Reproducing FPGen: Validation and Performance Analysis Across Program Complexity

Alexander LaChapelle
Department of Computer Science
Oregon State University
Corvallis, USA
lachapea@oregonstate.edu

Nikhil Kumar Gattu
Department of Computer Science
Oregon State University
Corvallis, USA
gattun@oregonstate.edu

***Abstract*—Floating-point error is a particularly difficult facet of automated error detection. While existing tools can detect floating-point inaccuracies, there are many competing tools with different strengths and weaknesses. In this paper we will discuss our plans for replicating and extending a performance analysis of FPGen, a symbolic execution-based tool. This research expands FPGen's scope by analyzing its performance across variable floating-point computational complexity. We propose to test for the complexity at which performance converges with randomized search methods and use this as a basis to determine where FPGen provides the most significant benefits, which could help guide its practical application in numerical software development.**

I. Introduction

Floating-point errors are a well-known challenge in numerical computing, affecting applications in scientific simulations, financial modeling, and engineering [1]. These errors arise due to the limited precision of floating-point representations in computers, leading to rounding errors, catastrophic cancellation, and unpredictable exceptions. While small errors may seem insignificant, they can accumulate over time and cause critical failures, a minor precision error in matrix computations propagates across multiple iterations, resulting in an incorrect final result.

Detecting floating-point errors requires generating test inputs that trigger inaccuracies in numerical computations. Traditional approaches, such as random testing [2], randomly select numerical inputs to find potential errors. However, this approach is inefficient, as it does not systematically explore all possible execution paths where floating-point errors might occur. A more structured approach is symbolic execution, which is a technique used in software analysis where a program is executed with symbols (variables) instead of actual values, allowing it to analyze multiple paths in a single run [1]. This method is particularly useful for finding rare errors that might be missed by conventional testing.

FPGen [1] is a floating-point error detection tool that leverages symbolic execution to generate numerical inputs that are highly likely to induce floating-point errors. In the study conducted by Guo and Rubio-González, FPGen was shown to generate test inputs that produce significantly larger floating-point errors compared to random testing, as well as two competing tools—S3FP and KLEE-Float [1]. Our goal is to replicate their findings to verify whether FPGen consistently outperforms these methods. Additionally, while their evaluation focused on a limited set of numerical programs, we aim to extend their study by analyzing FPGen's effectiveness on more computationally complex programs. FPGen's effectiveness has primarily been evaluated on small to medium-sized numerical programs, including common operations such as summation functions, statistical computations, and matrix calculations from the GNU Scientific Library (GSL) and Meschach Library [1]. However, its performance on highly complex programs such as those involving deep branching structures, higher-order mathematical computations, and large-scale simulations remains unexplored.

This research aims to analyze FPGen's effectiveness across different levels of program complexity by replicating previous findings and extending the evaluation to more computationally complex numerical programs. Specifically, our study is guided by two key research questions:

- RQ1: Is FPGen effective at generating error-inducing inputs than random search?
- RQ2: Does FPGen remain effective at generating error-inducing floating-point inputs when applied to computationally complex numerical programs?

To address these questions, we will:

1) **Replicate the original FPGen experiments** to verify previous findings regarding its effectiveness compared to random search and other tools.
2) **Extend FPGen's evaluation** by applying it to a broader set of numerical programs with varying computational complexity.
3) **Categorize programs** based on their mathematical complexity, control flow structure, and floating-point sensitivity to systematically assess FPGen's strengths and limitations.
4) **Analyze FPGen's performance trends** by comparing error magnitudes and computational overhead across different complexity levels.

By conducting this study, we aim to provide a deeper understanding of FPGen's capabilities and limitations, ultimately contributing to the improvement of floating-point error detection in real-world applications.

II. Background and Motivation

A. Background

Floating-Point Arithmetic Error: Floating-point numbers are the standard for representing real numbers in computer programming [1]. However, because the set of all floating point numbers is finite, representing the infinite set of real numbers requires rounding, and therefore a loss of accuracy. This inaccurate value can in turn be used in other floating point calculations, causing a compounding effect.

Error-inducing Test Inputs: When developing tests for software that contains many interdependent floating point operations, it is important to be able to choose from the values that cause the most noticeable amount of error. Several broad categories of tools have been developed for this purpose:

- **Search-Based Techniques:** Methods such as Bayesian optimization (Xscope) [3] refine input generation using statistical models to maximize floating-point errors in test cases.
- **Symbolic Execution-Based Approaches:** Tools like KLEE-Float [4] and FPGen [1] use constraint solving techniques to find inputs that are likely to trigger precision loss.
- **Hybrid Methods:** Some tools combine multiple approaches. For example, FPGen initially applies symbolic execution but resorts to concretization assigning semi-randomized concrete values when path explosion becomes computationally infeasible [1].

These tools vary in effectiveness, with symbolic execution offering more precision in error discovery but suffering from computational limitations, while random and search-based methods trade accuracy for scalability.

Symbolic Execution: Symbolic execution is a systematic program analysis technique in which symbolic values replace concrete inputs, allowing the exploration of multiple execution paths in a single run [1]. This technique is particularly useful for detecting rare but critical floating-point errors that might be missed by conventional testing.

However, symbolic execution has a significant limitation as the number of execution paths increases due to branching conditions, the required computational power grows exponentially, making it impractical for complex programs. This phenomenon, known as path explosion, is a well-documented challenge in symbolic execution research [4], [5].

FPGen employs symbolic execution in its floating-point test input generation but incorporates concretization to mitigate path explosion. Specifically, when symbolic execution encounters an excessive number of branches, FPGen assigns semi-randomized concrete values to symbolic variables that are no longer feasible to track, reducing computational complexity [1].

After symbolic execution completes, FPGen leverages SMT (Satisfiability Modulo Theories) solvers, such as Z3 [6], to solve remaining constraints. This approach allows FPGen to systematically generate test cases that expose numerical inaccuracies in software, making it a powerful tool for floating-point error detection.

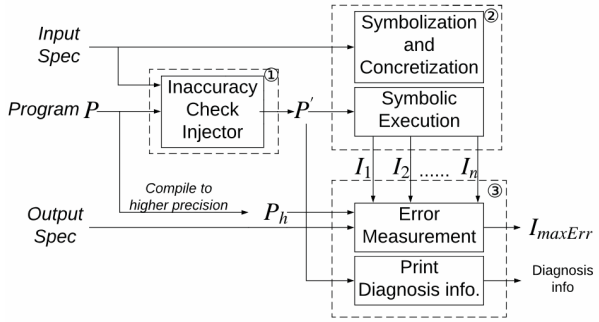

Fig. 1. FPGen workflow: Symbolic execution process involving inaccuracy check injection, floating-point constraint solving with Z3, and error measurement [1].

Complexity: Traditional complexity measures, such as Cyclomatic Complexity (which counts the number of independent paths through a program's control flow graph) [7], are often used to estimate program complexity. However, these measures do not accurately predict the computational cost of symbolic execution, particularly in the presence of floating-point constraints and deep branching structures.

To better quantify how program complexity impacts symbolic execution, we use Metrinome, a tool designed to analyze path complexity and predict path explosion in symbolic execution engines [8]. Path complexity, also known as Asymptotic Path Complexity (APC), provides a more effective measure for understanding how the number of execution paths grows as program input size increases.

We categorize test programs based on their path complexity: Low Complexity ($O(1)$) for minimal branching (e.g simple arithmetic operations), Polynomial Complexity ($O(n^2)$) for moderate nested loops, and Exponential Complexity ($O(n^n)$) for highly branching programs that lead to rapid path explosion.

By analyzing the dominant term in path complexity using Metrinome, we can predict where FPGen may struggle due to computational overhead.

B. Motivation

Floating-point errors are a critical yet often overlooked issue in numerical software, impacting fields such as scientific computing, engineering simulations, and financial modeling [1]. These errors accumulate over time, leading to significant deviations from expected results. Debugging floating-point inaccuracies is challenging because errors often remain undetected until they impact real-world computations, making it essential to develop systematic techniques for uncovering them.

A well-documented issue in scientific computing is catastrophic cancellation, where subtracting nearly equal numbers leads to a significant loss of precision [1], [9]. For example, numerical weather prediction models rely on floating-point

calculations for temperature, pressure, and wind speed simulations, but small rounding errors propagate across iterations, affecting long-term forecasts [9]. Similarly, in finite element simulations, precision loss in matrix computations can destabilize structural analysis models, leading to inaccurate stress or deformation predictions [10]. These examples highlight the need for automated tools that systematically detect floating-point inaccuracies in complex numerical programs before deployment.

However, detecting such errors is non-trivial, as many test input generation methods fail to expose worst-case numerical errors. Random testing does not systematically explore execution paths where floating-point errors accumulate, making it inefficient for complex numerical applications [2]. Symbolic execution tools, such as KLEE-Float, improve floating-point test generation by analyzing execution paths but struggle with handling deep execution paths and large constraint spaces, making them infeasible for computationally complex programs [4].

FPGen attempts to bridge this gap by leveraging symbolic execution to generate error-inducing floating-point inputs, while incorporating concretization to mitigate path explosion [1]. Prior studies have shown FPGen to be more effective than random search and KLEE-Float, but its performance on highly complex numerical programs remains unexplored. This study aims to evaluate FPGen's effectiveness across varying computational complexities, identifying the limits at which it can still generate meaningful error-inducing inputs for floating-point analysis.

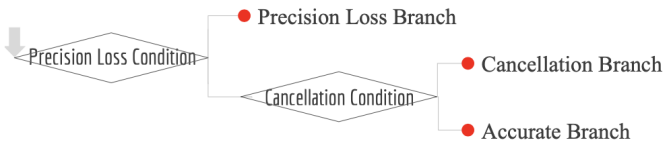

Fig. 2. FPGen detects floating-point inaccuracies by branching execution paths based on precision loss and cancellation conditions [1]

III. Related Work

A. Floating-Point Error Detection

1) Search Heuristics: S3fp [2] is often considered the baseline in generating error-inducing floating-point input [1]. Chiang et al [2] show that through use of additional heuristics, a random search for error-inducing floats can be made significantly more effective. S3fp specifically uses Binary Guided Random Testing, to select likely error-inducing inputs from the range of possible floating-point values. FPGen differs from s3fp in that it uses symbolic execution to avoid the need for random search.

2) Bayesian Optimization: Laguna et al test whether their float generation tool Xscope [3] can identify floating point exceptions in cases where the source code is not accessible (which excludes symbolic execution as an option). Xscope is notable for using Bayesian Optimization, a statistical approach that samples from an approximated probablity distribution. This research is specifically looking to find hard exceptions, such as division by zero, in the context of both CPU and GPU operations. In contrast our research is more concerned with rounding error, which may or may not trigger exceptions, and for CPU operations only.

B. Symbolic Execution

1) Floating Point Error Applications: KLEE-Float [4] was an earlier attempt to determine whether symbolic execution analysis could be applied in generating error-inducing floating points. KLEE-float extends KLEE [4], a symbolic execution engine, in order to solve for a set of floating point inputs. KLEE is the baseline state-of-the-art for solving symbolic execution problems. FPGen primarily adds to symbolic execution by using a concretization heuristic to handle branch explosion.

2) Alternative Symbolic Execution Approaches: Fu and Su [11] explore whether floating-point program testing can be improved through unconstrained programming instead of constraint-solving-based symbolic execution. Their method formulates floating-point test input generation as an optimization problem rather than a constraint-solving problem. It aims to maximize coverage by constructing a function that represents the numerical behavior of the program and optimizes this function for diverse test cases. Unlike FPGen, which uses Z3 SMT solving with symbolic execution, their method removes constraint solving entirely, treating input generation as a free optimization task. While this approach is computationally efficient, it does not systematically explore execution paths as FPGen does, making it less suited for complex program verification.

3) Exception Detection via Symbolic Execution: The work by [12] investigates how symbolic execution can be combined with value-range analysis to improve floating-point exception detection. This paper integrates symbolic execution with value-range analysis, allowing it to identify floating-point operations that might produce NaN, Inf, or overflows. It extends traditional symbolic execution by tracking numerical bounds on floating-point variables, reducing false positives in error detection. While FPGen detects precision loss and catastrophic cancellation, this approach focuses solely on floating-point exceptions (e.g, division by zero, overflows). FPGen's method is more generalized for floating-point accuracy analysis, whereas this paper is specialized for exception handling.

4) Empirical Study on Symbolic Execution for Floating-Point Programs: The study [5] evaluates multiple symbolic execution frameworks, including SMT solvers like Z3 and tools such as KLEE-Float, to determine their effectiveness in handling floating-point computations. This paper investigates "How effective and efficient are current symbolic execution techniques in analyzing floating-point programs?" The authors conduct an empirical study of five symbolic execution tools for floating-point analysis, measuring their scalability, precision in constraint solving, and computational performance. The study provides insights into the strengths and weaknesses of SMT solver-based floating-point analysis. While FPGen builds

upon symbolic execution (particularly KLEE-Float and Z3 SMT solving), this study provides a broader evaluation of existing techniques. Our work extends FPGen by not only utilizing symbolic execution but also assessing its efficiency on identifying where it performs best in high-complexity floating-point operations.

C. Floating-Point Error Analysis and Sensitivity Measurement

1) Dynamic Floating-Point Error Analysis: Herbgrind [9] dynamically monitors floating-point computations to detect precision loss. Herbgrind explores how dynamic execution monitoring can be used to detect and diagnose floating-point errors in large-scale numerical applications. It tracks floating-point computations at runtime, identifying discrepancies between floating-point results and their real-number counterparts. It also traces dependencies between operations and abstracts erroneous computations into simplified fragments for debugging. Herbgrind passively monitors execution to diagnose errors, while FPGen actively generates inputs that trigger numerical inaccuracies. Herbgrind is useful for debugging, whereas FPGen systematically tests numerical robustness across computations.

2) Static Floating-Point Error Analysis: FPCC [13] analyzes floating-point operation chains to detect error propagation. FPCC investigates how floating-point operation chains can be analyzed to detect potential numerical error amplification through static code analysis. It applies chain conditions to model sequences of floating-point operations, helping identify which computations are most susceptible to cumulative precision loss. Unlike dynamic analysis, FPCC does not require program execution. FPGen actively generates inputs that cause errors, whereas FPCC statically identifies error-prone code sections without execution. FPCC is limited to static analysis, while FPGen explores execution paths and evaluates efficiency on computational clusters.

3) Automatic Sensitivity and Error Propagation Analysis: CHEF-FP [14] uses Automatic Differentiation (AD) to measure floating-point sensitivity. CHEF-FP explores how automatic differentiation can be leveraged to quantify the sensitivity of floating-point computations to numerical precision errors. It constructs computational graphs of floating-point operations to measure how small perturbations in precision affect numerical results. This helps developers understand which parts of an algorithm are most sensitive to floating-point errors. CHEF-FP quantifies floating-point sensitivity but does not generate adversarial inputs. FPGen actively searches for worst-case inputs that induce errors. CHEF-FP is suited for error analysis, while FPGen tests robustness across computational clusters.

IV. Approach

Our approach consists of two phases: Replication (to verify FPGen's original results) and Extension (to evaluate its scalability on complex programs).

A. Replication Step (RQ1 - Verifying FPGen's Baseline Performance)

The first phase of our study replicates the benchmark testing for FPGen, originally conducted by Guo and Rubio-González [1]. This ensures that FPGen consistently outperforms random testing in generating floating-point error-inducing inputs.

1) Reproducing the FPGen Benchmark Tests:
   We run FPGen [1] on the same benchmark dataset used in the original study. This dataset consists of floating-point numerical programs designed to test precision loss and rounding errors. Unlike the original study, we do not include S3FP or KLEE-Float comparisons, as they are outside our replication scope.
2) Comparing FPGen with Random Testing (S3FP Alternative):
   We measure relative error magnitude, number of inputs generated, and execution time. Random testing is used as a baseline to evaluate whether FPGen continues to offer a significant accuracy improvement over naïve input generation methods [2].
3) Validating Results Against the Original Study:
   The floating-point errors detected by FPGen in our replication will be compared to the reported results in [1]. Any significant deviations will be analyzed for possible causes, including differences in environment, tool versions, or experimental constraints.

B. Extension Step (RQ2 - Assessing FPGen on Complex Programs)

The second phase extends the original study by evaluating FPGen on programs with increased computational complexity.

1) Measuring Complexity Using Metrinome:
   We use Metrinome [8] to quantify the computational complexity of test programs, recording:
   - Cyclomatic Complexity (CC) – Measures branching complexity [7].
   - NPath Complexity – Captures the total number of unique execution paths.
   - Asymptotic Path Complexity (APC) – Estimates how symbolic execution scales as program size increases.
2) Dataset Expansion:
   We expand the dataset beyond the original benchmarks by including: Programs from the GNU Scientific Library (GSL) [15], covering real-world numerical algorithms. Synthetic test cases, where we create programs with controlled floating-point operations and high branching complexity.
3) This ensures we test FPGen's performance across a broad spectrum of numerical computations.
4) Running FPGen on Extended Test Cases
   We apply the same experimental methodology as in the replication step. The goal is to determine how FPGen's effectiveness scales with complexity. We track:

- Error magnitude – Does FPGen continue to find large floating-point errors?
- Runtime overhead – Does symbolic execution slow down as complexity increases?
- Path Explosion – At what level of complexity does FPGen struggle to generate useful test cases?

5) Comparing FPGen Performance Across Complexity Levels
   We categorize programs into different computational complexity classes based on Metrinome's complexity analysis:
   - Low Complexity ($O(1)$) – Simple arithmetic operations.
   - Moderate Complexity ($O(n^2)$) – Standard numerical methods, nested loops.
   - High Complexity ($O(2^n)$) – Recursive computations, highly branched programs.
6) By evaluating FPGen's error-detection capability at each level, we assess whether it remains effective even for highly complex programs.

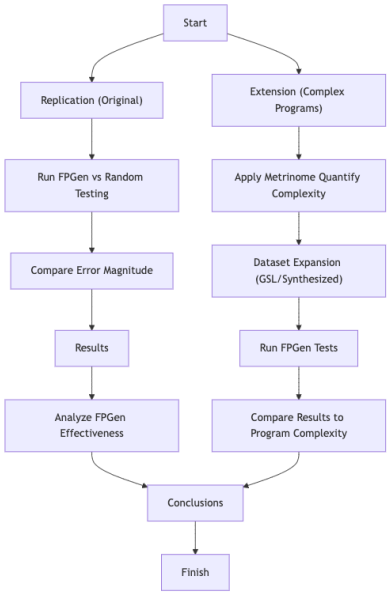

Fig. 3. Proposed Approach: Evaluating FPGen's effectiveness across different program complexities. The approach involves replication of prior work, complexity quantification using Metrinome, dataset expansion, and performance benchmarking.

V. Evaluation

A. Dataset

The evaluation consists of two datasets:

- Replication Dataset (Original FPGen Benchmark) Uses the dataset from FPGen's GitHub artifact [16]. Includes numerical operations from: Gnu Scientific Library (GSL), and Meschach Library. Covers summation functions, statistical computations, and matrix operations, reflecting FPGen's initial evaluation scope.
- Extended Dataset (Complexity Augmentation) The dataset is expanded to include computationally complex numerical programs from two key sources. The first source consists of additional tests selected from the GNU Scientific Library (GSL), chosen based on their increased branching and recursion depth. These tests ensure a more challenging evaluation of FPGen's capabilities. The second source includes synthetic high-complexity programs, which are custom-designed to assess FPGen's scalability in handling intricate floating-point computations.
  To introduce variations in complexity, the dataset incorporates programs that compute summations raised to high exponential powers. Additionally, deeply nested loops and higher branching structures are integrated to stress-test FPGen, evaluating its effectiveness in analyzing complex execution paths and detecting floating-point precision issues under extreme conditions.

This dataset expansion ensures a progressive increase in program complexity, allowing us to analyze FPGen's performance across varying computational challenges.

B. Evaluation Metrics

To address both RQ1 (Replication) and RQ2 (Extension), we define several quantitative metrics for evaluating FPGen's performance and effectiveness.

1) Error Magnitude (Relative Error): The **Error Magnitude** metric is used to evaluate RQ1 by comparing FPGen against Random Testing in terms of floating-point precision loss. This metric quantifies the deviation of the computed result from the expected result, ensuring that the generated inputs lead to meaningful precision loss. It is formally defined as:

$$|r - r_0| / \max\{\epsilon, |r_0|\}$$

where r represents the computed result, r_0 is the expected result, and ϵ is the smallest positive floating-point number [1]. This formulation follows the evaluation setup used in the original FPGen study, enabling a direct comparison to prior findings.

2) Program Complexity (Path Complexity & Symbolic Execution Difficulty): To evaluate RQ2, we categorize program complexity using Metrinome's path complexity metrics [8]. These metrics help assess FPGen's effectiveness in handling complex programs. Three key complexity measures are considered:

- Cyclomatic Complexity (CC) [7] – Quantifies the branching structure of a program.
- NPath Complexity – Measures the total number of possible execution paths.
- Asymptotic Path Complexity (APC) – Estimates how program complexity grows with increasing input size.

These measures provide a structured approach to understanding how FPGen performs across programs of varying complexity levels.

$$\text{APC} = O(1), O(n^2), O(2^n)$$

Complexity Class	Description	Example Programs
O(1) (Low Complexity)	Minimal branching	Basic arithmetic, simple summation
$O(n^2)$ (Moderate Complexity)	Polynomial growth in paths	Nested loops, matrix computations
$O(2^n)$ (High Complexity)	Exponential execution paths	Recursive functions, deep branch programs

TABLE I
COMPLEXITY CLASSIFICATION OF TEST PROGRAMS

This metric helps determine whether FPGen remains effective or suffers from symbolic execution limitations as complexity increases.

C. Experiment Procedure

To ensure our results are fully reproducible, we follow a structured approach for both the replication and extension phases.

1) Replication: Running FPGen on the Original Dataset: We follow the setup instructions from the FPGen GitHub artifact [16], configuring the test environment as described in the original study. The steps include:

1) Setting Up the FPGen Environment
   - Install required dependencies (KLEE, Z3, LLVM, and FPGen dependencies).
   - Clone the FPGen repository and verify correct configuration.
2) Executing FPGen and Random Testing
   - Run FPGen on the original dataset.
   - Run the random search baseline separately for comparison.
   - Ensure test programs are executed with identical time constraints as the original experiment.
3) Recording and Comparing Results
   - Extract error magnitude, execution time, and path complexity metrics.
   - Compare FPGen's error detection performance with the original paper's reported results.
   - Verify if FPGen still exhibits a significant advantage over random testing.

2) Complexity Quantification: Preparing for Extended Analysis: To systematically analyze FPGen's performance across different complexity levels, we use Metrinome [8] to compute:

- Cyclomatic Complexity (CC) [7] – Measures the number of independent paths in a program.
- NPath Complexity – Captures the number of possible execution paths.
- Asymptotic Path Complexity (APC) – Predicts how execution paths scale as input size increases.

These values allow us to categorize test programs and ensure a systematic comparison of FPGen performance across different levels of complexity.

3) Dataset Extension: Searching vs. Synthesizing Complex Programs: To extend the dataset, we introduce more complex numerical programs using two methods:

1) Searching for Complex Functions in GSL - We examine functions from the Gnu Scientific Library (GSL) [15], focusing on those with deeper branching and recursive structures.
2) Synthesizing Higher-Complexity Programs - If existing functions do not provide enough complexity variation, we construct new synthetic test cases with: Deep execution paths (multi-level branching structures), Recursive floating-point operations, Symbolic execution-intensive computations.

This combination ensures we cover a diverse range of computational complexities, testing FPGen's effectiveness in both real-world and edge-case scenarios.

4) Experimental Execution: Running FPGen on Extended Dataset: Once the dataset is extended, we conduct the following steps:

The first step involves preparing symbolic execution-compatible C programs. This requires modifying new test cases to ensure they use KLEE-compatible symbolic inputs, enabling symbolic execution. Additionally, FPGen instrumentation is integrated into these test cases to measure error magnitude, ensuring that precision loss is accurately recorded.

Next, FPGen is executed on each test case to evaluate its performance. The tool is run on all test programs, and key metrics such as error magnitude, runtime, and path complexity are logged for each case. This data collection ensures a comprehensive analysis of FPGen's behavior across different program complexities.

Finally, the collected data is used to analyze the relationship between program complexity and FPGen's performance. Error magnitude is plotted against complexity metrics, including Asymptotic Path Complexity (APC), Cyclomatic Complexity (CC) [7], and NPath Complexity [8]. This analysis helps determine whether FPGen's effectiveness in detecting floating-point errors diminishes as program complexity increases. Additionally, FPGen's runtime scalability is examined to assess its suitability for handling highly complex programs.

This structured approach ensures our methodology is fully reproducible and provides insight into how FPGen scales as complexity increases.

D. Results

1) RQ1: The comparative measured error for FPGen is expected to show similar improvements over random generation as were measured in (the original experiment).

2) RQ2: As complexity increases, concretization should cause the generated inputs to become increasingly more random. Therefore, we expect that performance between randomized inputs and FPGen will converge as a program's cyclomatic complexity increases.

VI. DISCUSSION AND THREATS TO VALIDITY

While FPGen provides an effective mechanism for generating error-inducing floating-point inputs, several limitations

must be considered. A key challenge is the scalability of symbolic execution, which suffers from *path explosion* in highly complex programs [8]. As complexity increases (e.g: $O(2^n)$ recursive computations), FPGen may struggle to explore all execution paths efficiently. Additionally, despite our dataset expansion beyond the original FPGen study, our test cases still cover a limited subset of real-world numerical applications. Evaluating FPGen on large-scale software, such as machine learning models and scientific simulations, remains an open challenge. Another limitation is the dependency on hardware-specific floating-point arithmetic implementations, which may lead to variations in results across different architectures [1]. Moreover, our evaluation adhered to the two-hour execution time per test case from the original FPGen study, potentially limiting the discovery of more error-inducing inputs for highly complex programs.

Several factors may affect the validity of our findings. While we carefully followed FPGen's experimental methodology [16], external influences such as compiler optimizations, OS-level scheduling, and hardware differences could introduce minor inconsistencies. Our dataset extension aimed to incorporate diverse computational complexities, but the selection of test cases both from the GSL [15] and synthetic programs may still not fully generalize to all real-world numerical computations. Additionally, we focused on comparing FPGen with random search [2], leaving out alternative symbolic execution-based tools such as S3FP and KLEE-Float [4]. This omission means that FPGen's performance advantage may be overstated if superior alternatives exist. Another threat to validity lies in the use of relative error magnitude as the primary metric, which may not always reflect the severity of floating-point inaccuracies in practical applications.

To address these limitations, future work should explore hybrid approaches that combine symbolic execution with machine learning for improved path selection and reduced path explosion [3]. Expanding the benchmark scope to large-scale numerical software, such as deep learning frameworks, computational physics models, and financial simulations, would provide a broader evaluation of FPGen's effectiveness. Additionally, FPGen's performance across different computing architectures, including GPUs and ARM processors, should be analyzed to ensure robustness in heterogeneous environments [9]. Incorporating alternative error measurement techniques, such as error propagation analysis and Monte Carlo-based stochastic error modeling, could offer deeper insights into floating-point inaccuracies [10]. Finally, conducting a comparative study between FPGen, S3FP, and KLEE-Float would help determine the most effective tool for floating-point error detection in different computational scenarios [1]. By addressing these challenges, FPGen can be further refined to handle increasingly complex computations and deliver more reliable floating-point error detection.

VII. Contributions

This study investigates the effectiveness of FPGen in detecting floating-point errors across programs of varying computational complexities. We begin by replicating the original FPGen benchmark experiments [16], validating its ability to generate error-inducing floating-point inputs compared to random testing. To extend the evaluation scope, we introduce a dataset containing higher-complexity numerical programs sourced from the Gnu Scientific Library (GSL) [15] and custom synthesized test cases. By categorizing test programs based on their path complexity ranging from low ($O(1)$) to exponential ($O(2^n)$) we systematically analyze how FPGen's performance scales with increasing computational difficulty.

To quantify program complexity, we leverage Metrinome [8], computing cyclomatic complexity, NPath complexity, and asymptotic path complexity. This allows us to evaluate FPGen's ability to generate high-error inputs in complex branching structures and recursive numerical functions. Our results highlight FPGen's strengths in handling moderate-complexity programs but reveal its scalability limitations in highly branching computations due to symbolic execution's inherent path explosion. We also assess FPGen's runtime trends, providing insights into its efficiency trade-offs in different computational scenarios. Finally, we discuss future research directions, including hybrid symbolic execution strategies, alternative error measurement techniques, and expanding benchmark datasets to improve floating-point error detection in large-scale numerical computing applications.

VIII. Data Availability

All details of synthesizing the dataset for RQ2 available in RQ2 Dataset repository

References

[1] H. Guo and C. Rubio-González, "Efficient generation of error-inducing floating-point inputs via symbolic execution," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1261–1272. [Online]. Available: https://doi.org/10.1145/3377811.3380359

[2] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev, "Efficient search for inputs causing high floating-point errors," *SIGPLAN Not.*, vol. 49, no. 8, p. 43–52, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2692916.2555265

[3] I. Laguna, A. Tran, and G. Gopalakrishnan, "Finding inputs that trigger floating-point exceptions in heterogeneous computing via bayesian optimization," *Parallel Computing*, vol. 117, p. 103042, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819123000480

[4] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zähl, and K. Wehrle, "Floating-point symbolic execution: a case study in n-version programming," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '17. IEEE Press, 2017, p. 601–612.

[5] G. Zhang, Z. Chen, and Z. Shuai, "Symbolic execution of floating-point programs: How far are we?" *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 179–188, 2022.

[6] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2008, pp. 337–340.

[7] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[8] G. Bessler, J. Cordova, S. Cullen-Baratloo, S. Dissem, E. Lu, S. Devin, I. Abugharath, and L. Bang, "Metrinome: Path complexity predicts symbolic execution path explosion," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2021, pp. 29–32.

[9] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. Tatlock, "Finding root causes of floating point error with herbgrind," 2018. [Online]. Available: https://arxiv.org/abs/1705.10416

[10] H. Yang, J. Xu, J. Hao, Z. Zhang, and B. Zhou, "Detecting floating-point expression errors based improved pso algorithm," *IET Software*, vol. 2023, no. 1, p. 6681267, 2023. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/2023/6681267

[11] Z. Fu and Z. Su, "Achieving high coverage for floating-point code via unconstrained programming," 2017. [Online]. Available: https://arxiv.org/abs/1704.03394

[12] X. Wu, L. Li, and J. Zhang, "Symbolic execution with value-range analysis for floating-point exception detection," *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 1–10, 2017.

[13] D. Zou, M. Zeng, Y. Xiong, Z. Fu, L. Zhang, and Z. Su, "FPCC: Detecting floating-point errors via chain conditions," *Proc. ACM Program. Lang.*, pp. 1504–1531, 2024.

[14] G. Singh, B. Kundu, H. Menon, A. Penev, D. J. Lange, and V. Vassilev, "Fast and automatic floating point error analysis with chef-fp," 2023. [Online]. Available: https://arxiv.org/abs/2304.06441

[15] B. Gough, *GNU scientific library reference manual*. Network Theory Ltd., 2009.

[16] J. Guo and C. Rubio-González, "Fpgen: Efficient generation of error-inducing floating-point inputs via symbolic execution - artifact," https://github.com/ucd-plse/FPGen, 2020, accessed: 2024-02-16.