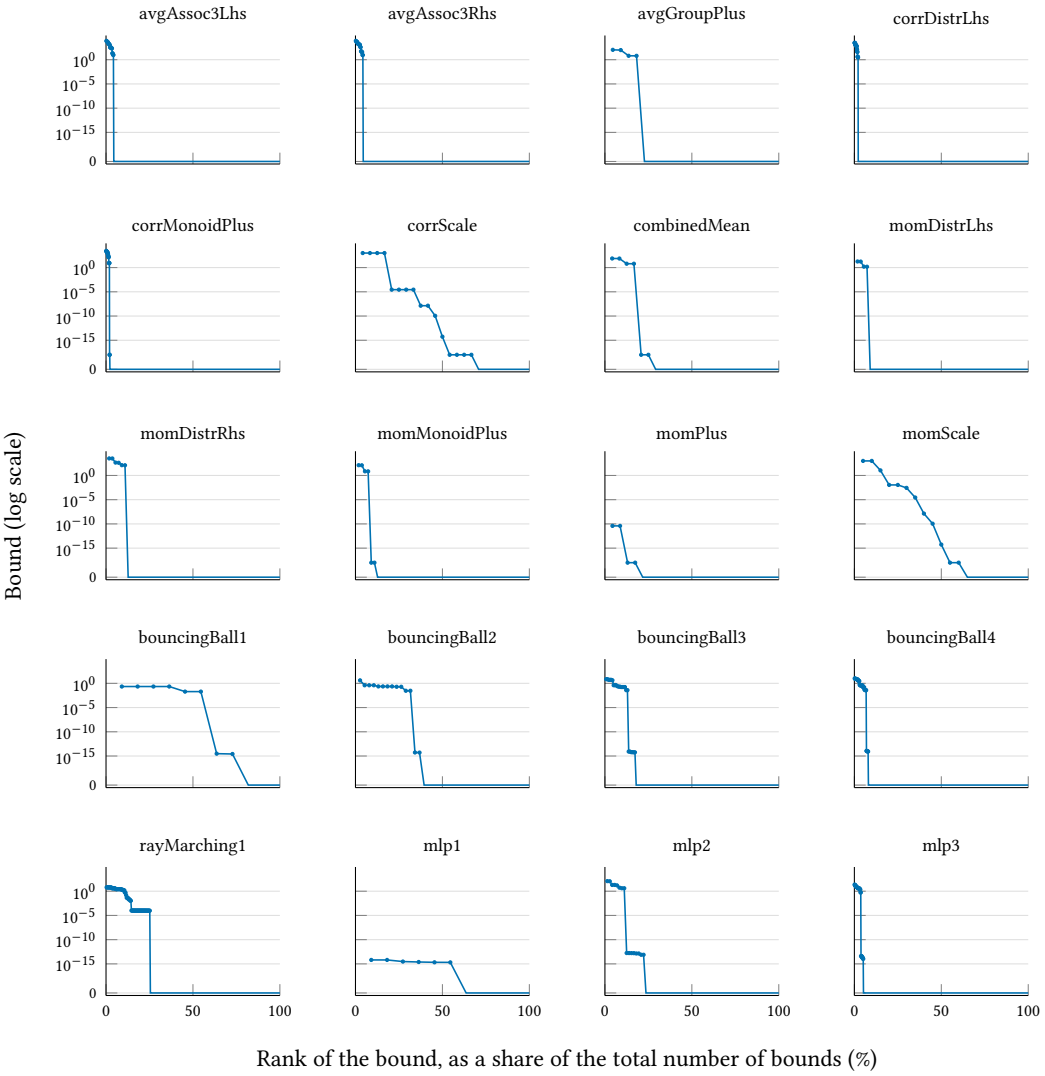


Fig. 5. Bounds of all subtree comparisons for each stress test and case study benchmark (Sec. 6) on which Stainless does not time out, on a log scale. Stainless runs without refinement, without the early stop of Sec. 3.4, and with a 30 s timeout. For each benchmark, the comparisons are sorted from the largest bound.

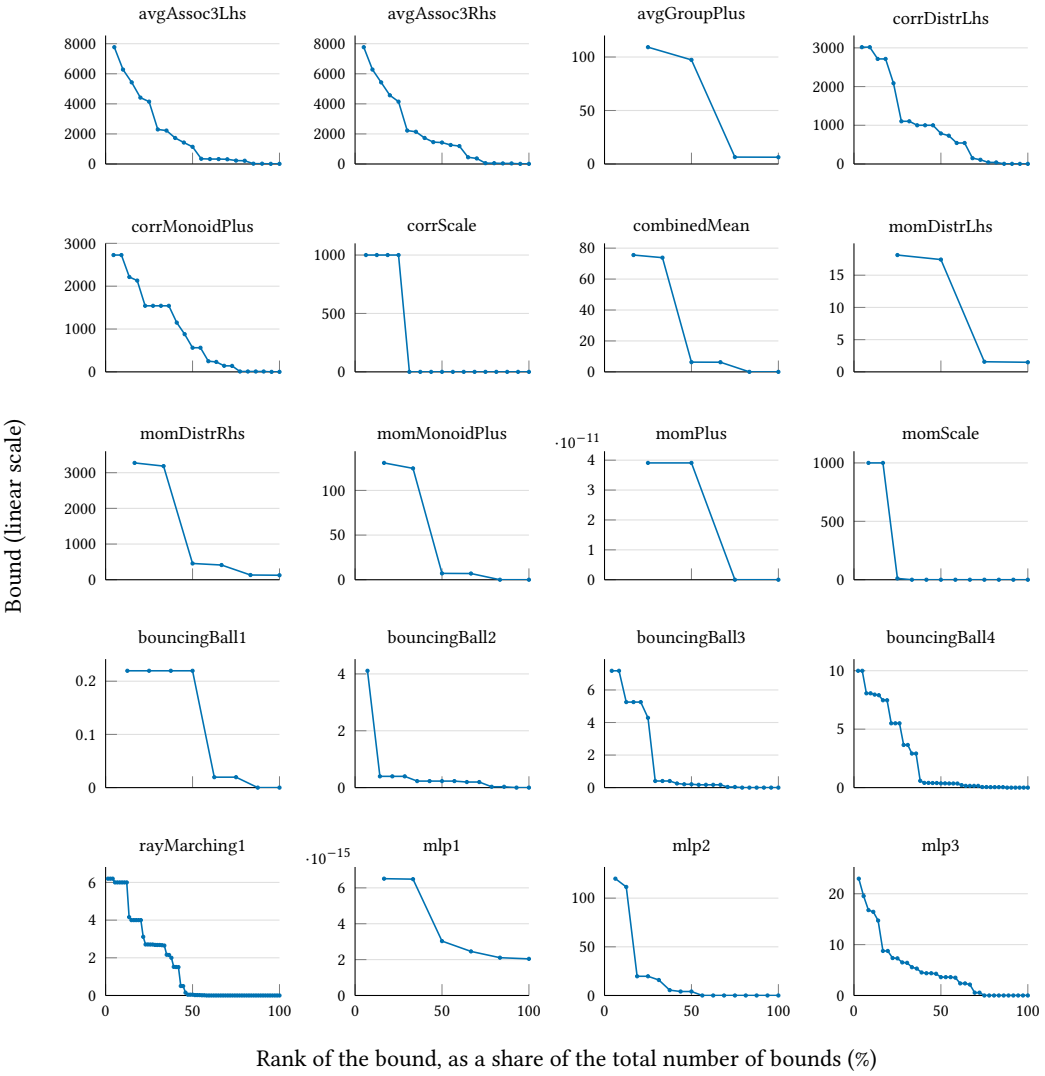


Fig. 6. The bounds of Fig. 5 that are not 0 on a linear scale, from the largest to the smallest.

Table 6. Absolute error bounds on the FPBench benchmarks, assuming rounded inputs. For each output, *tot* is the total error bound and *rnd* the roundoff error bound assuming no divergence. Stainless runs with timeouts of 5, 30 and 600 s, the other tools with a 10-minute timeout, and their analysis time is in brackets. Fluctuat runs without (fast) and with (tight) 10k interval subdivisions. The tightest total bound of each output is in bold.

Benchmark	#p		Stainless			Fluctuat		PRECiSA	Rosa
			5s	30s	600s	fast	tight		
cav10	2	tot	2.9	2.9	**2.9**	1.0e2 (<10 ms)	**2.9** (3.0 s)	1.0e2 (0.9 ms)	**2.9** (6.2 s)
		rnd	3.2e−14	3.2e−14	3.2e−14	6.3e−15	2.1e−15	3.2e−14	1.8e−14
smartRoot	4	tot	∞/NaN	∞/NaN	∞/NaN	NaN	NaN	NaN	**5.3e−15** (25 s)
		rnd							5.3e−15
squareRoot3Invalid	2	tot	1.2e−9	1.2e−9	**1.2e−9**	0.43 (<10 ms)	1.9e−8 (2.0 s)	2.7 (0.7 ms)	**1.2e−9** (5.8 s)
		rnd	1.1e−15	1.1e−15	1.1e−15	1.1e−15	6.1e−16	4.2e−8	1.1e−15
squareRoot3	2	tot	1.3e−11	1.3e−11	**1.3e−11**	0.43 (<10 ms)	3.0e−8 (2.0 s)	2.7 (0.8 ms)	**1.3e−11** (5.7 s)
		rnd	1.1e−15	1.1e−15	1.1e−15	1.1e−15	6.1e−16	4.2e−8	1.1e−15
triangleSorted	2	tot	∞/NaN	∞/NaN	∞/NaN	NaN	NaN	NaN	NaN
		rnd							

Table 7. Absolute error bounds on the case-study functions; the rows and columns are as in Table 6. Only outputs whose Stainless bound includes a divergence error, or that Stainless cannot bound, are shown.

Benchmark	#p		Stainless			Fluctuat		PRECiSA	Rosa
			5s	30s	600s	fast	tight		
avgGroupPlus.value	8	tot	2.5e2	29	**1.3**	1.9e9 (<10 ms)	1.7e8 (12 s)	3.1e8 (290 s)	unsupported
		rnd	0.058	0.058	0.058	0.058	7.3e−4	0.0037	
corrScale.c2	2	tot	8.6e−5	8.6e−5	**8.6e−5**	2.5e11 (<10 ms)	1.0e10 (16 s)	2.5e11 (2.2 ms)	3.8e6 (1.4 s)
		rnd	5.8e−5	5.8e−5	5.8e−5	5.8e−5	5.8e−5	5.8e−5	5.8e−5
corrScale.m0	2	tot	3.4e−10	3.4e−10	**3.4e−10**	2.5e5 (<10 ms)	2.1e4 (16 s)	1.0e6 (2.2 ms)	15 (1.0 s)
		rnd	2.3e−10	2.3e−10	2.3e−10	2.3e−10	2.3e−10	2.3e−10	2.3e−10
corrScale.m1x	2	tot	1.0e3	1.0e3	**1.0e3**	**1.0e3** (<10 ms)	**1.0e3** (16 s)	**1.0e3** (2.2 ms)	**1.0e3** (735 ms)
		rnd	5.7e−14	5.7e−14	5.7e−14	5.7e−14	5.7e−14	5.7e−14	5.7e−14
corrScale.m1y	2	tot	1.0e3	1.0e3	**1.0e3**	**1.0e3** (<10 ms)	**1.0e3** (16 s)	**1.0e3** (2.2 ms)	**1.0e3** (613 ms)
		rnd	5.7e−14	5.7e−14	5.7e−14	5.7e−14	5.7e−14	5.7e−14	5.7e−14
corrScale.m2x	2	tot	8.6e−5	8.6e−5	**8.6e−5**	6.3e10 (<10 ms)	5.2e9 (16 s)	2.5e11 (2.2 ms)	3.8e6 (734 ms)
		rnd	5.8e−5	5.8e−5	5.8e−5	5.8e−5	5.8e−5	5.8e−5	5.8e−5
corrScale.m2y	2	tot	8.6e−5	8.6e−5	**8.6e−5**	6.3e10 (<10 ms)	5.2e9 (16 s)	2.5e11 (2.2 ms)	3.8e6 (790 ms)
		rnd	5.8e−5	5.8e−5	5.8e−5	5.8e−5	5.8e−5	5.8e−5	5.8e−5
decayedAverageFrom	5	tot	∞/NaN	∞/NaN	∞/NaN	unsupported	unsupported	unsupported	unsupported
		rnd							
decayedPlus.scaledTime	8	tot	∞/NaN	∞/NaN	∞/NaN	unsupported	unsupported	unsupported	unsupported
		rnd							
decayedPlus.value	8	tot	∞/NaN	∞/NaN	∞/NaN	unsupported	unsupported	unsupported	unsupported
		rnd							
decayedScale.scaledTime	4	tot	∞/NaN	∞/NaN	∞/NaN	unsupported	unsupported	unsupported	unsupported
		rnd							
decayedScale.value	4	tot	∞/NaN	∞/NaN	∞/NaN	unsupported	unsupported	unsupported	unsupported
		rnd							
decayedValueAsOf	8	tot	∞/NaN	∞/NaN	∞/NaN	unsupported	unsupported	unsupported	unsupported
		rnd							
combinedMean	8	tot	6.6e2	33	**1.7**	1.9e9 (<10 ms)	1.7e8 (11 s)	3.1e8 (171 s)	t/o
		rnd	0.12	0.12	0.12	0.12	0.0065	0.0055	
momMonoidPlus.m1	9	tot	8.8e2	43	**2.0**	1.9e9 (10 ms)	5.7e8 (76 s)	t/o	t/o
		rnd	0.12	0.12	0.12	0.12	0.029		
momPlus.m1	8	tot	0.029	0.029	**0.029**	5.0e8 (<10 ms)	1.0e8 (32 s)	2.3e7 (275 s)	t/o
		rnd	0.029	0.029	0.029	0.029	0.0036	2.1e−5	
momScale.m0D	2	tot	3.4e−10	3.4e−10	**3.4e−10**	5.0e5 (<10 ms)	2.0e4 (10 s)	1.0e6 (2.0 ms)	8.0e6 (1.3 s)
		rnd	2.3e−10	2.3e−10	2.3e−10	2.3e−10	2.3e−10	2.3e−10	2.3e−10
momScale.m1	2	tot	1.0e3	1.0e3	**1.0e3**	**1.0e3** (<10 ms)	**1.0e3** (10 s)	**1.0e3** (2.0 ms)	**1.0e3** (767 ms)
		rnd	5.7e−14	5.7e−14	5.7e−14	5.7e−14	5.7e−14	5.7e−14	5.7e−14
momScale.m2	2	tot	8.6e−5	8.6e−5	**8.6e−5**	1.2e11 (<10 ms)	5.0e9 (10 s)	2.5e11 (2.0 ms)	2.0e12 (918 ms)
		rnd	5.8e−5	5.8e−5	5.8e−5	5.8e−5	5.8e−5	5.8e−5	5.8e−5
momScale.m3	2	tot	0.038	0.038	**0.038**	1.0e14 (<10 ms)	5.0e12 (10 s)	1.0e14 (2.0 ms)	8.0e14 (836 ms)
		rnd	0.027	0.027	0.027	0.027	0.027	0.027	0.027
momScale.m4	2	tot	38	38	**38**	5.0e16 (<10 ms)	2.5e15 (10 s)	1.0e17 (2.0 ms)	8.0e17 (666 ms)
		rnd	27	27	27	27	27	27	27

Table 8. Absolute error bounds on the case-study tests; the rows and columns are as in Table 6. Only outputs whose Stainless bound includes a divergence error, or that Stainless cannot bound, are shown.

Benchmark	#p		Stainless			Fluctuat		PRECiSA	Rosa
			5s	30s	600s	fast	tight		
avgAssoc3Lhs.value	64	tot	t/o	2.7e3	**2.9e3**	1.9e7 (10 ms)	7.7e5 (31 s)	t/o	unsupported
		rnd		1.1e−7	1.1e−7	1.4e−7	1.7e−9		
avgAssoc3Rhs.value	64	tot	t/o	2.3e3	**1.1e3**	1.5e7 (10 ms)	7.4e5 (32 s)	t/o	unsupported
		rnd		1.1e−7	1.1e−7	1.1e−7	1.7e−9		
avgAssoc4Lhs.value	512	tot	t/o	t/o	**5.4e3**	2.9e7 (10 ms)	2.9e6 (73 s)	t/o	unsupported
		rnd			1.5e−7	1.9e−7	6.3e−9		
avgAssoc4Rhs.value	512	tot	t/o	t/o	**8.4e3**	1.9e9 (10 ms)	5.0e7 (67 s)	t/o	unsupported
		rnd			1.2e−5	1.8e−5	1.4e−7		
corrDistrLhs.m1x	130	tot	t/o	54	**0.63**	1.9e5 (10 ms)	8.4e4 (111 s)	error	error
		rnd		1.5e−9	1.5e−9	1.5e−9	5.6e−10		
corrDistrLhs.m1y	130	tot	t/o	1.3e3	**2.2e2**	1.9e5 (10 ms)	8.4e4 (111 s)	error	error
		rnd		1.5e−9	1.5e−9	1.5e−9	5.6e−10		
corrDistrRhs.m1x	260	tot	t/o	t/o	**88**	∞	∞	t/o	t/o
		rnd			1.3e−8				
corrDistrRhs.m1y	260	tot	t/o	t/o	**1.2e3**	∞	∞	t/o	t/o
		rnd			1.3e−8				
corrMonoidPlus.m1x	65	tot	t/o	1.5e2	**4.1**	1.9e9 (10 ms)	8.4e8 (113 s)	t/o	t/o
		rnd		0.12	0.12	0.12	0.044		
corrMonoidPlus.m1y	65	tot	t/o	1.3e3	**8.8e2**	1.9e9 (10 ms)	8.4e8 (113 s)	t/o	t/o
		rnd		0.12	0.12	0.12	0.044		
momAssocLhs.m1	81	tot	t/o	t/o	**6.1e2**	1.9e7 (20 ms)	4.2e6 (271 s)	error	error
		rnd			2.4e−7	2.9e−7	5.3e−8		
momAssocLhs.m2	81	tot	t/o	t/o	**1.1e8**	6.0e14 (20 ms)	3.4e13 (271 s)	error	error
		rnd			3.4	5.1	0.24		
momAssocLhs.m3	81	tot	t/o	t/o	**1.8e15**	1.5e22 (20 ms)	2.1e20 (271 s)	error	error
		rnd			3.9e7	8.7e7	1.1e6		
momAssocLhs.m4	81	tot	t/o	t/o	**7.5e18**	3.6e29 (20 ms)	1.2e27 (271 s)	error	error
		rnd			4.7e14	1.2e15	4.2e12		
momAssocRhs.m1	81	tot	t/o	t/o	**6.1e2**	1.6e7 (20 ms)	3.6e6 (268 s)	error	error
		rnd			2.4e−7	2.4e−7	4.4e−8		
momAssocRhs.m2	81	tot	t/o	t/o	**2.2e8**	6.0e14 (20 ms)	3.4e13 (268 s)	error	error
		rnd			3.4	5.1	0.24		
momAssocRhs.m3	81	tot	t/o	t/o	**1.7e15**	1.5e22 (20 ms)	2.1e20 (268 s)	error	error
		rnd			3.9e7	8.6e7	1.0e6		
momAssocRhs.m4	81	tot	t/o	t/o	**7.5e18**	3.6e29 (20 ms)	1.2e27 (268 s)	error	error
		rnd			4.7e14	1.2e15	4.0e12		
momDistrLhs.m1	18	tot	1.6e2	7.5	**0.29**	1.9e5 (10 ms)	7.3e4 (78 s)	error	error
		rnd	1.5e−9	1.5e−9	1.5e−9	1.5e−9	4.7e−10		
momDistrRhs.m1	36	tot	t/o	3.3e2	**57**	∞	error	t/o	error
		rnd		1.3e−8	1.3e−8				

Table 9. Absolute error bounds on the stress tests; the rows and columns are as in Table 6, except that the other tools run with a 20-minute timeout.

Benchmark	#p		Stainless			Fluctuat		PRECiSA	Rosa
			5s	30s	600s	fast	tight		
bouncingBall1	8	tot	0.22	0.22	**0.22**	5.2 (<10 ms)	2.2 (7.4 s)	38 (25 ms)	7.8 (15 s)
		rnd	1.2e−15	1.2e−15	1.2e−15	1.3e−15	0.040	5.7e−15	error
bouncingBall2	32	tot	2.1	0.40	**0.40**	7.3 (<10 ms)	4.4 (13 s)	t/o	9.4 (1149 s)
		rnd	2.0e−15	2.0e−15	2.0e−15	2.4e−15	0.040		error
bouncingBall3	128	tot	t/o	0.57	**0.57**	9.7 (<10 ms)	6.7 (20 s)	error	t/o
		rnd		3.0e−15	3.0e−15	3.5e−15	0.040		
bouncingBall4	512	tot	t/o	10	**0.75**	12 (10 ms)	9.2 (28 s)	error	t/o
		rnd		4.0e−15	4.0e−15	4.8e−15	1.8		
bouncingBall5	2.0k	tot	t/o	t/o	**0.93**	15 (10 ms)	12 (38 s)	error	t/o
		rnd			5.5e−15	6.6e−15	3.0		
bouncingBall6	8.2k	tot	t/o	t/o	t/o	18 (10 ms)	**15** (49 s)	error	t/o
		rnd				8.6e−15	4.5		
rayMarching1	256	tot	6.2	2.2	**0.020**	4.8 (<10 ms)	2.3 (20 s)	t/o	t/o
		rnd	1.2e−15	1.2e−15	1.2e−15	1.0e−15	8.1e−16		
rayMarching2	33.0k	tot	t/o	t/o	20	**14** (10 ms)	error	t/o	error
		rnd			5.0e−15	3.2e−15			
rayMarching3	4.2M	tot	t/o	t/o	t/o	**27** (10 ms)	error	error	error
		rnd				8.2e−15			
stepResponseArea1	4	tot	2.1e−5	2.0e−5	**2.0e−5**	3.4 (10 ms)	2.4e−4 (57 s)	1.4 (109 s)	unsupported
		rnd	5.2e−15	5.2e−15	5.2e−15	4.1e−15	1.2e−15	9.8e−15	
stepResponseArea2	34	tot	t/o	0.18	**2.4e−4**	3.4 (10 ms)	0.023 (149 s)	error	unsupported
		rnd		9.9e−15	9.9e−15	4.9e−15	1.5e−15		
stepResponseArea3	2.3k	tot	t/o	t/o	2.1	3.4 (30 ms)	**0.042** (345 s)	error	unsupported
		rnd			2.0e−14	6.2e−15	1.9e−15		
stepResponseArea4	10.7M	tot	t/o	t/o	t/o	**3.4** (70 ms)	3.6 (778 s)	t/o	unsupported
		rnd				8.2e−15	2.5e−15		
mlp1	9	tot	9.8e−15	9.8e−15	**9.8e−15**	3.4 (<10 ms)	0.067 (9.2 s)	13 (252 ms)	13 (15 s)
		rnd	3.1e−15	3.1e−15	3.1e−15	3.2e−15	1.7e−15	4.7e−15	error
mlp2	81	tot	20	3.0e−13	**3.0e−13**	4.4e2 (<10 ms)	3.7 (18 s)	t/o	t/o
		rnd	1.3e−13	1.3e−13	1.3e−13	6.4e−14	2.5e−14		
mlp3	729	tot	t/o	6.4e−14	**6.4e−14**	40 (<10 ms)	0.41 (24 s)	error	t/o
		rnd		2.3e−14	2.3e−14	1.0e−14	6.2e−15		
mlp4	6.6k	tot	t/o	t/o	**1.5e−12**	1.1e3 (10 ms)	error	error	t/o
		rnd			7.1e−13	1.5e−13			
mlp5	59.0k	tot	t/o	t/o	t/o	**1.6e3** (10 ms)	error	error	error
		rnd				2.0e−13			

Table 10. Ablation study on the case-study tests. The columns and the bold are as in Table 2.

Benchmark	#p	Pairwise			Without refinement			With refinement		
		5s	30s	600s	5s	30s	600s	5s	30s	600s
avgAssoc3Lhs.value	64	t/o	t/o	6.3e3	t/o	2.8e3	2.8e3	t/o	**2.7e3**	2.9e3
avgAssoc3Rhs.value	64	t/o	t/o	9.1e3	t/o	2.4e3	2.6e3	t/o	2.3e3	**1.1e3**
avgAssoc4Lhs.value	512	t/o	t/o	t/o	t/o	t/o	5.5e4	t/o	t/o	**5.4e3**
avgAssoc4Rhs.value	512	t/o	t/o	t/o	t/o	t/o	1.6e4	t/o	t/o	**8.4e3**
corrDistrLhs.m1x	130	t/o	t/o	t/o	t/o	35	**0.21**	t/o	54	0.63
corrDistrLhs.m1y	130	t/o	t/o	t/o	t/o	1.3e3	1.1e3	t/o	1.3e3	**2.2e2**
corrDistrRhs.m1x	260	t/o	t/o	t/o	t/o	t/o	**52**	t/o	t/o	88
corrDistrRhs.m1y	260	t/o	t/o	t/o	t/o	t/o	1.3e3	t/o	t/o	**1.2e3**
corrMonoidPlus.m1x	65	t/o	t/o	t/o	t/o	59	**1.8**	t/o	1.5e2	4.1
corrMonoidPlus.m1y	65	t/o	t/o	t/o	t/o	1.5e3	1.1e3	t/o	1.3e3	**8.8e2**
momAssocLhs.m1	81	t/o	t/o	t/o	t/o	t/o	3.0e3	t/o	t/o	**6.1e2**
momAssocLhs.m2	81	t/o	t/o	t/o	t/o	t/o	1.7e8	t/o	t/o	**1.1e8**
momAssocLhs.m3	81	t/o	t/o	t/o	t/o	t/o	**1.8e15**	t/o	t/o	**1.8e15**
momAssocLhs.m4	81	t/o	t/o	t/o	t/o	t/o	1.5e19	t/o	t/o	**7.5e18**
momAssocRhs.m1	81	t/o	t/o	t/o	t/o	t/o	3.0e3	t/o	t/o	**6.1e2**
momAssocRhs.m2	81	t/o	t/o	t/o	t/o	t/o	**2.2e8**	t/o	t/o	**2.2e8**
momAssocRhs.m3	81	t/o	t/o	t/o	t/o	t/o	2.4e15	t/o	t/o	**1.7e15**
momAssocRhs.m4	81	t/o	t/o	t/o	t/o	t/o	1.4e19	t/o	t/o	**7.5e18**
momDistrLhs.m1	18	t/o	1.6e2	4.7	1.8e2	5.1	**0.19**	1.6e2	7.5	0.29
momDistrRhs.m1	36	t/o	1.9e3	2.3e2	t/o	3.4e2	**47**	t/o	3.3e2	57

Table 11. Ablation study of the dynamic analysis on all outputs for which Stainless reports a possible divergence. The columns and the shading are as in Table 4.

		Upper	Largest roundoff error					Diverging conditions					Unique diverging inputs				
Benchmark	#p	bound	All	−M	−I	−B	−IM	All	−M	−I	−B	−IM	All	−M	−I	−B	−IM
Stress tests																	
bouncingBall1	8	0.22	0.22	0.22	0.22	0.22	0.22	2	2	2	2	2	14k	15k↑	9.8k	171.5	8.1k
bouncingBall2	32	0.40	0.40	0.40	0.39	0.40	0.39	3	3	3	3	3	17k	18k	13k	200.5	10k
bouncingBall3	128	0.57	0.57	0.57	0.57	0.57	0.57	5	5	5	5	5	27k	27k	18k	1.4k	13k
bouncingBall4	512	0.75	0.75	0.75	0.75	0.75	0.75	7	7	7	7	7	40k	39k	23k	2.6k	16k
bouncingBall5	2.0k	0.93	0.93	0.93	0.92	0.93	0.92	10	10	10	10	10	39k	38k	25k	2.2k	17k
rayMarching1	256	0.011	1.0e−4	1.0e−4	0	1.0e−4	0	1	1	0	1	0	24k	15k	0	1.2k	0
rayMarching2	33.0k	20	2.1e−4	2.1e−4	1.0e−4	2.1e−4	1.0e−4	2	2	1	1	1	102k	24k	89k	445.5	8.3k
stepResponseArea1	4	2.0e−5	2.0e−5	2.0e−5	2.0e−5	2.0e−5	2.0e−5	1	1	1	1	1	50k	30k	39k	30k	1.5k
stepResponseArea2	34	2.4e−4	2.4e−4	2.4e−4	2.4e−4	2.4e−4	2.4e−4	3	3	3	3	3	88k	65k	49k	18k	11k
mlp1	9	9.8e−15	4.5e−16	4.5e−16	4.2e−16	4.5e−16	4.2e−16	3	3	3	2	3	74k	40k	115k↑	34k	41k
mlp2	81	3.0e−13	9.9e−15	1.0e−14	9.0e−15	9.7e−15	8.8e−15	8	8	8	8	8	81k	66k	62k	22k	33k
mlp3	729	6.4e−14	1.5e−15	1.5e−15	1.9e−15↑	1.3e−15	1.8e−15↑	14	14	14	14	14	48k	42k	51k↑	3.5k	24k
mlp4	6.6k	1.5e−12	3.5e−14	3.2e−14	2.8e−14	3.1e−14	2.6e−14	20	20	20	19	20	83k	61k	73k	23k	30k
Case study																	
avgAssoc3Lhs.value	64	2.9e3	2.3e−13	2.2e−13	2.5e−13↑	2.0e−13	2.4e−13↑	9	9	8	10↑	8	145k	96k	101k	228k↑	48k
avgAssoc3Rhs.value	64	1.1e3	2.2e−13	2.1e−13	2.5e−13↑	2.0e−13	2.4e−13↑	10	10	8	10	8	172k	110k	101k	243k↑	48k
avgAssoc4Lhs.value	512	5.4e3	2.2e−13	2.1e−13	2.4e−13↑	2.1e−13	2.3e−13↑	27	29	19	20	20	136k	63k	104k	284k↑	33k
avgAssoc4Rhs.value	512	8.7e3	2.6e−13	2.7e−13	2.8e−13	2.7e−13	2.7e−13	26	26	19	24	19	180k	118k	121k	292k↑	66k
avgGroupPlus.value	8	1.3	2.0e−13	1.9e−13	2.2e−13↑	0	2.0e−13	2	2	2	0	2	107k	65k	74k	0	27k
corrDistrLhs.m1x	130	0.63	2.4e−13	2.3e−13	2.3e−13	1.1e−13	2.0e−13	2	2	2	1	2	184k	109k	87k	3.2k	2.5k
corrDistrLhs.m1y	130	3.3e2	2.3e−13	2.3e−13	2.3e−13	4.0e−29	2.0e−13	2	2	2	1	2	184k	109k	87k	3.2k	2.5k
corrDistrRhs.m1x	260	88	2.5e−13	2.5e−13	2.4e−13	1.3e−13	2.2e−13	3	3	3	1	3	161k	90k	85k	13k	4.7k
corrDistrRhs.m1y	260	1.2e3	2.5e−13	2.5e−13	2.4e−13	1.3e−13	2.3e−13	3	3	3	1	3	161k	90k	85k	13k	4.7k
corrMonoidPlus.m1x	65	3.7	2.2e−13	2.1e−13	2.3e−13	0	2.0e−13	2	2	2	0	2	131k	52k	86k	0	3.3k
corrMonoidPlus.m1y	65	8.8e2	2.2e−13	2.2e−13	2.3e−13↑	0	2.1e−13	2	2	2	0	2	131k	52k	86k	0	3.3k
combinedMean	8	1.3	2.2e−13	1.8e−13	2.4e−13↑	0	2.0e−13	2	2	2	0	2	76k	32k	46k	0	2.4k
momAssocLhs.m1	81	8.3e2	2.7e−13	2.5e−13	2.9e−13↑	8.6e−14	2.6e−13	14	14	12	4	12	167k	92k	90k	5.3k	8.2k
momAssocLhs.m2	81	1.1e8	8.9e−5	8.9e−5	8.9e−5	4.6e−5	8.8e−5	14	14	12	4	12	167k	92k	90k	5.3k	8.2k
momAssocLhs.m3	81	1.8e15	0.20	0.18	0.18	0.023	0.14	14	14	12	4	12	167k	92k	90k	5.3k	8.2k
momAssocLhs.m4	81	7.5e18	4.6e2	4.5e2	4.3e2	39	3.4e2	14	14	12	4	12	167k	92k	90k	5.3k	8.2k
momAssocRhs.m1	81	8.3e2	2.7e−13	2.5e−13	2.9e−13↑	1.4e−13	2.6e−13	14	14	12	4	12	167k	91k	90k	5.8k	8.2k
momAssocRhs.m2	81	2.2e8	8.9e−5	8.9e−5	8.9e−5	4.6e−5	8.7e−5	14	14	12	4	12	167k	91k	90k	5.8k	8.2k
momAssocRhs.m3	81	1.7e15	0.18	0.17	0.17	0.020	0.15	14	14	12	4	12	167k	91k	90k	5.8k	8.2k
momAssocRhs.m4	81	7.5e18	3.5e2	3.3e2	4.1e2↑	40	3.5e2	14	14	12	4	12	167k	91k	90k	5.8k	8.2k
momDistrLhs.m1	18	0.19	2.0e−13	1.7e−13	2.2e−13↑	0	2.0e−13	2	2	2	0	2	167k	90k	83k	0	2.0k
momDistrRhs.m1	36	57	2.2e−13	2.1e−13	2.4e−13↑	1.0e−28	2.2e−13	3	3	3	1	3	145k	77k	81k	21k	4.6k
momMonoidPlus.m1	9	1.7	2.2e−13	1.8e−13	2.4e−13↑	0	2.0e−13	2	2	2	0	2	112k	37k	83k	0	3.2k
momPlus.m1	8	0.029	1.4e−13	3.4e−14	1.6e−13	0	8.5e−14	1	1	1	0	1	455k	416k	74k	0	9